《编译原理和技术》

导论

李诚

中国科学技术大学
计算机科学与技术学院
2022-08-29

本节提纲

- 从C代码到可执行程序
- 编译原理课程简介
- 编译器的一般构造
- 学好了编译，能干啥?

从C程序到可执行文件

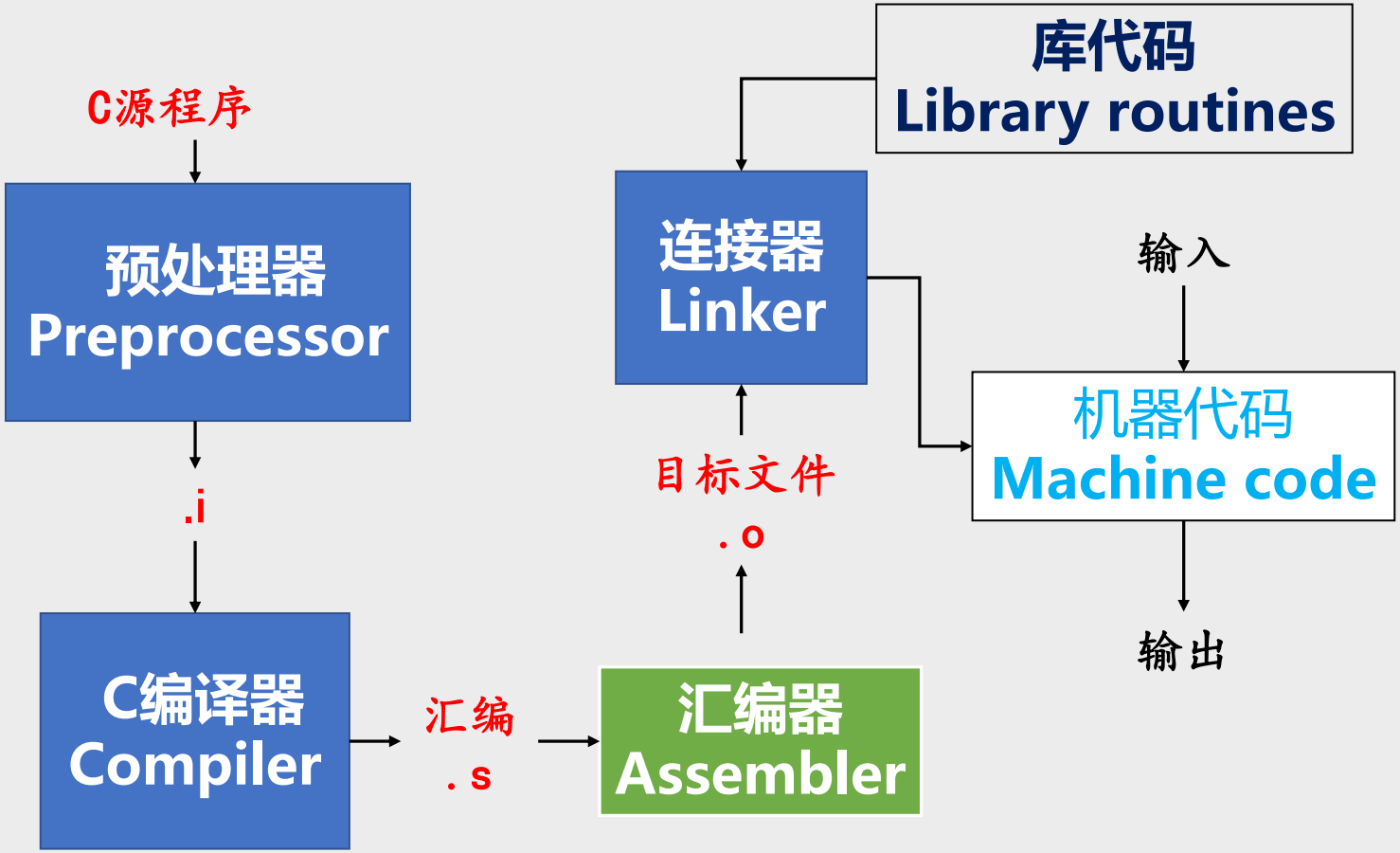
```c
#include <stdio.h>
int main()
{
  printf("hello, world!\n");
}

/* helloworld.c */
```

[root@host ~]# clang helloworld.c -o helloworld

[root@host ~]# ./helloworld

hello, world!

C程序的编译过程分解


C源程序
预处理器
Preprocessor
.i
C编译器
Compiler
汇编
.s
汇编器
Assembler
目标文件
.o
连接器
Linker
库代码
Library routines
输入
机器代码
Machine code
输出

C程序的编译过程分解

- **预处理：**
  - clang -E helloworld.c > helloworld.i
  - 将stdio.h内容放到helloworld.c中，并生成新文件
- **编译：**
  - clang -S helloworld.i
  - 生成汇编代码，生成helloworld.s文件
- **汇编：**
  - clang -c helloworld.s -o helloworld.o
  - 翻译为机器码
- **链接：**
  - clang helloworld.o -lm

C程序的编译过程分解

- **思考题：**
  - 参考clang的编译过程，请用gcc编译器进行上述过程的分解！
- **参考链接：**
  - http://www.cs-fundamentals.com/c-programming/how-to-compile-c-program-using-gcc.php

什么是/为什么需要编译器?

- **高级语言**
  - 直接面向开发者
  - 与数学公式类似
  - 编程效率高
- **机器语言**
  - 驱动硬件完成具体任务
  - 编程效率低
- **编译器提供程序开发的便捷性**
  - 实现人机交流，将人类易懂的高级语言翻译成硬件可执行的目标机器语言

编译器还有哪些重要功能?

编译器还有哪些重要功能?

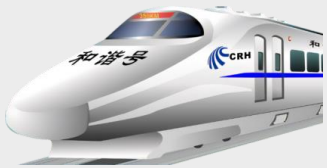
- **错误恢复!**
  - int !a;
  - int1 a;
  - f(a,b,c,d;
  - 使用了没有定义的变量
  - 指针没有初始化
- **编译器如何自动识别错误的?**

编译器还有哪些重要功能?

- **错误恢复!**
  - int !a;
  - int1 a;
  - f(a,b,c,d;
  - 使用了没有定义的变量
  - 指针没有初始化
- **编译器如何自动识别错误的?**
  - 基于编程语言理论的合法性检查
  - 代码分析

编译器还有哪些重要功能?

编程人员无感知的代码优化!



优化等级	简要说明
-Ofast	在-O3级别的基础上，开启更多**激进优化项**，该优化等级不会严格遵循语言标准
-O3	在-O2级别的基础上，开启了更多的**高级优化项**，以编译时间、代码大小、内存为代价获取更高的性能。
-Os	在-O2级别的基础上，开启**降低生成代码体量**的优化
-O2	开启了大多数**中级优化**，会改善编译时间开销和最终生成代码性能
-O/-O1	优化效果介于-O0和-O2之间
-O0	默认优化等级，即**不开启编译优化**，只尝试减少编译时间

延伸阅读：https://clang.llvm.org/docs/CommandGuide/clang.html#code-generation-options

案例演示——优化对代码性能的影响

1000000000次循环迭代累加

```
#include <stdio.h>
#include <time.h>
int main() {
    int loop = 1000000000;
    long sum = 0;
    int start_time = clock();
    int index = 0;
    for (index = 0; index < loop; index ++)
    {
        sum += index;
    }
    int end_time = clock();
    printf("Sum : %ld, Time Cost : %lf \n", sum, (end_time - start_time) * 1.0 / CLOCKS_PER_SEC);
    return 0;
}
```

循环次数定义

开始计时

循环体

结束计时

代码运行时间输出

案例演示——优化对代码性能的影响

- gcc –O0 无优化执行

```
gloit@gloit-x1c ~/2022_compiler_demo master gcc -O0 add.c
gloit@gloit-x1c ~/2022_compiler_demo master ./a.out
Sum: 49999999500000000, Time Cost: 3.415244
```

- gcc –O1 中级优化执行

```
gloit@gloit-x1c ~/2022_compiler_demo master gcc -O1 add.c
gloit@gloit-x1c ~/2022_compiler_demo master ./a.out
Sum: 49999999500000000, Time Cost: 0.554717
```

性能提升5倍

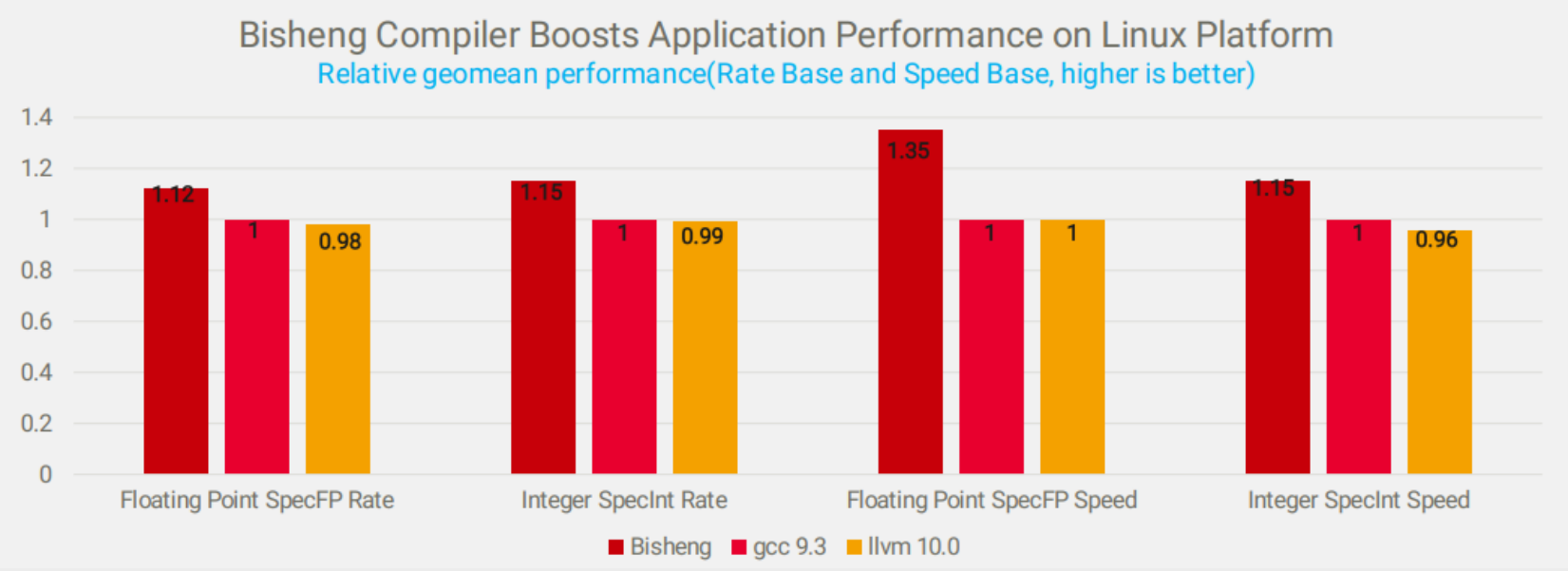
- gcc –O2 高级优化执行

```
gloit@gloit-x1c ~/2022_compiler_demo master gcc -O2 add.c
gloit@gloit-x1c ~/2022_compiler_demo master ./a.out
Sum: 49999999500000000, Time Cost: 0.000002
```

性能提升数十万倍

国产开源编译器——毕昇编译器

- 毕昇编译器通过**编译优化**提升鲲鹏硬件平台上业务的性能体验，SPEC2017性能较业界编译器平均高15%以上。



SPEC作为业界芯片性能评分标准，SPEC的分数可以直观体现出硬件的性能，越高越好

编译器还有哪些重要功能?

- **运行时管理，支持程序正确的运行**
  - 全局与局部变量的维护
  - 函数间参数的传递
  - 函数调用状态管理

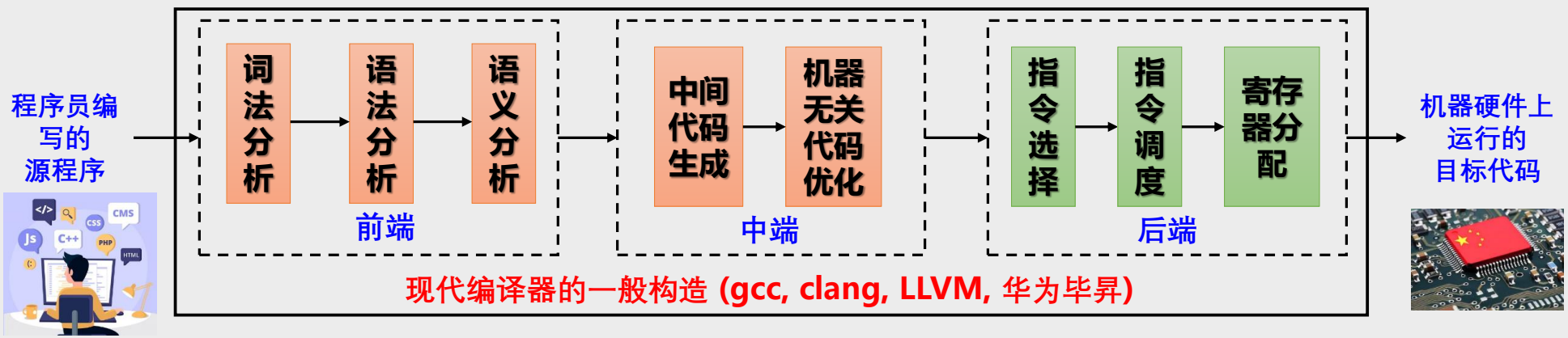
编译器还有哪些重要功能?

- **交叉编译**
  - 支持多种指令集体系结构（ISA）
    - llc –version （clang是LLVM编译器工具集的前端）
  - 支持多语言
    - clang支持C/C++/Objective-C/Objective-C++
    - gcc支持C/C++/Objective-C/Fortran/Pascal/Java/Ada/Go

本节提纲

- 从C代码到可执行程序
- 编译原理课程简介
- 编译器的一般构造
- 学好了编译，能干啥?

编译课程简介



“编写编译器的原理和技术具有普遍的意义，以至于在每个计算机科学家的研究生涯中，该书中的原理和技术都会反复用到。”

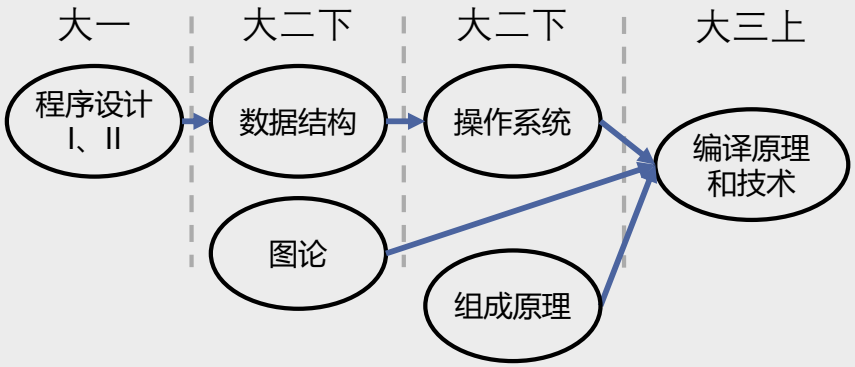
——著名计算机专家 *Alfred V. Aho*

“在供应链可控性上，仍存在编译工具依赖国外的情况。”

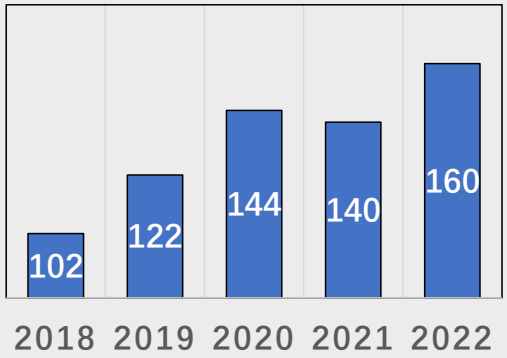
“我们应重点突破操作系统内核、编译器等关键技术。”

——《中国信息技术产品安全可控年度发展报告》

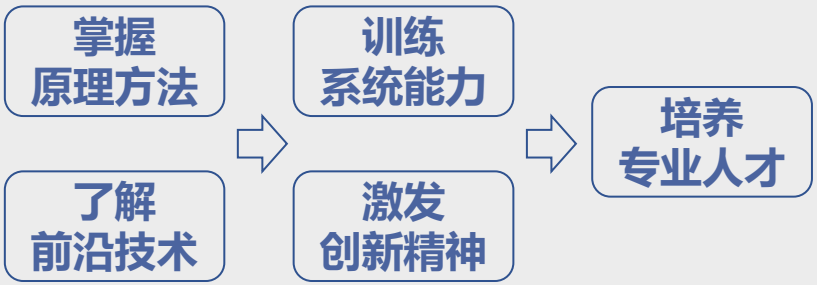
学情分析



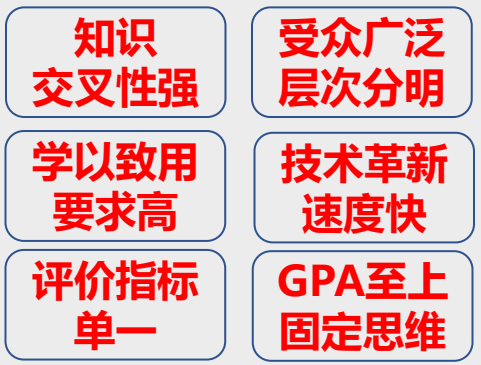
专业核心课及其诸多先导课程



选课人数逐年增多



课程教学目标艰巨



教学过程中面临很多挑战

教学目标

- **基本目标：使学生掌握从零开始构造一个能将类-C简单语言编写的程序翻译为简单目标机器代码（三地址码）的编译器**
  - 正确程序能运行、简单错误能处理
  - 部分代码性能得到优化（机器无关）
  - 较为熟练地使用产业界先进的编程思想和系统工具
- **高阶目标：使学生掌握从零开始构造一个能将类-C语言编写的程序翻译为LoongArch代码的编译器**
  - 手写寄存器分配算法
  - 后端指令选择与调度
  - 高级机器无关/相关优化
  - 语言特性能扩展

课程设置

- 时间：每周一(6,7)、三(3,4)
- 地点：3C303
- qq群（290551715）：发布通知、非业务交流
- 课程实验平台：
  - 发布讲义+实验信息+课程讨论
  - 已搭建完成，近期发布

参考资料

教材和参考书

- 陈意云、张昱，编译原理（第3版），高等教育出版社，2014
- A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, Compilers: Principles, Techniques, and Tools , 2nd edition, Addison-Wesley, 2007
- A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman 著，赵建华等译，编译原理，机械工业出版社，2017

其他资料

- Stanford课程主页

http://web.stanford.edu/class/cs143/

- MIT课程主页：

http://6.035.scripts.mit.edu/fa18/

考核要求

- **考核内容包括理论学习、工程实践**
- **成绩组成：**
  - 平时考核（15%）：
    - 按时上课（特殊情况不能来需要书面请假）
    - 按时完成课后作业
  - 工程实践（30%）：**个人实验**（实验难度降低，体验度进一步提升）
  - 考试（55%）
  - Bonus加分（直接加到总评，细则后续公布）
    - issue发帖不再加分
    - 开放赛道加分

实验辅导团队

徐伟 博士，高级实验师

xuweihf@ustc.edu.cn

2012年博士毕业于中国科学技术大学计算机软件与理论专业。2012年~2019年在中国电子科技集团公司第三十八研究所先后担任工程师(2012~2015)和高级工程师(2015~2019)。2020年回校任职至今。工作于计算机学院实验教学中心，主要负责本学科软件类课程实验教学。

实验辅导团队

张钏楠　硕士二年级

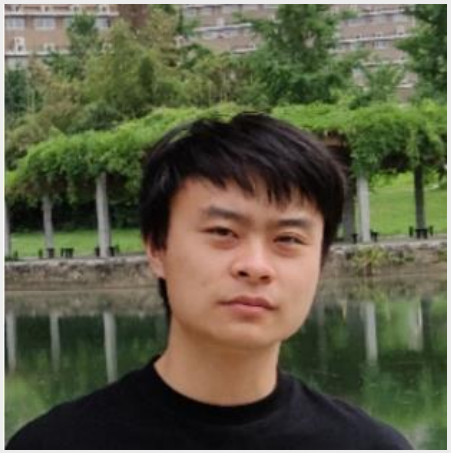
edwardzcn@mail.ustc.edu.cn

爱好广泛，兴趣很多的悲观理想主义者。期待和大家共同进步

郑环宇　硕士一年级

zhenghy22@mail.ustc.edu.cn

本科：清华大学



陈清源　硕士二年级

chen16614@mail.ustc.edu.cn

2020编译比赛冠军

实验辅导团队

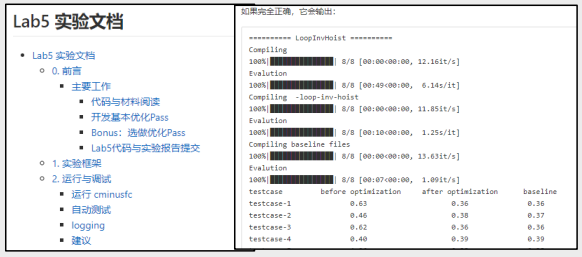
王晨晗 大四学生
wch0925@mail.ustc.edu.cn

李庆 大四学生
lq_2019@mail.ustc.edu.cn

张栋澈 大四学生
farmerzhang1@mail.ustc.edu.cn
编译比赛三等奖

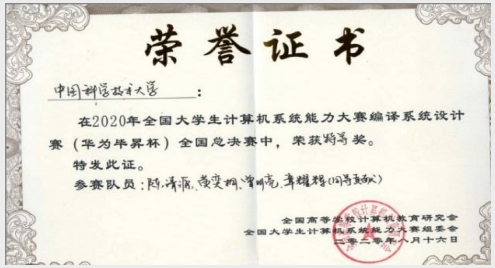
甘文迪 大四学生
gwdx@mail.ustc.edu.cn
编译比赛三等奖

已取得的成绩

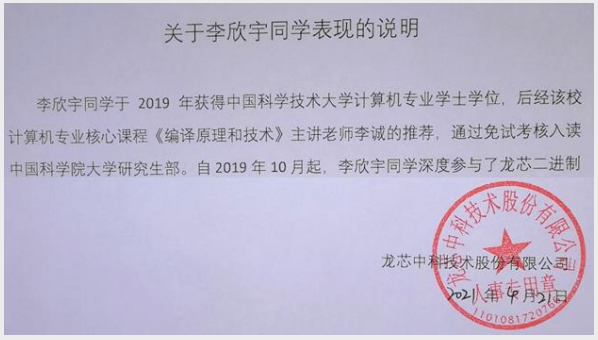


前沿且开放的实践框架
4.5万字文档、4000行框架代码

正在编撰《面向自主指令集的编译器设计与实现》书稿，将由高教社出版



指导学生获全国编译系统比赛
特等(1)、二等(2)、三等(3)



毕业学生从事龙芯编译生态工作
获得积极正面评价

毕业生问卷调查中
编译课程Top10/1500

荣获安徽省第五届青教赛
工科组一等奖、全国创新教学大赛
安徽省正高组二等奖

实验相关的视频

- https://www.bilibili.com/s/video/BV1oR4y1G72s

本节提纲

- 从C代码到可执行程序
- 编译原理课程简介
- **编译器的一般构造**
- **学好了编译，能干啥?**

编译器的输入

- **标准的指令式语言(Java, C, C++)**
  - 状态
    - 变量
    - 结构
    - 数组
  - 计算
    - 表达式 (arithmetic, logical, etc.)
    - 赋值语句
    - 条件语句 (conditionals, loops)
    - 函数

编译器的输出

- **状态**
  - 寄存器
  - 内存单元
- **机器码 – load/store architecture**
  - Load, store instructions
  - 寄存器操作– Arithmetic, logical operations
  - 分支指令– Branch instructions

编译器的构造/阶段

Lexical Analyzer 词法分析器

Source code 源程序

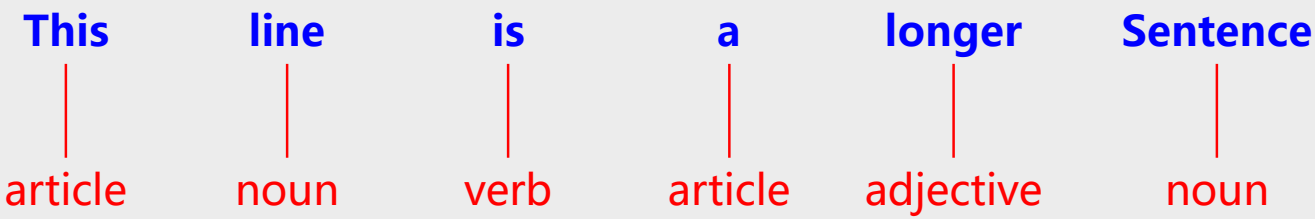
Token Stream 记号流

Symbol Table 符号表

Error Handler 错误处理

词法分析

- 人类在理解自然语言时，首先要识文断字

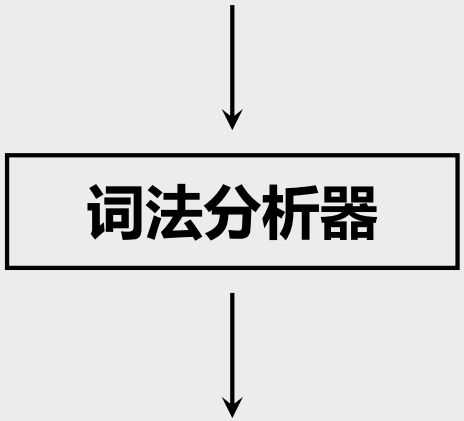


- 通常称为线性分析（linear analysis）

词法分析

- 将程序字符流分解为记号（Token）序列
  - 形式：<token_name, attribute_value>

position = initial + rate * 60 ← 字符流



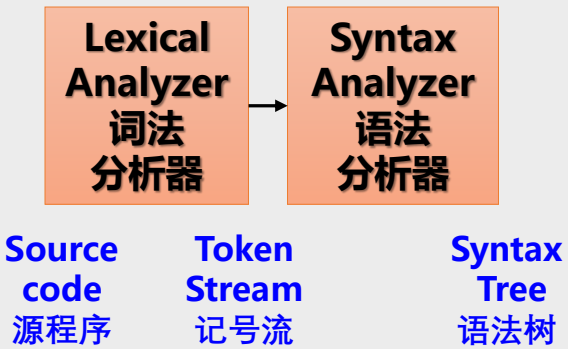
符 号 表

1	position	· · ·
2	initial	· · ·
3	rate	· · ·

⟨id, 1⟩ ⟨=⟩ ⟨id, 2⟩ ⟨+⟩ ⟨id, 3⟩ ⟨*⟩ ⟨60⟩ ← 记号流

命令行输入：clang -cc1 -dump-tokens test.c

编译器的构造/阶段

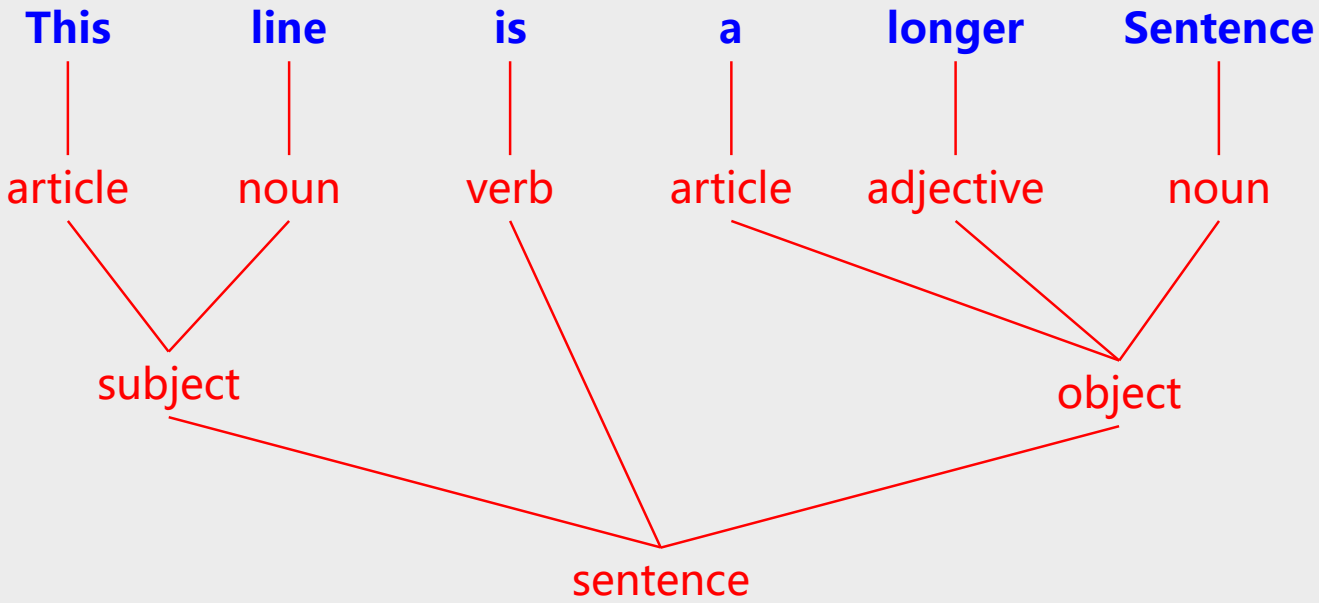


Symbol Table 符号表

Error Handler 错误处理

语法分析

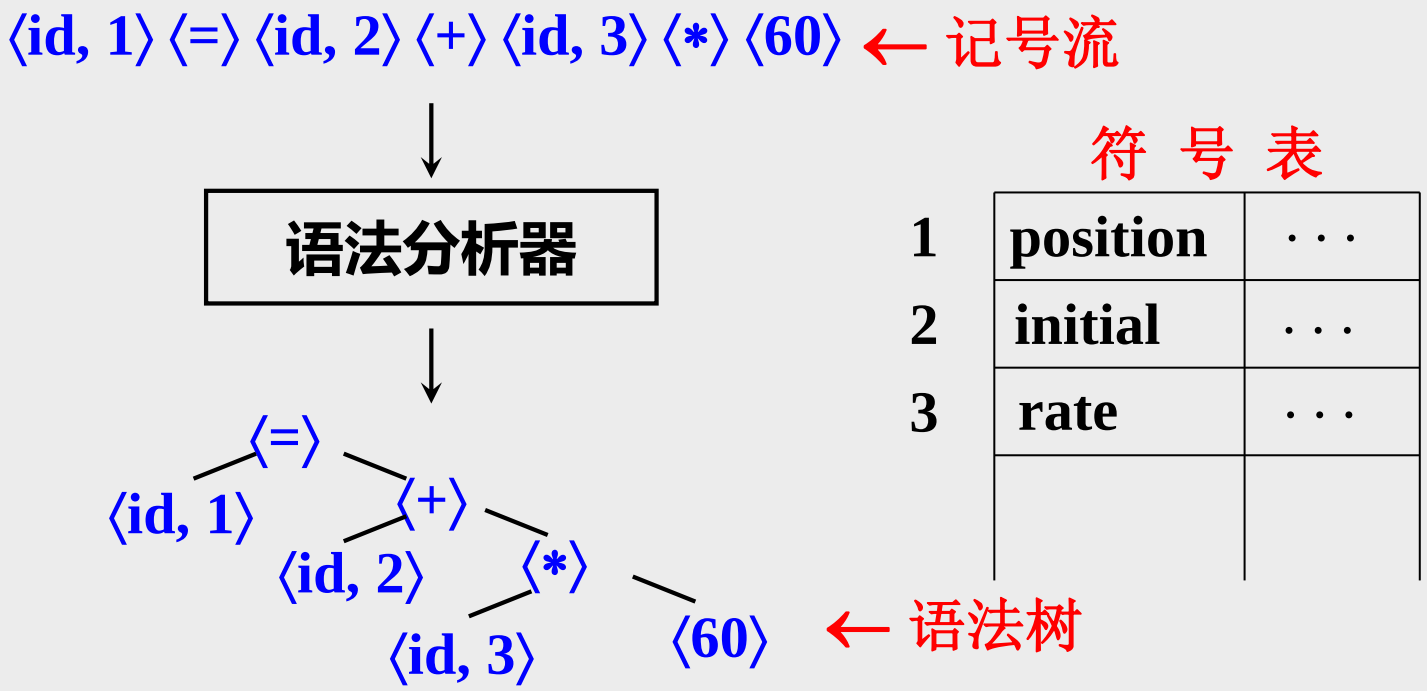
- 人类在理解自然语言时，其次要理解句子结构



- 也称为层次分析（Hierarchical analysis）

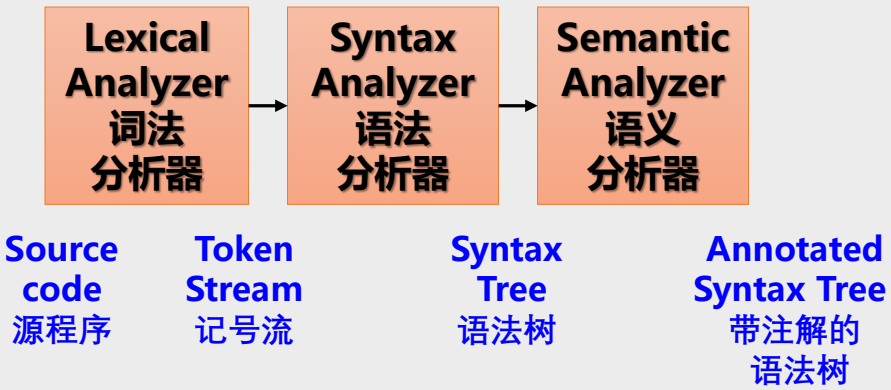
语法分析

- 也称为解析（Parsing），在词法记号的基础上，创建语法结构



命令行输入：clang -fsyntax-only -Xclang -ast-dump test.c

编译器的构造/阶段



Symbol Table 符号表

Error Handler 错误处理

语义分析

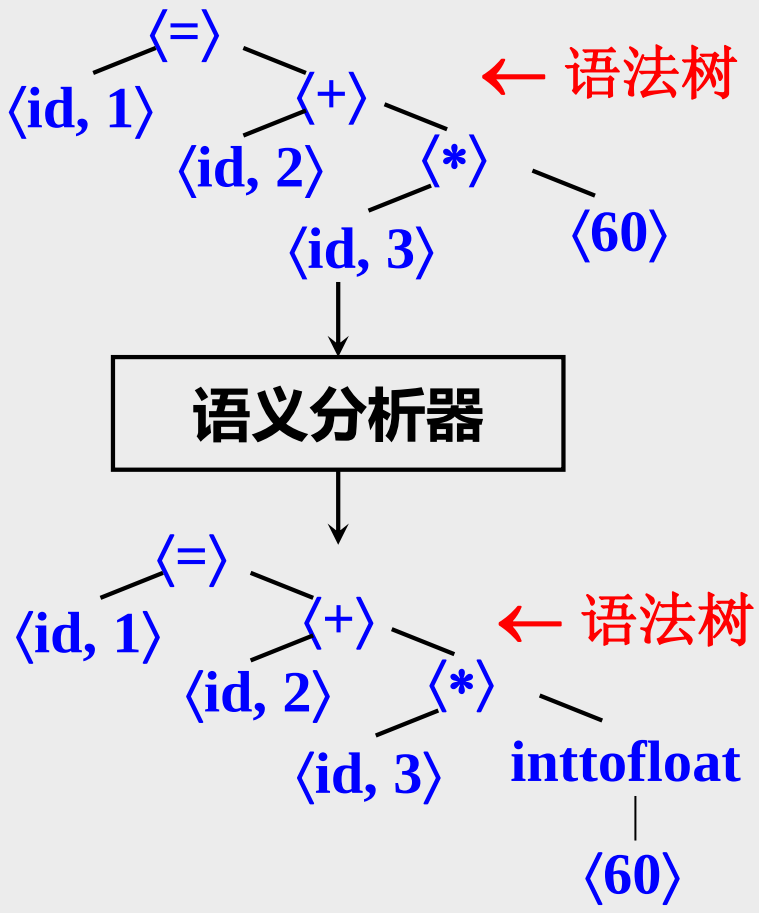
- **人类在理解自然语言时，最后要理解句子的含义**
  - Jack said Jerry left his assignment at home.
    - What does “his” refer to? Jack or Jerry?

语义分析

- 编译器会检查程序中的不一致
  - 如：类型检查（type checking）

符 号 表

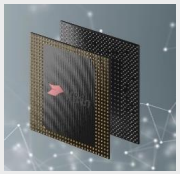
1	position	· · ·
2	initial	· · ·
3	rate	· · ·

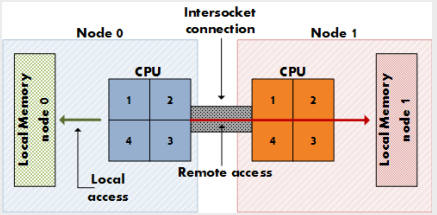


注：类型转换在astdump时已经完成

编译器面临的挑战

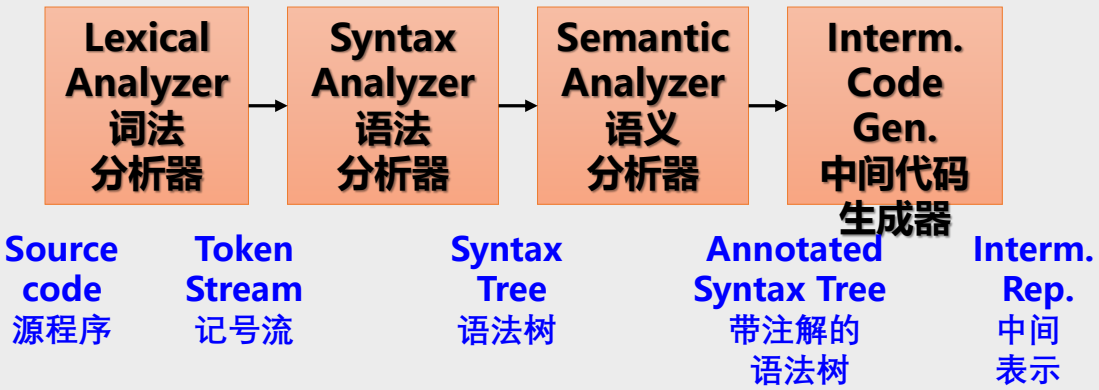
- 计算机是不断进化的
  - 体系结构的改变→编译器的改变
  - 新的特征产生新的问题





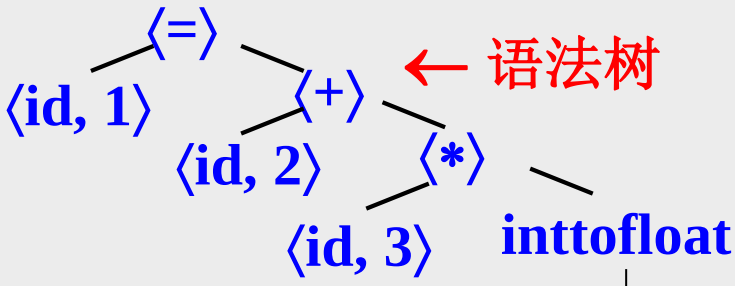
- 语言也在不断演化
  - C - C90, C99, C11； C++ - 1998, 2003, 2006, 2011, 2014
  - 新的语言不断诞生：Go (2009), Rust (2010), Elixir (2011), Swift (2014)

编译器的构造/阶段


Lexical Analyzer 词法 分析器
Syntax Analyzer 语法 分析器
Semantic Analyzer 语义 分析器
Interm. Code Gen. 中间代码 生成器
Source code 源程序
Token Stream 记号流
Syntax Tree 语法树
Annotated Syntax Tree 带注解的 语法树
Interm. Rep. 中间 表示
Symbol Table 符号表
Error Handler 错误处理

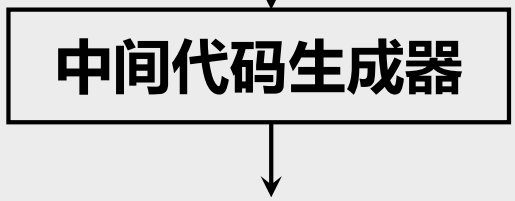
中间代码生成

- 是源语言与目标语言之间的桥梁



符 号 表

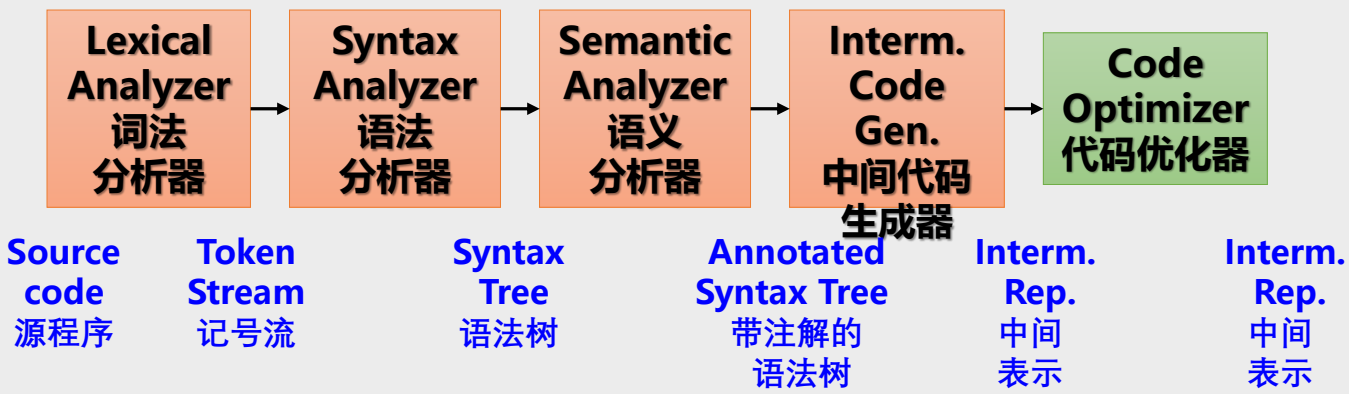
1	position	· · ·
2	initial	· · ·
3	rate	· · ·



t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2 ← 中间代码
id1 = t3

命令行输入：clang -cc1 test.c -emit-llvm -o test.ll

编译器的构造/阶段


Lexical Analyzer 词法 分析器
Syntax Analyzer 语法 分析器
Semantic Analyzer 语义 分析器
Interm. Code Gen. 中间代码 生成器
Code Optimizer 代码优化器
Source code 源程序
Token Stream 记号流
Syntax Tree 语法树
Annotated Syntax Tree 带注解的 语法树
Interm. Rep. 中间 表示
Interm. Rep. 中间 表示
Symbol Table 符号表
Error Handler 错误处理

代码优化

- 机器无关的代码优化便于生成**执行时间更快、更短或能耗更低**的目标代码

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

← 中间代码

符 号 表

1	position	· · ·
2	initial	· · ·
3	rate	· · ·

↓

代码优化器

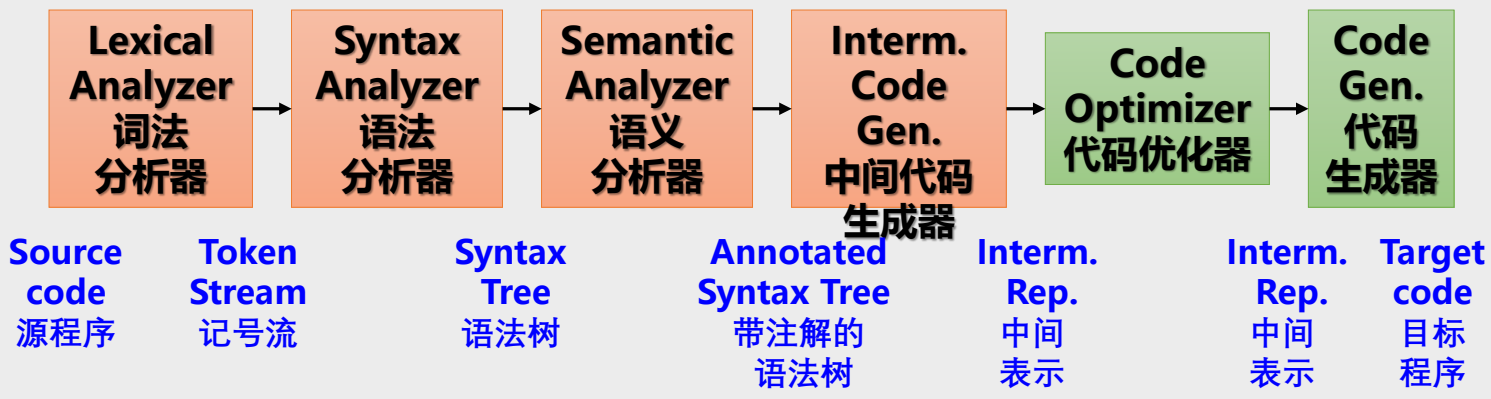
↓

```
t1 = id3 * 60.0
id1 = id2 + t1
```

← 中间代码

命令行输入：clang -S -emit-llvm -O3 test.c -o test-new.ll

编译器的构造/阶段


Lexical Analyzer 词法分析器
Syntax Analyzer 语法分析器
Semantic Analyzer 语义分析器
Interm. Code Gen. 中间代码生成器
Code Optimizer 代码优化器
Code Gen. 代码生成器
Source code 源程序
Token Stream 记号流
Syntax Tree 语法树
Annotated Syntax Tree 带注解的语法树
Interm. Rep. 中间表示
Interm. Rep. 中间表示
Target code 目标程序
Symbol Table 符号表
Error Handler 错误处理

代码生成

- 如果目标语言是机器代码，必须为变量选择寄存器或内存位置

```
t1 = id3 * 60.0
id1 = id2 + t1
```
← 中间代码

↓

代码生成器

↓

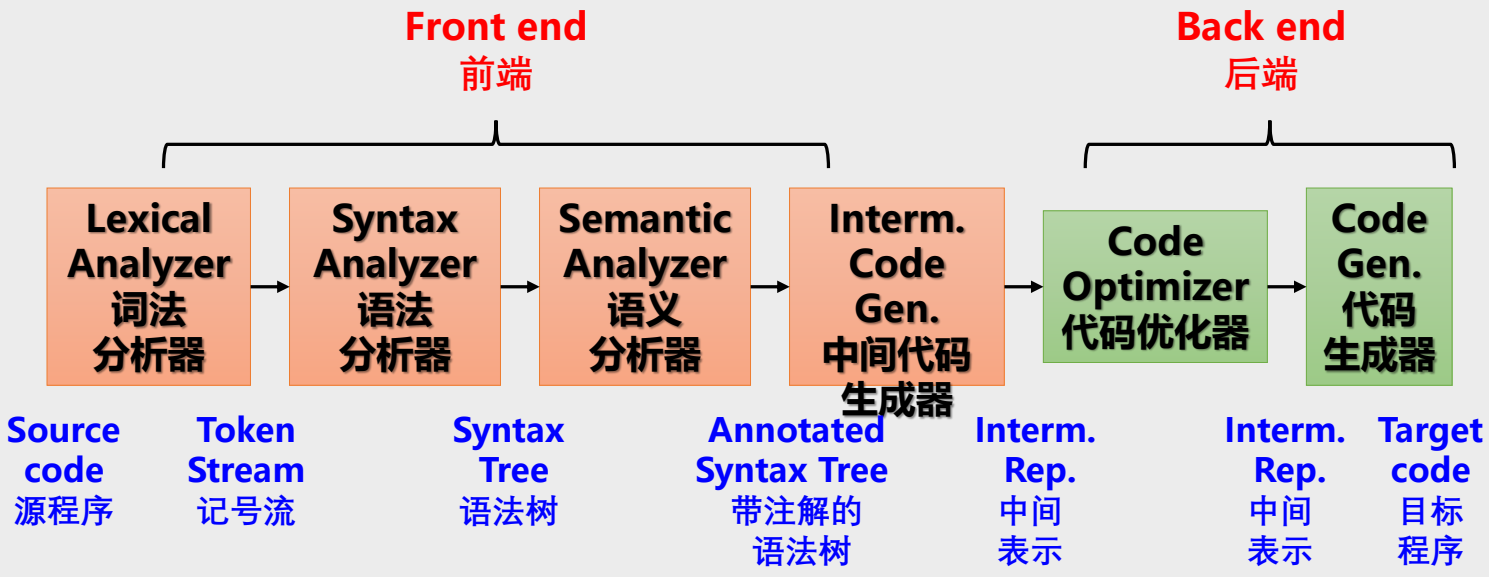
```
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```
← 汇编代码

符 号 表

1	position	· · ·
2	initial	· · ·
3	rate	· · ·

命令行输入：llc-14 test.ll -o test.s

编译器的构造/阶段


Front end
前端
Back end
后端
Lexical Analyzer 词法 分析器
Syntax Analyzer 语法 分析器
Semantic Analyzer 语义 分析器
Interm. Code Gen. 中间代码 生成器
Code Optimizer 代码优化器
Code Gen. 代码 生成器
Source code 源程序
Token Stream 记号流
Syntax Tree 语法树
Annotated Syntax Tree 带注解的 语法树
Interm. Rep. 中间 表示
Interm. Rep. 中间 表示
Target code 目标 程序
Symbol Table 符号表
Error Handler 错误处理

本节提纲

- 从C代码到可执行程序
- 编译原理课程简介
- 编译器的一般构造
- **学好了编译，能干啥?**

宽阔的科研道路

- 图灵奖自1966年颁发以来，共有75名获奖者，其中编译相关的科研人员有21位，占比28%，Alan J. Perlis因编译技术贡献成为第一位获得图灵奖的科学家。

年份	科学家	贡献	年份	科学家	贡献
1966	Alan J. Perlis	高级程序设计技巧，编译器构造	1972	Edsger Dijkstra	程序设计语言的科学与艺术
1974	Donald E. Knuth	算法分析、程序设计语言的设计、程序设计	1976	Michael O. Rabin Dana S. Scott	非确定性自动机
1977	John Backus	高级编程系统，程序设计语言规范的形式化定义	1979	Kenneth E. Iverson	程设语言和数学符号，互动系统的设计，程设语言的理论与实践
1987	John Cocke	编译理论，大型系统的体系结构，及RISC计算机的开发	2005	Peter Naur	Algol 60语言
2006	Frances E. Allen	优化编译器	2020	Jeffrey David Ullman Alfred Vaino Aho	推进编程语言实现的基础算法和理论、教材撰写

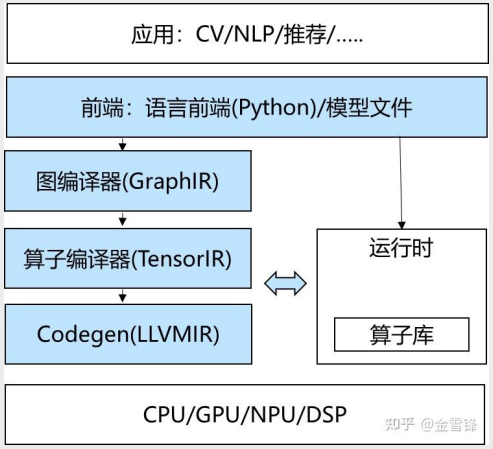
宽阔的科研道路

- 图灵奖自1966年颁发以来，共有75名获奖者，其中编译相关的科研人员有21位，占比28%，Alan J. Perlis因编译技术贡献成为第一位获得图灵奖的科学家。

年份	科学家	贡献	年份	科学家	贡献
1980	C. Antony R. Hoare	程序设计语言的定义与设计	1983	Ken Thompson Dennis M. Ritchie	UNIX操作系统和C语言
1984	Niklaus Wirth	程序设计语言设计、程序设计	2001	Ole-Johan Dahl Kristen Nygaard	面向对象编程
2003	Alan Kay	面向对象编程	2008	Barbara Liskov	编程语言和系统设计的实践与理论
2021	Jack J. Dongarra	通过对线性代数运算的高效数值算法、并行计算编程机制和性能评估工具的贡献，引领了高性能计算的世界。			

工业界关注的焦点

- **A New Golden Age for Computer Architecture**
  - By John L. Hennessy, David A. Patterson；Communications of the ACM
- **新硬件推陈出新的时代**
  - GPU、DPU、TPU、NPU、xPU
  - 量子计算机
- **新应用不断涌现的时代**
  - AI计算、科学计算（AlphaFold）、量子计算
- **编译技术的需求日益旺盛**
  - 华为方舟、毕昇编译器、龙芯二进制翻译、AI编译器（XLA、TVM）



产业界人才紧缺

编译器 [杭州·滨江区·长河]

25-50K 经验不限 本科 刘女士 | 编译器与芯片...

华为

计算机软件 不需要融资 10000人以上

Go | 后端开发

加班补助，补充医疗保险，定期体检，五险一金，年终奖，...

编译器开发 [西安·雁塔区·高新软件园]

20-40K·16薪 经验不限 硕士 陈先生 | 研发工程师

龙芯中科

电子/半导体/集成电路 已上市 500-999人

编译器 | Loongarch | 龙芯

定期体检，住房补贴，生日福利，员工旅游，包吃，补充医...

编译器工程师 [上海·浦东新区·金桥]

18-35K 1-3年 本科 王先生 | 软件研发工程师

华为技术有限公司

计算机软件 不需要融资 10000人以上

数据结构 | 架构师 | ARM开发

住房补贴，定期体检，离职补偿，零食下午茶，补充医疗保...

编译器工程师 [上海·徐汇区·徐家汇]

20-30K·15薪 经验不限 本科 韩女士 | 技术经理

麒麟软件

计算机软件 未融资 1000-9999人

算法基础 | Linux | 编译工具链 | 汇编 | 反汇编

年终奖，免费班车，员工旅游，定期体检，五险一金，员工...

其他…

- **对相关领域的支撑作用**
  - 网络空间安全、设计领域专用语言、自然语言处理
- **个人能力提升**
  - 写出更高质量的代码
  - 锻炼工程实践能力
  - 启发创新思想
  - and more…

《编译原理和技术》

导论

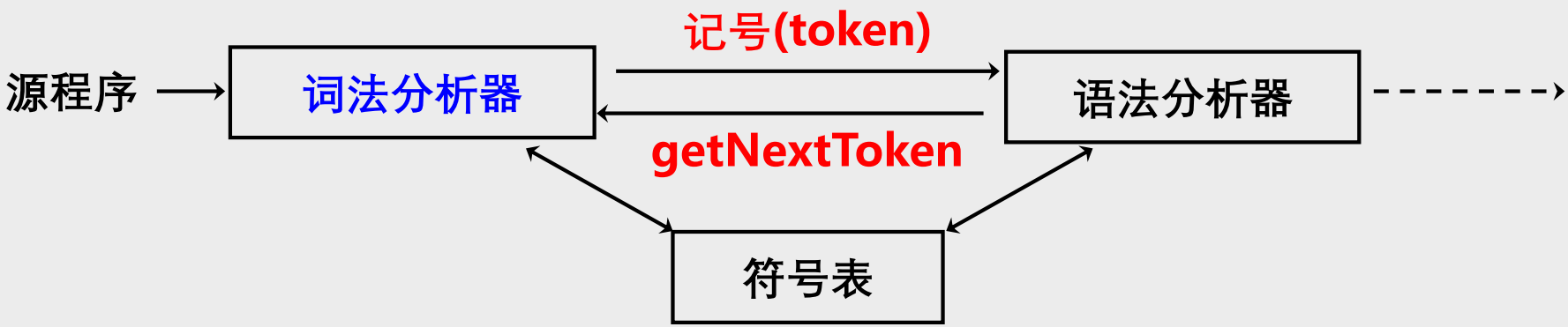
谢谢！

《编译原理和技术》

词法分析

中科大计算机学院
李 诚
2022-08-31

本节提纲



- **词法分析概述**
- **词法分析器的自动生成**
  - 词法单元的描述：正则式
  - 词法单元的识别：转换图
  - 有限自动机：NFA、DFA
  - 正则表达式→NFA → DFA →化简的DFA

词法分析 (Lexical Analysis)

- **程序示例：**

```
if (i == j)
   printf("equal!");
else
   num5 = 1;
```

- **程序是以字符串的形式传递给编译器的**

\tif (i == j)\n\t\tprintf("equal!");\n\telse\n\t\tnum5 = 1;

- **目的：将输入字符串识别为有意义的子串**
  - 子串的种类（Name）
  - 可帮助解释和理解该子串的属性（Attribute）
  - 可描述具有相同特征的子串的模式（Pattern）

词法单元 token

词法单元 (Token)

- **由一个记号名和一个可选的属性值（可以为空）组成**
  - token := <token_name, attribute_value>
- **属性记录词法单元的附加属性**
  - 例：标识符id的属性包括词素、类型、第一次出现的位置等
  - 保存在符号表（Symbol table）中，以便编译的各个阶段取用

源程序
position = initial +
rate * 60

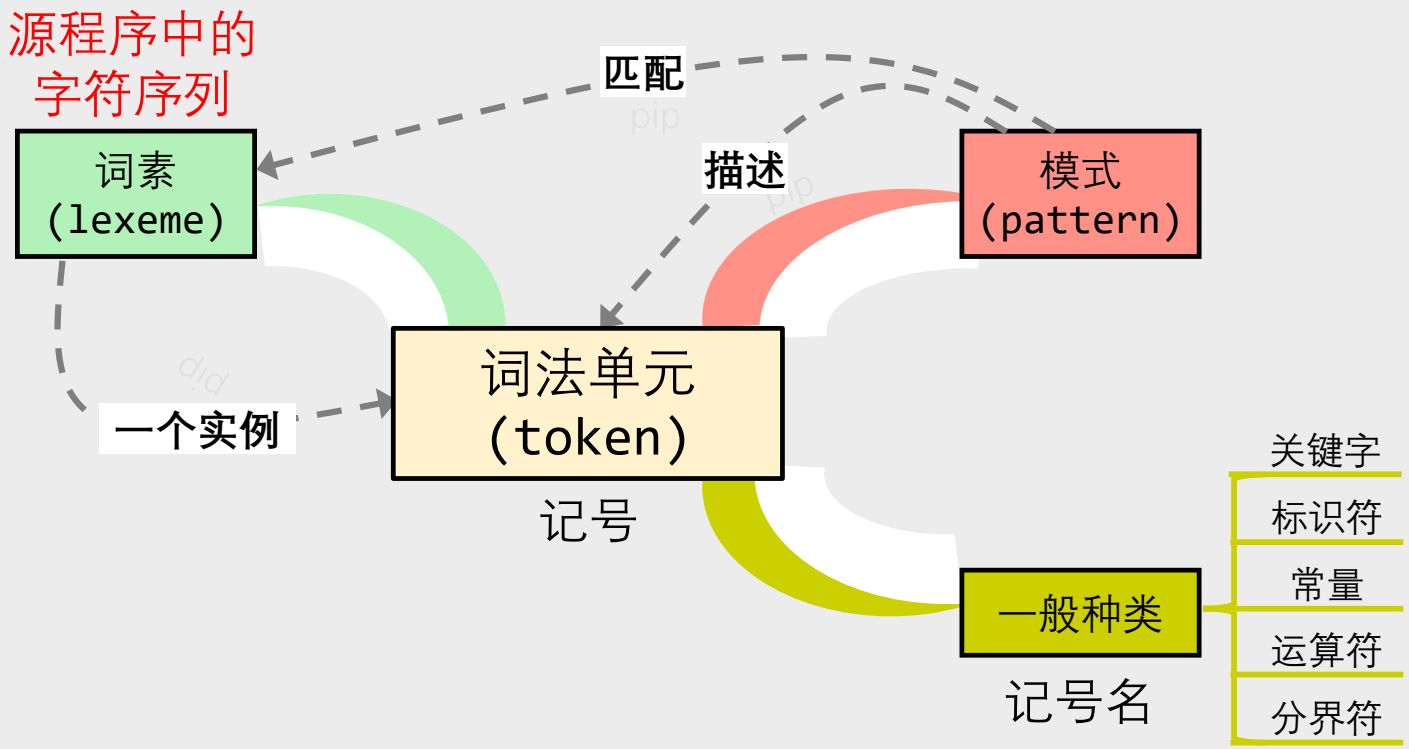
⟨id，指向符号表中position条目的指针⟩
⟨assign _ op⟩
⟨id，指向符号表中initial条目的指针⟩
⟨add_op⟩
⟨id，指向符号表中rate条目的指针⟩
⟨mul_ op⟩
⟨number，整数值60⟩

符 号 表

1	position	· · ·
2	initial	· · ·
3	rate	· · ·

词素
(实例)

四个关键术语

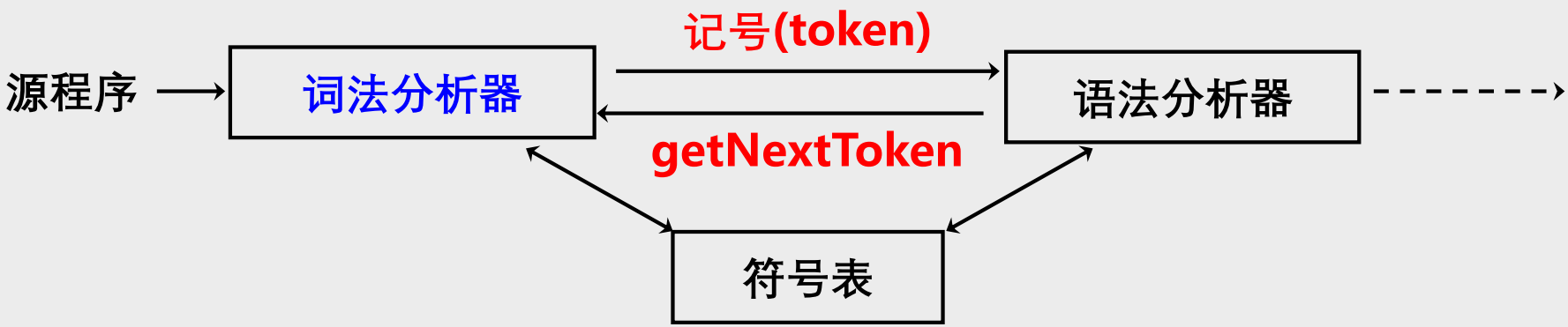

源程序中的
字符序列
匹配
描述
词素
(lexeme)
模式
(pattern)
一个实例
词法单元
(token)
记号
关键字
标识符
常量
运算符
分界符
一般种类
记号名

词法单元(记号)、实例与模式

```
if (i == j) printf("equal!");
else num5 = 1;
```

记号名	实例（词素）	模式的非形式描述
if	if	字符i, f
else	else	字符e, l, s, e
relation	==, < , <= , …	== 或 < 或 <= 或 …
id	i, j, num5	由字母开头的字母数字串
number	1, 3.1, 10, 2.8 E12	任何数值常数
literal	“equal!”	引号“和”之间任意不含引号本身的字符串

本节提纲



- 词法分析概述
- **词法分析器的自动生成**
  - 词法单元的描述：正则式
  - 词法单元的识别：转换图
  - 有限自动机：NFA、DFA
  - 正则表达式→NFA → DFA →化简的DFA

正整数的描述

- **正整数描述了一个集合**
  - 最基本的构成单元：0、1、2、3、…、9
  - 组合形式：10、123、1001、19461、…
    - 可以看做由基本单元不断拼接而形成的串

正整数的描述

正整数描述了一个集合

- 最基本的构成单元：0、1、2、3、…、9
- 组合形式：10、123、1001、19461、…
  - 可以看做由基本单元不断拼接而形成的串

字母表

可以从0-9中任选一个数字
| 表示选择运算符

digit → 0|1|2|…|9

digits → digit digit*

***是闭包运算，表示零次或多次出现**

由数字不断拼接形成（至少有一个数字）
两个元素顺序放置表示拼接操作

正整数的描述

- **正整数描述了一个集合**
  - 最基本的构成单元：0、1、2、3、…、9
  - 组合形式：10、123、1001、19461、…
    - 可以看做由基本单元不断拼接而形成的串

digit → 0|1|2|…|9

digits → digit digit*

正则表达式
(Regular Expression)

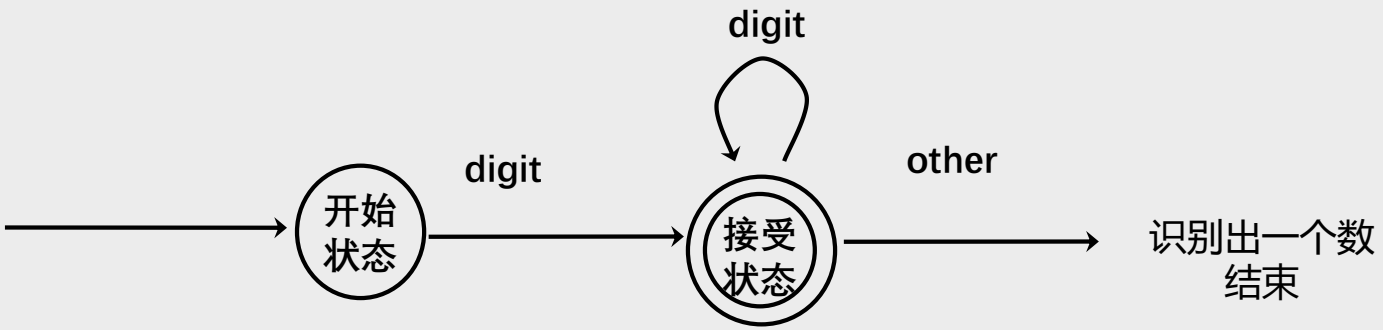
正整数的识别

- **正整数描述了一个集合**
  - 最基本的构成单元：0、1、2、3、…、9
  - 组合形式：10、123、1001、19461、…
    - 可以看做由基本单元不断拼接而形成的串

正则表达式

digit → 0|1|2|…|9

digits → digit digit*



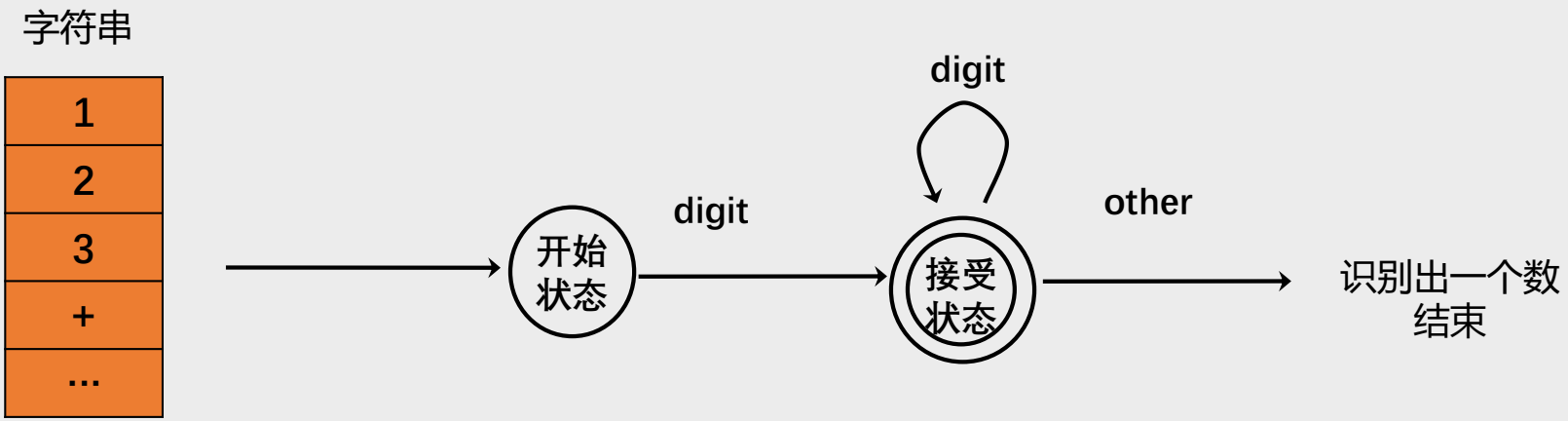
正整数的识别

- **正整数描述了一个集合**
  - 最基本的构成单元：0、1、2、3、…、9
  - 组合形式：10、123、1001、19461、…
    - 可以看做由基本单元不断拼接而形成的串

正则表达式

digit → 0|1|2|…|9

digits → digit digit*



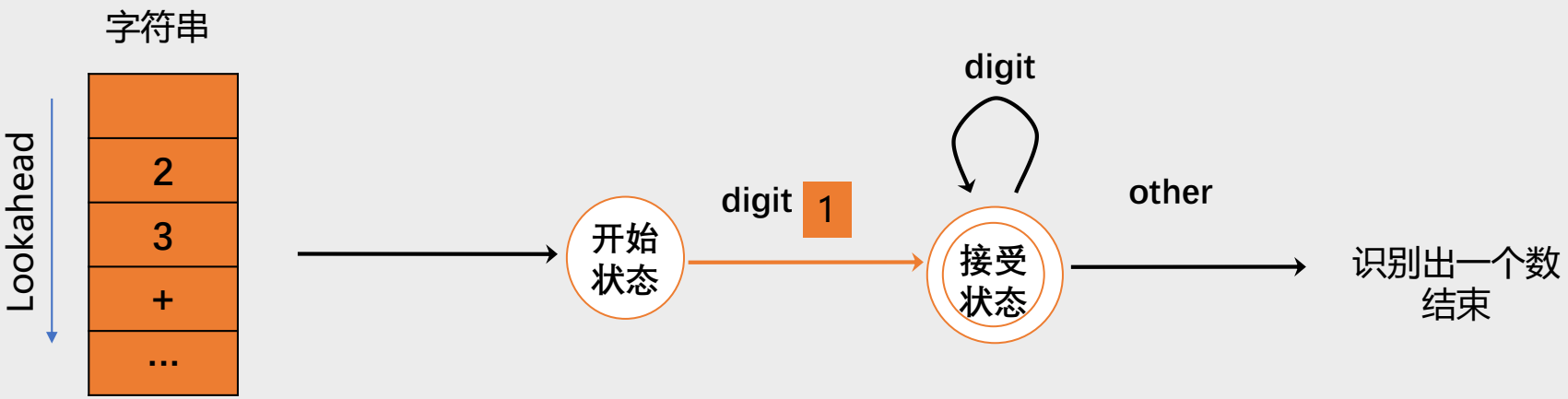
正整数的识别

- **正整数描述了一个集合**
  - 最基本的构成单元：0、1、2、3、…、9
  - 组合形式：10、123、1001、19461、…
    - 可以看做由基本单元不断拼接而形成的串

正则表达式

digit → 0|1|2|…|9

digits → digit digit*



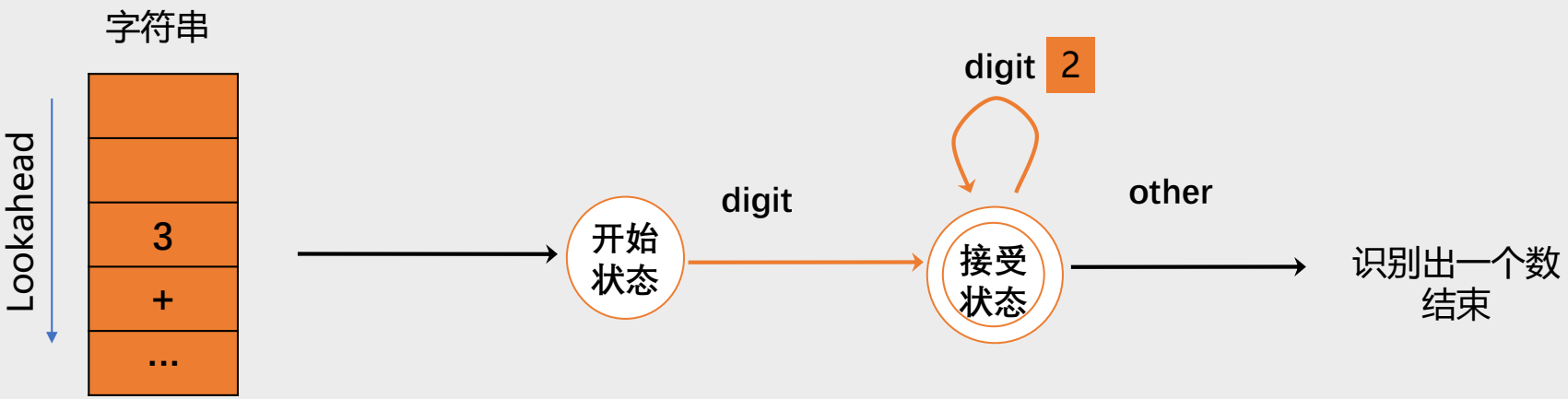
正整数的识别

- **正整数描述了一个集合**
  - 最基本的构成单元：0、1、2、3、…、9
  - 组合形式：10、123、1001、19461、…
    - 可以看做由基本单元不断拼接而形成的串

正则表达式

digit → 0|1|2|…|9

digits → digit digit*



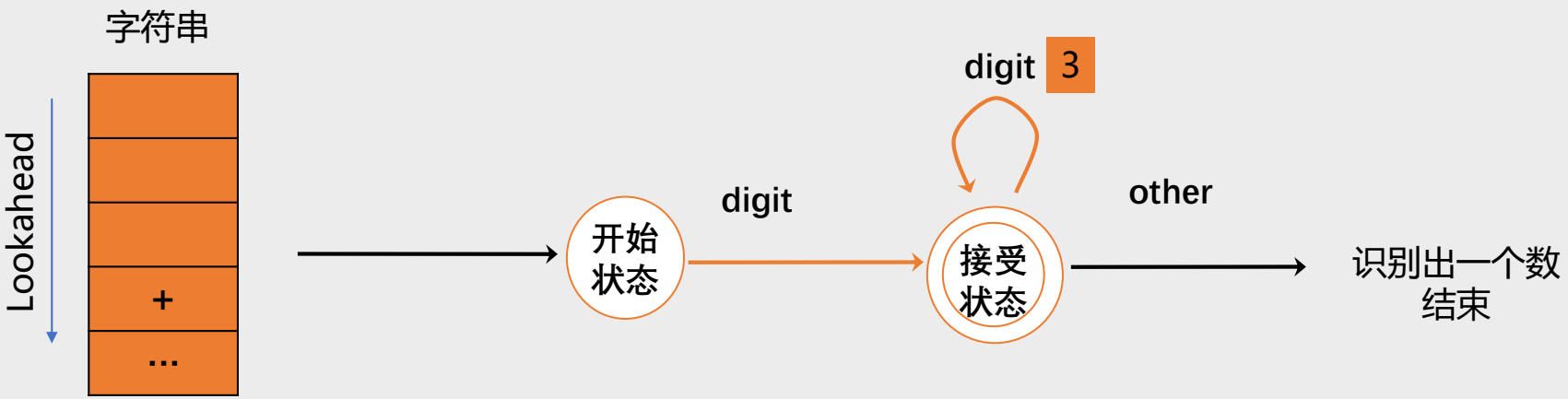
正整数的识别

- **正整数描述了一个集合**
  - 最基本的构成单元：0、1、2、3、…、9
  - 组合形式：10、123、1001、19461、…
    - 可以看做由基本单元不断拼接而形成的串

正则表达式

digit → 0|1|2|…|9

digits → digit digit*



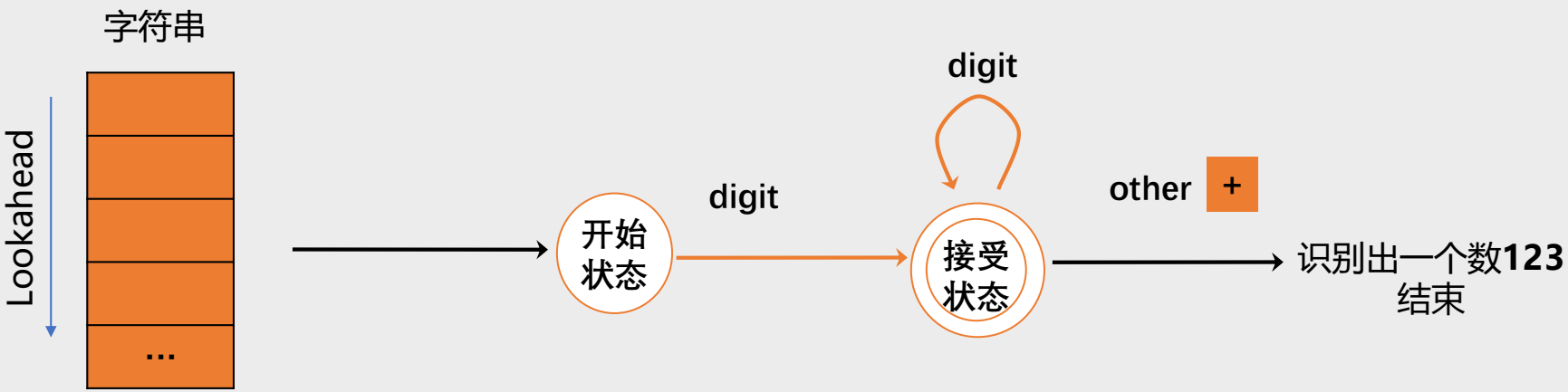
正整数的识别

- **正整数描述了一个集合**
  - 最基本的构成单元：0、1、2、3、…、9
  - 组合形式：10、123、1001、19461、…
    - 可以看做由基本单元不断拼接而形成的串

正则表达式

digit → 0|1|2|…|9

digits → digit digit*



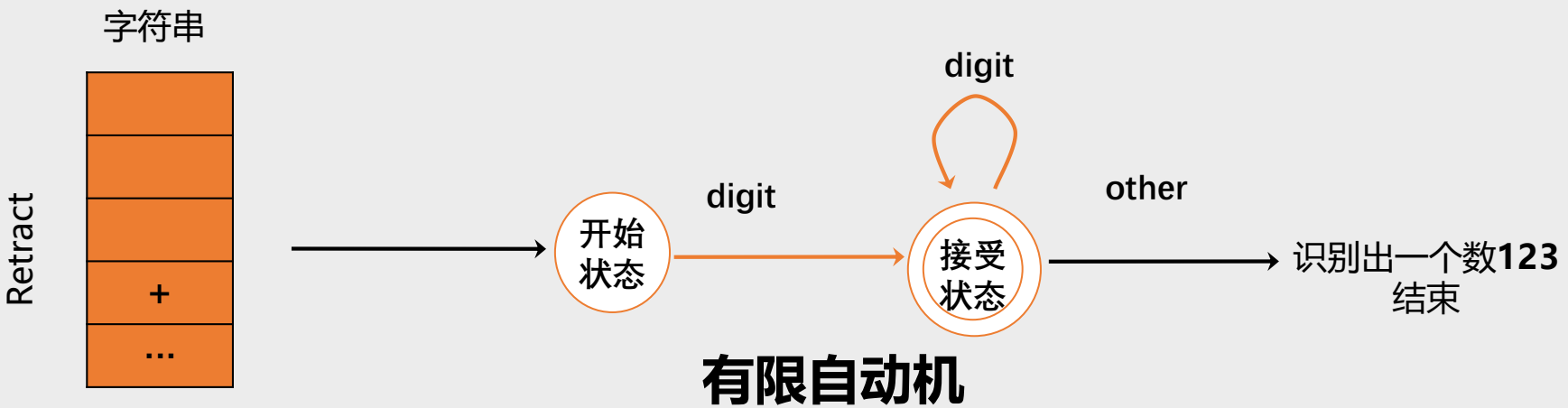
正整数的识别

- **正整数描述了一个集合**
  - 最基本的构成单元：0、1、2、3、…、9
  - 组合形式：10、123、1001、19461、…
    - 可以看做由基本单元不断拼接而形成的串

正则表达式

digit → 0|1|2|…|9

digits → digit digit*



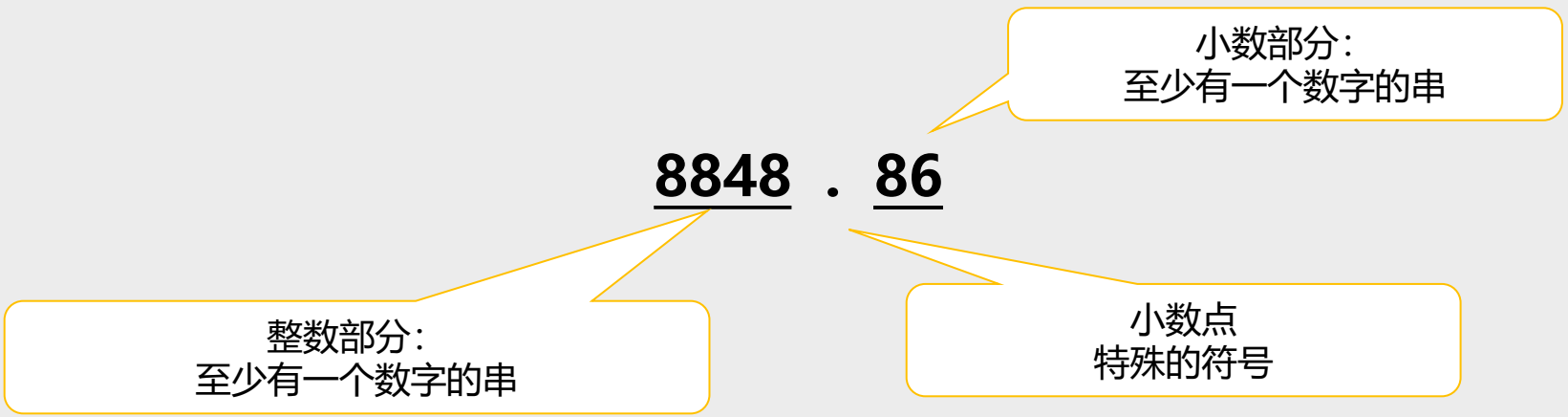
有限自动机
(Finite Automata)

带小数的数如何识别?

- 1.5, 10.28, 237.8, 8848.86（2020年测定的珠穆朗玛峰高度）

带小数的数如何识别?

- 1.5, 10.28, 237.8, 8848.86（2020年测定的珠穆朗玛峰高度）



带小数的数如何识别?

- 1.5, 10.28, 237.8, 8848.86（2020年测定的珠穆朗玛峰高度）

基本数字 digit → 0|1|2|…|9

整数部分 digits → digit digit*

小数部分 digits → digit digit*

带小数的数字串number→digit digit*.digit digit*

正则表达式
(Regular Expression)

带小数的数如何识别?

- 1.5, 10.28, 237.8, 8848.86（2020年测定的珠穆朗玛峰高度）

基本数字 digit → [0-9]

整数部分 digits → $digit^+$

小数部分 digits → $digit^+$

带小数的数字串 number → $digit^+$. $digit^+$

简写形式

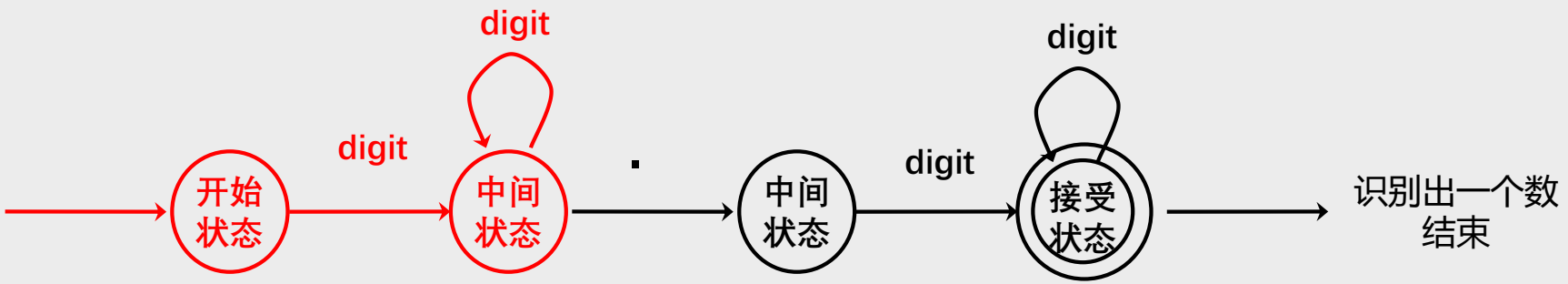
正则表达式
(Regular Expression)

带小数的数如何识别?

- 1.5, 10.28, 237.8, 8848.86

正则表达式

number → digit$^+$. digit$^+$

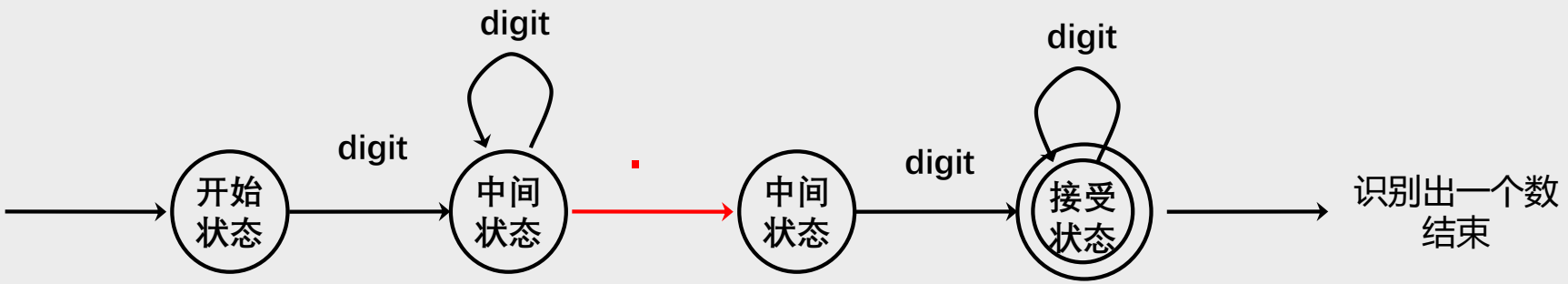


带小数的数如何识别?

- 1.5, 10.28, 237.8, 8848.86

正则表达式

$$number \rightarrow digit^{+} \,.\, digit^{+}$$

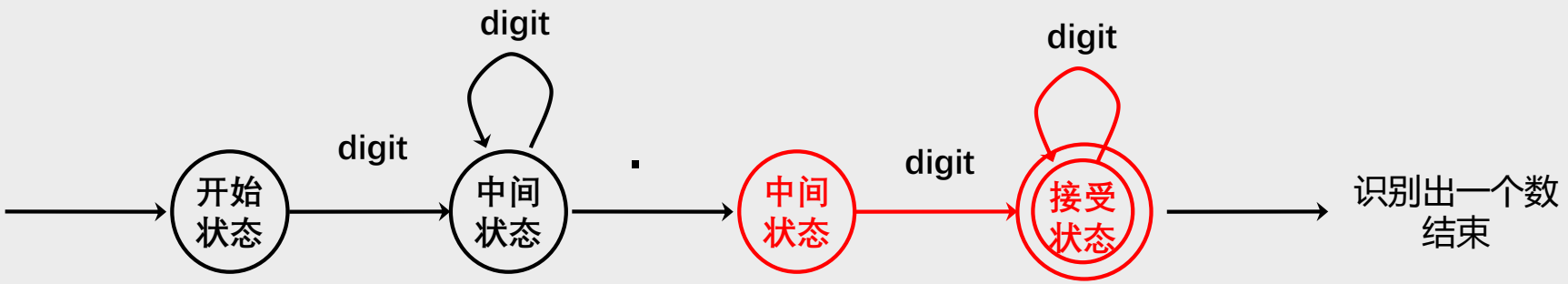


带小数的数如何识别?

- 1.5, 10.28, 237.8, 8848.86

正则表达式

number → digit$^+$. digit$^+$



串和语言

术语

- 字母表：符号的有限集合，例：$\Sigma = \{0, 1\}$
- 串：符号的有穷序列，例：0110，ε
- 语言：字母表上的一个串集

  $\{\varepsilon, 0, 00, 000, \ldots\}$，$\{\varepsilon\}$，$\varnothing$
- 句子：属于语言的串

注意区别：
ε，$\{\varepsilon\}$，$\varnothing$

串的运算

- 连接（积）：　　xy，$s\varepsilon = \varepsilon s = s$
- 指数（幂）：　　s^0为ε，s^i为$s^{i-1}s$（$i > 0$）

串和语言

语言的运算

- 并：　$L \cup M = \{s \mid s \in L$ 或 $s \in M\}$
- 连接：　$LM = \{st \mid s \in L$ 且 $t \in M\}$
- 幂：　L^0是$\{\varepsilon\}$，L^i是$L^{i-1}L$
- 闭包：　$L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$
- 正闭包：　$L^+ = L^1 \cup L^2 \cup \ldots$

优先级：
幂〉连接〉并

示例

L: { $A, B, \ldots, Z, a, b, \ldots, z$ }, D: { 0, 1, …, 9 }

$L \cup D$, LD, L^6, L^*, $L(L \cup D)^*$, D^+

正则表达式（Regular Expr）

- $\Sigma = \{a, b\}$

优先级：
闭包*〉连接〉选择 |

$a \mid b$	$\{a, b\}$
$(a \mid b)\,(a \mid b)$	$\{aa, ab, ba, bb\}$
$aa \mid ab \mid ba \mid bb$	$\{aa, ab, ba, bb\}$
a^*	由字母a构成的所有串集
$(a \mid b)^*$	由a和b构成的所有串集

- 复杂的例子

(00 | 11 | ((01 | 10) (00 | 11) * (01 | 10)))*

句子：0100110100001000001011001

正则定义的例子

- C语言的标识符是字母、数字和下划线组成的串

$$\text{letter\_} \rightarrow A \mid B \mid \cdots \mid Z \mid a \mid b \mid \cdots \mid z \mid \_$$

$$\text{digit} \rightarrow 0 \mid 1 \mid \cdots \mid 9$$

$$\text{id} \rightarrow \text{letter\_}(\text{letter\_} \mid \text{digit})^*$$

正则表达式（Regular Expr）

- 正则式用来表示简单的语言

正则式	定义的语言	备注
ε	{ε}	
a	{*a*}	$a \in \Sigma$
(*r*)	*L*(*r*)	*r*是正则式
(*r*) | (*s*)	*L*(*r*)∪*L*(*s*)	*r*和*s*是正则式
(r)(s)	*L*(*r*)*L*(*s*)	*r*和*s*是正则式
$(r)^*$	$(L(r))^*$	*r*是正则式

((*a*) (b)*)| (*c*)可以写成*ab**| *c*

优先级：
闭包*〉连接〉选择 |

正则定义

bottom-up方法

- 对于比较复杂的语言，为了构造简洁的正则式，可先构造简单的正则式，再将这些正则式组合起来，形成一个与该语言匹配的正则序列。

$$d_1 \rightarrow r_1$$

$$d_2 \rightarrow r_2$$

$$\cdots$$

$$d_n \rightarrow r_n$$

- 各个d_i的名字都不同，是新符号，not in $\sum$
- 每个r_i都是$\sum \cup \{d_1, d_2, \ldots, d_{i-1}\}$上的正则式

正则定义的例子

- 无符号数集合，例1946,11.28,63E8,1.99E–6

正则定义的例子

- 无符号数集合，例1946,11.28,63E8,1.99E–6

digit → 0 | 1 | ⋯ | 9

digits → digit digit*

optional_fraction → . digits | ε

optional_exponent → (E (+ | – | ε) digits) | ε

number→digits optional_fraction optional_exponent

正则定义的例子

- 无符号数集合，例1946,11.28,63E8,1.99E–6

digit → 0 | 1 | … | 9 [0-9]

digits → digit digit*

optional_fraction → . digits | ε

optional_exponent → (E (+ | – | ε) digits) | ε

number→digits optional_fraction optional_exponent

- 简化表示

number → $digit^+$ (.$digit^+$)? (E[+–]? $digit^+$)?

注意区分：
？ 和 *

正则定义的例子

while → while

do → do

relop → < | < = | = | < > | > | > =

letter_ → [A-Za-z_]

id → letter_ (letter_ | digit)*

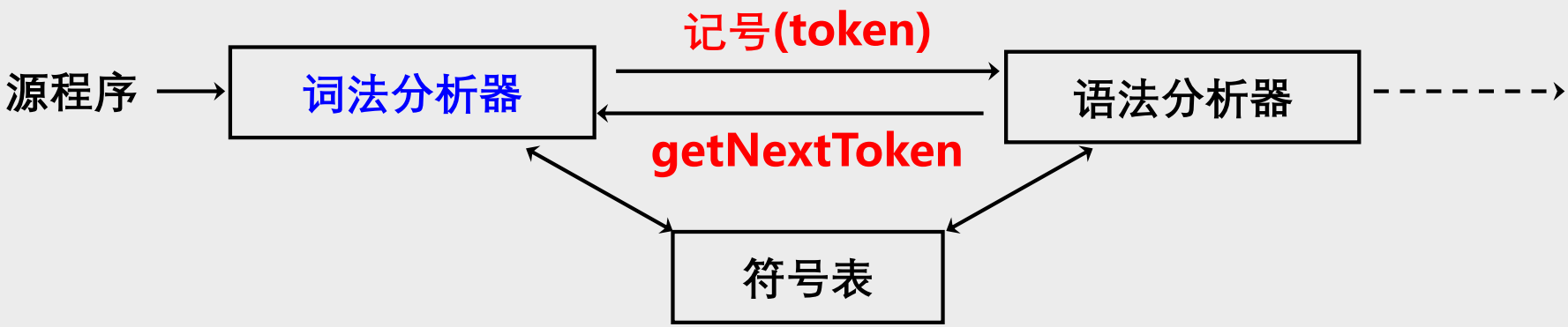
number → digit$^+$ (.digit$^+$)? (E[+–]? digit$^+$)?

delim → blank | tab | newline

ws → delim$^+$

问题：正则式是静态的定义，如何通过正则式动态识别输入串？

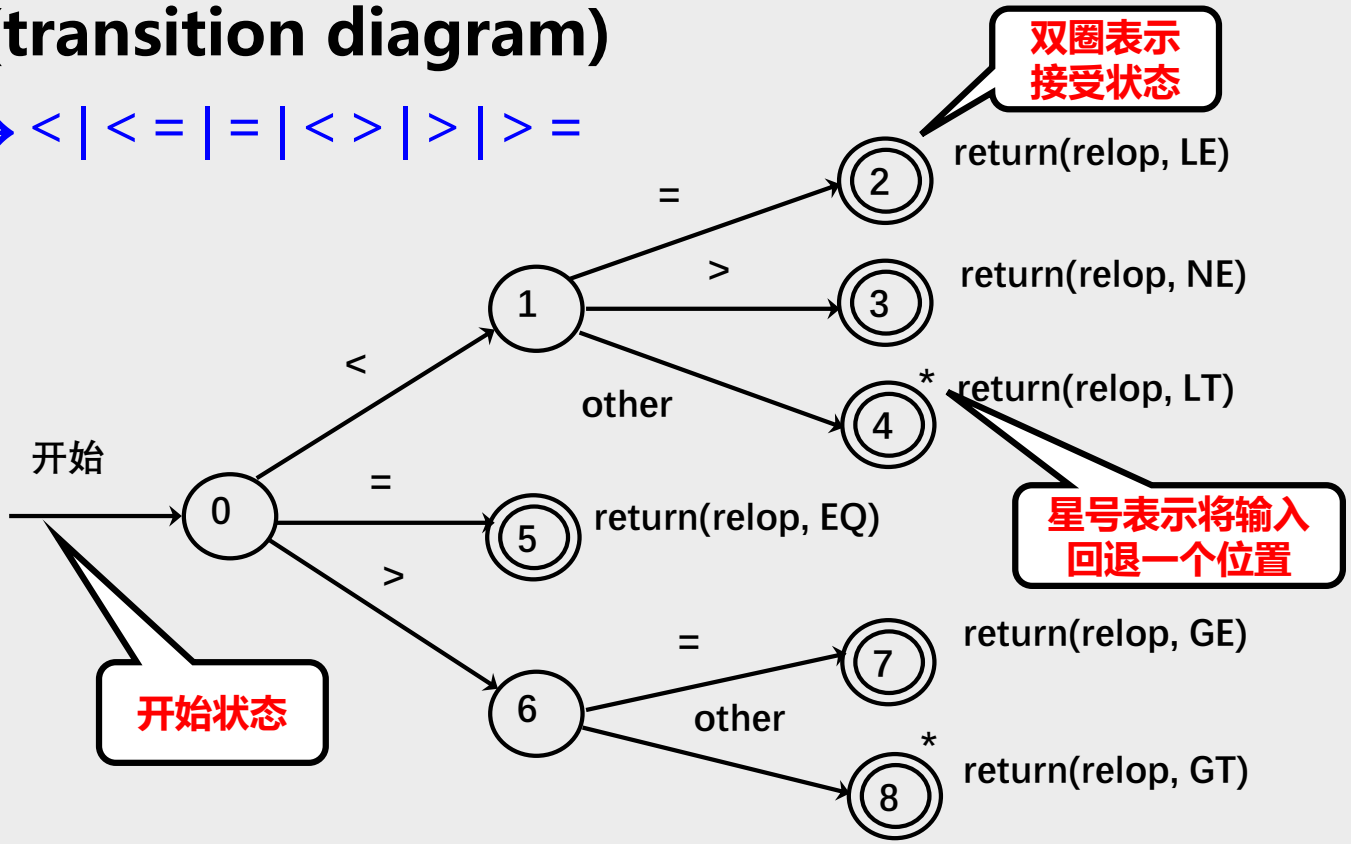
本节提纲



- ❑ 词法分析概述
- ❑ **词法分析器的自动生成**
  - ❖ 词法单元的描述：正则式
  - ❖ 词法单元的识别：转换图
  - ❖ 有限自动机：NFA、DFA
  - ❖ 正则表达式→NFA → DFA →化简的DFA

词法记号的识别：转换图

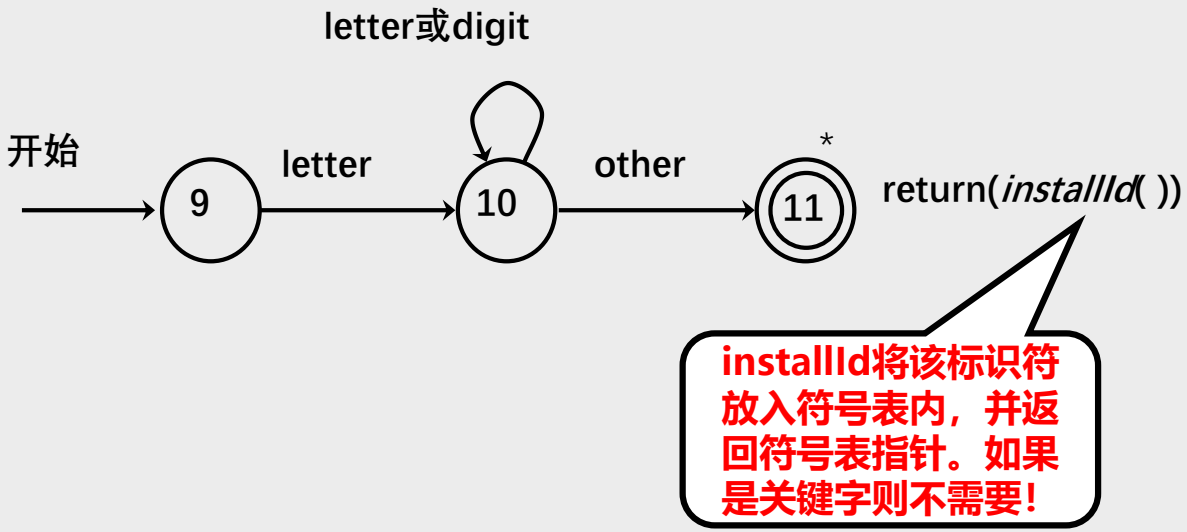
- 转换图(transition diagram)
  - relop → < | < = | = | < > | > | > =



状态转换图

标识符和关键字的转换图

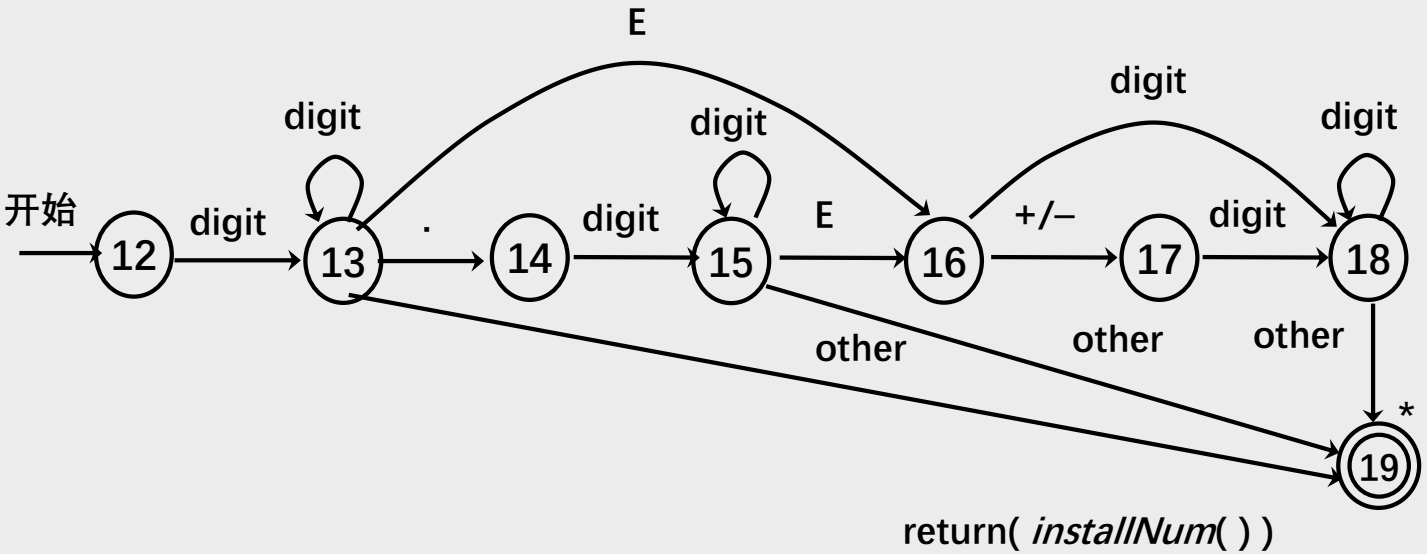
- id → letter (letter | digit)*



状态转换图

无符号数的转换图

number → digit$^+$ (.digit$^+$)? (E[+–]? digit$^+$)?

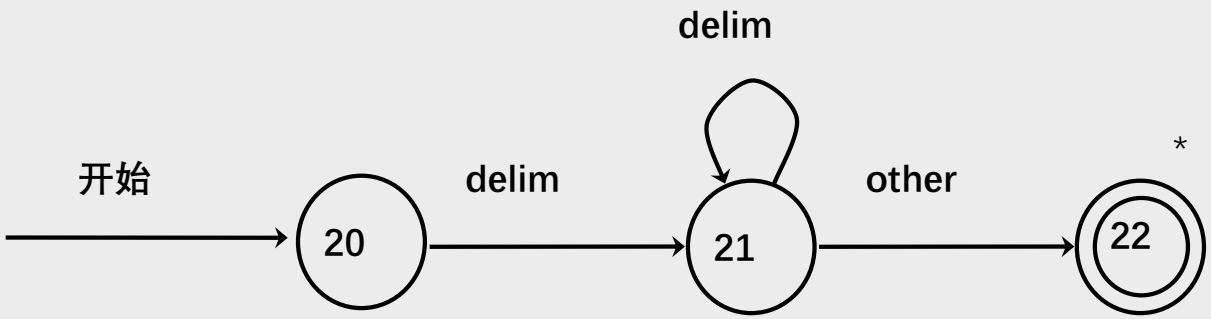


状态转换图

- **空白的转换图**

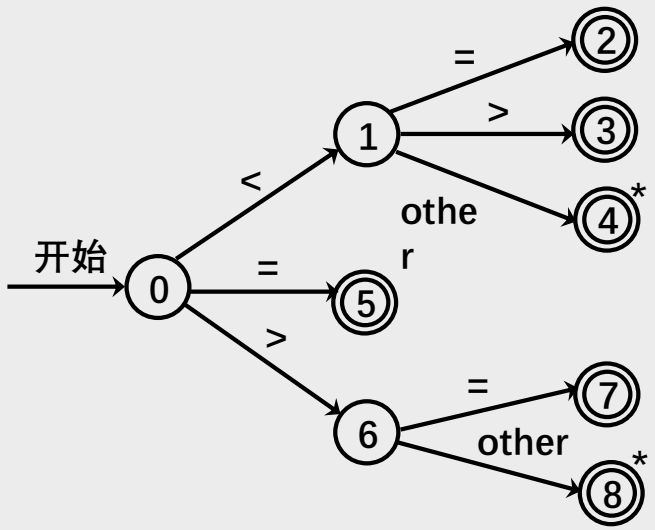
delim → blank | tab | newline

ws → delim+



基于转换图的词法分析

- 例：relop的转换图的概要实现



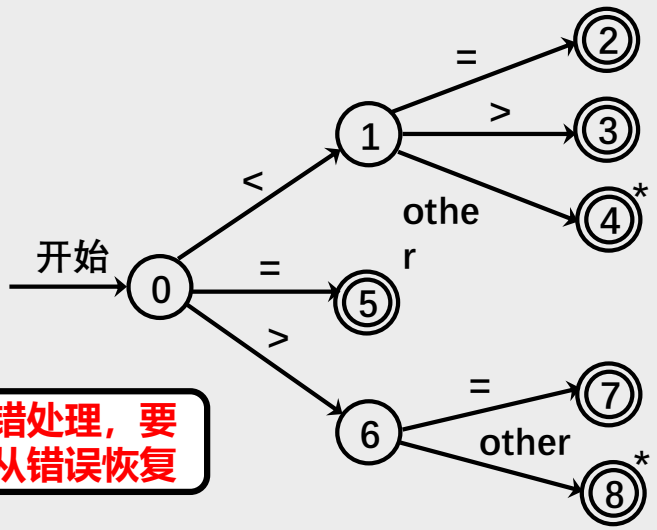
基于转换图的词法分析

例：relop的转换图的概要实现

```
TOKEN getRelop() {
   TOKEN retToken = new(RELOP);
   while (1) {
      switch (state) {
      case 0: c = nextChar();
              if (c == '<') state = 1;
              else if (c == '=') state = 5;
              else if (c == '>') state = 6;
              else fail();
              break;
      case 1: ...
      ...
      case 8: retract();
               retToken.attribute = GT;
      return(retToken);
      }
   }
}
```

出错处理，要能从错误恢复

回退



词法分析中的冲突及解决

R = Whitespace | Integer | Identifier| ‘+’

识别 “foo+3”

- “f” 匹配 R, 更精确地说是 Identifier
- 但是 “fo”也匹配 R, “foo” 也匹配, 但 “foo+”不匹配

如何处理输入？如果

- $x_1 \ldots x_i \in L(R)$ 并且 $x_1 \ldots x_k \in L(R)$

Maximal match 规则:

- 选择匹配 R 的最长前缀

词法分析中的冲突及解决

R = Whitespace| ‘new’ | Integer| Identifier识别 “new foo”

- “new” 匹配 R, 更精确地说是 ‘new’
- 但是 “new”也匹配 Identifier

如何处理输入？ 如果

- $x_1 \ldots x_i \in L(R_j)$ 并且 $x_1 \ldots x_i \in L(R_k)$

优先 match 规则:

- 选择先列出的模式 (j 如果 j < k)
- 必须将 ‘new’列在 Identifier的前面

词法错误

- **词法分析器对源程序采取非常局部的观点**
  - 例：难以发现下面的错误

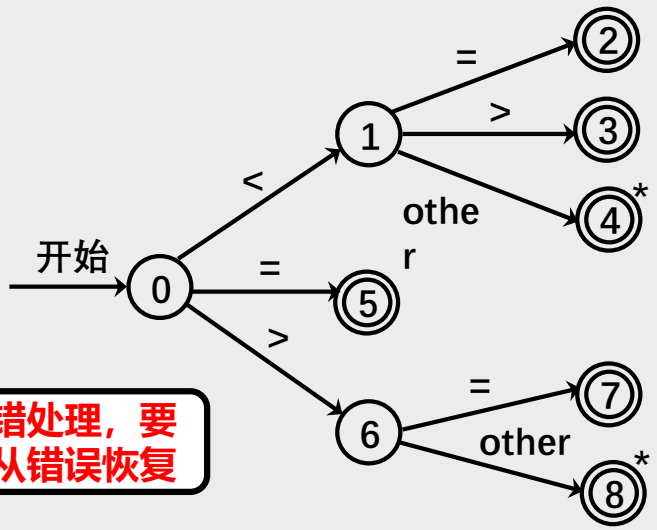
    fi (a == f (x)) …
- **在实数是“数字串.数字串”格式下**
  - 可以发现 **123.x** 中的错误
- **紧急方式的错误恢复**
  - 删掉当前若干个字符，直至能读出正确的记号
  - 会给语法分析器带来混乱
- **错误修补**
  - 进行增、删、替换和交换字符的尝试
  - 变换代价太高，不值得

基于转换图的词法分析

- 例：relop的转换图的概要实现

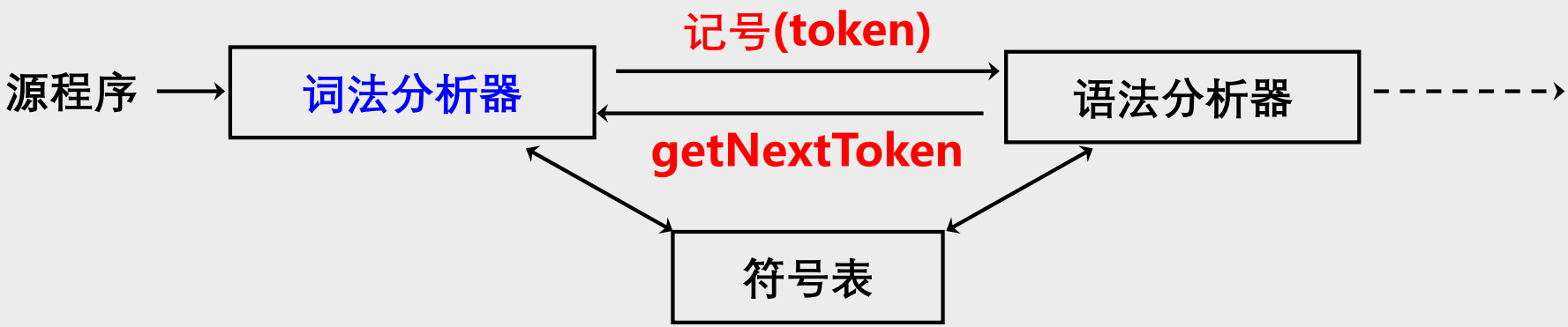
```
TOKEN getRelop() {
    TOKEN retToken = new(RELOP);
    while (1) {
        switch (state) {
        case 0: c = nextChar();
                if (c == '<') state = 1;
                else if (c == '=') state = 5;
                else if (c == '>') state = 6;
                else fail();
                break;
        case 1: ...
        ...
        }
    }
}
```

出错处理，要能从错误恢复



问题：怎么为每一个正则定义自动找到一个状态转换图?

本节提纲



- 词法分析概述
- **词法分析器的自动生成**
  - 词法单元的描述：正则式
  - 词法单元的识别：转换图
  - 有限自动机：NFA、DFA
  - 正则表达式→NFA → DFA →化简的DFA

有限自动机的定义

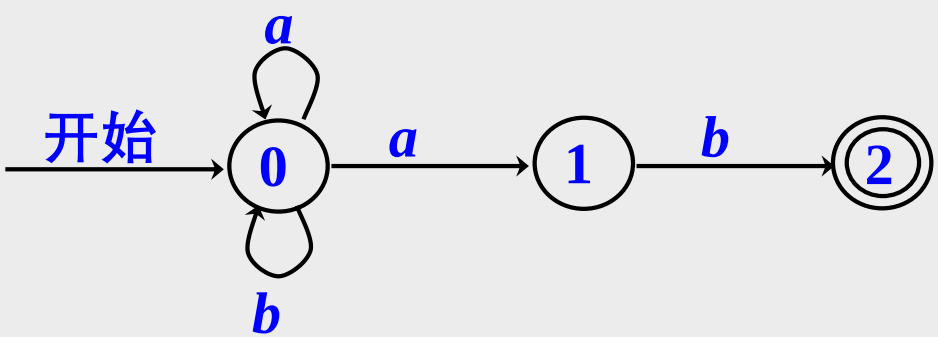
- （不确定的）有限自动机NFA是一个数学模型，它包括：
  - 有限的状态集合S
  - 输入符号集合$\sum$
  - 转换函数$move: S \times (\sum \cup \{\varepsilon\}) \rightarrow P(S)$
  - 状态s_0是唯一的开始状态
  - $F \subseteq S$是接受状态集合

幂集

有限自动机的定义

- （不确定的）有限自动机NFA是一个数学模型，它包括：
  - 有限的状态集合S
  - 输入符号集合Σ
  - 转换函数$move: S \times (\Sigma \cup \{\varepsilon\}) \rightarrow P(S)$
  - 状态s_0是唯一的开始状态
  - $F \subseteq S$是接受状态集合

识别语言
$(a|b)^*ab$
的NFA

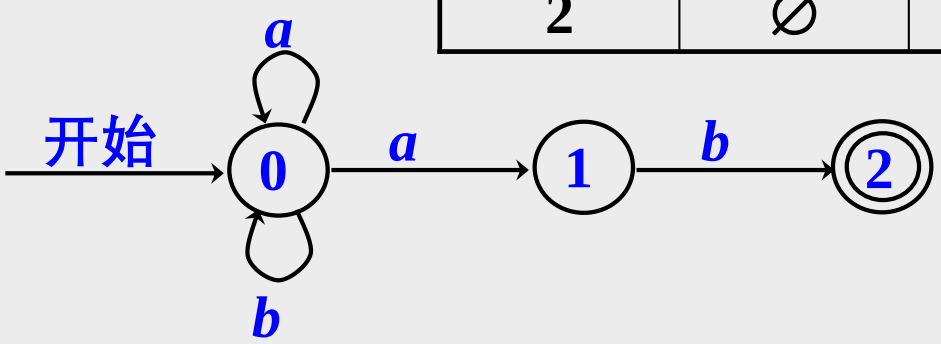


有限自动机的实现

- 构造状态之间的转换表，在读入字符串的过程中，不停查表，直至到达接受状态
- 或者，报告非法输入

	输 入 符 号	
	a	*b*
0	{0, 1}	{0}
1	∅	{2}
2	∅	∅

识别语言
$(a|b)^*ab$
的NFA

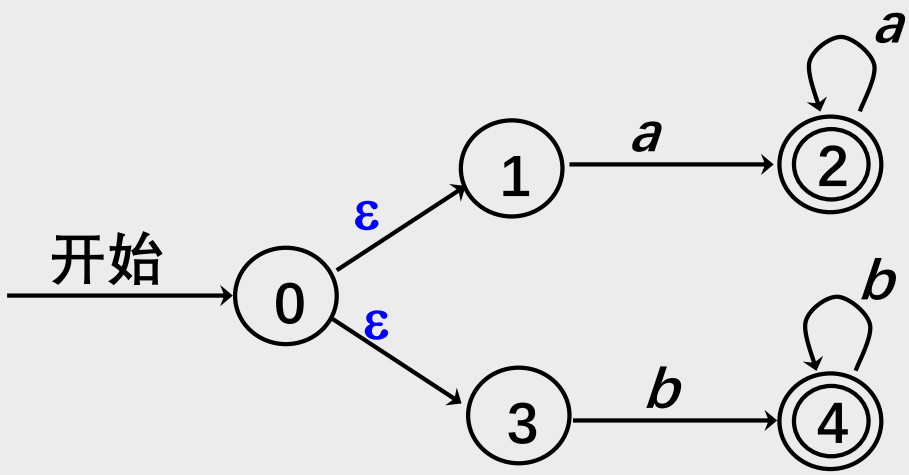


有限自动机

- 例 识别aa*|bb*的NFA

有限自动机

- 例 识别aa*|bb*的NFA



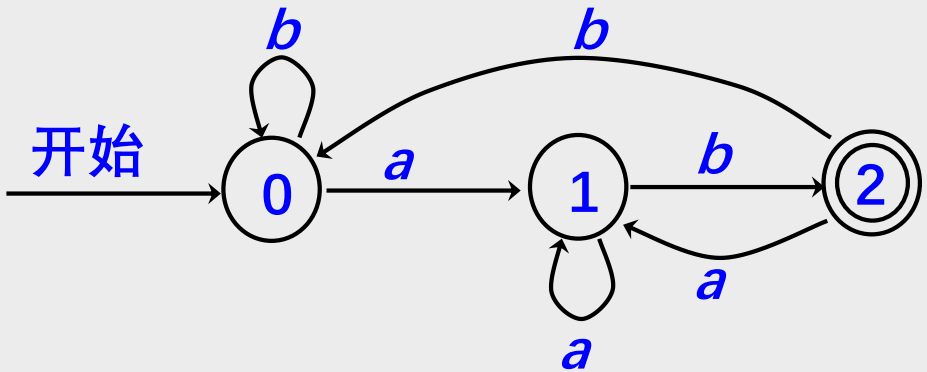
利用NFA识别token的问题

- 转换函数move : $S \times (\Sigma \cup \{\varepsilon\}) \rightarrow P(S)$
- 对于一个token，
  - 有可能要尝试很多不同的路径，
  - 大部分路径都是白费功夫
  - 尝试+回退的方式 => 效率很低
  - 考虑很多project，百万行代码+
- 思考：**有没有一种确定的形式化描述，对于输入的一个符号，只有唯一的跳转?**

有限自动机

- 确定的有限自动机（简称DFA)也是一个数学模型，包括:
  - 有限的状态集合S
  - 输入符号集合$\sum$
  - 转换函数$move: S \times \sum \to S$，且可以是部分函数
  - 状态s_0是唯一的开始状态
  - $F \subseteq S$是接受状态集合

识别语言
$(a|b)^*ab$
的DFA

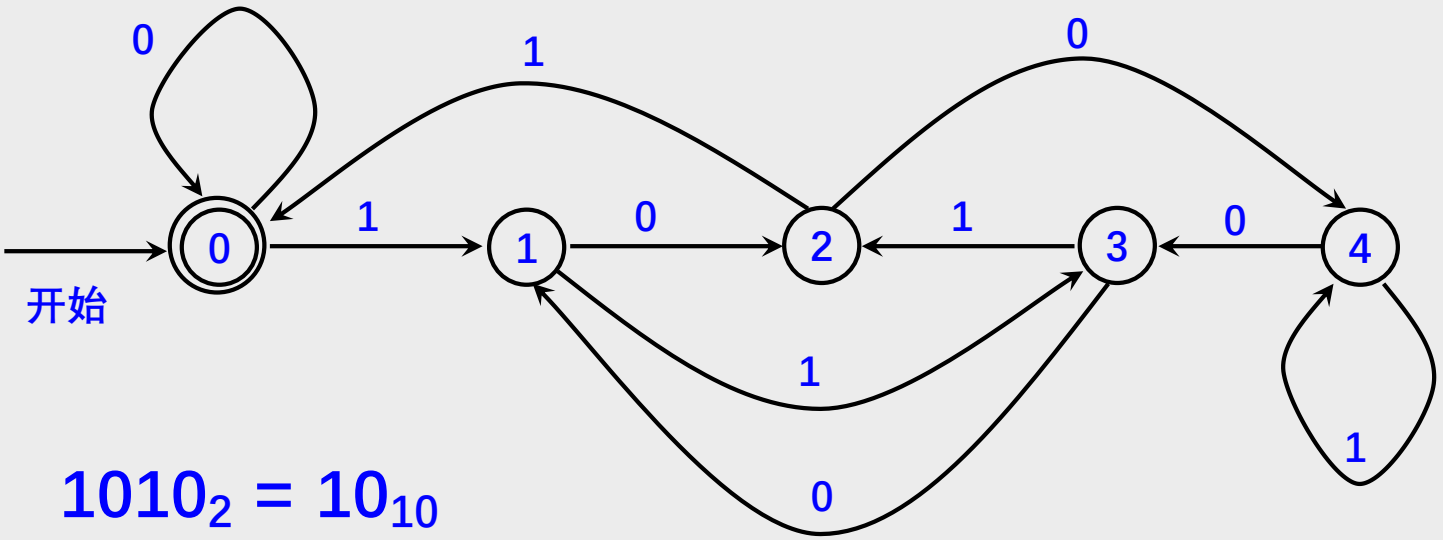


有限自动机

- 例 DFA，识别{0,1}上能被5整除的二进制数

有限自动机

- 例 DFA，识别{0,1}上能被5整除的二进制数

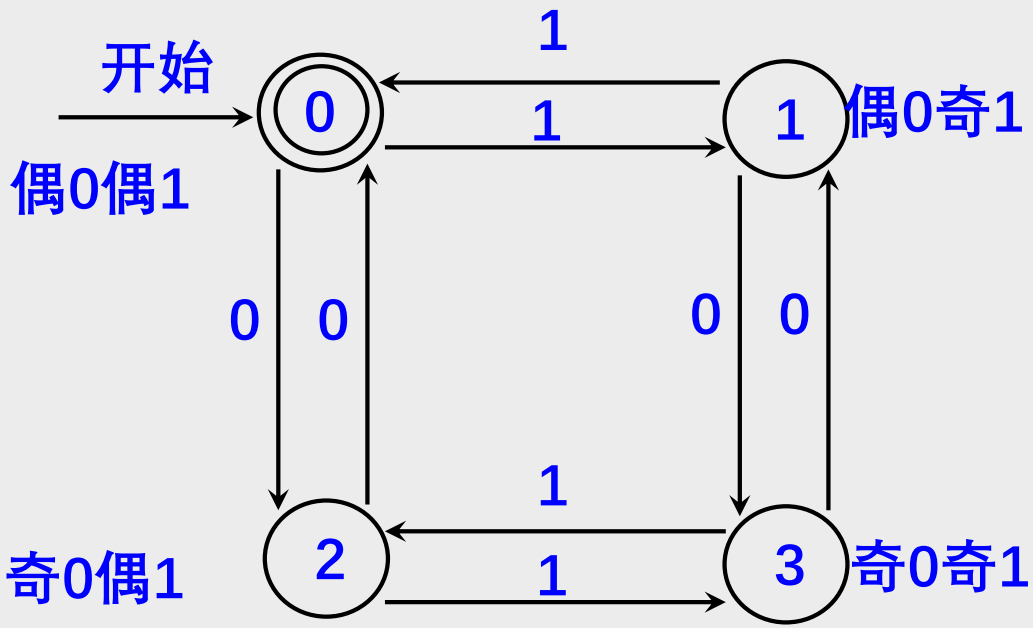


$1010_2 = 10_{10}$

$111_2 = 7_{10}$

有限自动机

- 例 DFA,接受 0和1的个数都是偶数的字符串

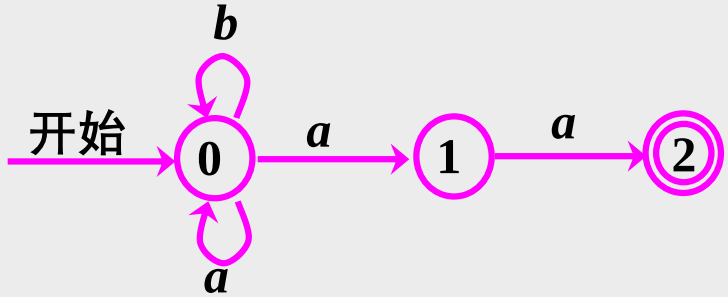


NFA vs. DFA

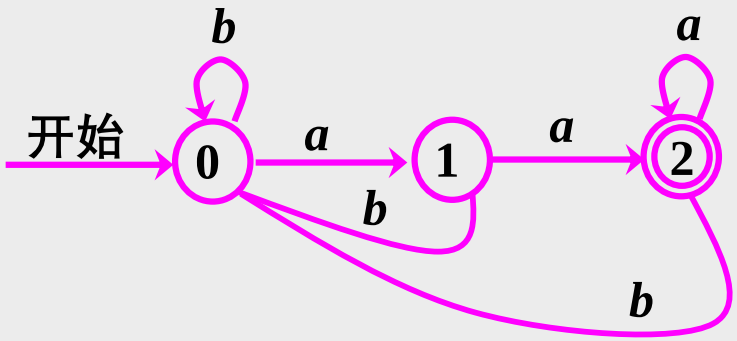
- NFAs and DFAs recognize the same set of languages (regular languages)
- Major differences:
  - Move function
    - $S \times (\Sigma \cup \{\varepsilon\}) \rightarrow P(S)$ NFA
    - $S \times \Sigma \rightarrow S$ DFA
  - DFA does not accept ε as input
- DFAs are faster to execute
  - There are no choices to consider

NFA vs. DFA

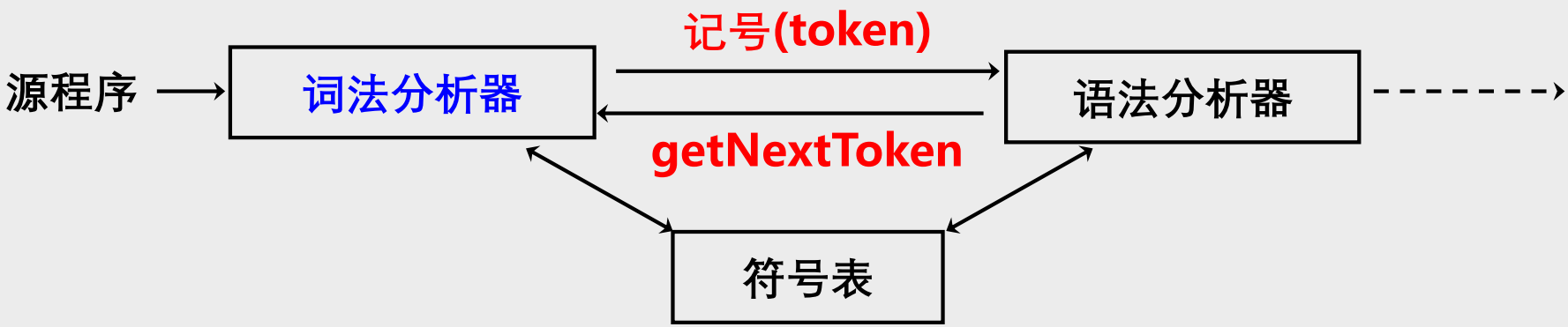
- For a given language NFA can be simpler than DFA



- DFA can be exponentially larger than NFA



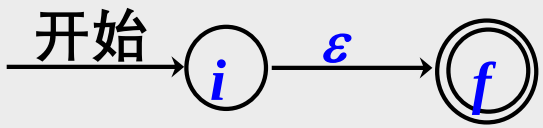
本节提纲



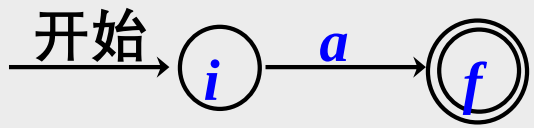
- 词法分析概述
- **词法分析器的自动生成**
  - 词法单元的描述：正则式
  - 词法单元的识别：转换图
  - 有限自动机：NFA、DFA
  - 正则表达式→NFA → DFA →化简的DFA

语法制导的构造算法

- 首先构造识别ε和字母表中一个符号a的NFA
  - 重要特点：仅一个接受状态，它没有向外的转换



识别正则表达式ε的NFA

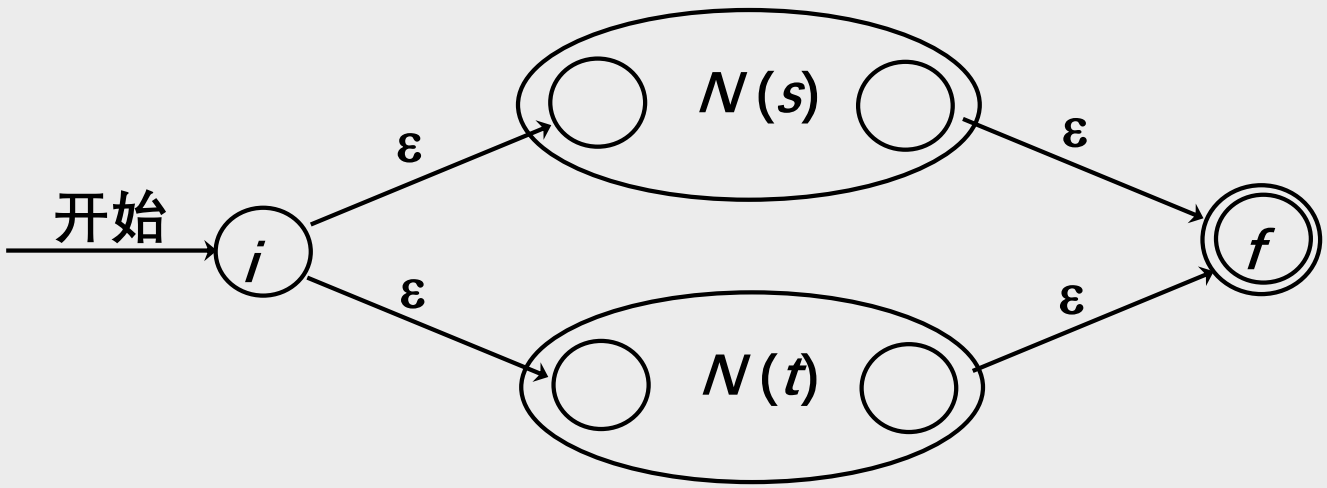


识别正则表达式a的NFA

- 对于加括号的正则表达式(s)，其NFA可用s的NFA（用N(s)表示）代替

语法制导的构造算法

- 构造识别主算符为选择的正则表达式的NFA
  - 重要特点：仅一个接受状态，它没有向外的转换

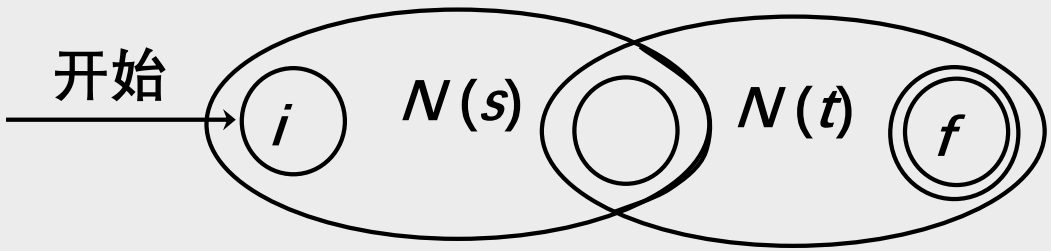


识别正则表达式$s \mid t$的NFA

语法制导的构造算法

构造识别主算符为连接的正则表达式的NFA

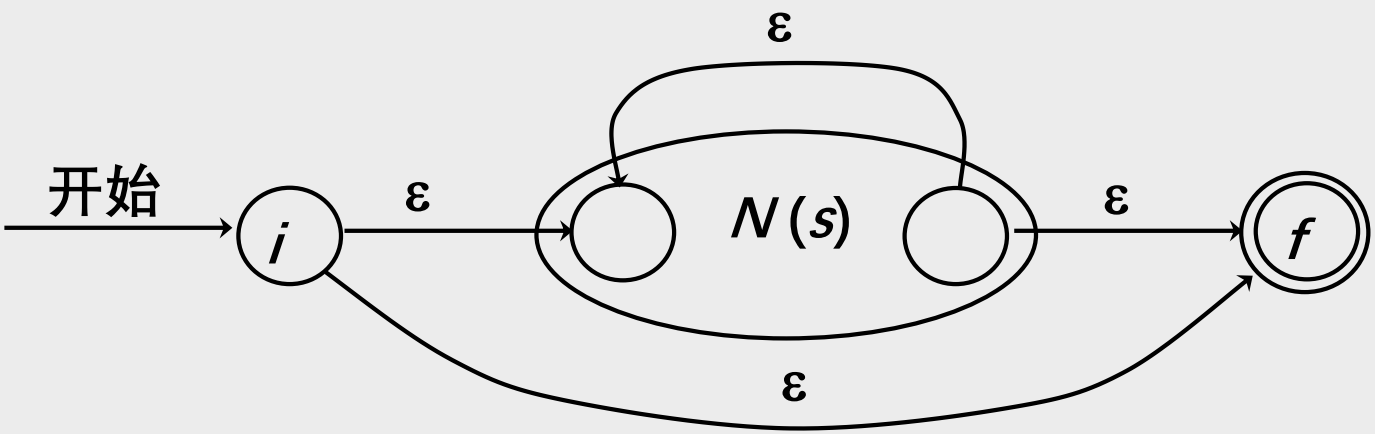
- 重要特点：仅一个接受状态，它没有向外的转换



识别正则表达式*st* 的NFA

语法制导的构造算法

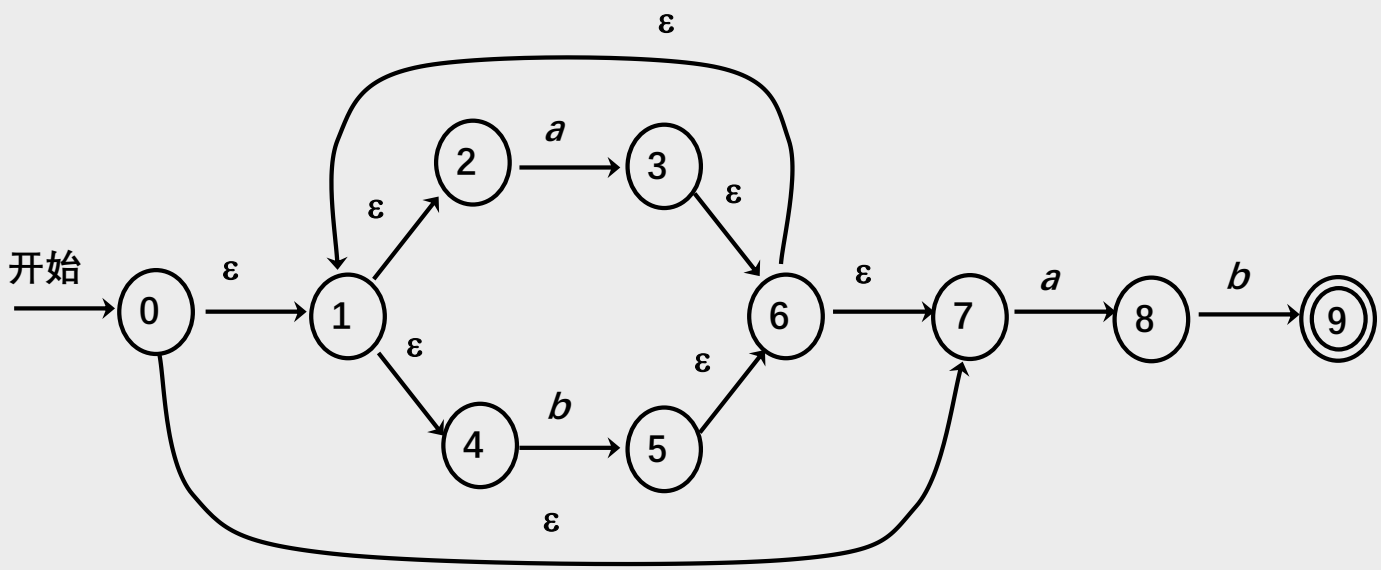
- 构造识别主算符为闭包的正则表达式的NFA
  - 重要特点：仅一个接受状态，它没有向外的转换



识别正则表达式s^* 的NFA

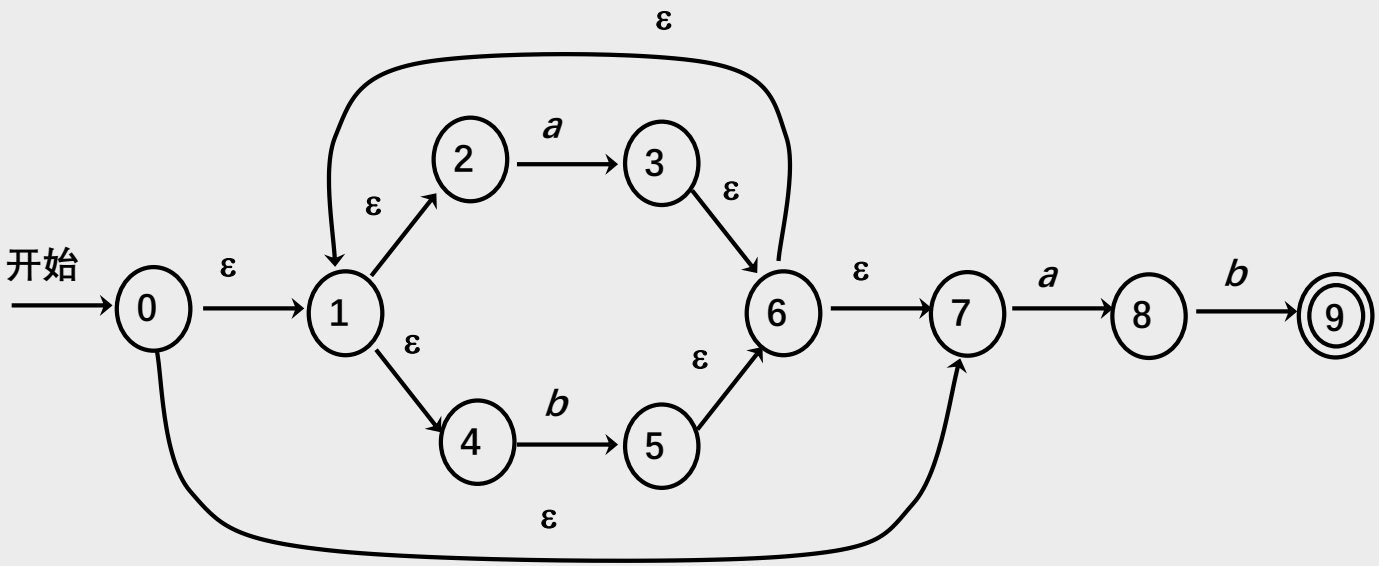
语法制导的构造算法

- 由本方法产生的NFA具有下列性质：
  - N(r)的状态数最多是r中符号和算符总数的两倍
  - N(r)只有一个接受状态，接受状态没有向外的转换



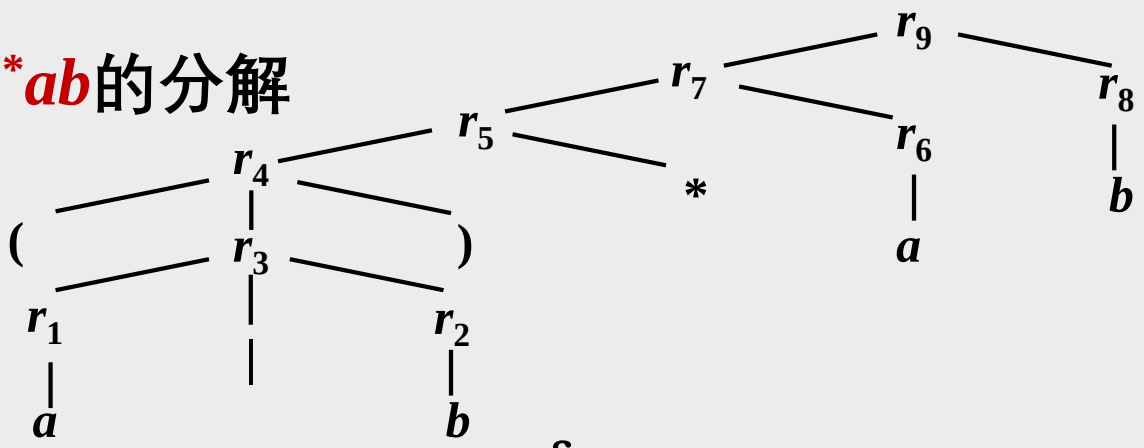
语法制导的构造算法

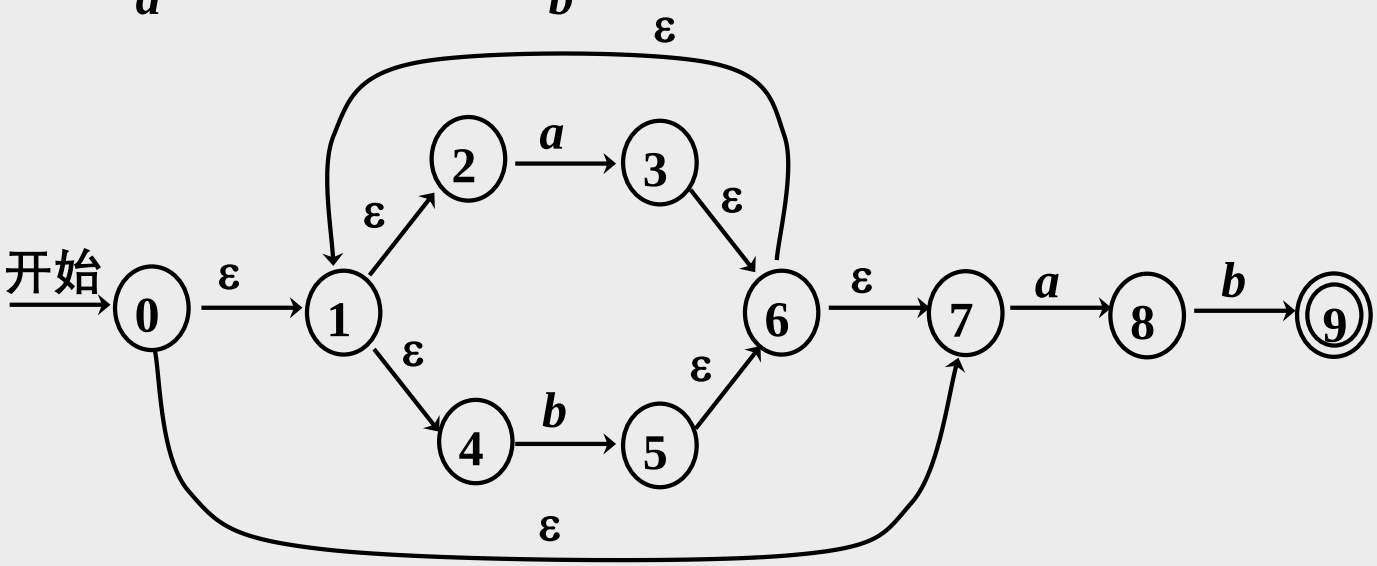
- 由本方法产生的NFA具有下列性质：
  - N(r)的每个状态有(1)一个其标号为∑中符号的指向其它状态的转换，或者(2)最多两个指向其它状态的ε转换



NFA构造过程举例

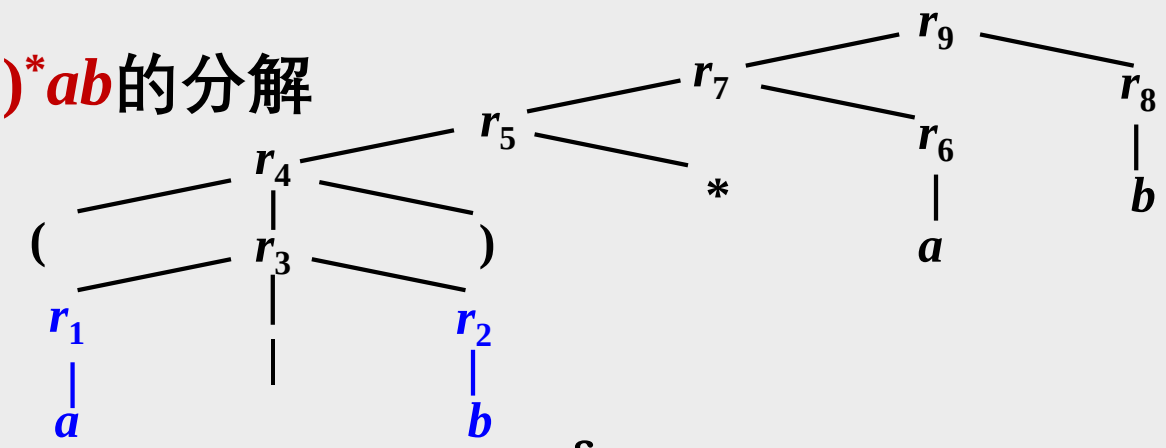
$(a|b)^*ab$的分解

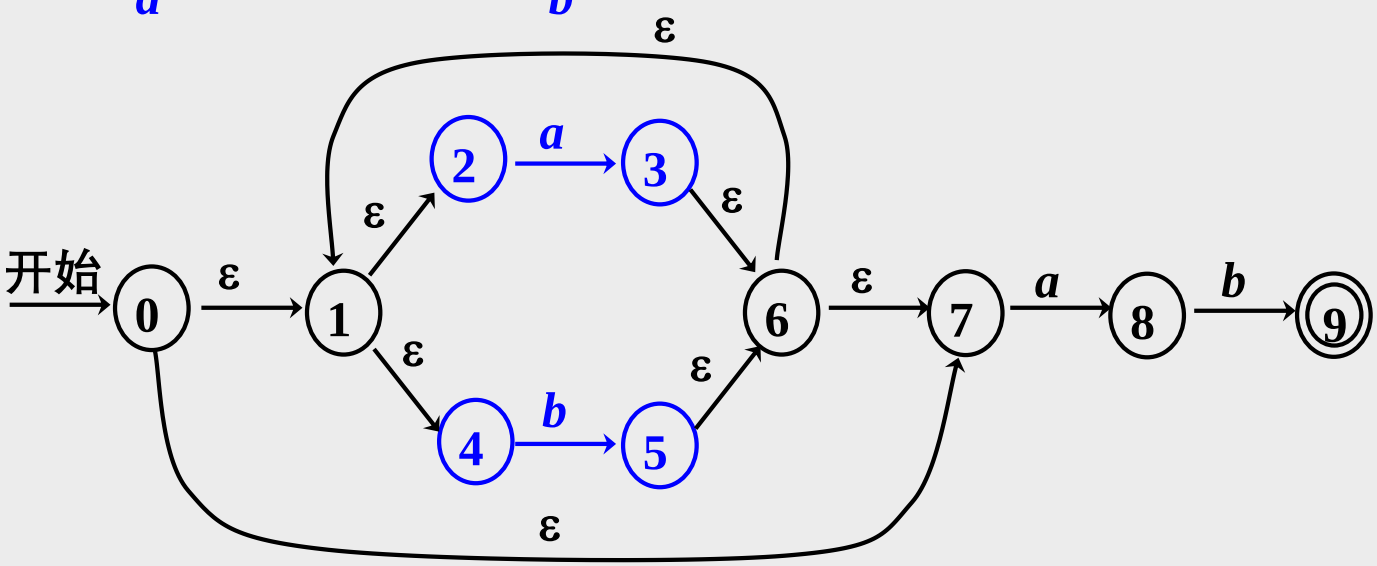




NFA构造过程举例

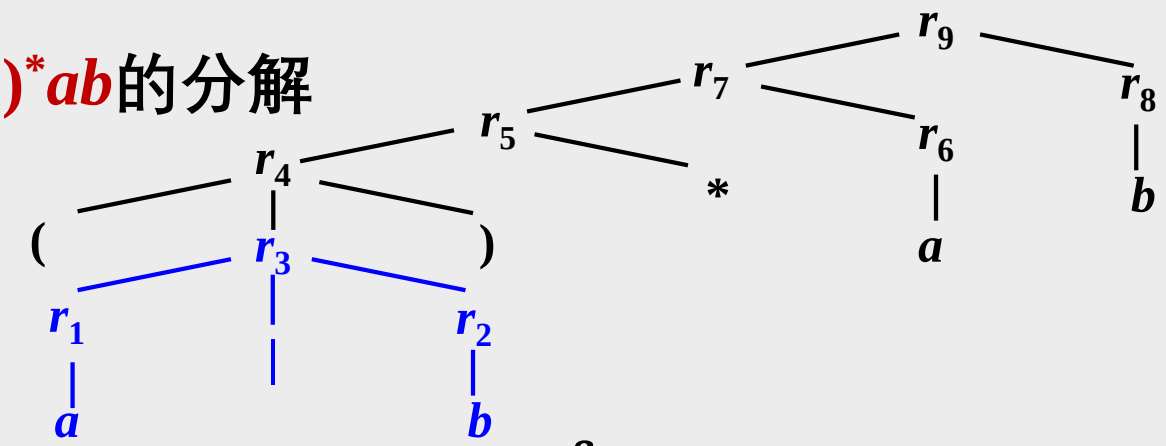
$(a|b)^*ab$的分解

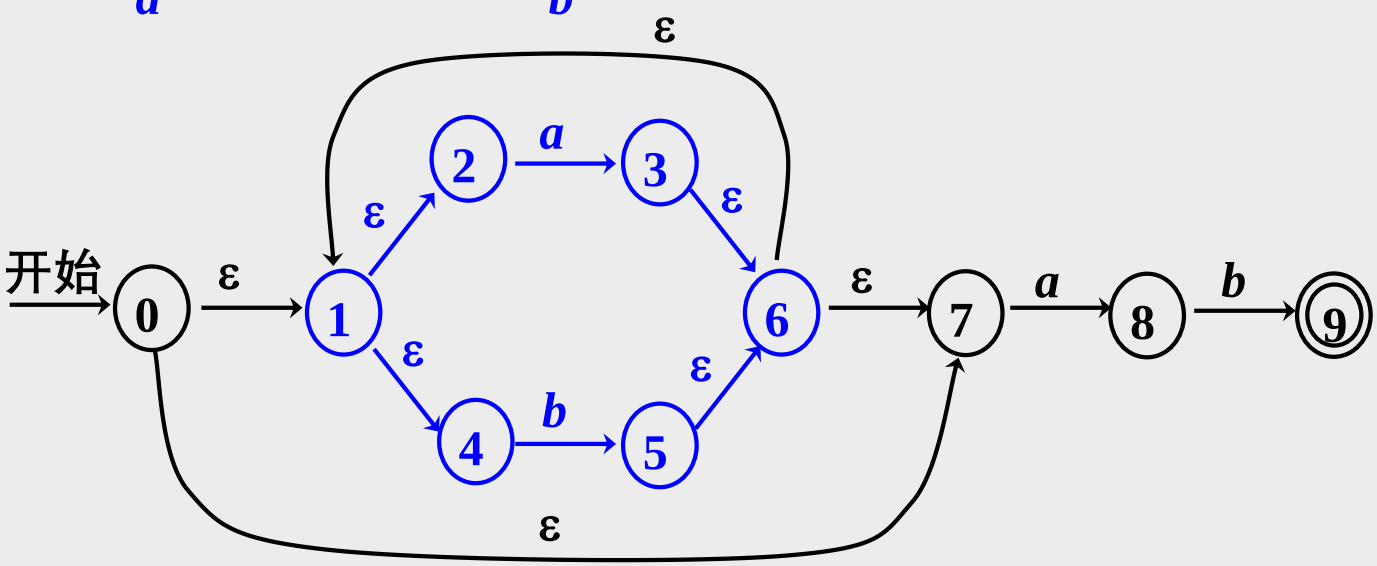




NFA构造过程举例

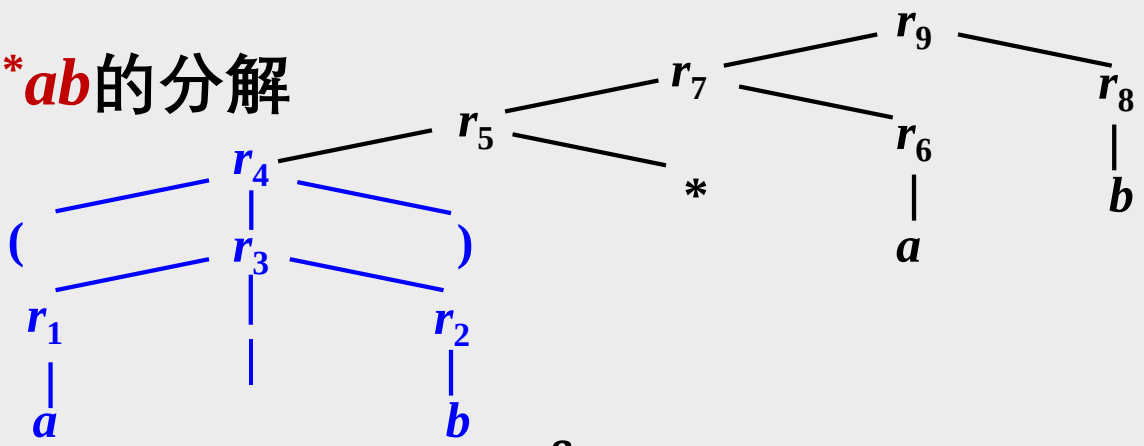
$(a|b)^*ab$的分解

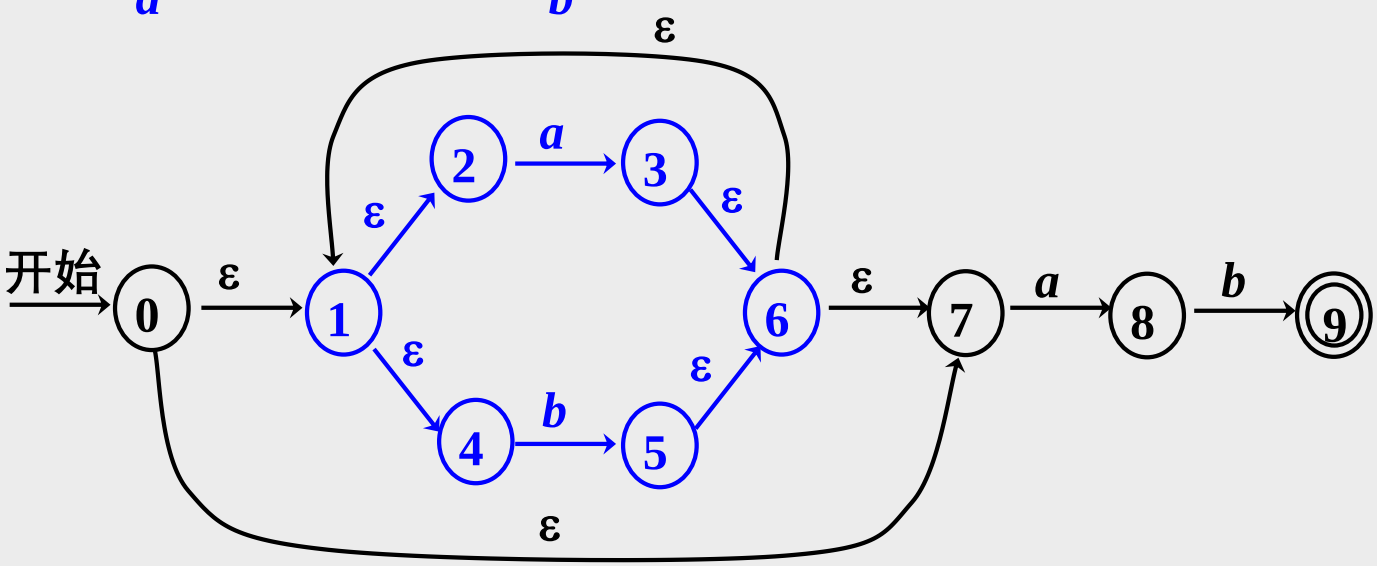




NFA构造过程举例

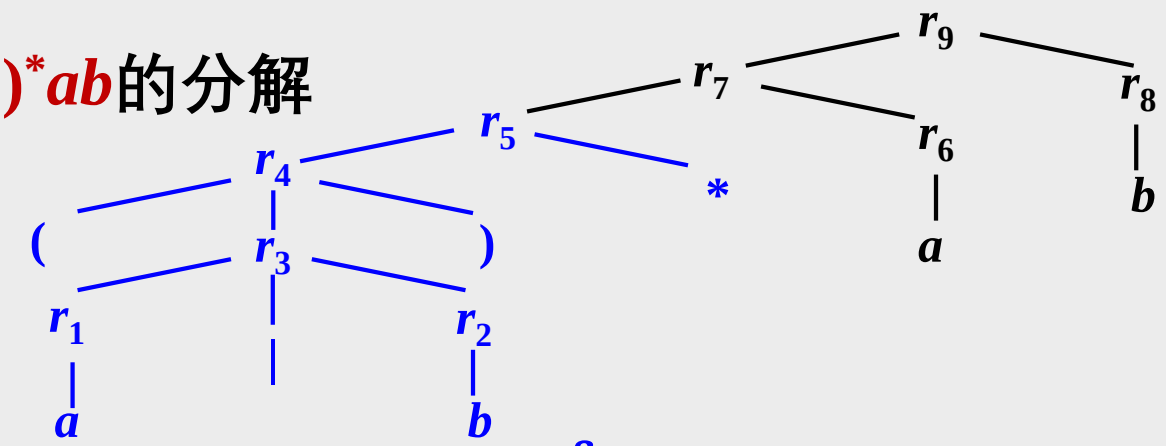
$(a|b)^*ab$的分解

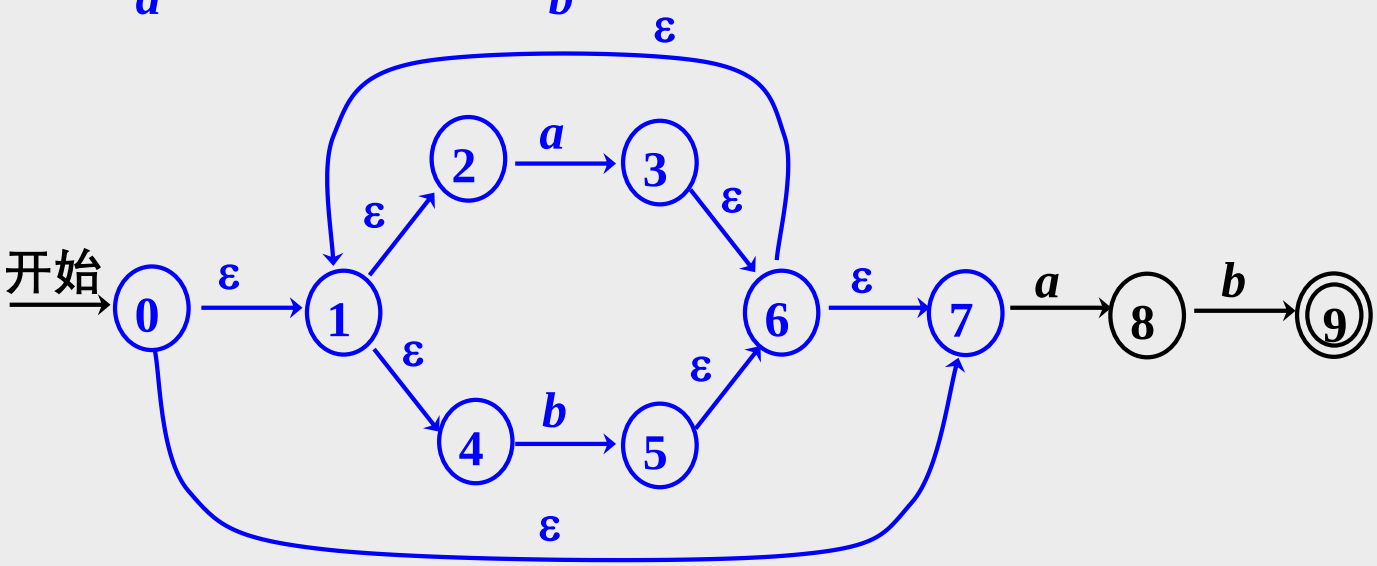




NFA构造过程举例

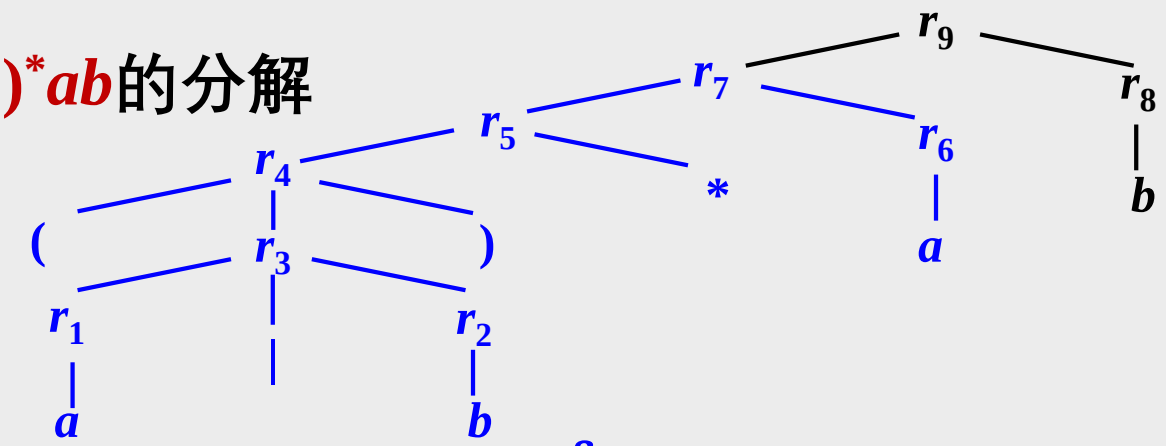
$(a|b)^*ab$的分解

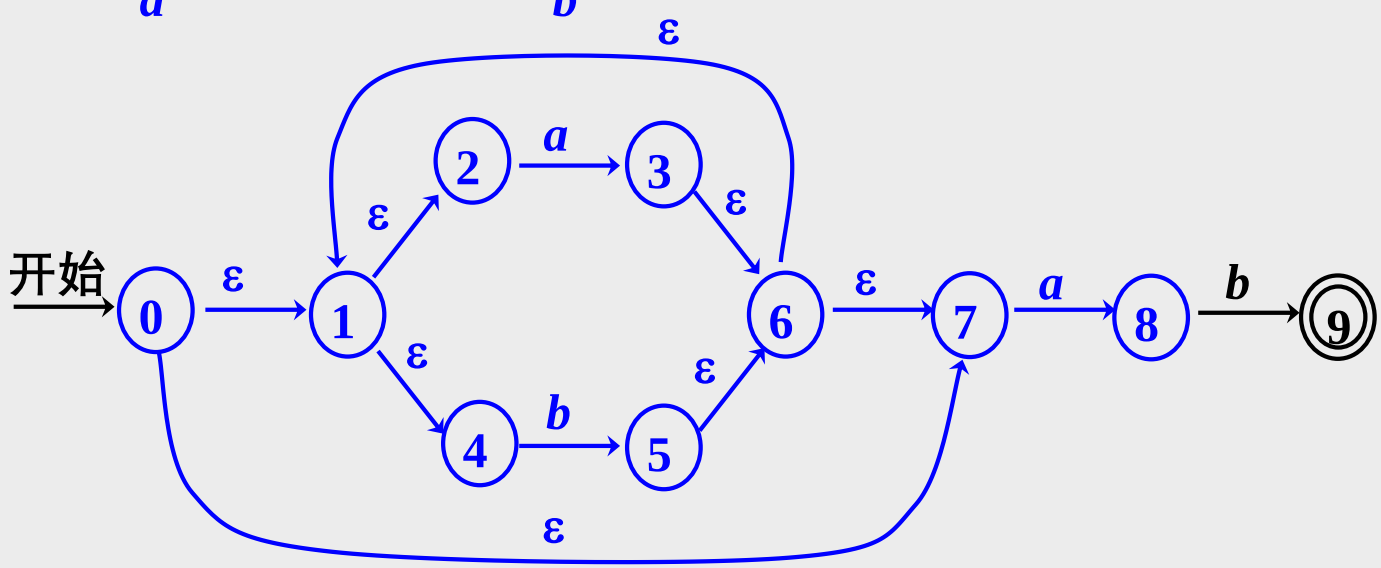




NFA构造过程举例

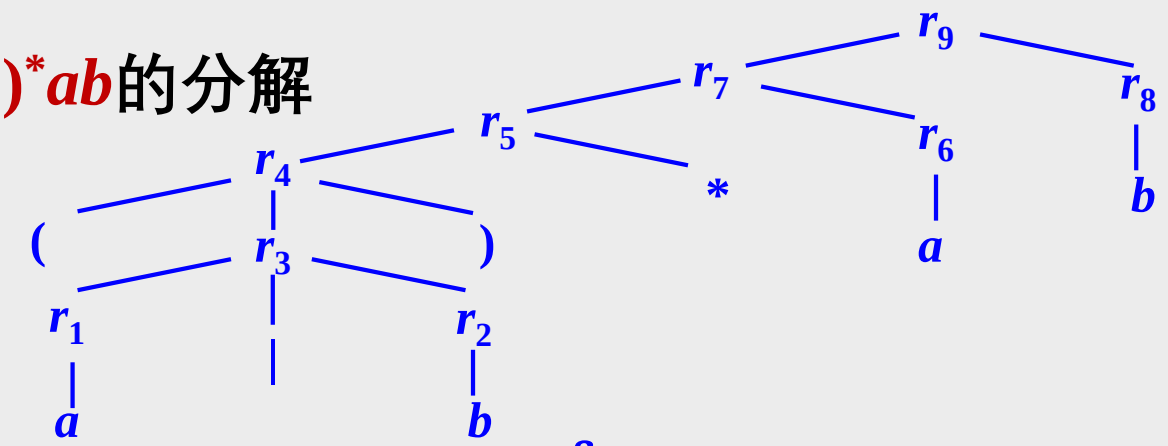
$(a|b)^*ab$的分解

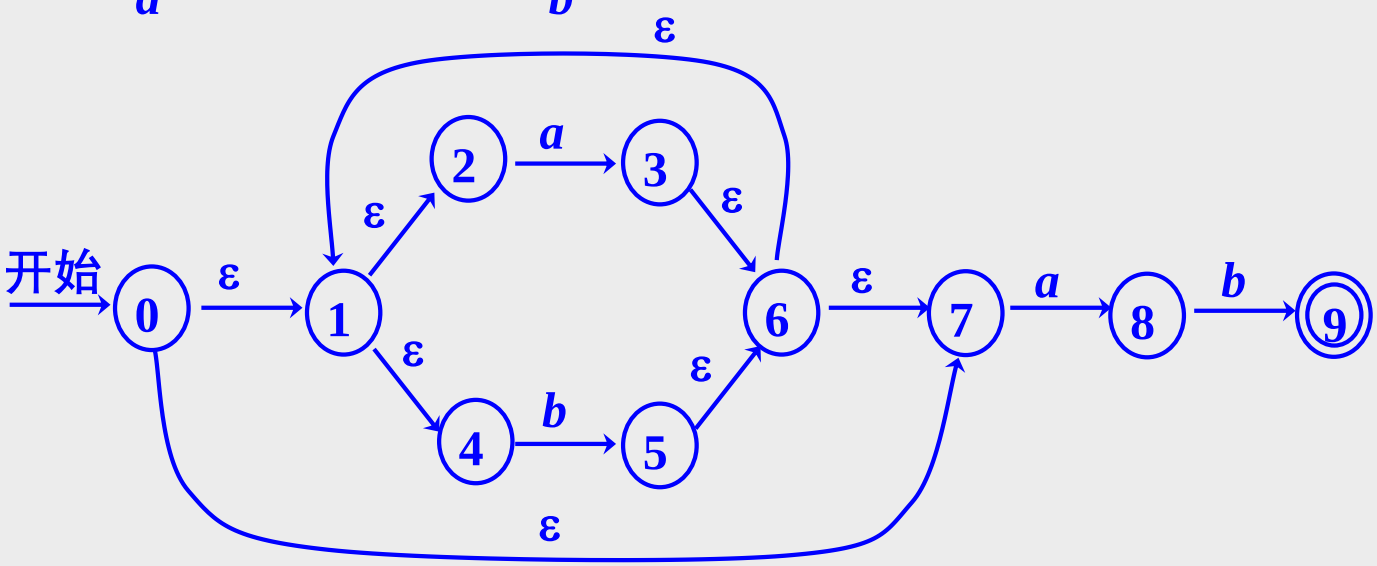




NFA构造过程举例

$(a|b)^*ab$的分解

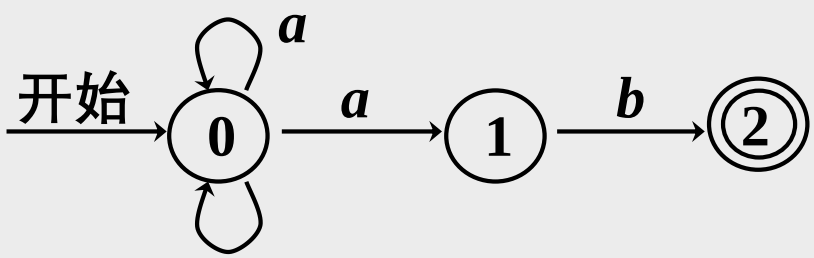




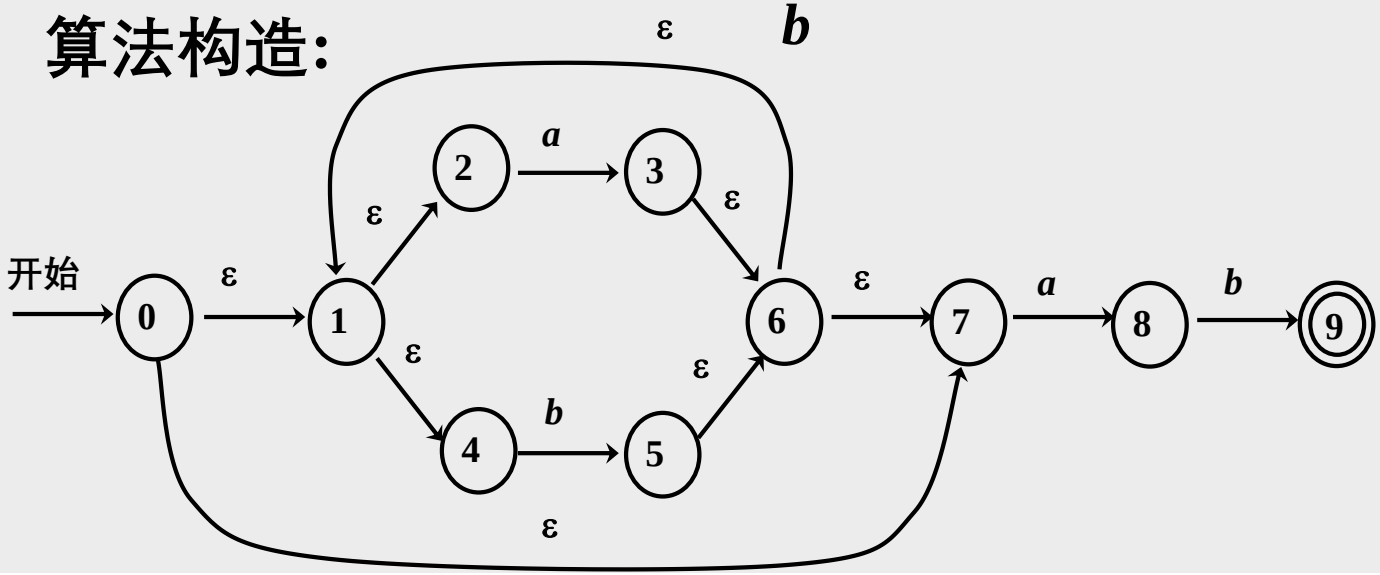
NFA构造过程举例

- (a|b)*ab的两个NFA的比较

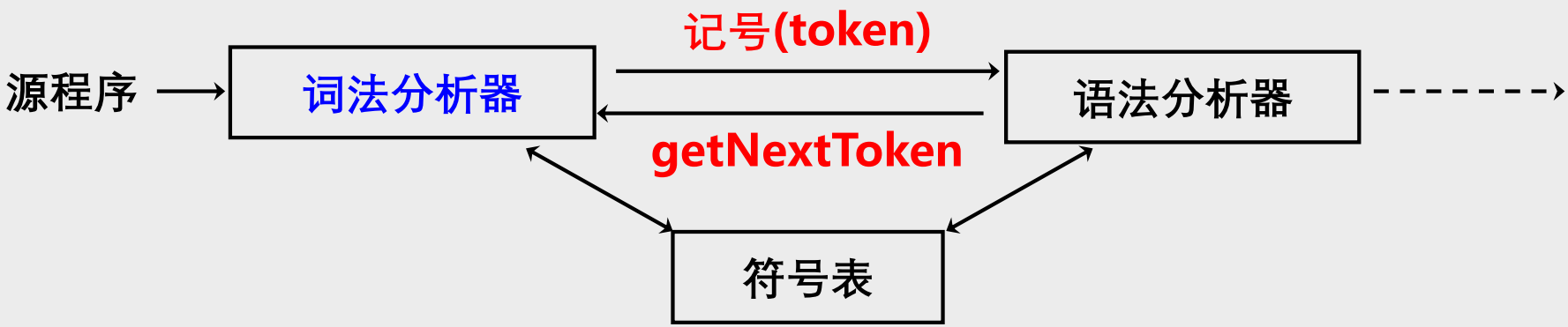
手工构造:



算法构造:



本节提纲



- 词法分析概述
- **词法分析器的自动生成**
  - 词法单元的描述：正则式
  - 词法单元的识别：转换图
  - 有限自动机：NFA、DFA
  - 正则表达式→NFA → DFA →化简的DFA

NFA到DFA的变换

子集构造法

- DFA的一个状态是NFA的一个状态集合
- 读了输入$a_1\ a_2 \ldots a_n$后，
  NFA能到达的所有状态： $s_1, s_2, \ldots, s_k$， 则
  DFA到达状态$\{s_1, s_2, \ldots, s_k\}$

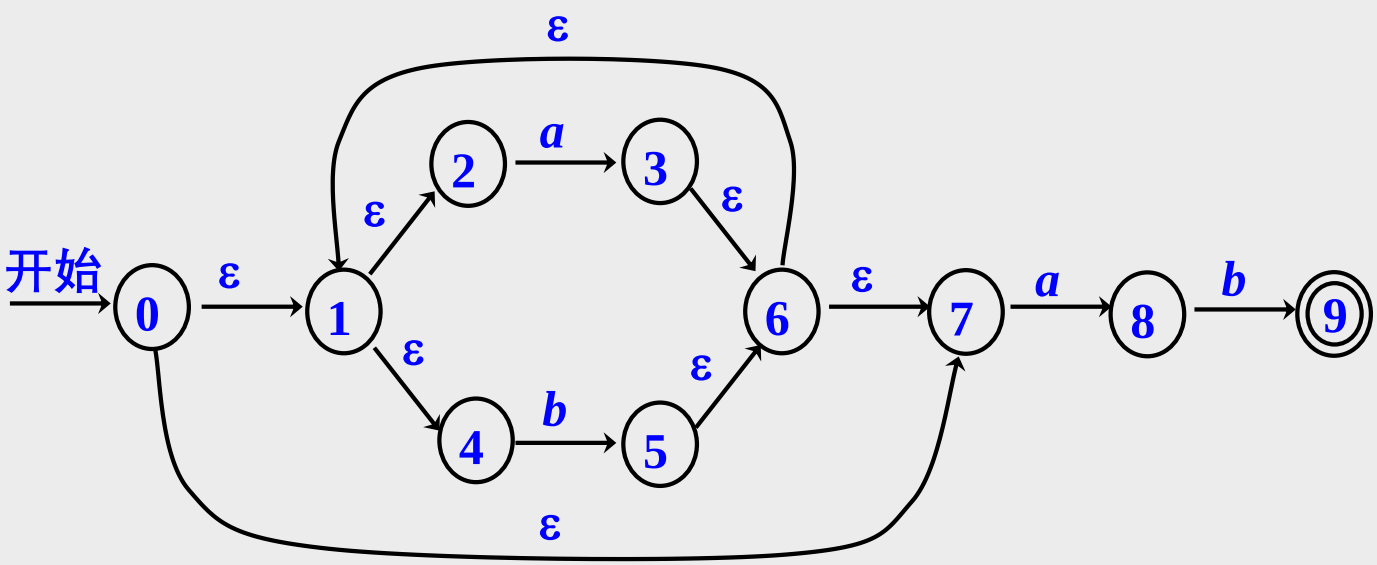
NFA到DFA的变换

子集构造法(subset construction)

- *ε*-闭包 (*ε- closure*)：状态*s* 的*ε*-闭包是 *s* 经 *ε*转换所能到达的状态集合
- NFA的初始状态的 *ε*-闭包对应于DFA的初始状态
- 针对每个DFA 状态 – NFA状态子集A，求输入每个a_i 后能到达的NFA状态的ε-闭包并集 ($\varepsilon\text{- closure}(move(A, a_i))$)，该集合对应于DFA中的一个已有状态，或者是一个要新加的DFA状态

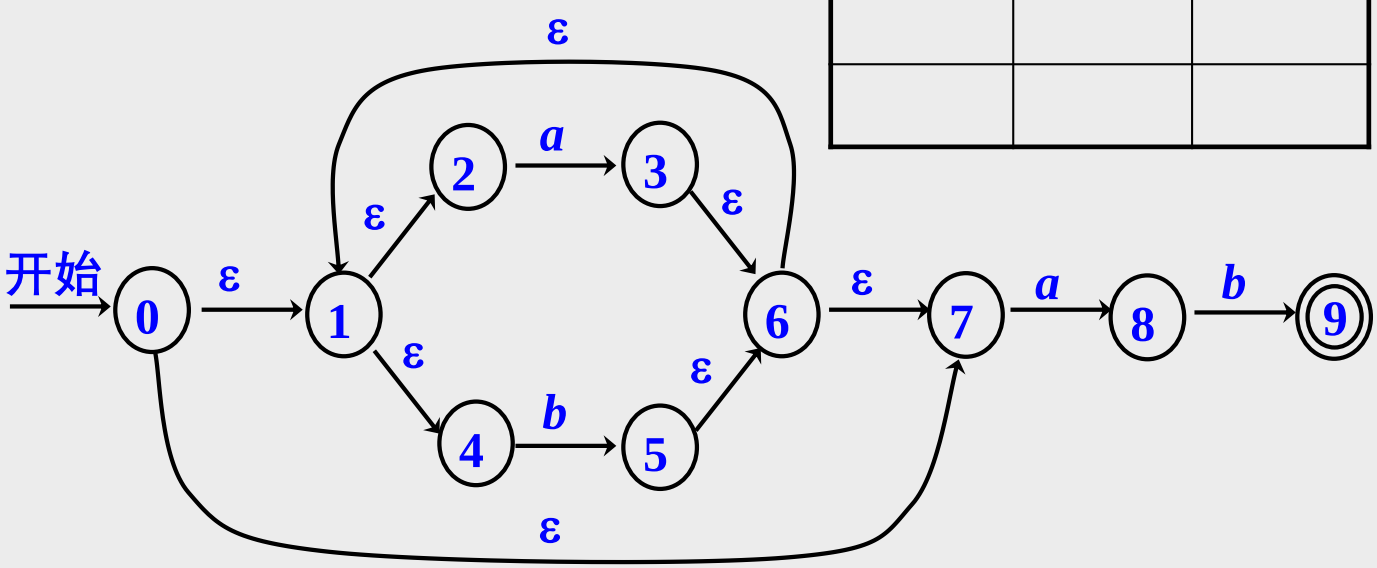
NFA到DFA的变换

- 例(a|b)*ab，NFA如下，把它变换为DFA



NFA到DFA的变换

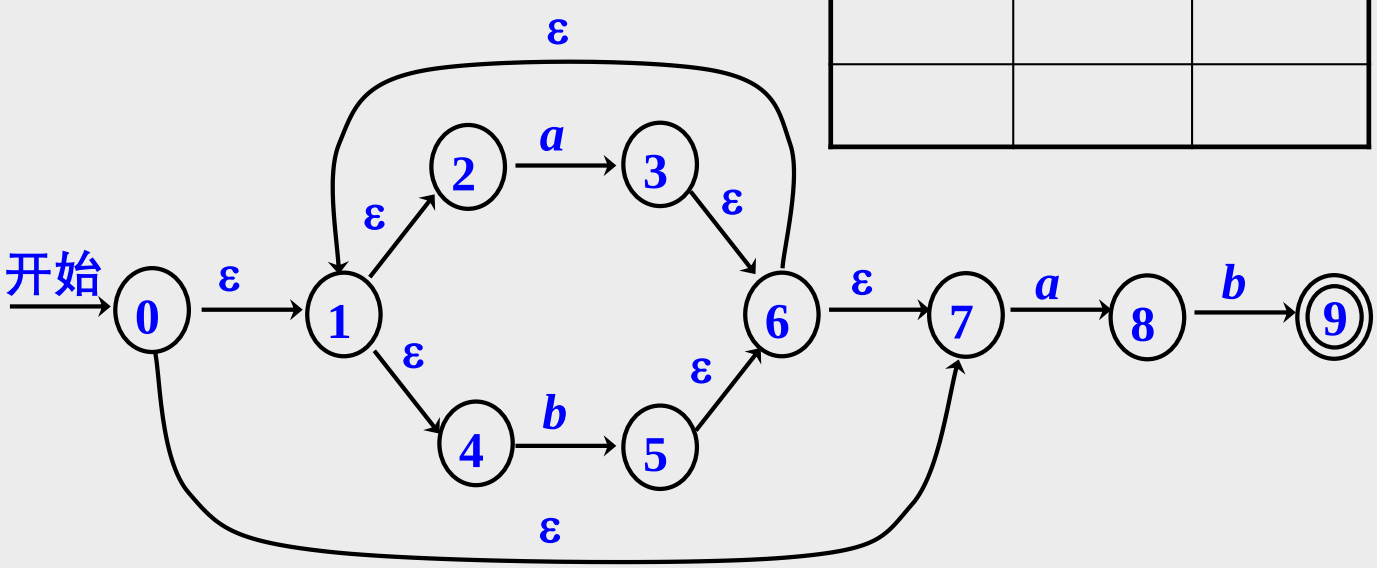
状态	输入符号	
	a	*b*



NFA到DFA的变换

$A = \{0, 1, 2, 4, 7\}$

状态	输入符号	
	a	*b*
A		

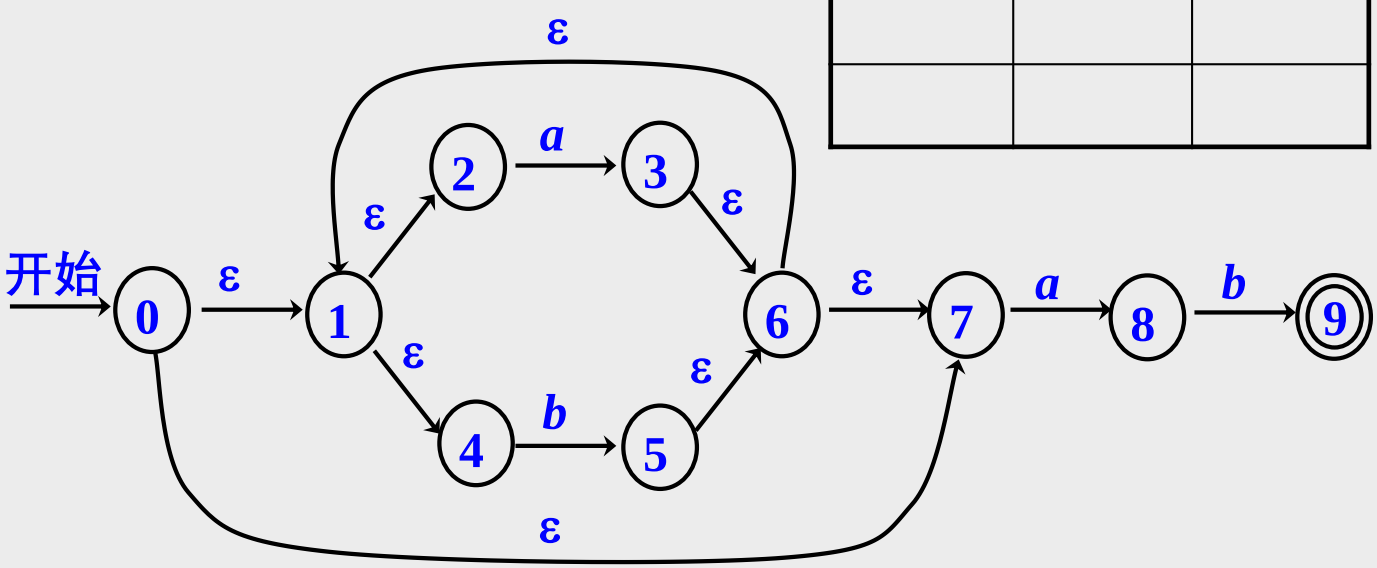


NFA到DFA的变换

$A = \{0, 1, 2, 4, 7\}$

$B = \{1, 2, 3, 4, 6, 7, 8\}$

状态	输入符号	
	a	*b*
A	*B*	



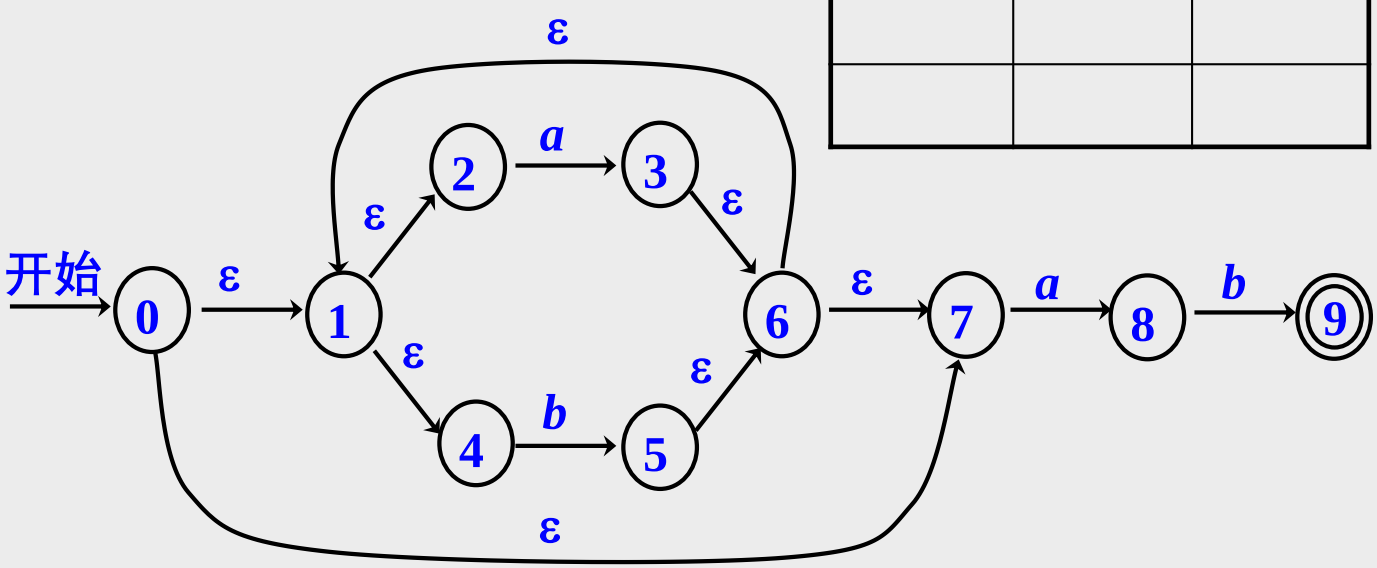
NFA到DFA的变换

A = {0, 1, 2, 4, 7}

B = {1, 2, 3, 4, 6, 7, 8}

C = {1, 2, 4, 5, 6, 7}

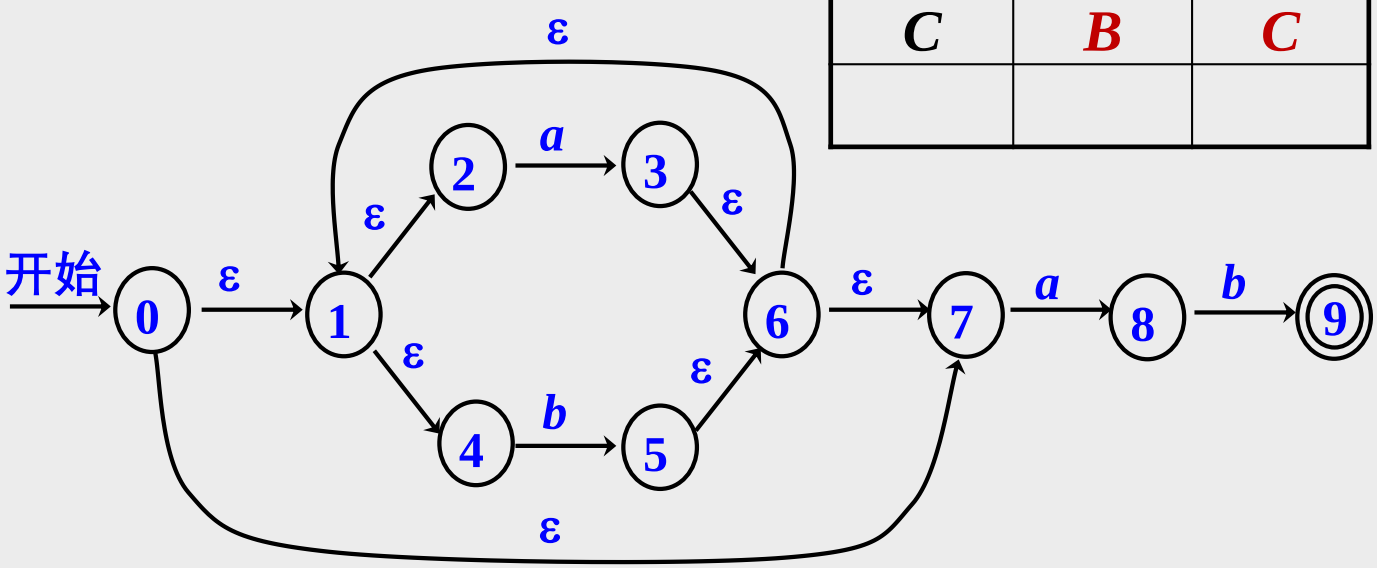
状态	输入符号	
	a	*b*
A	*B*	*C*



NFA到DFA的变换

$A = \{0, 1, 2, 4, 7\}$
$B = \{1, 2, 3, 4, 6, 7, 8\}$
$C = \{1, 2, 4, 5, 6, 7\}$

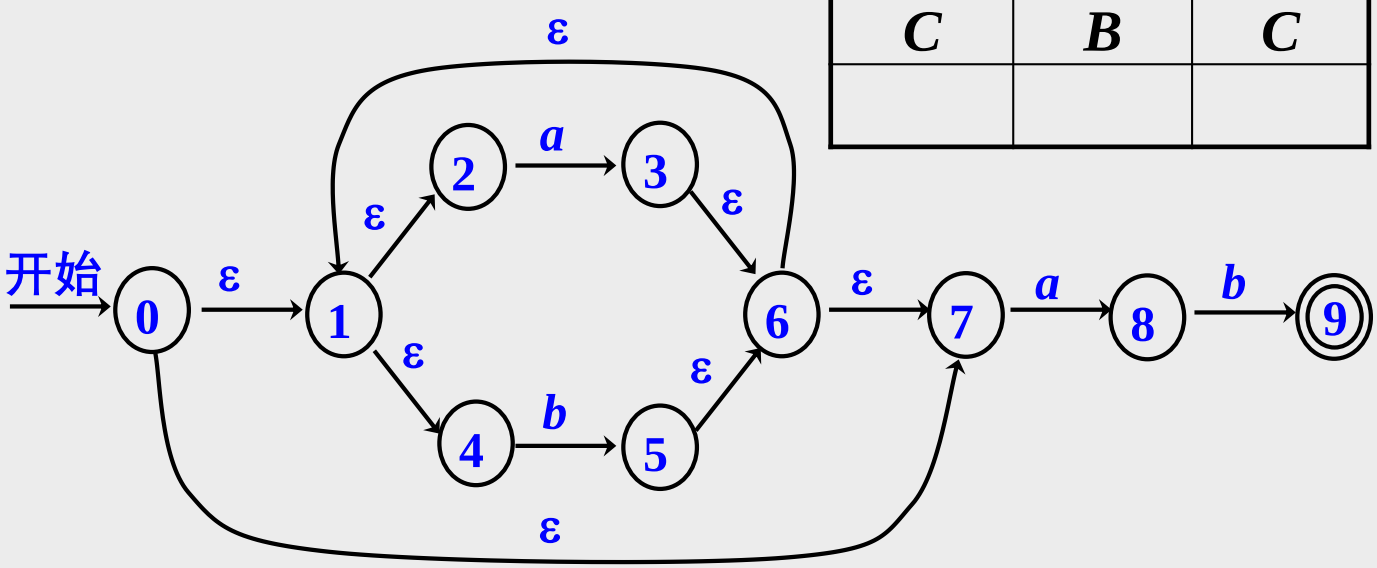
状态	输入符号	
	a	*b*
A	*B*	*C*
B	*B*	
C	*B*	*C*



NFA到DFA的变换

A = {0, 1, 2, 4, 7}
B = {1, 2, 3, 4, 6, 7, 8}
C = {1, 2, 4, 5, 6, 7}
D = {1, 2, 4, 5, 6, 7, 9}

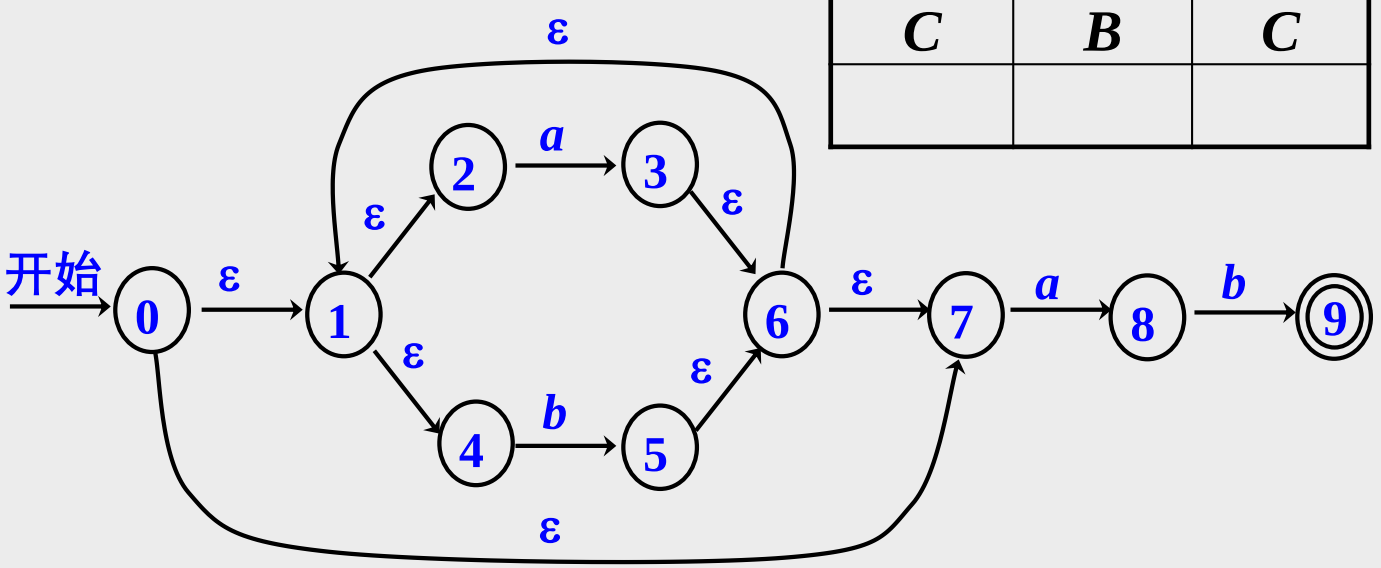
状态	输入符号	
	a	*b*
A	*B*	*C*
B	*B*	
C	*B*	*C*



NFA到DFA的变换

A = {0, 1, 2, 4, 7}
B = {1, 2, 3, 4, 6, 7, 8}
C = {1, 2, 4, 5, 6, 7}
D = {1, 2, 4, 5, 6, 7, 9}

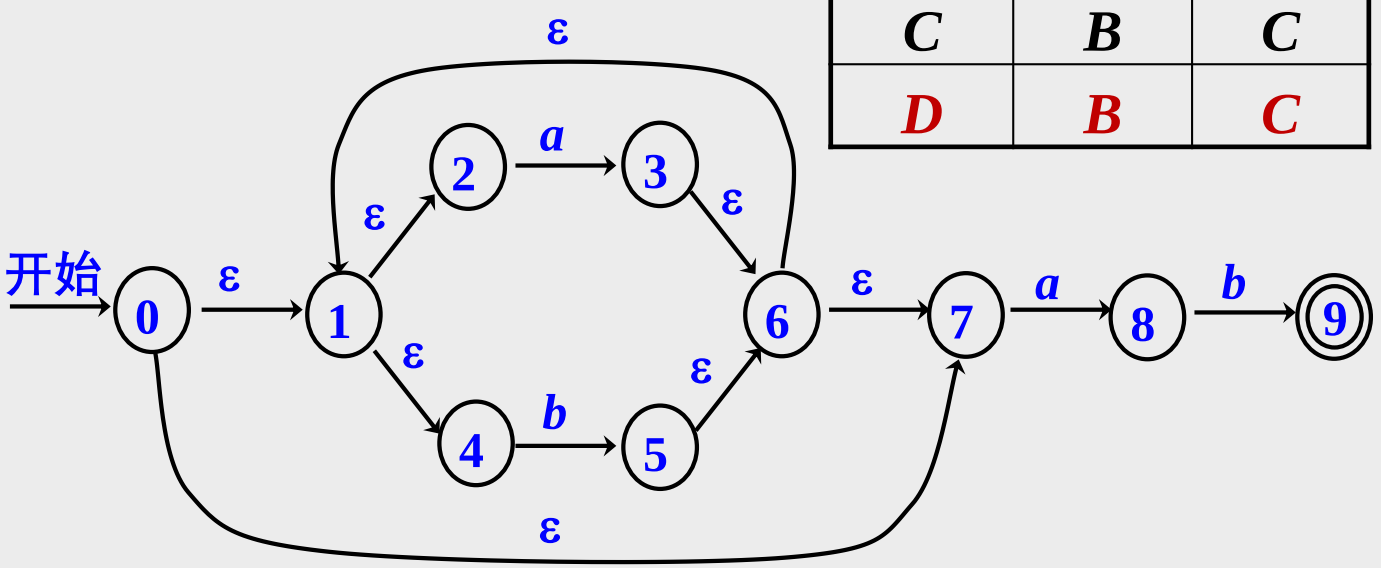
状态	输入符号	
	a	*b*
A	*B*	*C*
B	*B*	*D*
C	*B*	*C*



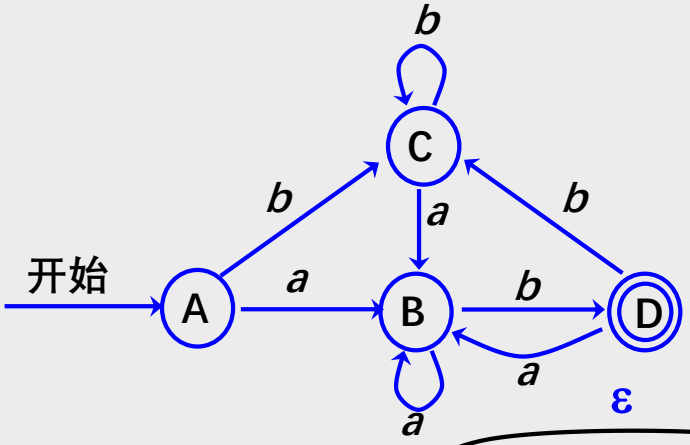
NFA到DFA的变换

A = {0, 1, 2, 4, 7}
B = {1, 2, 3, 4, 6, 7, 8}
C = {1, 2, 4, 5, 6, 7}
D = {1, 2, 4, 5, 6, 7, 9}

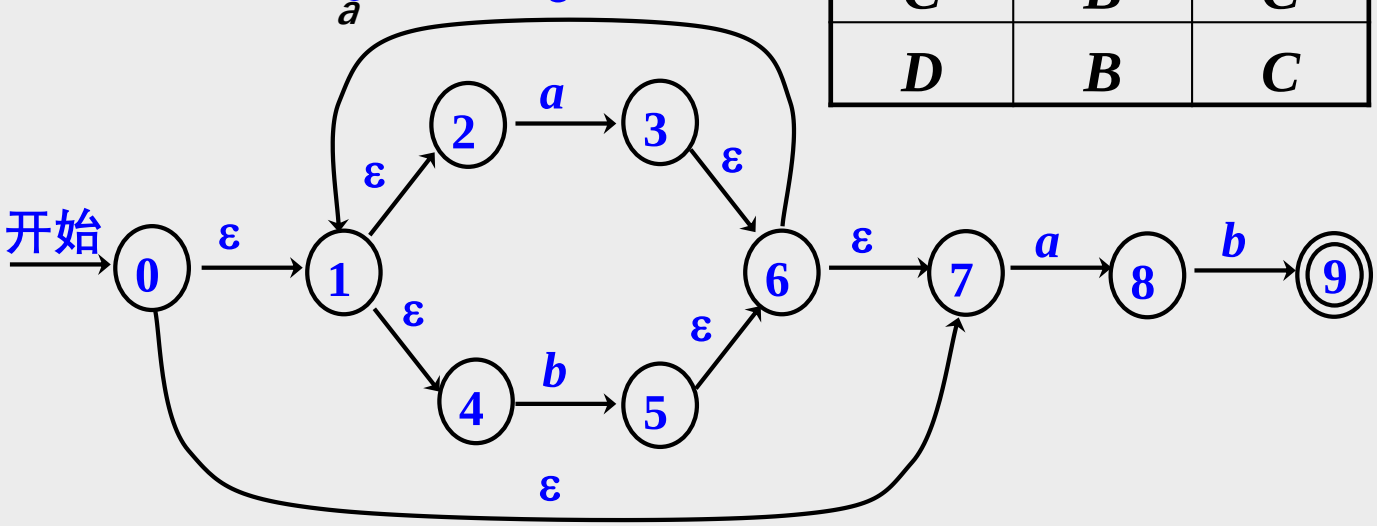
状态	输入符号	
	a	*b*
A	*B*	*C*
B	*B*	*D*
C	*B*	*C*
D	*B*	*C*



NFA到DFA的变换

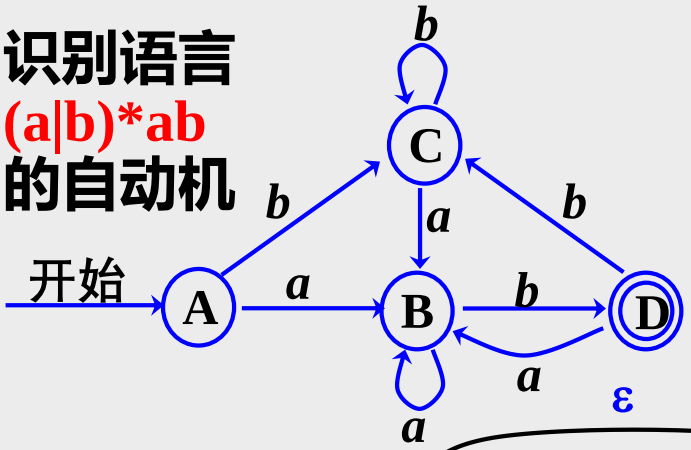


状态	输入符号	
	a	*b*
A	*B*	*C*
B	*B*	*D*
C	*B*	*C*
D	*B*	*C*

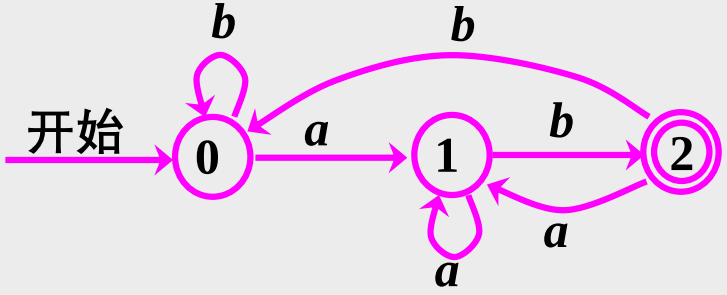


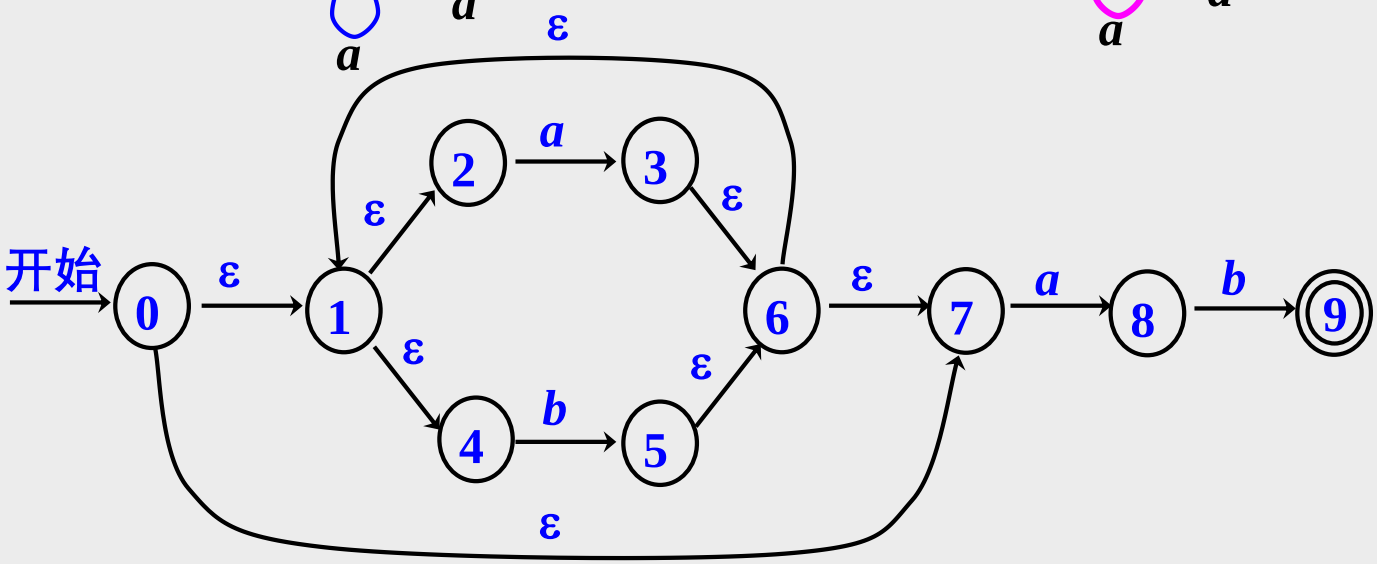
NFA到DFA的变换

识别语言
(a|b)*ab
的自动机

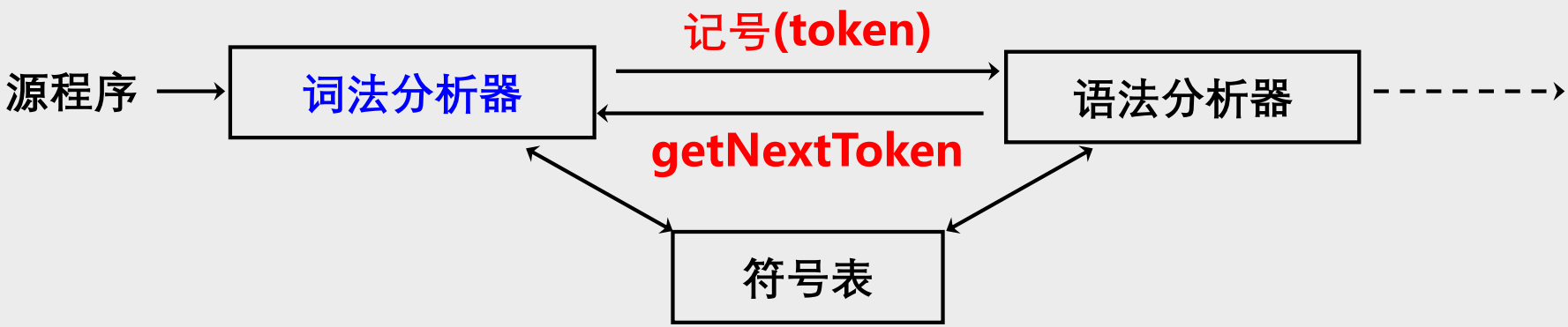


子集构造法不一
定得到最简DFA





本节提纲

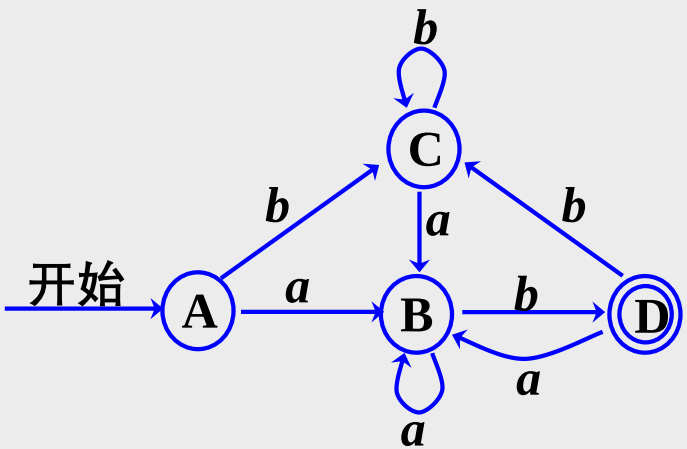


- 词法分析概述
- **词法分析器的自动生成**
  - 词法单元的描述：正则式
  - 词法单元的识别：转换图
  - 有限自动机：NFA、DFA
  - 正则表达式→NFA → DFA →化简的DFA

DFA的化简

- **A和B是可区别的状态**
  - 从A出发，读过单字符b构成的串，到达非接受状态C，而从B出发，读过串b，到达接受状态D
- **A和C是不可区别的状态**
  - 无任何串可用来像上面这样区别它们

可区别的状态要分开对待



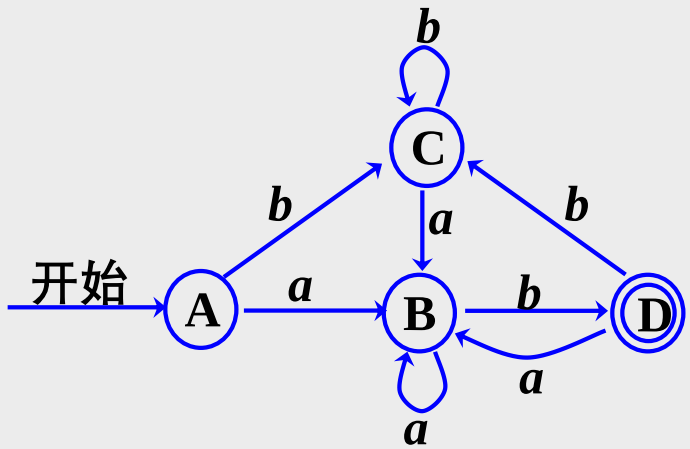
DFA的化简

1. 按是否是接受状态来区分

{A, B, C}, {D}

move({A, B, C}, a) = {B}

move({A, B, C}, b) = {C, D}



DFA的化简

- 1. 按是否是接受状态来区分

{A, B, C}, {D}

move({A, B, C}, a) = {B}

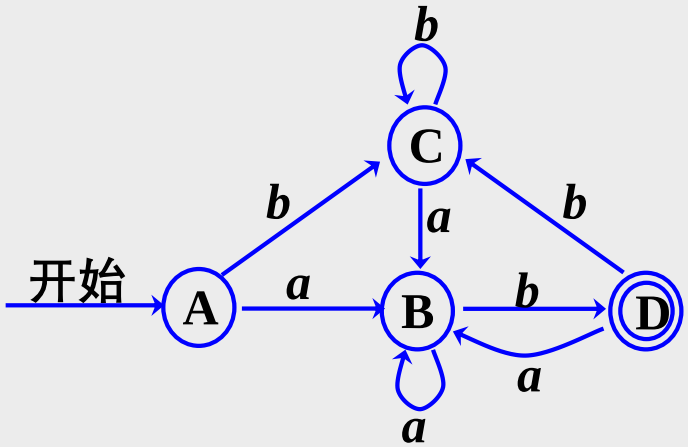
move({A, B, C}, b) = {C, D}

- 2. 继续分解

{A, C}, {B}, {D}

move({A, C}, a) = {B}

move({A, C}, b) = {C}



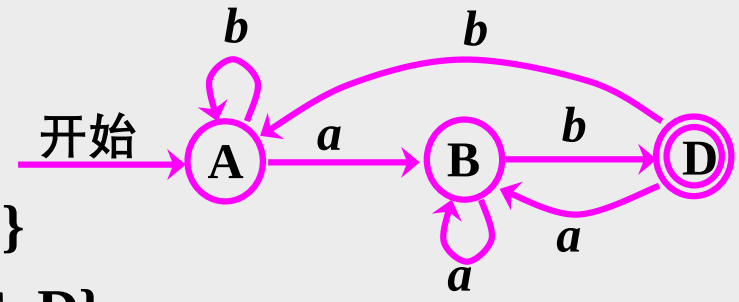
DFA的化简

- 1. 按是否是接受状态来区分

{A, B, C}, {D}

move({A, B, C}, a) = {B}

move({A, B, C}, b) = {C, D}

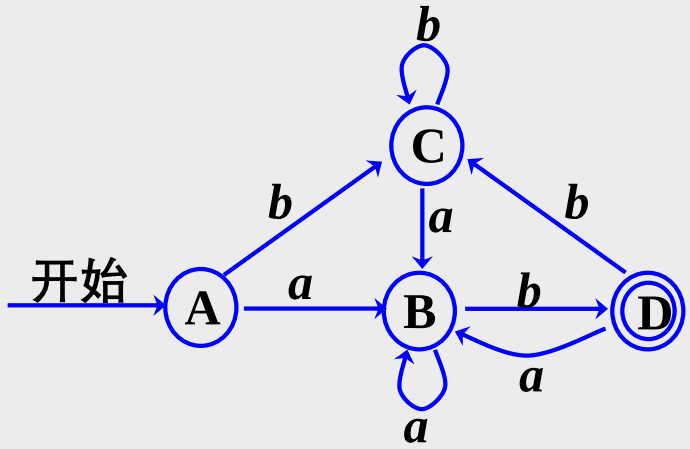


- 2. 继续分解

{A, C}, {B}, {D}

move({A, C}, a) = {B}

move({A, C}, b) = {C}



思考问题

- 正则表达式（a|b）*与（a*|b*）*是否等价?
  - 提示：可利用其最简化DFA的

- 有限自动机如何实现为代码?
  - 请课外阅读有限自动机的Python实现样例

本节总结

- 词法分析器的作用和接口，用高级语言编写词法分析器等内容
- 掌握下面涉及的一些概念，它们之间转换的技巧、方法或算法
  - 非形式描述的语言↔正则表达式
  - 正则表达式→ NFA
  - 非形式描述的语言↔ NFA
  - NFA → DFA
  - DFA →最简DFA
  - 非形式描述的语言 ↔ DFA（或最简DFA）

《编译原理和技术》

词法分析

谢谢！

《编译原理和技术》

语法分析 I

中科大计算机学院
李 诚
2022-09-07

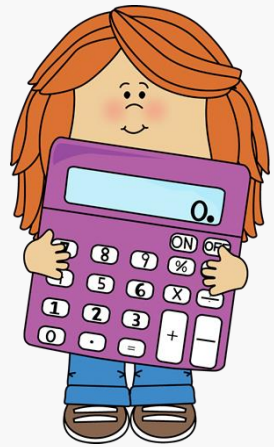
简单计算器程序

1 + (2 – 3) 合法

1 + 2 – 3 + 非法

1 2 + – 3 非法

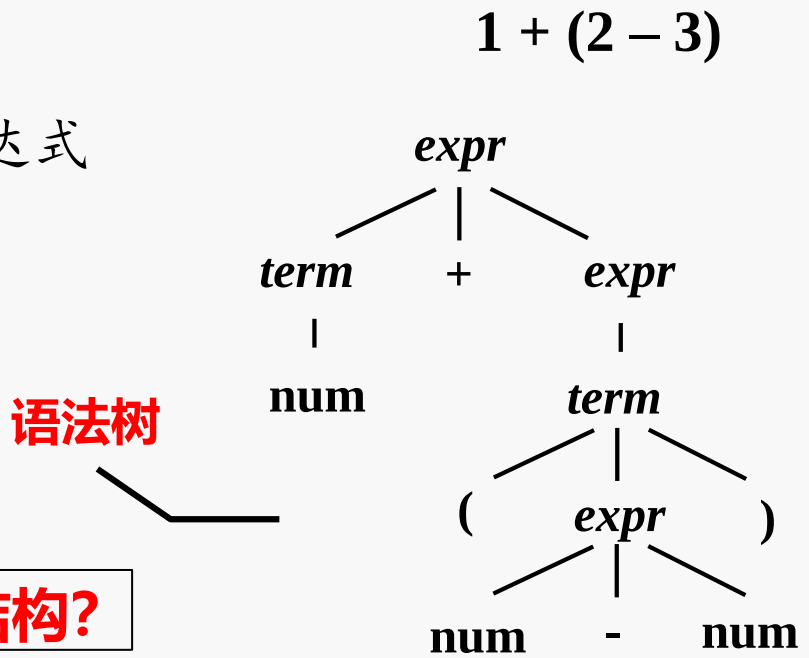
1 + 2 – *a* 非法



语法分析的目的是教会计算机判断输入合法性

如何判定输入合法性呢?

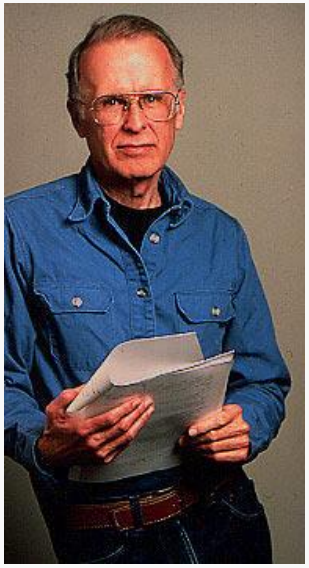
- **首先要规定好合法的基本单元——词法分析**
  - 由0-9组成的数字(num)和符号+、-、(、)
- **其次要理解算术表达式的构成**
  - 大表达式(expr)可拆为若干子表达式
  - 拆解过程是递归的
  - 直至看到基本单元



问题一：如何描述编程语言的语法结构?

John Backus -1977图灵奖

- **提出了多种高级编程语言**
  - Speedcoding -> FORTRAN -> ALGOL 58 -> ALGOL 60 -> FP
- **提出了编译技术的理论基础**
  - 巴科斯范式（Backus–Naur Form）
  - 上下文无关文法
- **对计算机科学影响巨大**
  - 诞生了许多理论研究成果
  - 现代编译器还保留了FORTRAN I的大概架构



弗吉尼亚大学化学专业，哥伦比亚大学数学专业，曾服务于阿波罗登月计划

上下文无关文法——描述语言的语法结构

- 上下文无关文法的形式化定义：

$$(V_T, V_N, S, P)$$

V_T：终结符集合

- 终结符：是文法所定义的语言的基本符号，也称为“*token*”
- 例：V_T = { num, +, -, (,)}

上下文无关文法——描述语言的语法结构

上下文无关文法的形式化定义：

$$(V_T, V_N, S, P)$$

V_N：非终结符集合

- 非空有限集合，$V_T \cap V_N = \emptyset$
- 非终结符：表示语法成分的符号， 存放中间结果，也称为“语法变量”
- 例：$V_N = \{ expr, term \}$

上下文无关文法——描述语言的语法结构

- 上下文无关文法的形式化定义：

$$(V_T, V_N, S, P)$$

S：开始符号

- 属于非终结符，是该文法中最大的语法成分，分析开始的地方
- 例：$S = expr$

上下文无关文法——描述语言的语法结构

- 上下文无关文法的形式化定义：

$$(V_T, V_N, S, P)$$

P：产生式集合

- 产生式：描述了将终结符和非终结符组合成串的方法
- 例：$P = \{expr \rightarrow term \mid term + expr \mid term - expr$

$term \rightarrow \text{num} \mid (expr)\}$

举例

- 例：描述简单计算器的上下文无关文法

四元组：(V_T, V_N, S, P)

$(\{\text{num}, +, -, (,)\}, \{expr, term\}, expr, P)$

$$P = \{expr \rightarrow term \mid term + expr \mid term - expr$$

$$term \rightarrow \text{num} \mid (expr)\}$$

问题二：给定文法，如何判定输入串属于文法规定的语言呢?

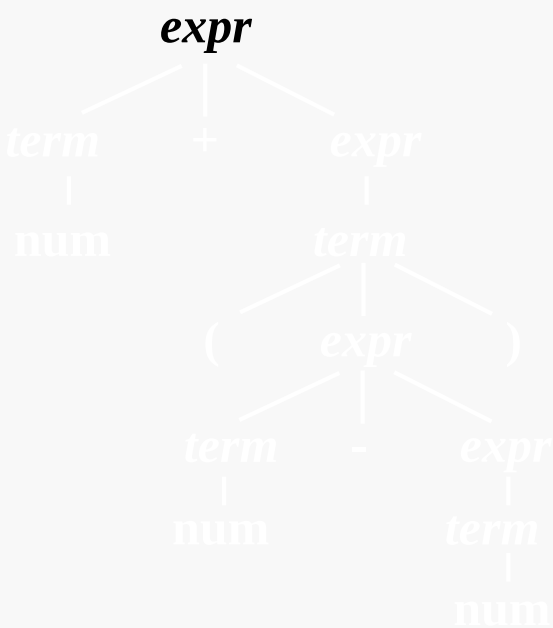
图形化演示构造过程

- 例：对于文法 $expr \rightarrow term \mid term + expr \mid term - expr$

  $term \rightarrow num \mid (expr)$

  - 展示1 + (2 – 3) 的构造过程

$$
\begin{aligned}
expr &\Rightarrow term + expr \\
&\Rightarrow num + expr \\
&\Rightarrow num + term \\
&\Rightarrow num + (expr) \\
&\Rightarrow num + (term - expr) \\
&\Rightarrow num + (num - expr) \\
&\Rightarrow num + (num - term) \\
&\Rightarrow num + (num - num)
\end{aligned}
$$



图形化演示构造过程

- 例：对于文法 $expr \rightarrow term \mid term + expr \mid term - expr$

$$term \rightarrow num \mid (expr)$$

- 展示1 + (2 – 3) 的构造过程

$expr \Rightarrow term + expr$

$\Rightarrow num + expr$

$\Rightarrow num + term$

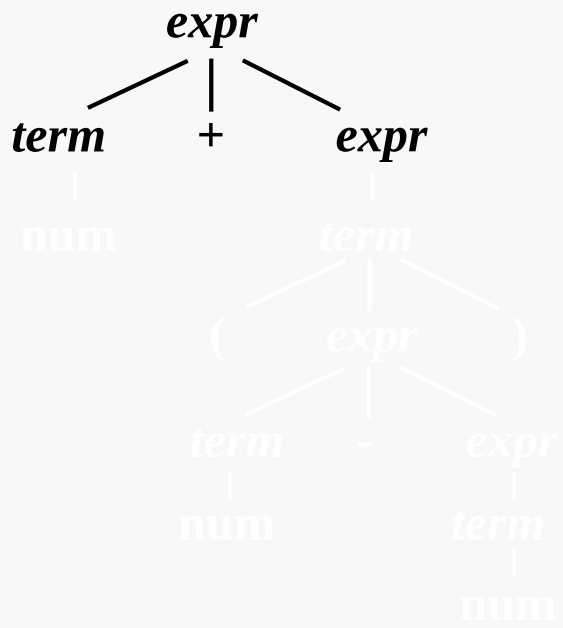
$\Rightarrow num + (expr)$

$\Rightarrow num + (term - expr)$

$\Rightarrow num + (num - expr)$

$\Rightarrow num + (num - term)$

$\Rightarrow num + (num - num)$



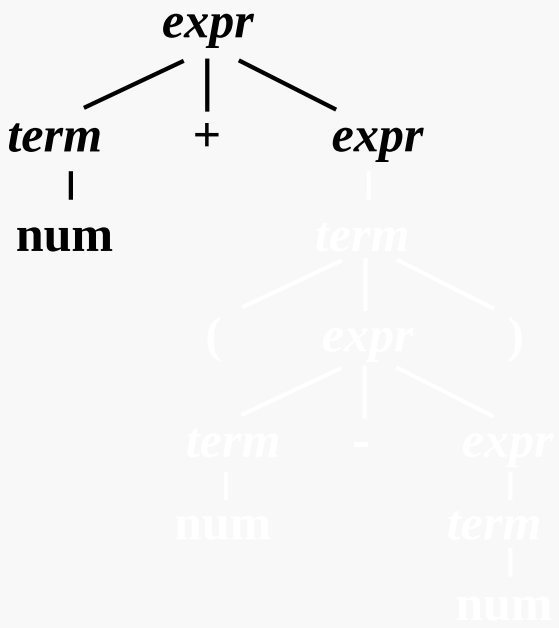
图形化演示构造过程

- 例：对于文法 $expr \rightarrow term \mid term + expr \mid term - expr$

  $term \rightarrow num \mid (expr)$

  - 展示1 + (2 – 3) 的构造过程

$$
\begin{aligned}
expr &\Rightarrow term + expr \\
&\Rightarrow num + expr \\
&\Rightarrow num + term \\
&\Rightarrow num + (expr) \\
&\Rightarrow num + (term - expr) \\
&\Rightarrow num + (num - expr) \\
&\Rightarrow num + (num - term) \\
&\Rightarrow num + (num - num)
\end{aligned}
$$



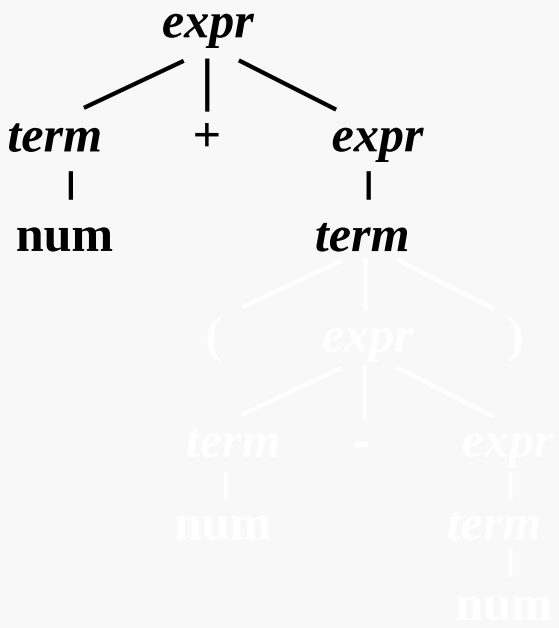
图形化演示构造过程

- 例：对于文法 $expr \rightarrow term \mid term + expr \mid term - expr$

$$term \rightarrow num \mid (expr)$$

- 展示1 + (2 – 3) 的构造过程

$$\begin{aligned} expr &\Rightarrow term + expr \\ &\Rightarrow num + expr \\ &\Rightarrow num + term \\ &\Rightarrow num + (expr) \\ &\Rightarrow num + (term - expr) \\ &\Rightarrow num + (num - expr) \\ &\Rightarrow num + (num - term) \\ &\Rightarrow num + (num - num) \end{aligned}$$



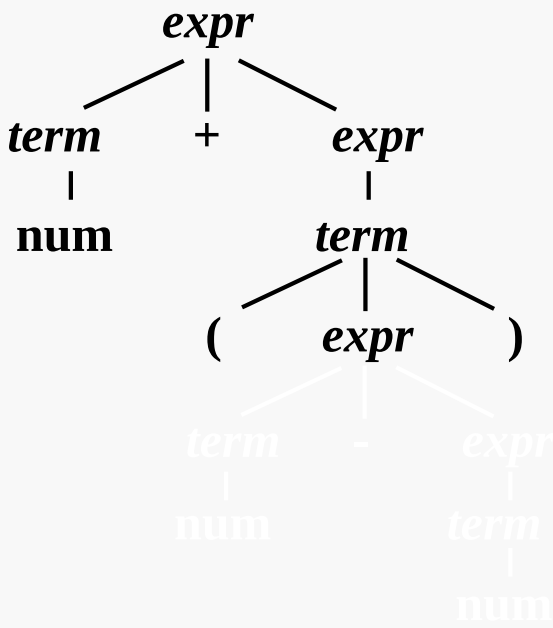
图形化演示构造过程

- 例：对于文法 *expr* → *term* | *term* + *expr* | *term* - *expr*

  term → num | (*expr*)

  - 展示1 + (2 – 3) 的构造过程

expr ⇒ *term* + *expr*
⇒ num + *expr*
⇒ num + *term*
⇒ num + (*expr*)
⇒ num + (*term* – *expr*)
⇒ num + (num – *expr*)
⇒ num + (num – *term*)
⇒ num + (num – num)



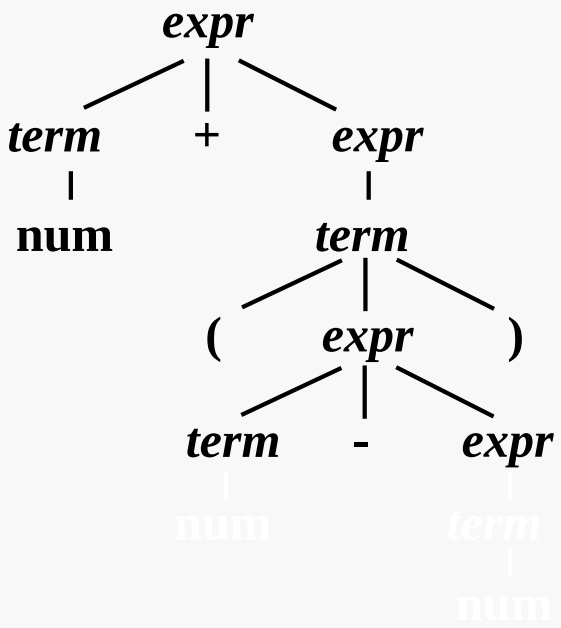
图形化演示构造过程

- 例：对于文法 $expr \rightarrow term \mid term + expr \mid term - expr$

$$term \rightarrow num \mid (expr)$$

- 展示1 + (2 – 3) 的构造过程

$$\begin{aligned} expr &\Rightarrow term + expr \\ &\Rightarrow num + expr \\ &\Rightarrow num + term \\ &\Rightarrow num + (expr) \\ &\Rightarrow num + (term - expr) \\ &\Rightarrow num + (num - expr) \\ &\Rightarrow num + (num - term) \\ &\Rightarrow num + (num - num) \end{aligned}$$



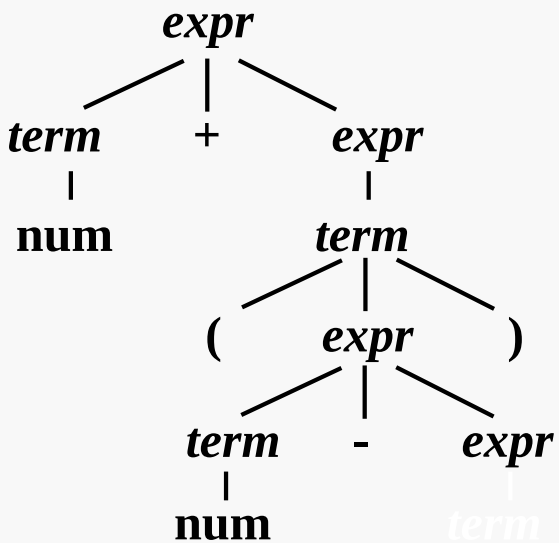
图形化演示构造过程

- 例：对于文法 $expr \rightarrow term \mid term + expr \mid term - expr$

  $term \rightarrow num \mid (expr)$

  - 展示1 + (2 – 3) 的构造过程

$$
\begin{aligned}
expr &\Rightarrow term + expr \\
&\Rightarrow num + expr \\
&\Rightarrow num + term \\
&\Rightarrow num + (expr) \\
&\Rightarrow num + (term - expr) \\
&\Rightarrow num + (num - expr) \\
&\Rightarrow num + (num - term) \\
&\Rightarrow num + (num - num)
\end{aligned}
$$



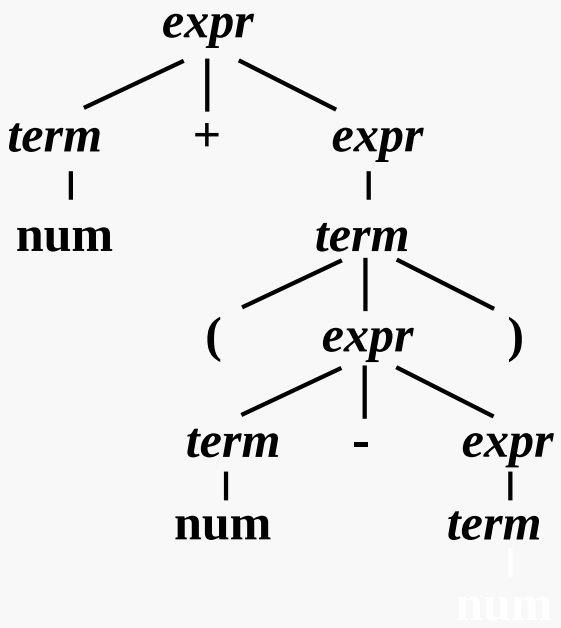
图形化演示构造过程

- 例：对于文法 $expr \rightarrow term \mid term + expr \mid term - expr$

$$term \rightarrow num \mid (expr)$$

- 展示1 + (2 – 3) 的构造过程

$$
\begin{aligned}
expr &\Rightarrow term + expr \\
&\Rightarrow num + expr \\
&\Rightarrow num + term \\
&\Rightarrow num + (expr) \\
&\Rightarrow num + (term - expr) \\
&\Rightarrow num + (num - expr) \\
&\Rightarrow num + (num - term) \\
&\Rightarrow num + (num - num)
\end{aligned}
$$



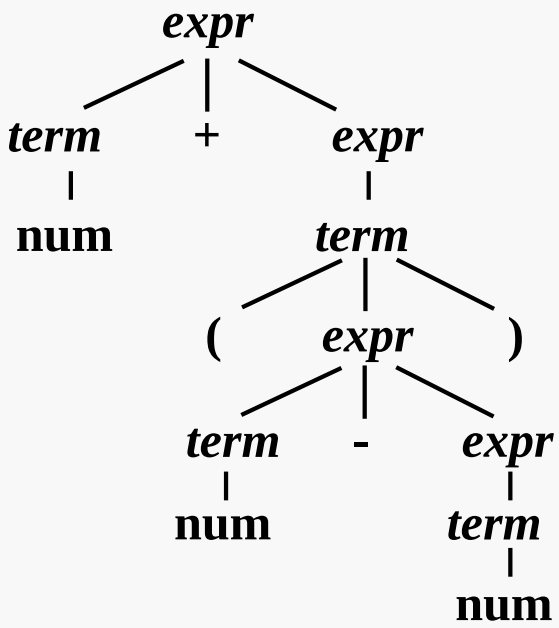
图形化演示构造过程

- 例：对于文法 $expr \rightarrow term \mid term + expr \mid term - expr$

  $term \rightarrow num \mid (expr)$

  - 展示1 + (2 – 3) 的构造过程

$$
\begin{aligned}
expr &\Rightarrow term + expr \\
&\Rightarrow num + expr \\
&\Rightarrow num + term \\
&\Rightarrow num + (expr) \\
&\Rightarrow num + (term - expr) \\
&\Rightarrow num + (num - expr) \\
&\Rightarrow num + (num - term) \\
&\Rightarrow num + (num - num)
\end{aligned}
$$



上下文无关文法的推导

- **推导 (Derivation)**
  - 是从文法推出文法所描述的语言中所包含的合法串集合的动作
  - 把产生式看成重写规则，把符号串中的非终结符用其产生式右部的串来代替

- **例** $E \to E + E \mid E * E \mid (E) \mid -E \mid \mathrm{id}$

  $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\mathrm{id} + E) \Rightarrow -(\mathrm{id} + \mathrm{id})$

- **记法:**
  - $S \Rightarrow^* \alpha$: 0步或多步推导
  - $S \Rightarrow^+ w$: 1步或多步推导

最左推导和最右推导

- 例 $E \to E + E \mid E * E \mid (E) \mid -E \mid \text{id}$

- **最左推导** (leftmost derivation)
  - 每步代换最左边的非终结符

$$E \Rightarrow_{lm} -E \Rightarrow_{lm} -(E) \Rightarrow_{lm} -(E + E)$$
$$\Rightarrow_{lm} -(\text{id} + E) \Rightarrow_{lm} -(\text{id} + \text{id})$$

- **最右推导** (rightmost or canonical derivation，规范推导)
  - 每步代换最右边的非终结符

$$E \Rightarrow_{rm} -E \Rightarrow_{rm} -(E) \Rightarrow_{rm} -(E + E)$$
$$\Rightarrow_{rm} -(E + \text{id}) \Rightarrow_{rm} -(\text{id} + \text{id})$$

思考题?

- **上下文无关是什么意思?**
  - 上下文无关指的是在文法推导的每一步

$$\alpha A \beta \Rightarrow \alpha \gamma \beta$$

符号串γ仅依据A的产生式推导，而无需依赖A的上下文α和β

语言、文法、句型、句子

上下文无关语言

- 上下文无关文法G产生的语言：从开始符号S出发，经$\Rightarrow^+$推导所能到达的所有仅由终结符组成的串
- 句型(sentential form)：$S \Rightarrow^* \alpha$，S是开始符号，α是由终结符和/或非终结符组成的串，则α是文法G的句型
- 句子(sentence)：仅由终结符组成的句型

等价的文法

- 它们产生同样的语言

最左推导和最右推导

- 例 $E \to E + E \mid E * E \mid (E) \mid -E \mid \mathrm{id}$

褐红色标出的均是句型

- **最左推导** (leftmost derivation)
  - 每步代换最左边的非终结符

$$E \Rightarrow_{lm} -E \Rightarrow_{lm} -(E) \Rightarrow_{lm} -(E + E)$$
$$\Rightarrow_{lm} -(\mathrm{id} + E) \Rightarrow_{lm} -(\mathrm{id} + \mathrm{id})$$

- **最右推导** (rightmost or canonical derivation,规范推导)
  - 每步代换最右边的非终结符

$$E \Rightarrow_{rm} -E \Rightarrow_{rm} -(E) \Rightarrow_{rm} -(E + E)$$
$$\Rightarrow_{rm} -(E + \mathrm{id}) \Rightarrow_{rm} -(\mathrm{id} + \mathrm{id})$$

最左推导和最右推导

- 例 $E \to E + E \mid E * E \mid (E) \mid -E \mid \mathrm{id}$

- **最左推导** (leftmost derivation)
  - 每步代换最左边的非终结符

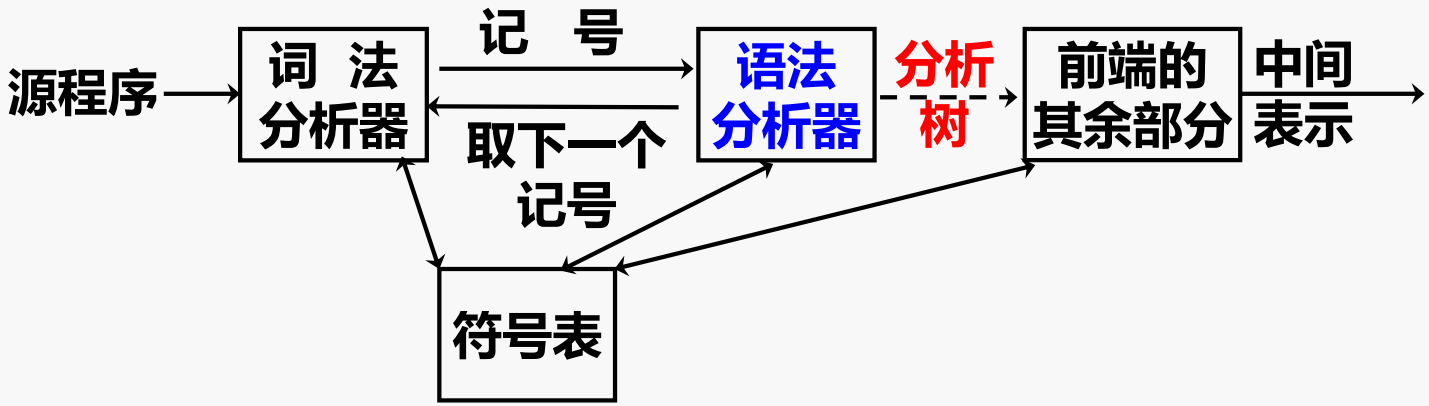
$$E \Rightarrow_{lm} -E \Rightarrow_{lm} -(E) \Rightarrow_{lm} -(E + E)$$
$$\Rightarrow_{lm} -(\mathrm{id} + E) \Rightarrow_{lm} -(\mathrm{id} + \mathrm{id})$$

褐红色标出的均是句子

- **最右推导** (rightmost or canonical derivation,规范推导)
  - 每步代换最右边的非终结符

$$E \Rightarrow_{rm} -E \Rightarrow_{rm} -(E) \Rightarrow_{rm} -(E + E)$$
$$\Rightarrow_{rm} -(E + \mathrm{id}) \Rightarrow_{rm} -(\mathrm{id} + \mathrm{id})$$

主要内容



- 语法分析器简介
- 上下文无关文法CFG：定义、推导
- 思考与拓展
  - 正则表达式与CFG的联系与区别
  - CFG二义性及消除方法

正则表达式的局限

正则表达式的表达能力

- 定义一些简单的语言，能表示给定结构的固定次数的重复或者没有指定次数的重复

  例：$a\,(ba)^5$, $a\,(ba)^*$

- 不能用于描述配对或嵌套的结构

  例1：配对括号串的集合，如不能表达 $(^n)^n$, $n \geq 0$

  例2：{wcw | w是a和b的串}

正则表达式的局限

正则表达式的表达能力

- 定义一些简单的语言，能表示给定结构的固定次数的重复或者没有指定次数的重复

  例：$a\,(ba)^5$, $a\,(ba)^*$

- 不能用于描述配对或嵌套的结构

  例1：配对括号串的集合，如不能表达 $(^n)^n$, $n \geq 0$

  例2：{wcw | w是a和b的串}

  原因：有限自动机无法记录访问同一状态的次数

思考题

- 请写出语言$\{(^n)^n \mid n \geq 0\}$ 的CFG文法

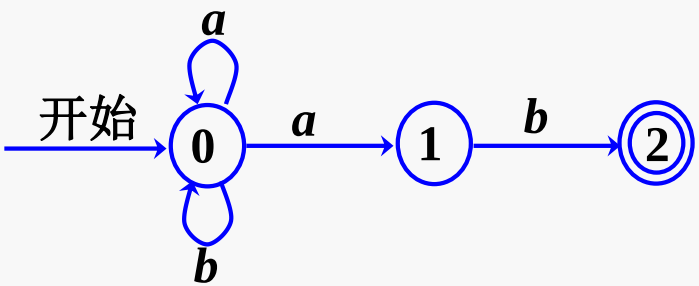
思考题

- 请写出语言$\{(^n)^n \mid n \geq 0\}$ 的CFG文法
  - $S \rightarrow (S) \mid \varepsilon$

正则表达式与CFG的区别

- 都能表示语言
- 能用正则表达式表示的语言都能用CFG表示
  - 正则表达式

    $(a|b)^*ab$



  - CFG文法

$A_0 \to a\,A_0 \mid b\,A_0 \mid a\,A_1$

$A_1 \to b\,A_2$

$A_2 \to \varepsilon$

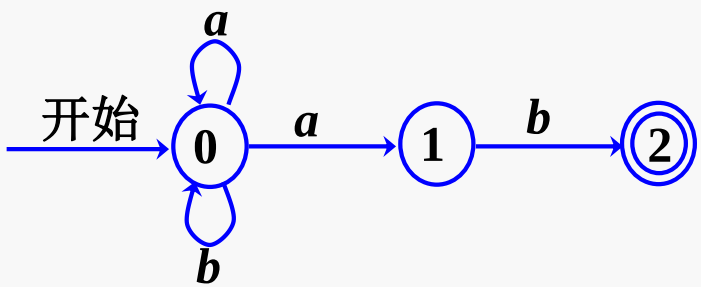
正则表达式与CFG的区别

- NFA →上下文无关文法
  - 确定终结符集合
  - 为每个状态引入一个非终结符Ai
  - 如果状态i有一个a转换到状态j,引入产生式Ai→aAj,如果i是接受状态，则引入Ai→ ε

$A_0 \rightarrow a\,A_0 \mid b\,A_0 \mid a\,A_1$

$A_1 \rightarrow b\,A_2$

$A_2 \rightarrow \varepsilon$



思考题

- 请为描述所有由0或1组成的回文字符串的语言设计CFG文法

思考题

- **请为描述所有由0或1组成的回文字符串的语言设计CFG文法**
  - $S \rightarrow 0S0 \mid 1S1 \mid 0 \mid 1 \mid \varepsilon$

文法的二义性

- 文法的某些句子存在**不止一种**最左(最右)推导， 或者**不止一棵**分析树，则该文法是二义的。

文法的二义性

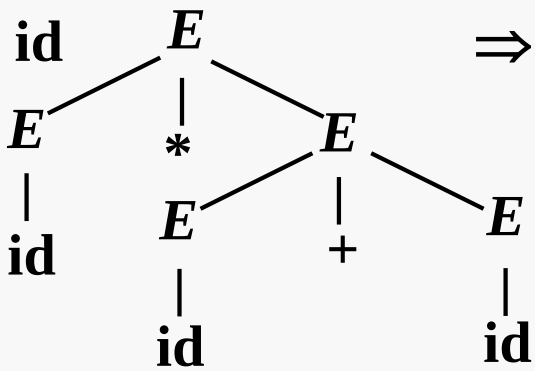
- 例 $E \to E + E \mid E * E \mid (E) \mid - E \mid \mathbf{id}$
  - id * id + id有两个不同的最左推导

$$
\begin{aligned}
E &\Rightarrow E * E \\
&\Rightarrow \mathbf{id} * E \\
&\Rightarrow \mathbf{id} * E + E \\
&\Rightarrow \mathbf{id} * \mathbf{id} + E \\
&\Rightarrow \mathbf{id} * \mathbf{id} + \mathbf{id}
\end{aligned}
\qquad
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow E * E + E \\
&\Rightarrow \mathbf{id} * E + E \\
&\Rightarrow \mathbf{id} * \mathbf{id} + E \\
&\Rightarrow \mathbf{id} * \mathbf{id} + \mathbf{id}
\end{aligned}
$$

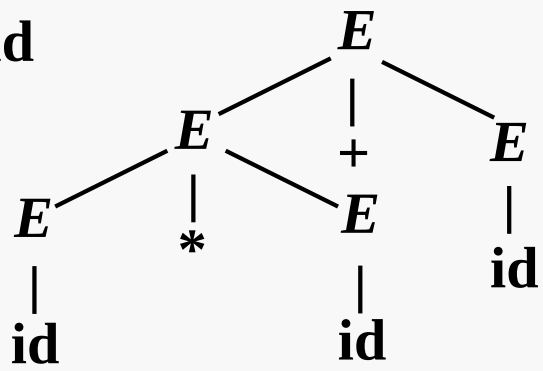
文法的二义性

- 例 $E \rightarrow E + E \mid E * E \mid (E) \mid - E \mid \mathbf{id}$
  - id * id + id有两棵不同的分析树

$E \Rightarrow E * E$
$\Rightarrow \mathbf{id} * E$
$\Rightarrow \mathbf{id} * E + E$
$\Rightarrow \mathbf{id} * \mathbf{id} + E$
$\Rightarrow \mathbf{id} * \mathbf{id} + \mathbf{id}$



$E \Rightarrow E + E$
$\Rightarrow E * E + E$
$\Rightarrow \mathbf{id} * E + E$
$\Rightarrow \mathbf{id} * \mathbf{id} + E$
$\Rightarrow \mathbf{id} * \mathbf{id} + \mathbf{id}$

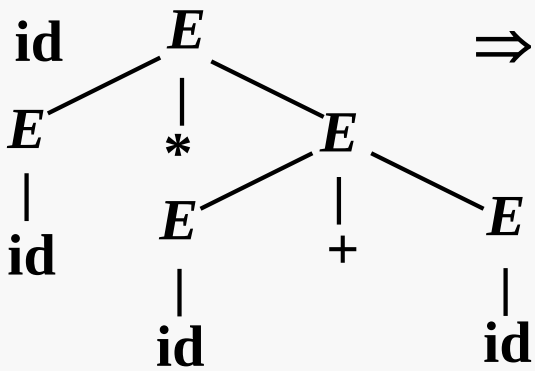


文法的二义性

- 例 $E \rightarrow E + E \mid E * E \mid (E) \mid - E \mid \mathrm{id}$
  - id * id + id有两棵不同的分析树

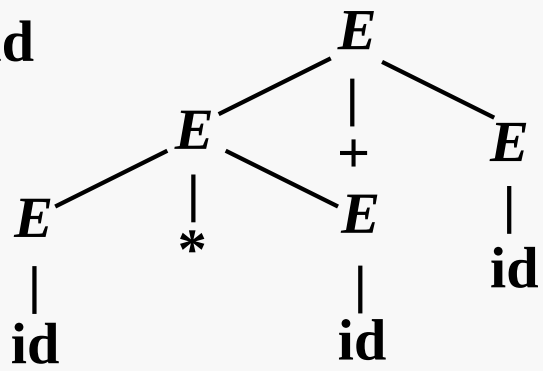
$E \Rightarrow E * E$
$\Rightarrow \mathrm{id} * E$
$\Rightarrow \mathrm{id} * E + E$
$\Rightarrow \mathrm{id} * \mathrm{id} + E$
$\Rightarrow \mathrm{id} * \mathrm{id} + \mathrm{id}$

3*4+5
->3*9
->27
Wrong!



$E \Rightarrow E + E$
$\Rightarrow E * E + E$
$\Rightarrow \mathrm{id} * E + E$
$\Rightarrow \mathrm{id} * \mathrm{id} + E$
$\Rightarrow \mathrm{id} * \mathrm{id} + \mathrm{id}$

3*4+5
->12+5
->17
Right!



消除二义性

- 表达式产生二义性的原因

$$E \rightarrow E + E \mid E * E \mid (E) \mid - E \mid \mathrm{id}$$

+，*操作都是左结合的，并且在运算中有不同的优先级，但是在这个文法中没有得到体现

消除二义性

- 表达式产生二义性的原因

- 没有一般性的方法，但，可通过**定义运算优先级和结合律**来消除二义性

消除二义性

- 用一种层次观点看待表达式
  - id * id * (id+id) + id * id + id
  - id * id * (id+id)

E → E + E
从不同的E推导得到不同的树

根据算符不同的优先级，引入新的非终结符

消除二义性

- 用一种层次观点看待表达式
  - id * id * (id+id) + id * id + id
  - id * id * (id+id)
- 新的非二义文法

$$E \rightarrow E + T \mid T$$

E → E + E
从不同的E推导得到不同的树

根据算符不同的优先级，引入新的非终结符

消除二义性

- 用一种层次观点看待表达式
  - id * id * (id+id) + id * id + id
  - id * id * (id+id)
- 新的非二义文法

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$E \rightarrow E + E$
从不同的E推导
得到不同的树

根据算符不同的
优先级，引入新
的非终结符

消除二义性

- 用一种层次观点看待表达式
  - id * id * (id+id) + id * id + id
  - id * id * (id+id)
- 新的非二义文法

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow \mathbf{id} \mid (E)$

E → E + E
从不同的E推导
得到不同的树

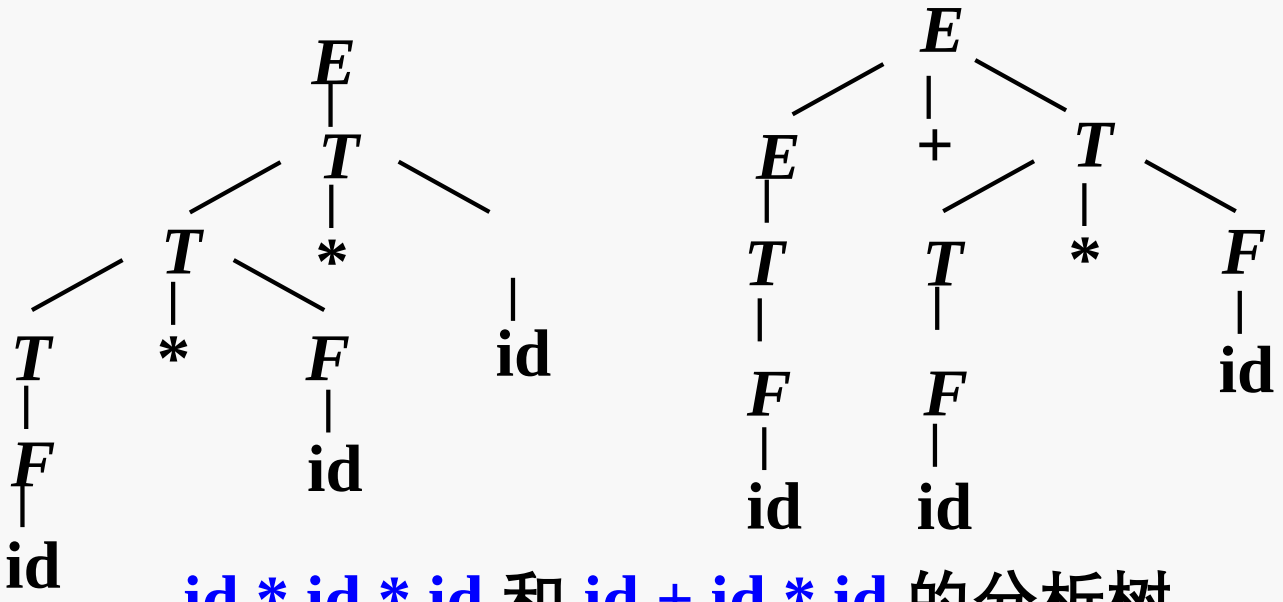
根据算符不同的
优先级，引入新
的非终结符

消除二义性

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow \mathrm{id} \mid (E)$



id * id * id 和 id + id * id 的分析树

思考题

- 悬空else文法

stmt → if *expr* then *stmt*
| if *expr* then *stmt* else *stmt*
| other

  - 判断该文法有无二义性
  - 如果存在二义性，如何消除

思考题

- 悬空else文法

  stmt → if *expr* then *stmt*
  | if *expr* then *stmt* else *stmt*
  | other

- 句型：if *expr* then if *expr* then *stmt* else *stmt*

思考题

- 悬空else文法

  stmt → if *expr* then *stmt*
  | if *expr* then *stmt* else *stmt*
  | other

- 句型：if *expr* then if *expr* then *stmt* else *stmt*

- 两个最左推导：

  stmt ⇒ if *expr* then *stmt*
  ⇒ if *expr* then if *expr* then *stmt* else *stmt*

  stmt ⇒ if *expr* then *stmt* else *stmt*
  ⇒ if *expr* then if *expr* then *stmt* else *stmt*

消除二义性

- 无二义的文法
  - 每个else与最近的尚未匹配的then匹配

stmt → *matched _stmt*
| *unmatched_stmt*

matched_stmt → if *expr* then *matched_stmt*
else *matched_stmt*
| other

unmatched_stmt → if *expr* then *stmt*
| if *expr* then *matched_stmt*
else *unmatched_stmt*

语言与文法

上下文无关文法的优点

- 文法给出了精确的，易于理解的语法说明
- 自动产生高效的分析器
- 可以给语言定义出层次结构
- 以文法为基础的语言的实现便于语言的修改

上下文无关文法的缺点

- 文法只能描述编程语言的大部分语法

分离词法分析器的理由

- **为什么要用正则表达式定义词法**
  - 词法规则非常简单，不必用上下文无关文法。
  - 对于词法记号，正则表达式描述简洁且易于理解。
  - 从正则表达式构造出的词法分析器效率高。

分离词法分析器的理由

- **为什么要用正则表达式定义词法**
  - 词法规则非常简单，不必用上下文无关文法。
  - 对于词法记号，正则表达式描述简洁且易于理解。
  - 从正则表达式构造出的词法分析器效率高。
- **分离词法分析和语法分析的好处（软件工程视角）**
  - 简化设计
  - 编译器的效率会改进
  - 编译器的可移植性加强
  - 便于编译器前端的模块划分

《编译原理和技术》

语法分析 I

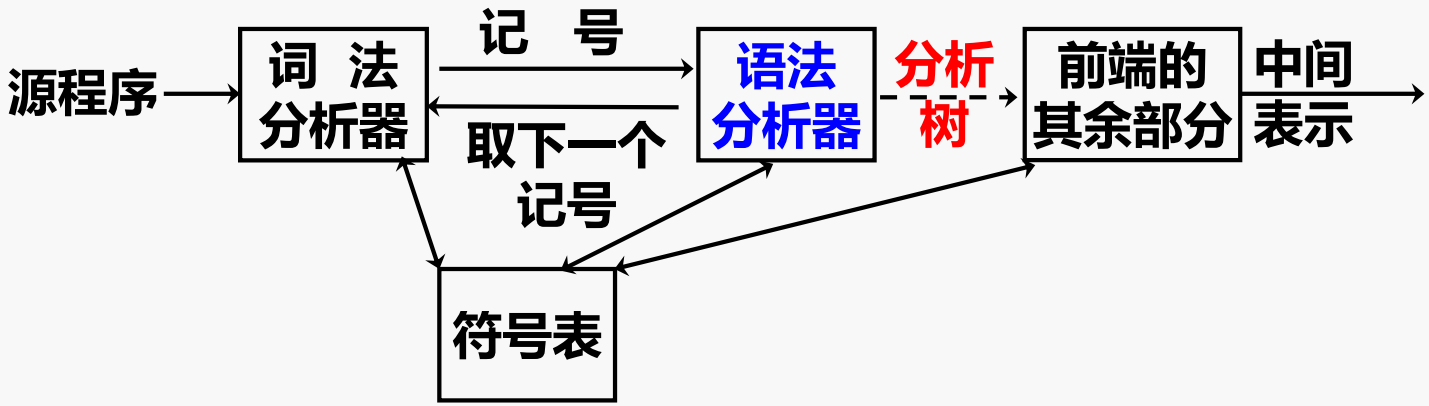
谢谢！

《编译原理和技术》

语法分析 II

中科大计算机学院
李 诚
2022-09-14

本节提纲



- 自顶向下与自底向上方法的区别
- 自顶向下分析方法
  - 递归下降分析方法
  - 消除左递归、提取左公因子

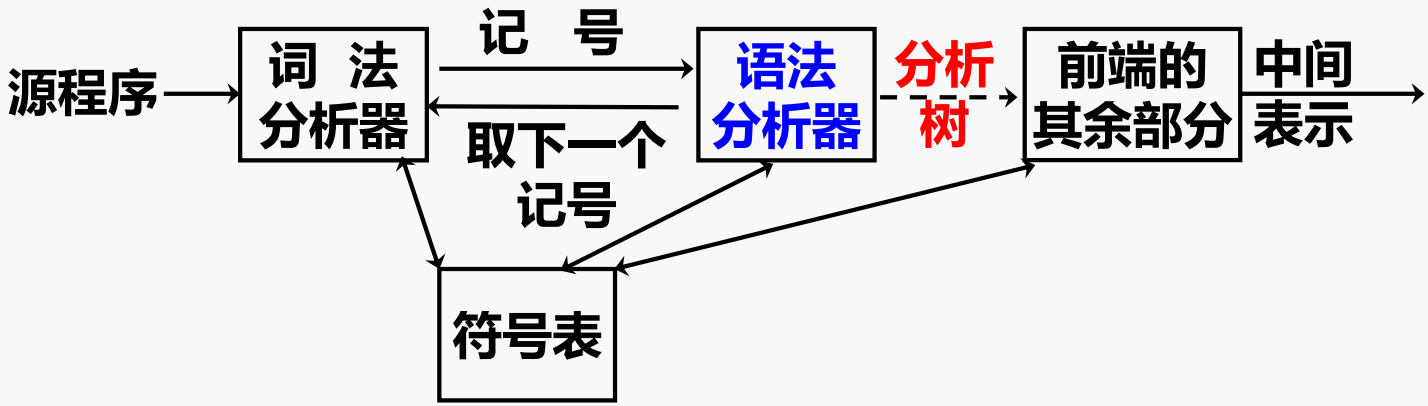
语法分析的主要方法

- **自顶向下（Top-down）**
  - 针对输入串，从文法的开始符号出发，尝试根据产生式规则推导（derive）出该输入串。
- **自底向上（Bottom-up）**
  - 针对输入串，尝试根据产生式规则归约（reduce）到文法的开始符号。

语法分析的主要方法

- **自顶向下（Top-down）**
  - 针对输入串，从文法的开始符号出发，尝试根据产生式规则推导（derive）出该输入串。
  - 分析树的构造方法
    - 从根部开始
- **自底向上（Bottom-up）**
  - 针对输入串，尝试根据产生式规则归约（reduce）到文法的开始符号。
  - 分析树的构造方法：
    - 从叶子开始

本节提纲



- 自顶向下与自底向上方法的区别
- **自顶向下分析方法**
  - 递归下降分析方法
  - 消除左递归、提取左公因子

递归下降语法分析

- **数据结构**
  - 一个输入缓冲区和向前看指针*lookahead*
- **分析过程**
  - 自左向右扫描输入串
  - 设计一个辅助过程*match()*，将*lookahead*指向的位置与产生式迭代生成的终结符进行匹配，如匹配，将*lookahead*挪到下一个位置
  - 为每一个非终结符写一个分析过程
    - 该过程可以调用其他非终结符的过程及match
    - 这些过程可能是递归的

递归下降语法分析——程序模拟推导

- 考虑以下文法：

$expr \rightarrow term$

$\quad | \ term + expr$

$\quad | \ term - expr$

$term \rightarrow num \ | \ (expr)$

- 分析过程：
  - 从左到右扫描输入串
  - 开始符号：*expr*
  - 按顺序尝试产生式

```
void expr() {
     term();
     if (lookahead == '+/-') {
         match('+/-');
         expr();
     }
     report("语法正确");
}

void term(){
    if (lookahead is num){
        match(lookahead);
    } else{ if (lookahead == '(') {
        match('(');
        expr();
        match(')');
    } else report("语法错误");}
}
```

递归下降语法分析——演示过程

- 考虑以下文法：

$expr \rightarrow term$

$\quad | \ term + expr$

$\quad | \ term - expr$

$term \rightarrow num \ | \ (expr)$

(	2	-	3	)

```
void expr() {
     term();
     if (lookahead == '+/-') {
         match('+/-');
         expr();
     }
     report("语法正确");
}

void term(){
    if (lookahead is num){
        match(lookahead);
    } else{ if (lookahead == '(') {
        match('(');
        expr();
        match(')');
    } else report("语法错误");}
}
```

递归下降语法分析——演示过程

- 考虑以下文法：

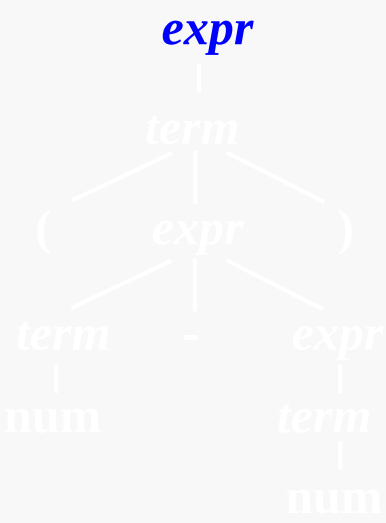
expr → *term*

| *term* + *expr*

| *term* - *expr*

term → num | (*expr*)

(	2	-	3	)




```
void expr() {
     term();
     if (lookahead == '+/-') {
         match('+/-');
         expr();
     }
     report("语法正确");
}

void term(){
    if (lookahead is num){
         match(lookahead);
    } else{ if (lookahead == '(') {
         match('(');
         expr();
         match(')');
    } else report("语法错误");}
}
```

递归下降语法分析——演示过程

- 考虑以下文法：

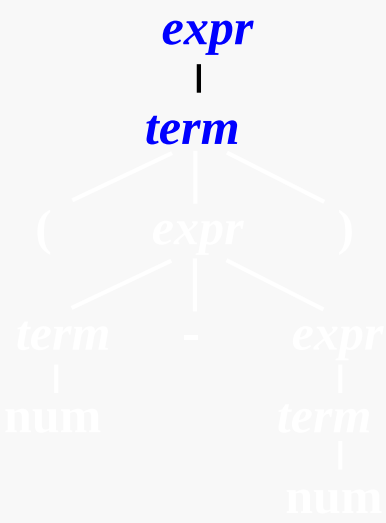
expr → *term*

| *term* + *expr*

| *term* - *expr*

term → num | (*expr*)

(	2	-	3	)




```
void expr() {
     term();
     if (lookahead == '+/-') {
         match('+/-');
         expr();
     }
     report("语法正确");
}

void term(){
    if (lookahead is num){
        match(lookahead);
    } else{ if (lookahead == '(') {
        match('(');
        expr();
        match(')');
    } else report("语法错误");}
}
```

递归下降语法分析——演示过程

- 考虑以下文法：

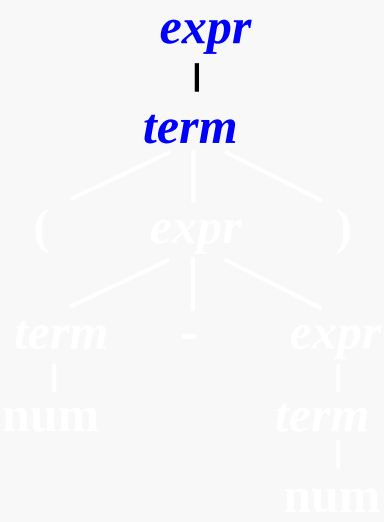
expr → *term*

| *term* + *expr*

| *term* - *expr*

term → num | (*expr*)

(	2	-	3	)




```
void expr() {
     term();
     if (lookahead == '+/-') {
         match('+/-');
         expr();
     }
     report(“语法正确");
}

void term(){
    if (lookahead is num){
         match(lookahead);
    } else{ if (lookahead == '(') {
         match('(');
         expr();
         match(')');
    } else report("语法错误");}
}
```

递归下降语法分析——演示过程

- 考虑以下文法：

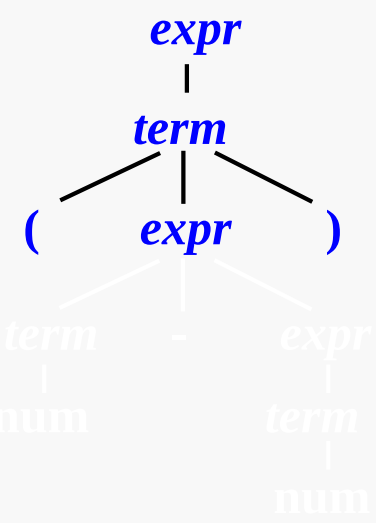
expr → *term*

| *term* + *expr*

| *term* - *expr*

term → num | (*expr*)

(	2	-	3	)




```
void expr() {
     term();
     if (lookahead == '+/-') {
         match('+/-');
         expr();
     }
     report(“语法正确");
}

void term(){
    if (lookahead is num){
        match(lookahead);
    } else{ if (lookahead == '(') {
        match('(');
        expr();
        match(')');
    } else report("语法错误");}
}
```

递归下降语法分析——演示过程

- 考虑以下文法：

expr → *term*

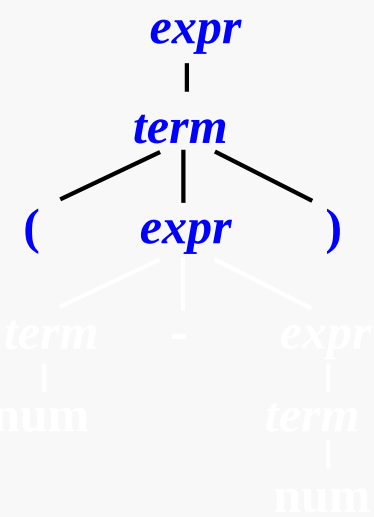
| *term* + *expr*

| *term* - *expr*

term → num | (*expr*)

(	2	-	3	)

匹配
箭头前进




```
void expr() {
     term();
     if (lookahead == '+/-') {
         match('+/-');
         expr();
     }
     report("语法正确");
}

void term(){
    if (lookahead is num){
         match(lookahead);
    } else{ if (lookahead == '(') {
         match('(');
         expr();
         match(')');
    } else report("语法错误");}
}
```

递归下降语法分析——演示过程

- 考虑以下文法：

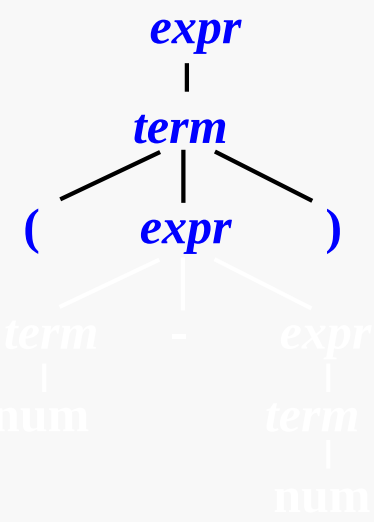
expr → *term*

| *term* + *expr*

| *term* - *expr*

term → num | (*expr*)

(	2	-	3	)




```
void expr() {
     term();
     if (lookahead == '+/-') {
         match('+/-');
         expr();
     }
     report("语法正确");
}

void term(){
    if (lookahead is num){
        match(lookahead);
    } else{ if (lookahead == '(') {
        match('(');
        expr();
        match(')');
    } else report("语法错误");}
}
```

递归下降语法分析——演示过程

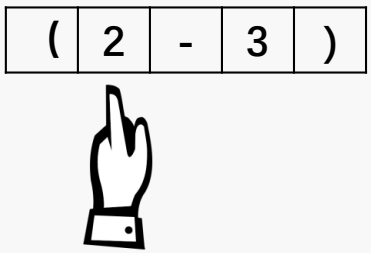
- 考虑以下文法：

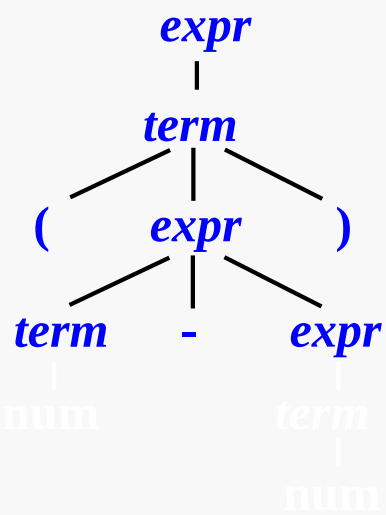
$expr \rightarrow term$

$\quad | \ term + expr$

$\quad | \ term - expr$

$term \rightarrow num \mid (expr)$






```
void expr() {
     term();
     if (lookahead == '+/-') {
         match('+/-');
         expr();
     }
     report("语法正确");
}

void term(){
    if (lookahead is num){
        match(lookahead);
    } else{ if (lookahead == '(') {
        match('(');
        expr();
        match(')');
    } else report("语法错误");}
}
```

递归下降语法分析——演示过程

- 考虑以下文法：

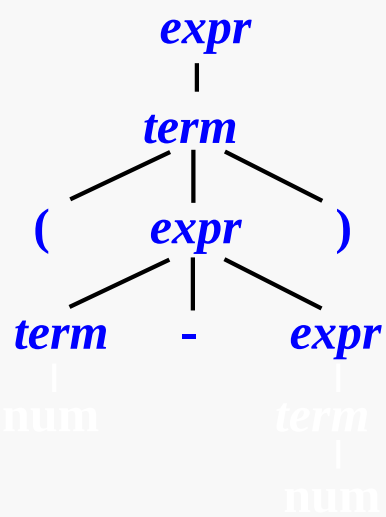
$expr \rightarrow term$

$\quad | \ term + expr$

$\quad | \ term - expr$

$term \rightarrow num \mid (expr)$

(	2	-	3	)




```
void expr() {
     term();
     if (lookahead == '+/-') {
          match('+/-');
          expr();
     }
     report("语法正确");
}

void term(){
    if (lookahead is num){
         match(lookahead);
    } else{ if (lookahead == '(') {
         match('(');
         expr();
         match(')');
    } else report("语法错误");}
}
```

递归下降语法分析——演示过程

- 考虑以下文法：

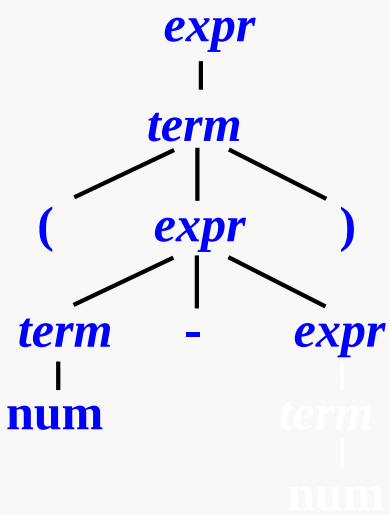
expr → *term*

| *term* + *expr*

| *term* - *expr*

term → num | (*expr*)

(	2	-	3	)




```
void expr() {
     term();
     if (lookahead == '+/-') {
         match('+/-');
         expr();
     }
     report(“语法正确");
}

void term(){
    if (lookahead is num){
        match(lookahead);
    } else{ if (lookahead == '(') {
        match('(');
        expr();
        match(')');
    } else report("语法错误");}
}
```

递归下降语法分析——演示过程

- 考虑以下文法：

$expr \rightarrow term$

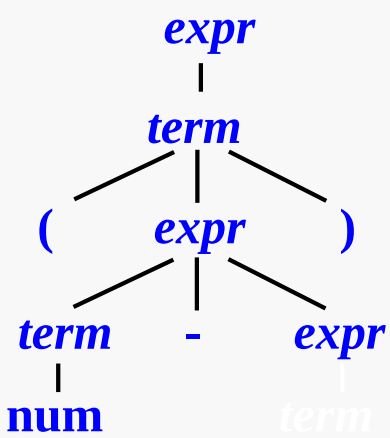
$\quad | \ term + expr$

$\quad | \ term - expr$

$term \rightarrow num \mid (expr)$

(	2	-	3	)

匹配
箭头前进




```
void expr() {
     term();
     if (lookahead == '+/-') {
         match('+/-');
         expr();
     }
     report("语法正确");
}

void term(){
    if (lookahead is num){
        match(lookahead);
    } else{ if (lookahead == '('){
        match('(');
        expr();
        match(')');
    } else report("语法错误");}
}
```

递归下降语法分析——演示过程

- 考虑以下文法：

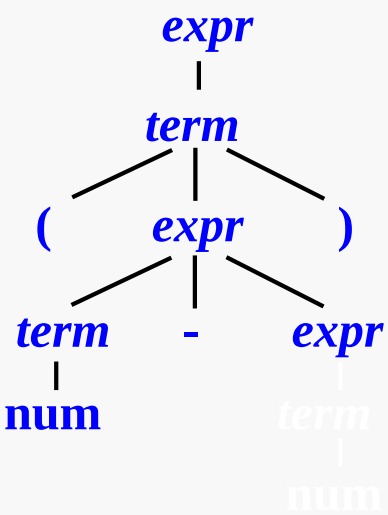
expr → *term*

| *term* + *expr*

| *term* - *expr*

term → num | (*expr*)

(	2	-	3	)




```
void expr() {
     term();
     if (lookahead == '+/-') {
          match('+/-');
          expr();
     }
     report("语法正确");
}

void term(){
    if (lookahead is num){
         match(lookahead);
    } else{ if (lookahead == '(') {
         match('(');
         expr();
         match(')');
    } else report("语法错误");}
}
```

递归下降语法分析——演示过程

- 考虑以下文法：

$expr \rightarrow term$

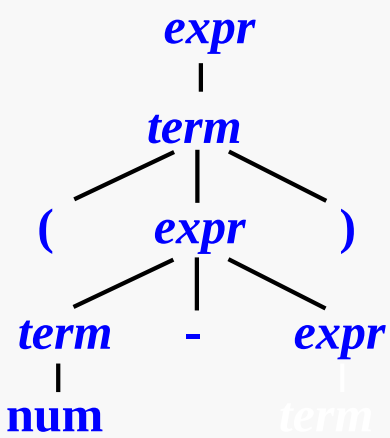
$\quad | \ term + expr$

$\quad | \ term - expr$

$term \rightarrow num \mid (expr)$

(	2	-	3	)

匹配
箭头前进




```
void expr() {
     term();
     if (lookahead == '+/-') {
         match('+/-');
         expr();
     }
     report("语法正确");
}

void term(){
    if (lookahead is num){
        match(lookahead);
    } else{ if (lookahead == '(') {
        match('(');
        expr();
        match(')');
    } else report("语法错误");}
}
```

递归下降语法分析——演示过程

- 考虑以下文法：

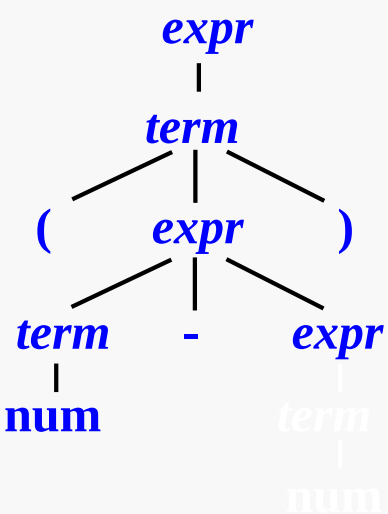
expr → *term*

| *term* + *expr*

| *term* - *expr*

term → num | (*expr*)

(	2	-	3	)




```
void expr() {
     term();
     if (lookahead == '+/-') {
          match('+/-');
          expr();
     }
     report("语法正确");
}

void term(){
    if (lookahead is num){
         match(lookahead);
    } else{ if (lookahead == '(') {
         match('(');
         expr();
         match(')');
    } else report("语法错误");}
}
```

递归下降语法分析——演示过程

- 考虑以下文法：

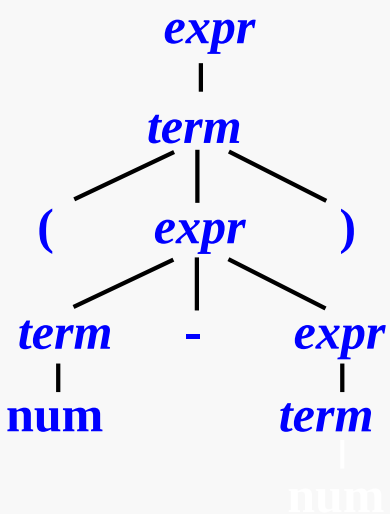
expr → *term*

| *term* + *expr*

| *term* - *expr*

term → num | (*expr*)

(	2	-	3	)




```
void expr() {
     term();
     if (lookahead == +/-) {
          match('+/-');
          expr();
     }
     report("语法正确");
}

void term(){
    if (lookahead is num){
         match(lookahead);
    } else{ if (lookahead == '(') {
         match('(');
         expr();
         match(')');
    } else report("语法错误");}
}
```

递归下降语法分析——演示过程

- 考虑以下文法：

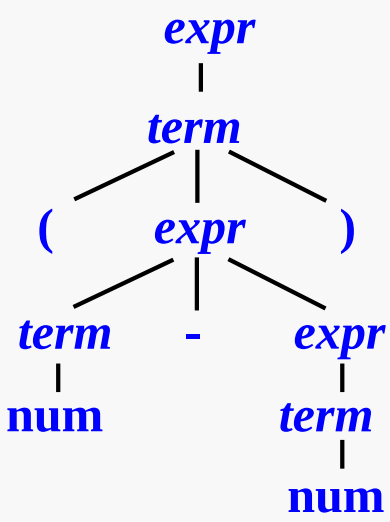
expr → *term*

| *term* + *expr*

| *term* - *expr*

term → num | (*expr*)

(	2	-	3	)




```
void expr() {
     term();
     if (lookahead == '+/-') {
         match('+/-');
         expr();
     }
     report(“语法正确");
}

void term(){
    if (lookahead is num){
        match(lookahead);
    } else{ if (lookahead == '(') {
        match('(');
        expr();
        match(')');
    } else report("语法错误");}
}
```

递归下降语法分析——演示过程

- 考虑以下文法：

$expr \rightarrow term$

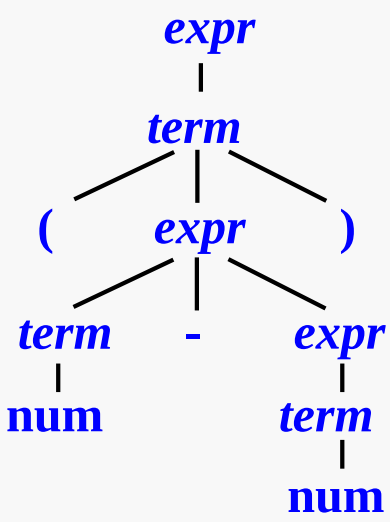
$\quad | \ term + expr$

$\quad | \ term - expr$

$term \rightarrow num \mid (expr)$

(	2	-	3	)

匹配
箭头前进




```
void expr() {
     term();
     if (lookahead == '+/-') {
         match('+/-');
         expr();
     }
     report("语法正确");
}

void term(){
    if (lookahead is num){
        match(lookahead);
    } else{ if (lookahead == '(') {
        match('(');
        expr();
        match(')');
    } else report("语法错误");}
}
```

递归下降语法分析——演示过程

- 考虑以下文法：

$expr \rightarrow term$

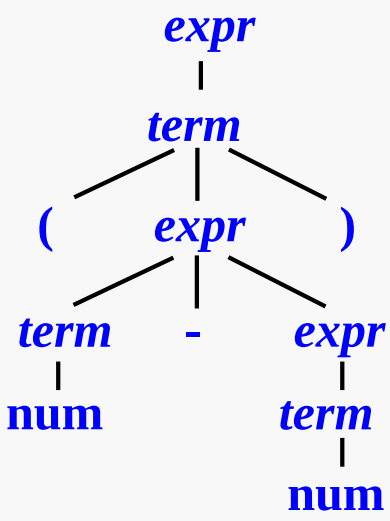
$\quad | \ term + expr$

$\quad | \ term - expr$

$term \rightarrow num \mid (expr)$

(	2	-	3	)

匹配
箭头前进




```
void expr() {
     term();
     if (lookahead == '+/-') {
          match('+/-');
          expr();
     }
     report("语法正确");
}

void term(){
    if (lookahead is num){
         match(lookahead);
    } else{ if (lookahead == '(') {
         match('(');
         expr();
         match(')');
    } else report("语法错误");}
}
```

递归下降语法分析——演示过程

- 考虑以下文法：

$expr \rightarrow term$

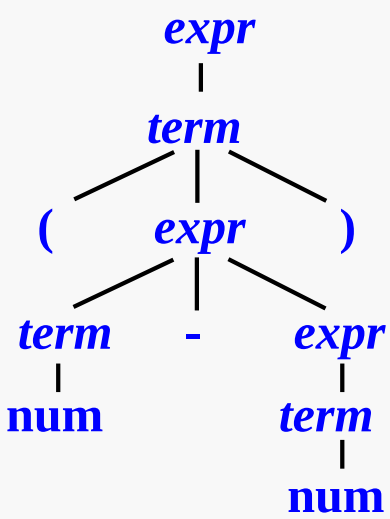
$\quad | \ term + expr$

$\quad | \ term - expr$

$term \rightarrow num \mid (expr)$

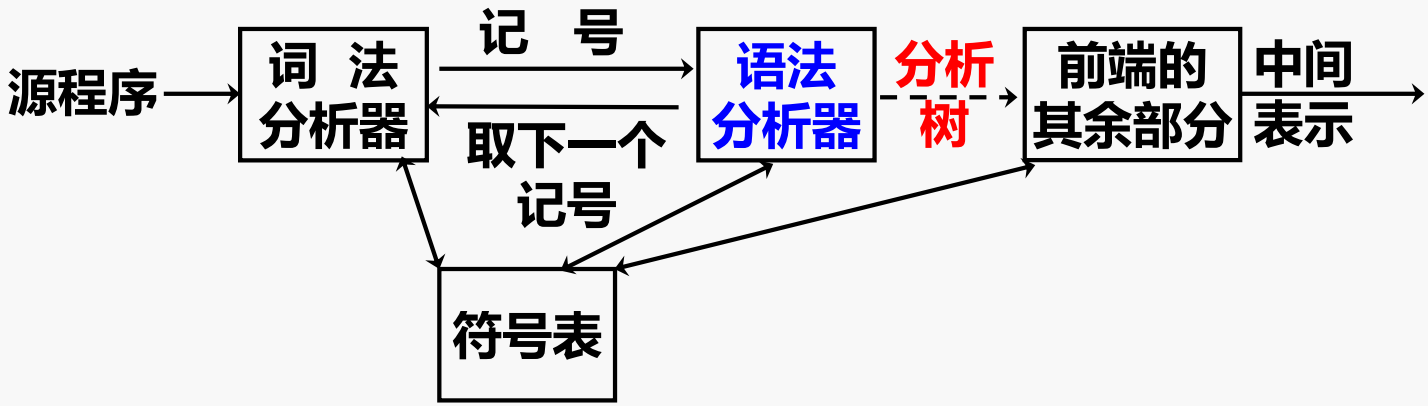
(	2	-	3	)

分析完毕
接受该串




```
void expr() {
     term();
     if (lookahead == '+/-') {
         match('+/-');
         expr();
     }
     report("语法正确");
}

void term(){
    if (lookahead is num){
        match(lookahead);
    } else{ if (lookahead == '(') {
        match('(');
        expr();
        match(')');
    } else report("语法错误");}
}
```

本节提纲



- 自顶向下与自底向上方法的区别
- 自顶向下分析方法
  - 递归下降预测分析方法
  - 消除左递归、提取左公因子

递归下降的问题1

- 可能进入无限循环
- 考虑以下文法

$$S \rightarrow Sa \mid b$$

- 该文法是左递归的(left-recursive)

递归下降的问题1

- 可能进入无限循环
- 考虑以下文法

$$S \rightarrow Sa \mid b$$

- 该文法是左递归的(left-recursive)
- 自顶向下分析方法无法处理左递归
  - Why?

递归下降的问题1

- 可能进入无限循环
- 考虑以下文法

  $S \rightarrow Sa \mid b$

- 该文法是左递归的(left-recursive)
- 自顶向下分析方法无法处理左递归
  - 考虑输入文法符号串为baaaaa
  - 最左推导如下：
    - $S \Rightarrow Sa \Rightarrow Saa \Rightarrow Saaa \Rightarrow Saaaa \ldots.$
    - 输入缓冲区lookahead指针纹丝未动

消除左递归

- **直接左递归**

  $A \to A\alpha \mid \beta$，**其中α，β不以A开头**

  - 串的特点 $\beta\alpha \ldots \alpha$ $(A \Rightarrow^{+} A\alpha)$

- **消除直接左递归**

  $A \to \beta A'$

  $A' \to \alpha A' \mid \varepsilon$

消除左递归

- **直接左递归**

  $A \to A\alpha \mid \beta$，其中α，β不以A开头

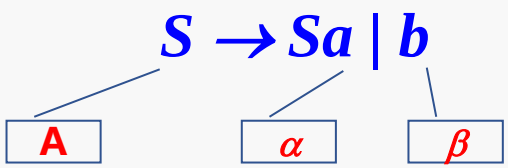
  - 串的特点 $\beta\alpha \ldots \alpha$ $(A \Rightarrow^{+} A\alpha)$

- **消除直接左递归**

  $A \to \beta A'$

  $A' \to \alpha A' \mid \varepsilon$

- **考虑之前的文法**



消除左递归

- **直接左递归**

  $A \to A\alpha \mid \beta$，其中α，β不以A开头

  - 串的特点 $\beta\alpha \ldots \alpha$ $(A \Rightarrow^{+} A\alpha)$

- **消除直接左递归**

  $A \to \beta A'$

  $A' \to \alpha A' \mid \varepsilon$

- **考虑之前的文法**

$$S \to Sa \mid b$$

A　α　β

$S \to bS'$

$S' \to aS' \mid \varepsilon$

baaaaa推导：

$S \Rightarrow bS' \Rightarrow baS' \Rightarrow baaS' \Rightarrow baaaS' \Rightarrow baaaaS'$

输入缓冲区指针不停地移动

消除左递归

- **直接左递归**

  $A \to A\alpha \mid \beta$，其中α，β不以A开头

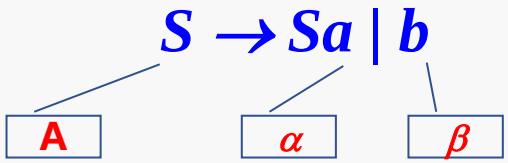
  - 串的特点 $\beta\alpha \ldots \alpha$ $(A \Rightarrow^{+} A\alpha)$

- **消除直接左递归**

  $A \to \beta A'$

  $A' \to \alpha A' \mid \varepsilon$

- **考虑之前的文法**



$S \to bS'$

$S' \to aS' \mid \varepsilon$

baaaaa推导：

$S \Rightarrow bS' \Rightarrow baS' \Rightarrow baaS' \Rightarrow baaaS' \Rightarrow baaaaS' \Rightarrow baaaaaS'$

输入缓冲区指针不停地移动

消除左递归

- 例 算术表达文法

$$E \to E + T \mid T \qquad (T + T \ldots + T)$$

$$T \to T * F \mid F \qquad (F * F \ldots * F)$$

$$F \to (E) \mid \text{id}$$

消除左递归

- 例 算术表达文法

$$E \to E + T \mid T \qquad (T + T \ldots + T)$$

$$T \to T * F \mid F \qquad (F * F \ldots * F)$$

$$F \to (E) \mid \text{id}$$

消除左递归

- 例 算术表达文法

$$E \to E + T \mid T \qquad (\ T + T \ldots + T\ )$$
$$T \to T * F \mid F \qquad (\ F * F \ldots * F\ )$$
$$F \to (\ E\ ) \mid \mathrm{id}$$

- 消除左递归后文法

$$E \to TE'$$
$$E' \to +\ TE' \mid \varepsilon$$
$$T \to FT'$$
$$T' \to *\ F\ T' \mid \varepsilon$$
$$F \to (\ E\ ) \mid \mathrm{id}$$

注明：红色部分代表了α，蓝色部分代表了β

消除左递归的推广

- 处理任意数量的A产生式

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

其中β_i都不以A开头

改为：

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$$

消除间接左递归

- 非直接左递归

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Sd \mid \varepsilon$$

消除间接左递归

- 非直接左递归

$$S \to Aa \mid b$$

$$A \to Sd \mid \varepsilon$$

- 先变换成直接左递归

$$S \to Aa \mid b$$

$$A \to Aad \mid bd \mid \varepsilon$$

- 再消除左递归

$$S \to Aa \mid b$$

$$A \to bdA' \mid A'$$

$$A' \to adA' \mid \varepsilon$$

递归下降的问题2

- **有左公因子的(left -factored)文法：**
  - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

- **提左公因子(left factoring)**
  - 推后选择产生式的时机，以便获取更多信息

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ **等价于**

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$

提左公因子(left factoring)

- 例 悬空*else*的文法

stmt → if *expr* then *stmt* else *stmt*

| if *expr* then *stmt*

| other

提左因子

stmt → if *expr* then *stmt optional_else_part*

| other

optional_else_part → else *stmt*

| ε

算法仍然二义！！！

递归下降的问题3

- **复杂的回溯→代价太高**
  - 非终结符有可能有多个产生式
  - 由于信息缺失，无法准确预测选择哪一个
  - 考虑到往往需要对多个非终结符进行推导展开，因此尝试的路径可能呈指数级爆炸
- **其分析过程类似于NFA**
- **问题：是否可以构造一个类似于DFA的分析方法?**

《编译原理和技术》

语法分析 II

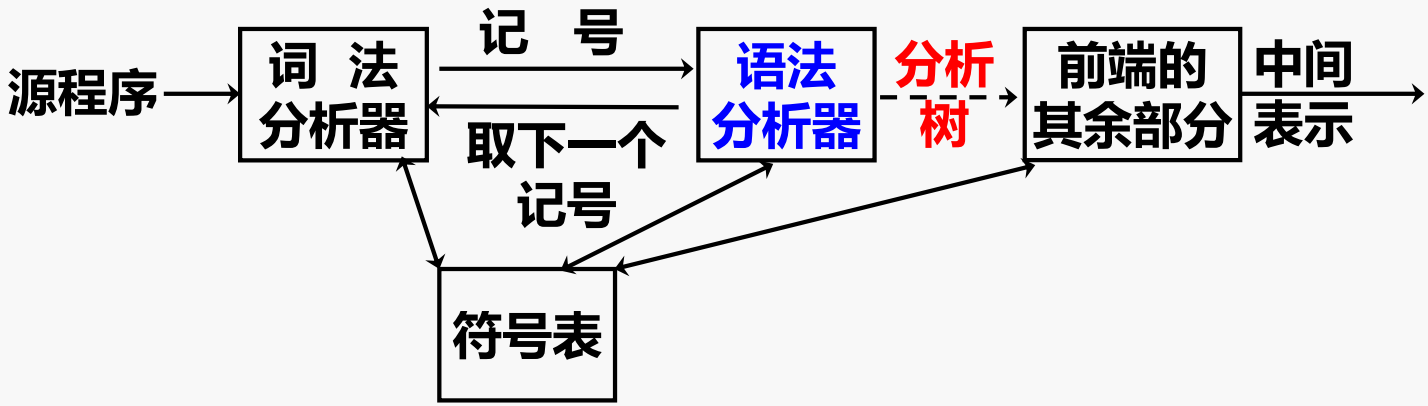
谢谢！

《编译原理和技术》

语法分析 III

中科大计算机学院
李 诚
2022-09-14/19

本节提纲



- 自顶向下分析方法
  - LL(1)文法
  - 非递归预测分析方法

预测分析法（Predictive parsing）

- **与递归下降法相似，但**
  - 不会对若干产生式进行尝试
  - 没有回溯
  - 通过向前看一些记号来预测需要用到的产生式

- **此方法接受LL(k)文法**
  - L-means “left-to-right” scan of input
  - L-means “leftmost derivation”
  - k-means “predict based on k tokens of lookahead”
  - In practice, LL(1) is used

LL(1)文法

- **对文法加什么样的限制可以保证没有回溯?**

- **先定义两个和文法有关的函数**
  - FIRST(α) = $\{a \mid \alpha \Rightarrow^* a\ldots, a \in V_T\}$

    意义：可从α推导得到的串的首符号的集合

  - FOLLOW(A) = $\{a \mid S \Rightarrow^* \ldots Aa\ldots,\ a \in V_T\}$

    意义：可能在推导过程中紧跟在A右边的终结符号的集合

LL(1)文法：FIRST(X)

- 计算FIRST(X), $X \in V_T \cup V_N$
  - $X \in V_T$, FIRST(X) = {X}
  - $X \in V_N$且 $X \to \varepsilon$

    则将 ε加入到FIRST(X)
  - $X \in V_N$ 且 $X \to Y_1\ Y_2\ \ldots Y_k$
    - 如果 $a \in$ FIRST(Y_i)且ε在FIRST(Y_1), …, FIRST(Y_{i-1})中， 则将 a加入到FIRST(X)
    - 如果 ε 在FIRST(Y_1), …, FIRST(Y_k)中， 则将ε 加入到FIRST(X)

FIRST集合只包括终结符和ε

表达式文法：无左递归的

- 例 $E \to TE'$
  $E' \to +TE' \mid \varepsilon$
  $T \to FT'$
  $T' \to *FT' \mid \varepsilon$
  $F \to (E) \mid \text{id}$

- $X \in V_T$, FIRST(X) = {X}
- $X \in V_N$且 $X \to \varepsilon$, $\varepsilon \in$ FIRST(X)
- $X \in V_N$且 $X \to Y_1 Y_2 \ldots Y_k$
  - 如果 $a \in$ FIRST(Y_i)且ε在FIRST(Y_1), …, FIRST(Y_{i-1})中，则$a \in$ FIRST(X)
  - 如果 ε 在FIRST(Y_1), …, FIRST(Y_k)中，则$\varepsilon \in$ FIRST(X)

FIRST(F) = { (, id }
FIRST(E') = {+, ε}
FRIST(T') = {*, ε}
FOLLOW(E) = FOLLOW(E') = {), $}
FOLLOW(T) = FOLLOW (T') = {+,), $}
FOLLOW(F) = {+, *,), $}

表达式文法：无左递归的

- 例 $E \to TE'$
  $E' \to +TE' \mid \varepsilon$
  $T \to FT'$
  $T' \to *FT' \mid \varepsilon$
  $F \to (E) \mid \text{id}$

- $X \in V_T$, FIRST(X) = {X}
- $X \in V_N$且 $X \to \varepsilon$, $\varepsilon \in$ FIRST(X)
- $X \in V_N$且 $X \to Y_1 Y_2 \ldots Y_k$
  - 如果 $a \in$ FIRST(Y_i)且ε在FIRST(Y_1), …, FIRST(Y_{i-1})中，则$a \in$ FIRST(X)
  - 如果 ε 在FIRST(Y_1), …, FIRST(Y_k)中，则$\varepsilon \in$ FIRST(X)

FIRST(F) = { (, id } = FIRST(T) = FIRST(E)

FIRST(E') = {+, ε}

FRIST(T') = {*, ε}

FOLLOW(E) = FOLLOW(E') = {), $}

FOLLOW(T) = FOLLOW (T') = {+,), $}

FOLLOW(F) = {+, *,), $}

表达式文法：无左递归的

- 例 $E \to TE'$
  $E' \to +TE' \mid \varepsilon$
  $T \to FT'$
  $T' \to *FT' \mid \varepsilon$
  $F \to (E) \mid \text{id}$

- $X \in V_T$, FIRST(X) = {X}
- $X \in V_N$且 $X \to \varepsilon$, $\varepsilon \in$ FIRST(X)
- $X \in V_N$且 $X \to Y_1 Y_2 \ldots Y_k$
  - 如果 $a \in$ FIRST(Y_i)且ε在FIRST(Y_1), …, FIRST(Y_{i-1})中，则$a \in$ FIRST(X)
  - 如果 ε 在FIRST(Y_1), …, FIRST(Y_k)中，则$\varepsilon \in$ FIRST(X)

FIRST(F) = { (, id } = FIRST(T) = FIRST(E)
FIRST(E') = {+, ε}
FRIST(T') = {*, ε}
FOLLOW(E) = FOLLOW(E') = {), $}
FOLLOW(T) = FOLLOW (T') = {+,), $}
FOLLOW(F) = {+, *,), $}

表达式文法：无左递归的

- 例 $E \to TE'$
  $E' \to +TE' \mid \varepsilon$
  $T \to FT'$
  $T' \to *FT' \mid \varepsilon$
  $F \to (E) \mid id$

- $X \in V_T$, FIRST(X) = {X}
- $X \in V_N$且 $X \to \varepsilon$, $\varepsilon \in$ FIRST(X)
- $X \in V_N$且 $X \to Y_1 Y_2 \ldots Y_k$
  - 如果 $a \in$ FIRST(Y_i)且ε在FIRST(Y_1), …, FIRST(Y_{i-1})中，则$a \in$ FIRST(X)
  - 如果 ε 在FIRST(Y_1), …, FIRST(Y_k)中，则$\varepsilon \in$ FIRST(X)

FIRST(F) = { (, id } = FIRST(T) = FIRST(E)
FIRST(E') = {+, ε}
FRIST(T') = {*, ε}
FOLLOW(E) = FOLLOW(E') = {), $}
FOLLOW(T) = FOLLOW (T') = {+,), $}
FOLLOW(F) = {+, *,), $}

表达式文法：无左递归的

- 例 $E \to TE'$
  $E' \to +TE' \mid \varepsilon$
  $T \to FT'$
  $T' \to *FT' \mid \varepsilon$
  $F \to (E) \mid id$

- $X \in V_T$, FIRST(X) = {X}
- $X \in V_N$且 $X \to \varepsilon$, $\varepsilon \in$ FIRST(X)
- $X \in V_N$且 $X \to Y_1 Y_2 \ldots Y_k$
  - 如果 $a \in$ FIRST(Y_i)且ε在FIRST(Y_1), …, FIRST(Y_{i-1})中， 则$a \in$ FIRST(X)
  - 如果 ε 在FIRST(Y_1), …, FIRST(Y_k)中，则$\varepsilon \in$ FIRST(X)

FIRST(E) = FIRST(T) = FIRST(F) = { (, id }
FIRST(E') = {+, ε}
FRIST(T') = {*, ε}
FOLLOW(E) = FOLLOW(E') = {), $}
FOLLOW(T) = FOLLOW (T') = {+,), $}
FOLLOW(F) = {+, *,), $}

LL(1)文法：FOLLOW(A)

- 计算FOLLOW(A), $A \in V_N$
  - $加入到FOLLOW($A$)，当A是开始符号，$是输入串的结束符号
  - 如果$A \to \alpha B\beta$，则FIRST(β)-{ε}加入到FOLLOW(B)
  - 如果$A \to \alpha B$或$A \to \alpha B\beta$且$\varepsilon \in$ FIRST(β)，则FOLLOW(A)加入到FOLLOW(B)

表达式文法：无左递归的

- 例 $E \to TE'$
  $E' \to +TE' \mid \varepsilon$
  $T \to FT'$
  $T' \to *FT' \mid \varepsilon$
  $F \to (E) \mid \text{id}$

- 当A是开始符号，$\$ \in$ FOLLOW(A)
- $A \to \alpha B\beta$，FIRST(β)-{ε} $\subseteq$ FOLLOW(B)
- $A \to \alpha B$ 或 $A \to \alpha B\beta$ 且 $\varepsilon \in$ FIRST(β)，FOLLOW(A) $\subseteq$ FOLLOW(B)

FIRST(E) = FIRST(T) = FIRST(F) = { (, id }
FIRST(E') = {+, ε}
FRIST(T') = {*, ε}
FOLLOW(E) = {), $} = FOLLOW($E'$)
FOLLOW(T) = {+,), $}
FOLLOW(F) = {+, *,), $}

表达式文法：无左递归的

- 例 $E \to TE'$
  $E' \to +TE' \mid \varepsilon$
  $T \to FT'$
  $T' \to *FT' \mid \varepsilon$
  $F \to (E) \mid \text{id}$

- 当A是开始符号，$ ∈ FOLLOW(A)
- $A \to \alpha B\beta$，FIRST(β)-{ε} ⊆ FOLLOW(B)
- $A \to \alpha B$ 或 $A \to \alpha B\beta$ 且 ε ∈ FIRST(β)，FOLLOW(A) ⊆ FOLLOW(B)

FIRST(E) = FIRST(T) = FIRST(F) = { (, id }
FIRST(E') = {+, ε}
FRIST(T') = {*, ε}
FOLLOW(E) = {), $} = FOLLOW($E'$)
FOLLOW(T) = {+,), $}
FOLLOW(F) = {+, *,), $}

表达式文法：无左递归的

- 例 $E \to TE'$
  $E' \to +TE' \mid \varepsilon$
  $T \to FT'$
  $T' \to *FT' \mid \varepsilon$
  $F \to (E) \mid \text{id}$

- 当A是开始符号，$ ∈ FOLLOW(A)
- $A \to \alpha B\beta$，FIRST(β)-{ε} ⊆ FOLLOW(B)
- $A \to \alpha B$ 或 $A \to \alpha B\beta$ 且 ε ∈ FIRST(β)，FOLLOW(A) ⊆ FOLLOW(B)

FIRST(E) = FIRST(T) = FIRST(F) = { (, id }
FIRST(E') = {+, ε}
FRIST(T') = {*, ε}
FOLLOW(E) = {), $} = FOLLOW($E'$)
FOLLOW(T) = {+,), $}
FOLLOW(F) = {+, *,), $}

表达式文法：无左递归的

- 例 $E \to TE'$
  $E' \to +TE' \mid \varepsilon$
  $T \to FT'$
  $T' \to *FT' \mid \varepsilon$
  $F \to (E) \mid \text{id}$

- 当A是开始符号，$ ∈ FOLLOW(A)
- $A \to \alpha B\beta$，FIRST(β)-{ε} ⊆ FOLLOW(B)
- $A \to \alpha B$ 或 $A \to \alpha B\beta$ 且 ε ∈ FIRST(β)，FOLLOW(A) ⊆ FOLLOW(B)

FIRST(E) = FIRST(T) = FIRST(F) = { (, id }
FIRST(E') = {+, ε}
FRIST(T') = {*, ε}
FOLLOW(E) = {), $} = FOLLOW($E'$)
FOLLOW(T) = {+,), $} = FOLLOW ($T'$)
FOLLOW(F) = {+, *,), $}

表达式文法：无左递归的

- 例 $E \to TE'$
  $E' \to +TE' \mid \varepsilon$
  $T \to FT'$
  $T' \to *FT' \mid \varepsilon$
  $F \to (E) \mid id$

- 当A是开始符号，$ ∈ FOLLOW(A)
- $A \to \alpha B \beta$，FIRST(β)-{ε} ⊆ FOLLOW(B)
- $A \to \alpha B$ 或 $A \to \alpha B \beta$ 且 ε ∈ FIRST(β)，FOLLOW(A) ⊆ FOLLOW(B)

FIRST(E) = FIRST(T) = FIRST(F) = { (, id }
FIRST(E') = {+, ε}
FRIST(T') = {*, ε}
FOLLOW(E) = {), $} = FOLLOW($E'$)
FOLLOW(T) = {+,), $} = FOLLOW ($T'$)
FOLLOW(F) = {*,+,), $}

LL(1)文法

LL(1)文法的定义

任何两个产生式$A \to \alpha \mid \beta$都满足下列条件：

- $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \varnothing$
- 若$\beta \Rightarrow^* \varepsilon$，那么$\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \varnothing$

LL(1)文法

LL(1)文法的定义

任何两个产生式$A \to \alpha \mid \beta$都满足下列条件：

- $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \varnothing$
- 若$\beta \Rightarrow^* \varepsilon$，那么$\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \varnothing$

该条件存在的必要性

- 容易理解
- 每次通过输入词法单元记号和FIRST集合匹配产生式的时候，需要有唯一的选择

LL(1)文法

LL(1)文法的定义

任何两个产生式$A \to \alpha \mid \beta$都满足下列条件：

- $\mathrm{FIRST}(\alpha) \cap \mathrm{FIRST}(\beta) = \varnothing$
- 若$\beta \Rightarrow^* \varepsilon$，那么$\mathrm{FIRST}(\alpha) \cap \mathrm{FOLLOW}(A) = \varnothing$

假设$\mathrm{FIRST}(\alpha) \cap \mathrm{FOLLOW}(A) = \{a\}$

$a \in \mathrm{FIRST}(\alpha)$: $A \Rightarrow^* a\alpha'$

$a \in \mathrm{FOLLOW}(A)$: $B \Rightarrow^* \ldots A\, a \ldots$

LL(1)文法

LL(1)文法的定义

任何两个产生式$A \to \alpha \mid \beta$都满足下列条件：

- $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \varnothing$
- 若$\beta \Rightarrow^* \varepsilon$，那么$\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \varnothing$

假设$\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \{a\}$

$a \in \text{FIRST}(\alpha)$: $A \Rightarrow^* a\alpha'$

$a \in \text{FOLLOW}(A)$: $B \Rightarrow^* \ldots A\, a \ldots$

由于$\beta \Rightarrow^* \varepsilon$，所以遇到$a$时，无法判断用哪一个产生式

- 可以用$A \to \alpha$来对A进行展开
- 亦可以用$A \to \beta$和$\beta \Rightarrow^* \varepsilon$最后把$A$消掉

LL(1)文法

LL(1)文法的定义

任何两个产生式$A \to \alpha \mid \beta$都满足下列条件：

- $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \varnothing$
- 若$\beta \Rightarrow^* \varepsilon$，那么$\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \varnothing$

例如，考虑下面文法

面临a...时，第2步推导不知用哪个产生式

$S \to A\ B$

$A \to a\ b \mid \varepsilon$　　$a \in \text{FIRST}(ab) \cap \text{FOLLOW}(A)$

$B \to a\ C$

$C \to \ldots$

LL(1)文法

LL(1)文法的定义

任何两个产生式$A \to \alpha \mid \beta$都满足下列条件：

- $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \varnothing$
- 若$\beta \Rightarrow^* \varepsilon$，那么$\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \varnothing$

LL(1)文法有一些明显的性质

- 没有公共左因子
- 不是二义的
- 不含左递归

表达式文法：无左递归的

- 例 $E \to TE'$

  $E' \to +TE' \mid \varepsilon$

  $T \to FT'$

  $T' \to *FT' \mid \varepsilon$

  $F \to (E) \mid \text{id}$

FIRST(E) = FIRST(T) = FIRST(F) = { (, id }

FIRST(E') = {+, ε}

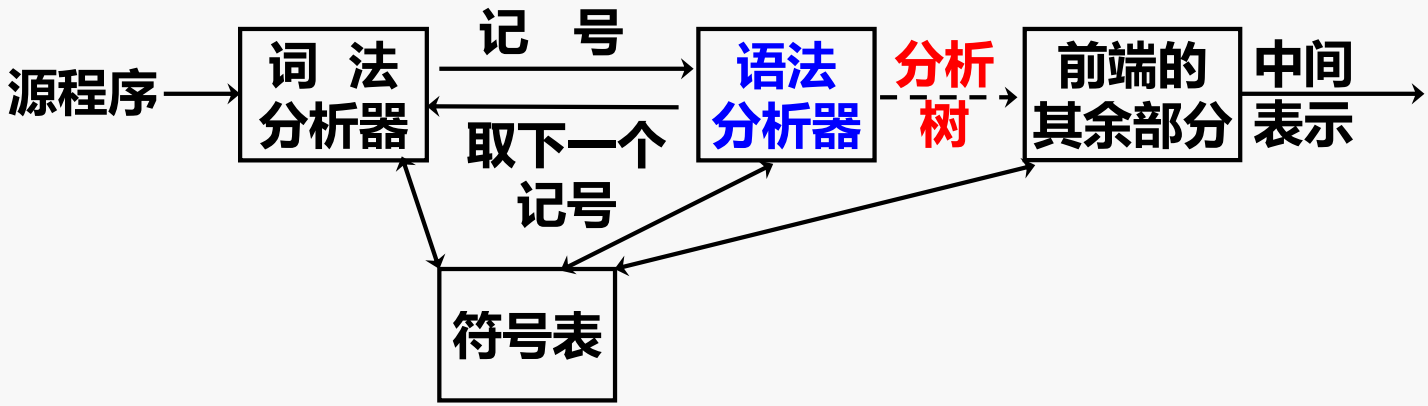
FRIST(T') = {*, ε}

FOLLOW(E) = FOLLOW(E') = {), $}

FOLLOW(T) = FOLLOW (T') = {+,), $}

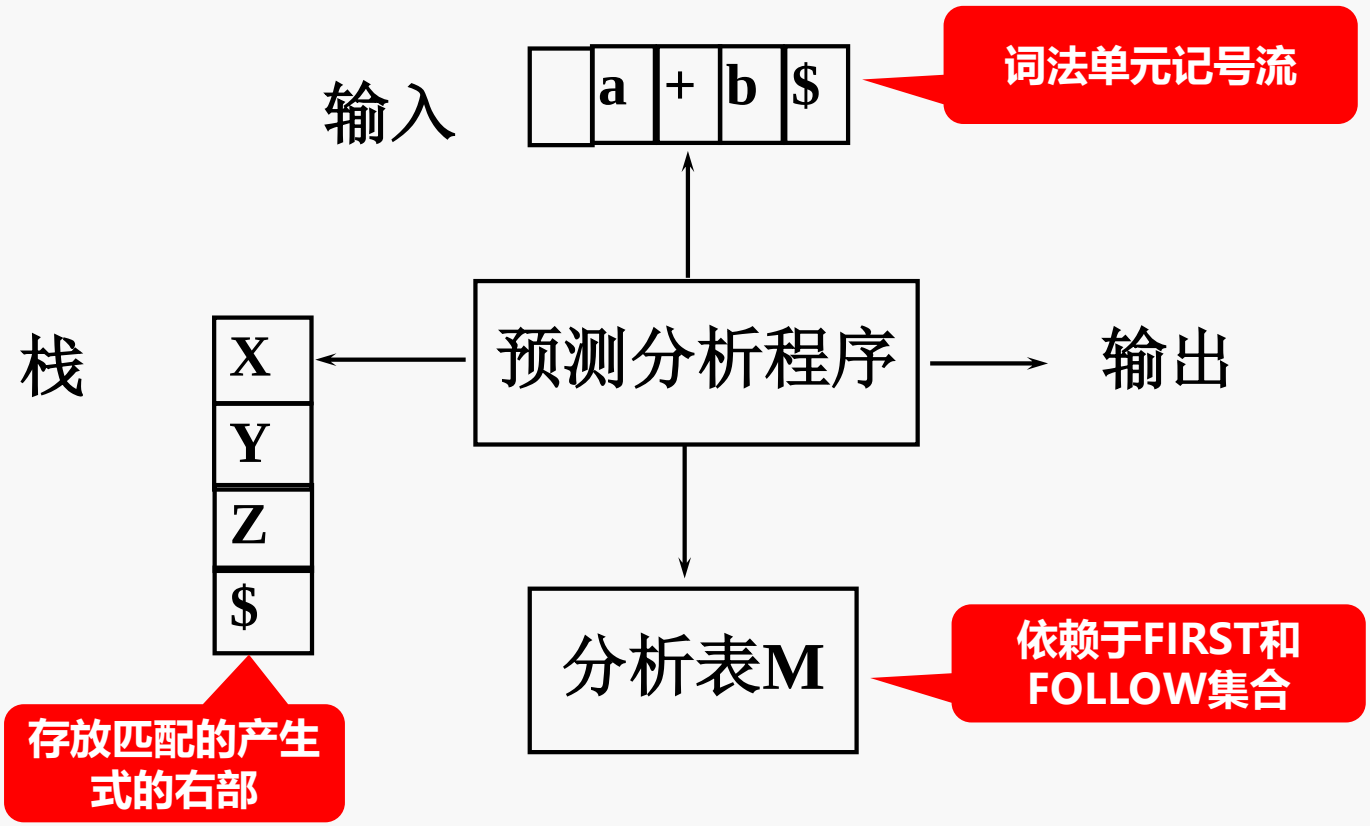
FOLLOW(F) = {+, *,), $}

本节提纲



- 自顶向下分析方法
  - LL(1)文法
  - 非递归预测分析方法

非递归的预测分析


输入
a
+
b
$
词法单元记号流
栈
X
Y
Z
$
预测分析程序
输出
分析表M
依赖于FIRST和FOLLOW集合
存放匹配的产生式的右部

预测分析表M的构造

- **对文法的每个产生式$A \to \alpha$，执行(1)和(2)**
  - （1）对FIRST(α)的每个终结符a，把$A \to \alpha$加入$M[A, a]$
  - （2）如果ε在FIRST(α)中，对FOLLOW(A)的每个终结符b（包括\$），把$A \to \alpha$加入$M[A, b]$

***M*中其它没有定义的条目都是error**

预测分析表M的构造

- 行：非终结符；列：终结符 或$ ；单元：产生式

非终结符	输 入 符 号					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow$id			$F \rightarrow (E)$		

预测分析举例

预测分析器接受输入id * id + id的前一部分动作

栈	输　入	输　出
$*E*	id * id + id$	

预测分析举例

预测分析器接受输入id * id + id的前一部分动作

栈	输　入	输　出
$$E$	id * id + id$	
$$E'T$	id * id + id$	$E \rightarrow TE'$

预测分析举例

预测分析器接受输入id * id + id的前一部分动作

栈	输　入	输　出
$$E$	id * id + id$	
$$E\,'T$	id * id + id$	$E \rightarrow TE'$
$$E\,'T\,'F$	id * id + id$	$T \rightarrow FT'$

预测分析举例

预测分析器接受输入id * id + id的前一部分动作

栈	输　入	输　出
$$E$	id * id + id$	
$$E\,'T$	id * id + id$	$E \rightarrow TE'$
$$E\,'T\,'F$	id * id + id$	$T \rightarrow FT'$
$$E\,'T\,'$ id	id * id + id$	$F \rightarrow$ id

预测分析举例

预测分析器接受输入id * id + id的前一部分动作

栈	输　入	输　出
$$E$	id * id + id$	
$$E'T$	id * id + id$	$E \rightarrow TE'$
$$E'T'F$	id * id + id$	$T \rightarrow FT'$
$$E'T'$ id	id * id + id$	$F \rightarrow$ id
$$E'T'$	* id + id$	匹配id

预测分析举例

预测分析器接受输入id * id + id的前一部分动作

栈	输　入	输　出
$$E$	id * id + id$	
$$E\,'T$	id * id + id$	$E \rightarrow TE'$
$$E\,'T\,'F$	id * id + id$	$T \rightarrow FT'$
$$E\,'T\,'$ id	id * id + id$	$F \rightarrow$ id
$$E\,'T\,'$	* id + id$	
$$E\,'T\,'F$ *	* id + id$	$T' \rightarrow *FT'$

预测分析举例

预测分析器接受输入id * id + id的前一部分动作

栈	输　入	输　出
$$E$	id * id + id$	
$$E'T$	id * id + id$	$E \rightarrow TE'$
$$E'T'F$	id * id + id$	$T \rightarrow FT'$
$$E'T'$ id	id * id + id$	$F \rightarrow$ id
$$E'T'$	* id + id$	
$$E'T'F$ *	* id + id$	$T' \rightarrow *FT'$
$$E'T'F$	id + id$	

预测分析举例

预测分析器接受输入id * id + id的前一部分动作

栈	输　入	输　出
$$E	id * id + id$	
$$E'T$	id * id + id$	$E \rightarrow TE'$
$$E'T'F$	id * id + id$	$T \rightarrow FT'$
$$E'T'$ id	id * id + id$	$F \rightarrow$ id
$$E'T'$	* id + id$	
$$E'T'F$ *	* id + id$	$T' \rightarrow *FT'$
$$E'T'F$	id + id$	
$$E'T'$ id	id + id$	$F \rightarrow$ id

预测分析举例

预测分析器接受输入id * id + id的前一部分动作

栈	输　入	输　出
$E′T′id	id + id$	$F \rightarrow id$
$E′T′	+ id$	匹配id
$E′	+ id$	$T′ \rightarrow \varepsilon$
$E′T+	+ id$	$E′ \rightarrow +TE′$
$E′T+	id$	匹配+
$E′T′F	id$	$T \rightarrow FT′$
$E′T′id	id$	$F \rightarrow id$
$E′T′	$	匹配id

预测分析举例

预测分析器接受输入id * id + id的所有动作

栈	输　入	输　出
$E′T′	$	$T′ \to \varepsilon$
$E′	$	$E′ \to \varepsilon$
$	$	Finished

多重定义

例：*stmt* → if *expr* then *stmt* *e_part* | other

e_part → else *stmt*| ε　　*expr* → *b*

非终结符	输入符号			
	other	*b*	else	. . .
stmt	*stmt* → other			
e_part			*e_part*→ else *stmt* *e_part* → ε	
expr		*expr* → *b*		

多重定义条目意味着文法左递归或者是二义的

多重定义的消除

例：删去*e_part* → ε，这正好满足else和近的then配对

LL(1)文法：预测分析表无多重定义的条目

非终结符	输入符号			
	other	*b*	else	. . .
stmt	*stmt* → other			
e_part			*e_part*→ else *stmt* ~~*e_part* → ε~~	
expr		*expr* → *b*		

《编译原理和技术》

语法分析 III

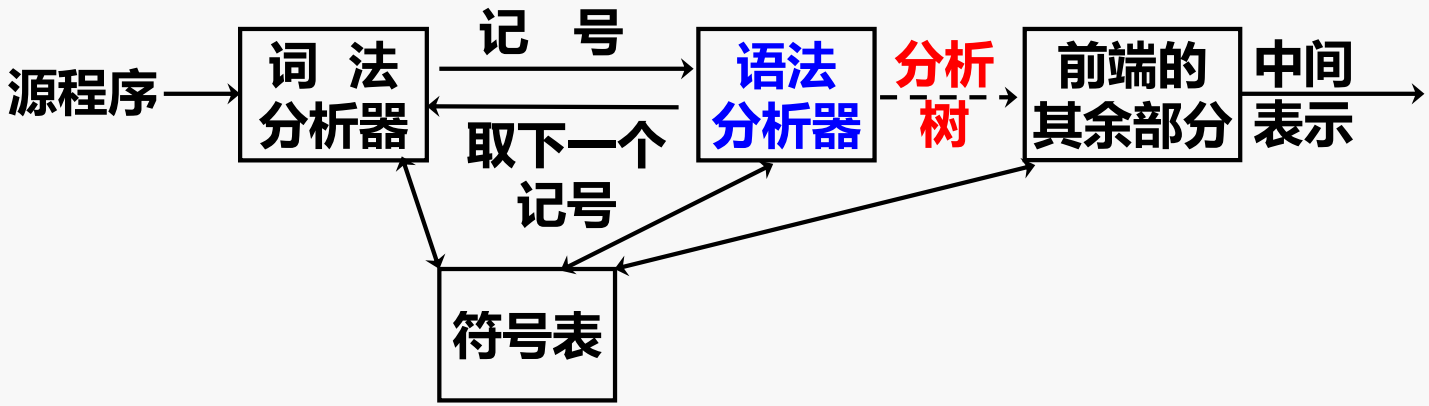
谢谢！

《编译原理和技术》

语法分析 IV

中科大计算机学院
李 诚
2022-09-19

本节提纲



- 自底向上分析方法
  - 归约(右推导的逆过程)
  - 句柄(可归约串), 可能不唯一
  - 移进-归约分析方法
  - 冲突：移进-归约、归约-归约

语法分析的主要方法

- **自顶向下（Top-down）**
  - 针对输入串，从文法的开始符号出发，尝试根据产生式规则推导（derive）出该输入串。
  - 即便是进行消除左递归、提取左公因子操作，仍然存在一些程序语言，他们对应的文法不是LL(1)
- **自底向上（Bottom-up）**
  - 针对输入串，尝试根据产生式规则归约（reduce）到文法的开始符号。
  - 比top-down分析方法更一般化

归约(Reduce)

- 每一步，特定子串被替换为相匹配的某个产生式左部的非终结符
- 最终，把输入串归约成文法的开始符号

归约(Reduce)

- 每一步，特定子串被替换为相匹配的某个产生式左部的非终结符
- 最终，把输入串归约成文法的开始符号

例 $S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$
输入串：abbcde

归约(Reduce)

- 每一步，特定子串被替换为相匹配的某个产生式左部的非终结符
- 最终，把输入串归约成文法的开始符号

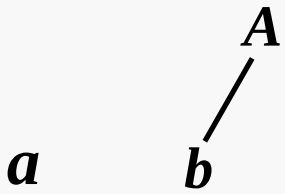
例 $S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$
输入串：abbcde
ab　　（读入ab）

a ***b***

归约(Reduce)

- 每一步，**特定子串**被替换为相匹配的某个**产生式左部的非终结符**
- 最终，把**输入串**归约成文法的**开始符号**

例 $S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$
输入串：abbcde
ab　　（读入ab）
aA　　（归约）

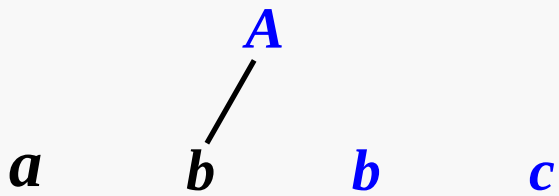


归约(Reduce)

- 每一步，特定子串被替换为相匹配的某个产生式左部的非终结符
- 最终，把输入串归约成文法的开始符号

例 $S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$
输入串：abbcde

ab	(读入ab)
aA	(归约)
aAbc	(再读入bc)



归约(Reduce)

- 每一步，特定子串被替换为相匹配的某个产生式左部的非终结符
- 最终，把输入串归约成文法的开始符号

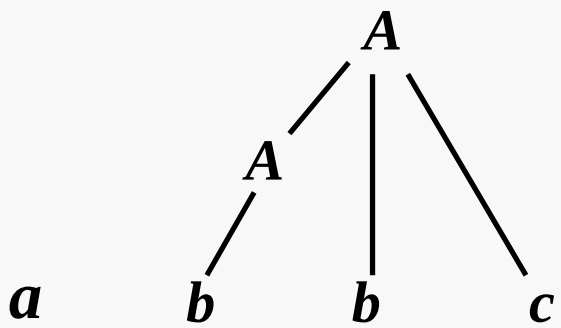
例 $S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

输入串：abbcde

ab	(读入ab)
aA	(归约)
aAbc	(再读入bc)
aA	(归约)



归约(Reduce)

- 每一步，特定子串被替换为相匹配的某个产生式左部的非终结符
- 最终，把输入串归约成文法的开始符号

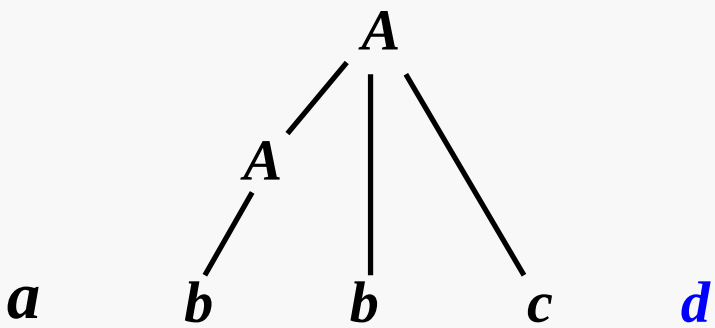
例 $S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

输入串：abbcde

ab	（读入ab）
aA	（归约）
aAbc	（再读入bc）
aA	（归约）
aAd	（再读入d）



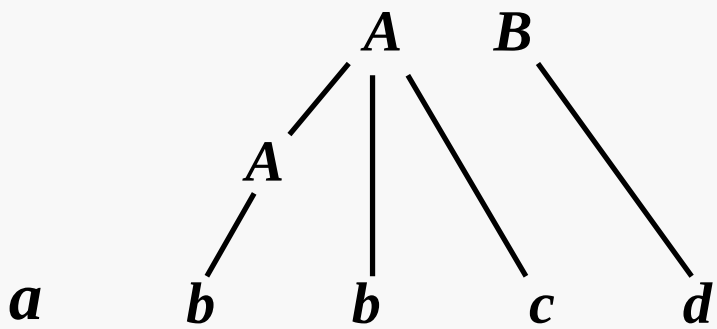
归约(Reduce)

- 每一步，特定子串被替换为相匹配的某个产生式左部的非终结符
- 最终，把输入串归约成文法的开始符号

例 $S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

输入串：abbcde

ab	(读入ab)
aA	(归约)
aAbc	(再读入bc)
aA	(归约)
aAd	(再读入d)
aAB	(归约)



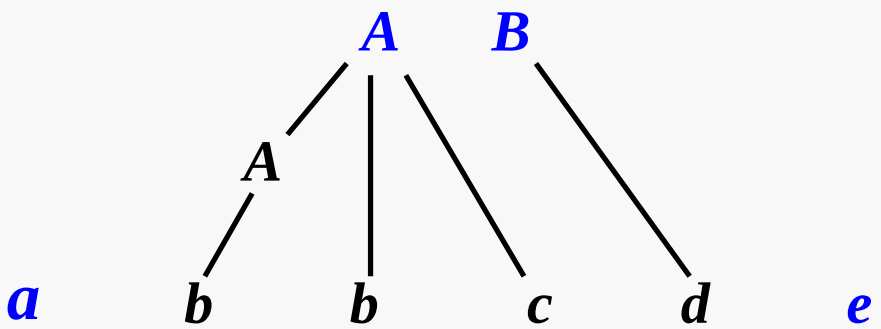
归约(Reduce)

- 每一步，特定子串被替换为相匹配的某个产生式左部的非终结符
- 最终，把输入串归约成文法的开始符号

例 $S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

输入串：abbcde

ab	(读入ab)
aA	(归约)
aAbc	(再读入bc)
aA	(归约)
aAd	(再读入d)
aAB	(归约)
aABe	(再读入e)



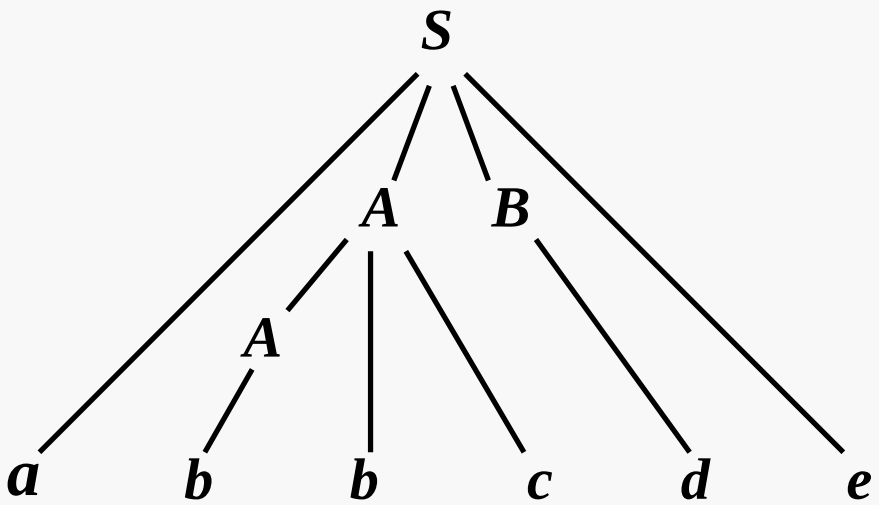
归约(Reduce)

- 每一步，**特定子串**被替换为相匹配的某个**产生式左部的非终结符**
- 最终，把**输入串**归约成文法的**开始符号**

例 $S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

输入串：abbcde

ab	(读入ab)
aA	(归约)
aAbc	(再读入bc)
aA	(归约)
aAd	(再读入d)
aAB	(归约)
aABe	(再读入e)
S	(归约)

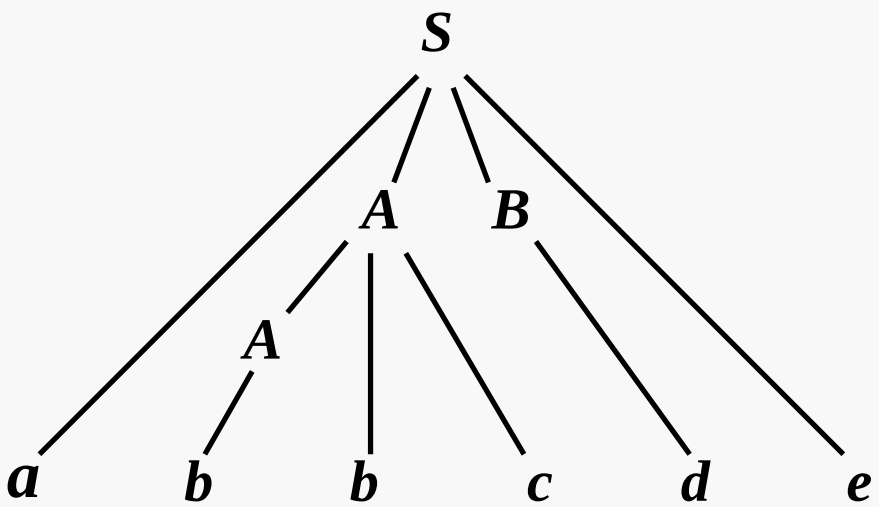


归约(Reduce)

- 每一步，特定子串被替换为相匹配的某个产生式左部的非终结符
- 最终，把输入串归约成文法的开始符号
- 归约是最右推导的逆过程

例 $S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$
输入串：abbcde

ab	(读入ab)
aAbc	(再读入bc)
aAd	(再读入d)
aABe	(再读入e)
S	(归约)

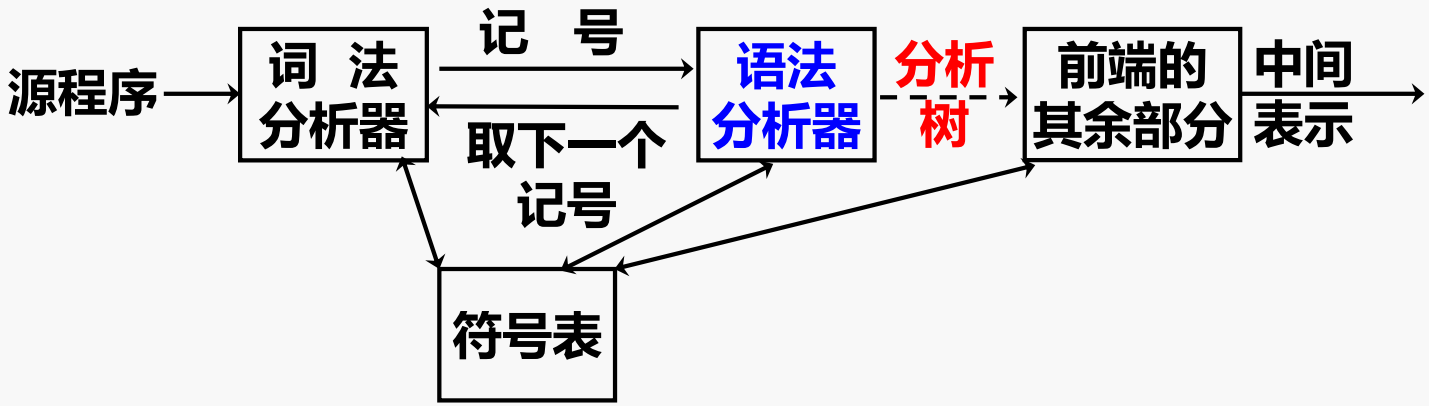


$$S \Rightarrow_{rm} aABe \Rightarrow_{rm} aAde \Rightarrow_{rm} aAbcde \Rightarrow_{rm} abbcde$$

归约(Reduce)

- **需要解决两个问题**
  - 在读入串的过程中，如何识别可以归约的子串？
  - 在进行归约的时候，选择哪一个产生式？

本节提纲



- 自底向上分析方法
  - 归约(右推导的逆过程)
  - 句柄(可归约串), 可能不唯一
  - 移进-归约分析方法
  - 冲突：移进-归约、归约-归约

句柄(Handles)

句型的句柄（可归约串）

- 该句型中和某产生式右部匹配的子串，并且
- 把它归约成该产生式左部的非终结符，代表了最右推导的逆过程的一步

$S \to aABe$

$A \to Abc \mid b$

$B \to d$

$$S \Rightarrow_{rm} aABe \Rightarrow_{rm} aAde \Rightarrow_{rm} aAbcde \Rightarrow_{rm} abbcde$$

- 句柄的右边仅含终结符
- 如果文法二义，那么句柄可能不唯一

例 句柄不唯一

$$E \to E + E \mid E * E \mid (E) \mid \mathrm{id}$$

例 句柄不唯一

$E \to E + E \mid E * E \mid (E) \mid \mathrm{id}$

$$
\begin{aligned}
E &\Rightarrow_{rm} E * E \\
&\Rightarrow_{rm} E * E + E \\
&\Rightarrow_{rm} E * E + \mathrm{id}_3 \\
&\Rightarrow_{rm} E * \mathrm{id}_2 + \mathrm{id}_3 \\
&\Rightarrow_{rm} \mathrm{id}_1 * \mathrm{id}_2 + \mathrm{id}_3
\end{aligned}
$$

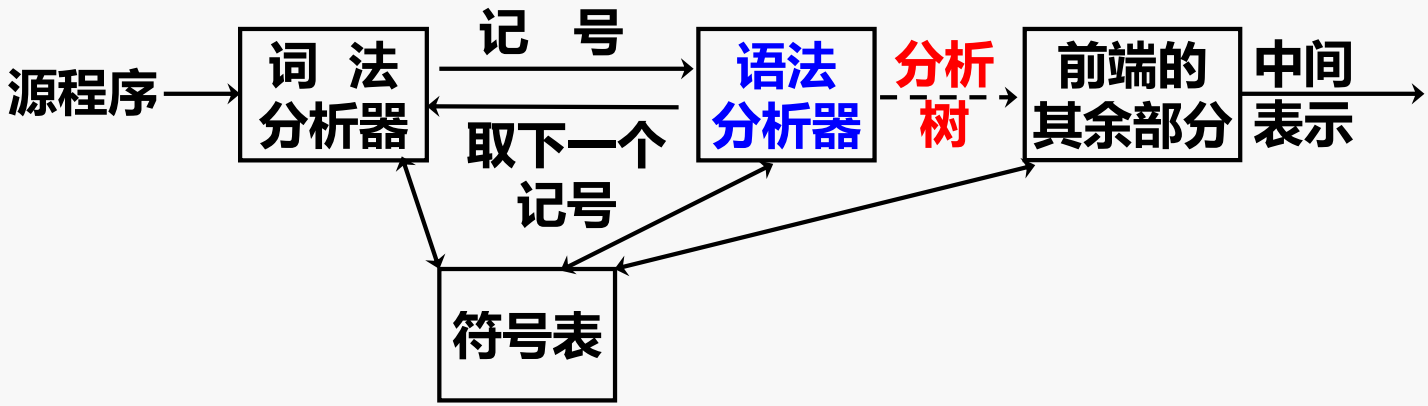
例 句柄不唯一

$E \to E + E \mid E * E \mid (E) \mid \mathrm{id}$

$$
\begin{aligned}
E &\Rightarrow_{rm} E * E \\
&\Rightarrow_{rm} E * E + E \\
&\Rightarrow_{rm} E * E + \mathrm{id}_3 \\
&\Rightarrow_{rm} E * \mathrm{id}_2 + \mathrm{id}_3 \\
&\Rightarrow_{rm} \mathrm{id}_1 * \mathrm{id}_2 + \mathrm{id}_3
\end{aligned}
\qquad
\begin{aligned}
E &\Rightarrow_{rm} E + E \\
&\Rightarrow_{rm} E + \mathrm{id}_3 \\
&\Rightarrow_{rm} E * E + \mathrm{id}_3 \\
&\Rightarrow_{rm} E * \mathrm{id}_2 + \mathrm{id}_3 \\
&\Rightarrow_{rm} \mathrm{id}_1 * \mathrm{id}_2 + \mathrm{id}_3
\end{aligned}
$$

在句型$E * E + \mathrm{id}_3$中，句柄不唯一

本节提纲



- 自底向上分析方法
  - 归约(右推导的逆过程)
  - 句柄(可归约串), 可能不唯一
  - 移进-归约分析方法
  - 冲突：移进-归约、归约-归约

移进-归约分析技术

用栈实现移进-归约分析

- 栈保存已扫描过的文法符号，缓冲区存放还未分析的其余符号
- 移进(shift)：将下一个输入符号放到栈顶，以形成句柄
- 归约(reduce)：将句柄替换为对应的产生式的左部非终结符
- 接受(accept)：分析成功
- 报错(error)：发现语法错误

移进-归约分析技术

用栈实现移进-归约分析

- 先通过分析输入串$id_1*id_2 + id_3$时的动作序列来了解移进-归约分析的工作方式

移进-归约$id_1*id_2 + id_3$

栈	输　入	动　作
$	$id_1 * id_2 + id_3$$	

移进-归约$id_1*id_2 + id_3$

栈	输　入	动　作
$	$id_1 * id_2 + id_3$$	移进

移进-归约$id_1*id_2 + id_3$

栈	输　入	动　作
$	$id_1 * id_2 + id_3$$	移进
$ id_1	$* id_2 + id_3$$	

移进-归约$id_1*id_2 + id_3$

栈	输　入	动　作
$	$id_1 * id_2 + id_3$$	移进
$ id_1	$* id_2 + id_3$$	按$E \rightarrow$ id归约

移进-归约$id_1*id_2 + id_3$

栈	输　入	动　作
$	$id_1 * id_2 + id_3$$	移进
$ id_1	$* id_2 + id_3$$	按$E \rightarrow$ id归约
$$E$	$* id_2 + id_3$$	

移进-归约$id_1*id_2 + id_3$

栈	输　入	动　作
$	$id_1 * id_2 + id_3$$	移进
$ id_1	$* id_2 + id_3$$	按$E \rightarrow$ id归约
$$E$	$* id_2 + id_3$$	移进

移进-归约$id_1*id_2 + id_3$

栈	输　入	动　作
$	$id_1 * id_2 + id_3$$	移进
$ id_1	$* id_2 + id_3$$	按$E \rightarrow$ id归约
$$E$	$* id_2 + id_3$$	移进
$$E$*	$id_2 + id_3$$	

移进-归约$id_1*id_2 + id_3$

栈	输　入	动　作
$	$id_1 * id_2 + id_3$$	移进
$ id_1	$* id_2 + id_3$$	按$E \rightarrow$ id归约
$$E$	$* id_2 + id_3$$	移进
$$E$*	$id_2 + id_3$$	移进

移进-归约$id_1*id_2 + id_3$

栈	输　入	动　作
$	$id_1 * id_2 + id_3$$	移进
$ id_1	$* id_2 + id_3$$	按$E \rightarrow$ id归约
$$E$	$* id_2 + id_3$$	移进
$$E*$	$id_2 + id_3$$	移进
$$E*id_2$	$+ id_3$$	

移进-归约$id_1*id_2 + id_3$

栈	输　入	动　作
$	$id_1 * id_2 + id_3$$	移进
$ id_1	$* id_2 + id_3$$	按$E \rightarrow$ id归约
$$E$	$* id_2 + id_3$$	移进
$$E*$	$id_2 + id_3$$	移进
$$E*id_2$	$+ id_3$$	按$E \rightarrow$ id归约

移进-归约$id_1*id_2 + id_3$

栈	输　入	动　作
$	$id_1 * id_2 + id_3$$	移进
$ id_1	$* id_2 + id_3$$	按$E \rightarrow$ id归约
$$E$	$* id_2 + id_3$$	移进
$$E*$	$id_2 + id_3$$	移进
$$E*id_2$	$+ id_3$$	按$E \rightarrow$ id归约
$$E*E$	$+ id_3$$	

移进-归约$id_1*id_2 + id_3$

栈	输　入	动　作
$	$id_1 * id_2 + id_3$$	移进
$ id_1	$* id_2 + id_3$$	按$E \rightarrow$ id归约
$$E$	$* id_2 + id_3$$	移进
$$E*$	$id_2 + id_3$$	移进
$$E*id_2$	$+ id_3$$	按$E \rightarrow$ id归约
$$E*E$	$+ id_3$$	移进

移进-归约$id_1*id_2 + id_3$

栈	输　入	动　作
$	$id_1 * id_2 + id_3$$	移进
$ id_1	$* id_2 + id_3$$	按$E \rightarrow$ id归约
$$E$	$* id_2 + id_3$$	移进
$$E*$	$id_2 + id_3$$	移进
$$E*id_2$	$+ id_3$$	按$E \rightarrow$ id归约
$$E*E$	$+ id_3$$	移进
$$E*E+$	$id_3$$	

移进-归约$id_1*id_2 + id_3$

栈	输　入	动　作
$	$id_1 * id_2 + id_3$$	移进
$ id_1	$* id_2 + id_3$$	按$E \rightarrow$ id归约
$$E$	$* id_2 + id_3$$	移进
$$E*$	$id_2 + id_3$$	移进
$$E*id_2$	$+ id_3$$	按$E \rightarrow$ id归约
$$E*E$	$+ id_3$$	移进
$$E*E+$	$id_3$$	移进

移进-归约$id_1*id_2 + id_3$

栈	输　入	动　作
$	$id_1 * id_2 + id_3$$	移进
$ id_1	$* id_2 + id_3$$	按$E \rightarrow$ id归约
$$E$	$* id_2 + id_3$$	移进
$$E*$	$id_2 + id_3$$	移进
$$E*id_2$	$+ id_3$$	按$E \rightarrow$ id归约
$$E*E$	$+ id_3$$	移进
$$E*E+$	$id_3$$	移进
$$E*E+id_3$	$$	

移进-归约$id_1*id_2 + id_3$

栈	输　入	动　作
$	$id_1 * id_2 + id_3$$	移进
$ id_1	$* id_2 + id_3$$	按$E \rightarrow$ id归约
$$E$	$* id_2 + id_3$$	移进
$$E*$	$id_2 + id_3$$	移进
$$E*id_2$	$+ id_3$$	按$E \rightarrow$ id归约
$$E*E$	$+ id_3$$	移进
$$E*E+$	$id_3$$	移进
$$E*E+id_3$	$$	按$E \rightarrow$ id归约

移进-归约$id_1*id_2 + id_3$

栈	输　入	动　作
$	$id_1 * id_2 + id_3$$	移进
$ id_1	$* id_2 + id_3$$	按$E \rightarrow$ id归约
$$E$	$* id_2 + id_3$$	移进
$$E*$	$id_2 + id_3$$	移进
$$E*id_2$	$+ id_3$$	按$E \rightarrow$ id归约
$$E*E$	$+ id_3$$	移进
$$E*E+$	$id_3$$	移进
$$E*E+id_3$	$$	按$E \rightarrow$ id归约
$$E*E+E$	$$	

移进-归约$id_1*id_2 + id_3$

栈	输　入	动　作
$	$id_1 * id_2 + id_3$$	移进
$ id_1	$* id_2 + id_3$$	按$E \rightarrow$ id归约
$$E$	$* id_2 + id_3$$	移进
$$E*$	$id_2 + id_3$$	移进
$$E*id_2$	$+ id_3$$	按$E \rightarrow$ id归约
$$E*E$	$+ id_3$$	移进
$$E*E+$	$id_3$$	移进
$$E*E+id_3$	$$	按$E \rightarrow$ id归约
$$E*E+E$	$$	按$E \rightarrow E+E$归约

移进-归约$id_1*id_2 + id_3$

栈	输　入	动　作
$	$id_1 * id_2 + id_3$$	移进
$ id_1	$* id_2 + id_3$$	按$E \rightarrow$ id归约
$$E$	$* id_2 + id_3$$	移进
$$E*$	$id_2 + id_3$$	移进
$$E*id_2$	$+ id_3$$	按$E \rightarrow$ id归约
$$E*E$	$+ id_3$$	移进
$$E*E+$	$id_3$$	移进
$$E*E+id_3$	$$	按$E \rightarrow$ id归约
$$E*E+E$	$$	按$E \rightarrow E+E$归约
$$E*E$	$$	

移进-归约$id_1*id_2 + id_3$

栈	输　入	动　作
$	$id_1 * id_2 + id_3$$	移进
$ id_1	$* id_2 + id_3$$	按$E \rightarrow$ id归约
$$E$	$* id_2 + id_3$$	移进
$$E*$	$id_2 + id_3$$	移进
$$E*id_2$	$+ id_3$$	按$E \rightarrow$ id归约
$$E*E$	$+ id_3$$	移进
$$E*E+$	$id_3$$	移进
$$E*E+id_3$	$$	按$E \rightarrow$ id归约
$$E*E+E$	$$	按$E \rightarrow E+E$归约
$$E*E$	$$	按$E \rightarrow E*E$归约

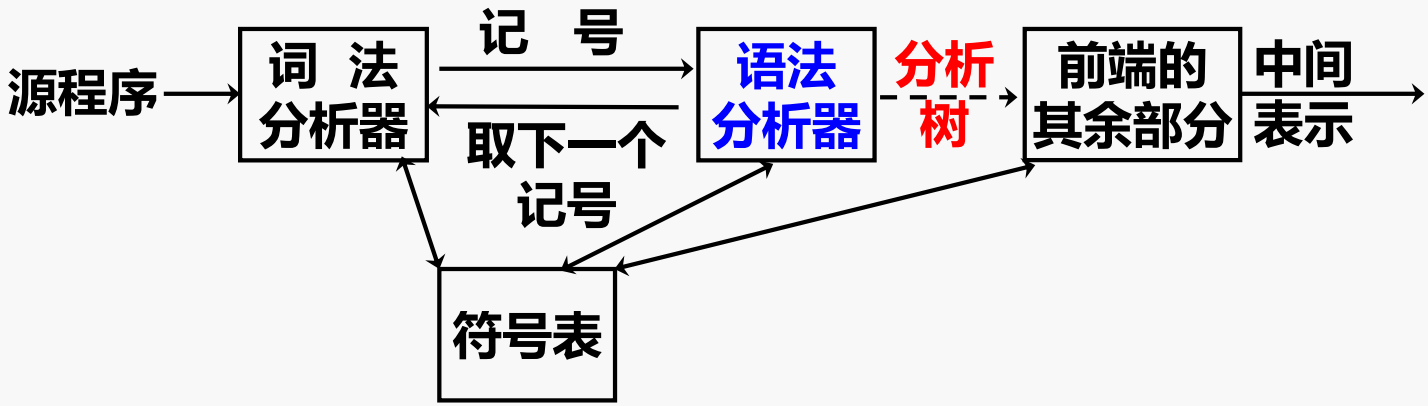
移进-归约$id_1*id_2 + id_3$

栈	输　入	动　作
$	$id_1 * id_2 + id_3$$	移进
$ id_1	$* id_2 + id_3$$	按$E \rightarrow$ id归约
$$E$	$* id_2 + id_3$$	移进
$$E*$	$id_2 + id_3$$	移进
$$E*id_2$	$+ id_3$$	按$E \rightarrow$ id归约
$$E*E$	$+ id_3$$	移进
$$E*E+$	$id_3$$	移进
$$E*E+id_3$	$$	按$E \rightarrow$ id归约
$$E*E+E$	$$	按$E \rightarrow E+E$归约
$$E*E$	$$	按$E \rightarrow E*E$归约
$$E$	$$	

移进-归约$id_1*id_2 + id_3$

栈	输　入	动　作
$	$id_1 * id_2 + id_3$$	移进
$ id_1	$* id_2 + id_3$$	按$E \rightarrow$ id归约
$$E$	$* id_2 + id_3$$	移进
$$E*$	$id_2 + id_3$$	移进
$$E*id_2$	$+ id_3$$	按$E \rightarrow$ id归约
$$E*E$	$+ id_3$$	移进
$$E*E+$	$id_3$$	移进
$$E*E+id_3$	$$	按$E \rightarrow$ id归约
$$E*E+E$	$$	按$E \rightarrow E+E$归约
$$E*E$	$$	按$E \rightarrow E*E$归约
$$E$	$$	接受

本节提纲



- 自底向上分析方法
  - 归约(右推导的逆过程)
  - 句柄(可归约串), 可能不唯一
  - 移进-归约分析方法
  - 冲突：移进-归约、归约-归约

移进–归约冲突

例 *stmt* → if *expr* then *stmt*

| if *expr* then *stmt* else *stmt*

| other

如果移进–归约分析器处于格局(configuration)

栈	输入
… if *expr* then *stmt*	else … $

归约?

移进?

归约–归约冲突

例 *stmt* → id (*parameter_list*) | *expr* = *expr*

parameter_list→*parameter_list*, *parameter* | *parameter*

parameter → id

expr → id (*expr_list*) | id

expr_list → *expr_list*, *expr* | *expr*

由*A*(*I*, *J*)开始的语句

归约成expr还是parameter？

栈	输入
… id (id	, id)…

归约–归约冲突

例 *stmt* → procid (*parameter_list*) | *expr* = *expr*
parameter_list→*parameter_list*, *parameter* | *parameter*
parameter → id
expr → id (*expr_list*) | id
expr_list → *expr_list*, *expr* | *expr*

由*A*(*I*, *J*)开始的语句（词法分析查符号表, 区分第一个id）

栈	输入
… procid (id	, id)…

需要修改文法中的第一个产生式，并利用栈中信息

《编译原理和技术》

语法分析 IV

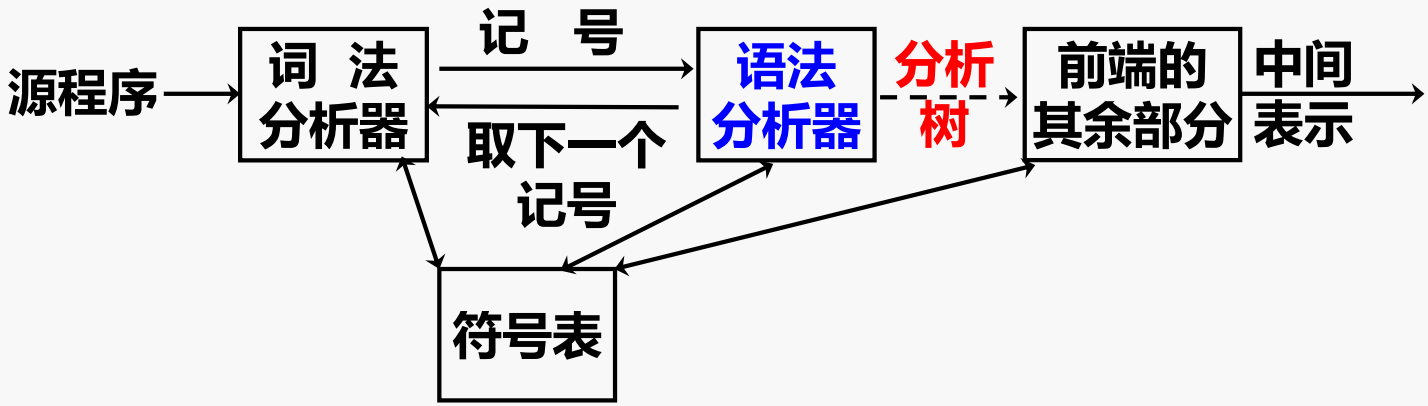
谢谢！

《编译原理和技术》

语法分析 V

中科大计算机学院
李 诚
2022-09-21

本节提纲

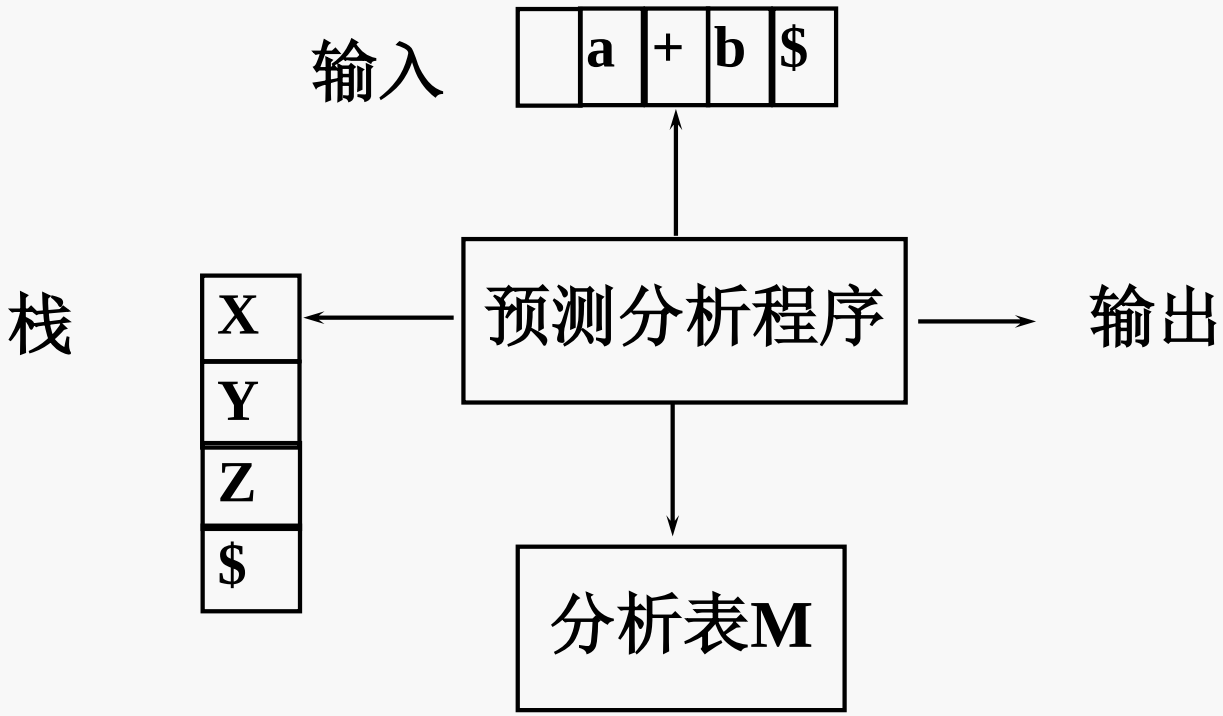


- **LR(k)分析技术**
  - LR分析器的简单模型
    - action, goto函数
  - 简单的LR方法（简称SLR）
    - 活前缀，识别活前缀的DFA/NFA，SLR算法
  - 规范的LR方法
  - 向前看的LR方法（简称LALR）

语法分析的主要方法

- **自顶向下（Top-down）**
  - 针对输入串，从文法的开始符号出发，尝试根据产生式规则推导（derive）出该输入串。
  - LL(1)文法及非递归预测分析方法
  - left-to-right scan + leftmost derivation
- **自底向上（Bottom-up）**
  - 针对输入串，尝试根据产生式规则归约（reduce）到文法的开始符号。
  - LR(k)文法及其分析器
  - left-to-right scan + rightmost derivation

复习：LL(1)非递归分析

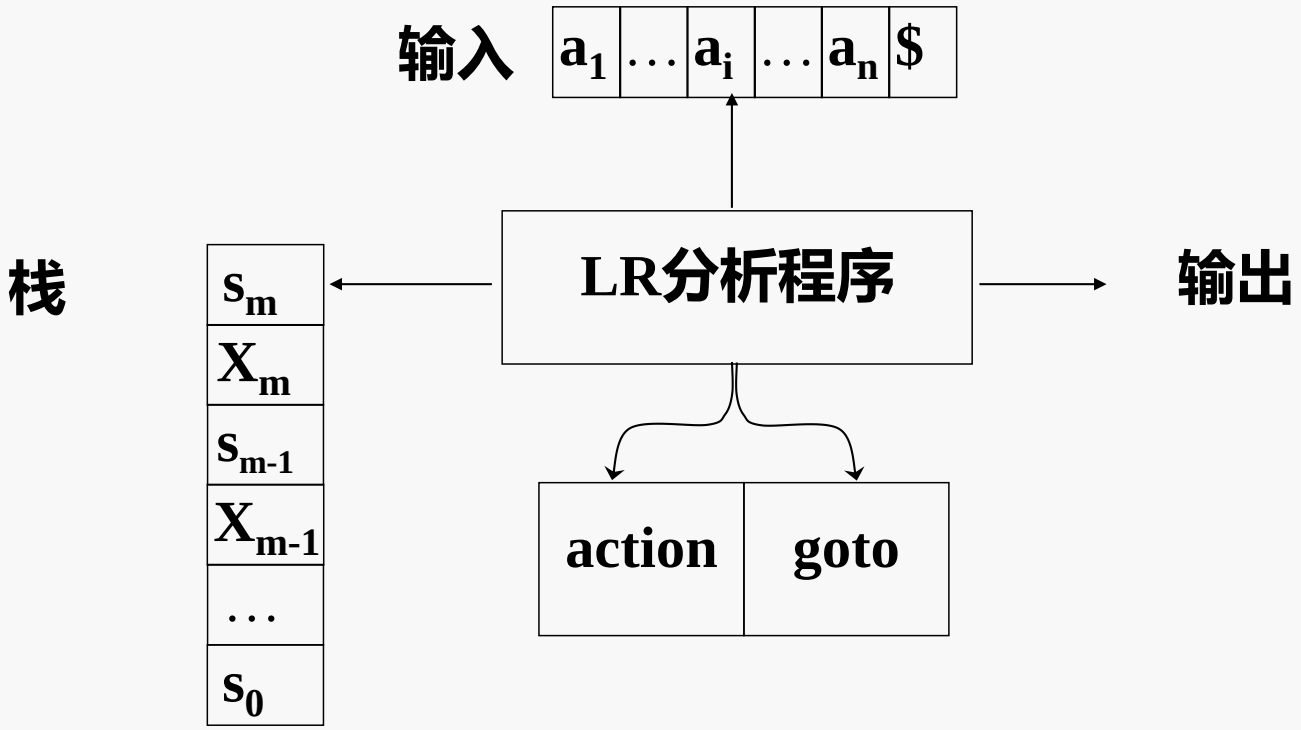


复习：LL(1)非递归分析

- 行：非终结符；列：终结符 或$ ；单元：产生式

非终结符	输 入 符 号					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow$id			$F \rightarrow (E)$		

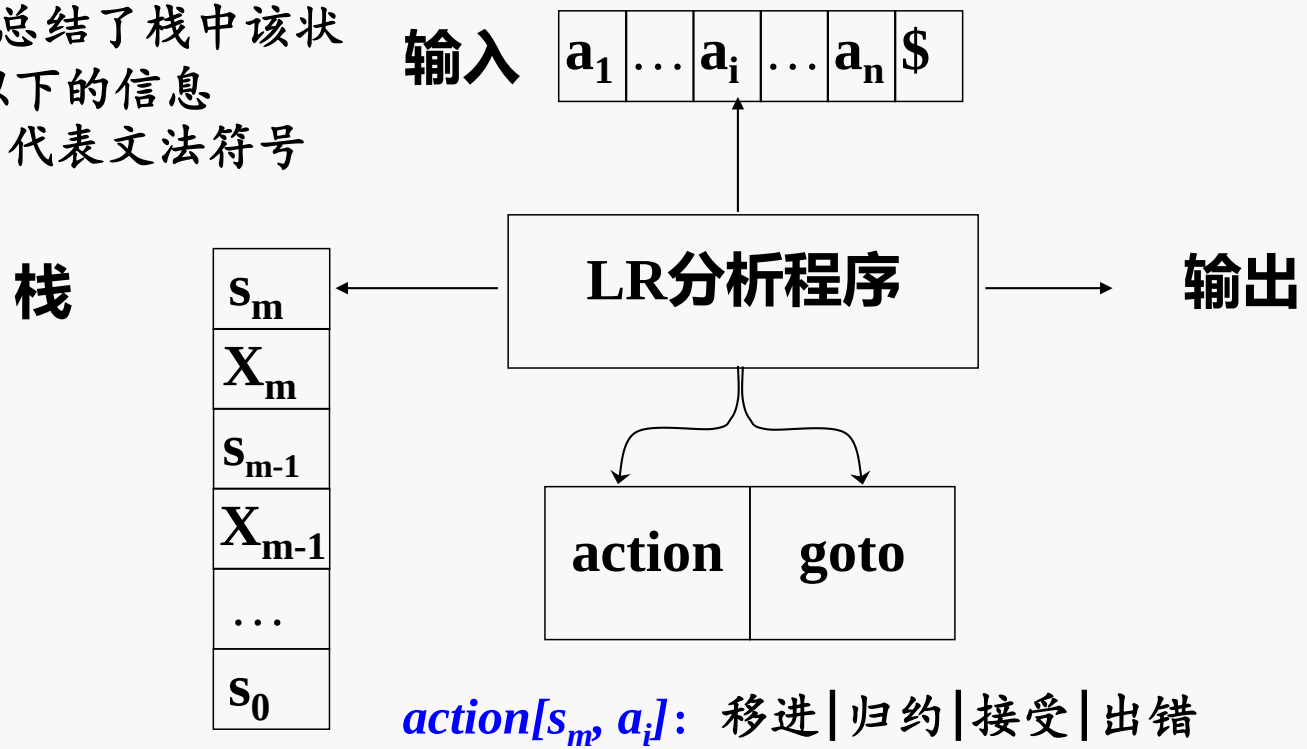
LR分析器



LR分析器

s_j：总结了栈中该状态以下的信息

X_i：代表文法符号



$action[s_m, a_i]$：移进|归约|接受|出错

$goto[s_{m-r}, A]=s_j$：移进A和s_j（归约后使用）

LR分析算法：举例

例 (1) $E \to E + T$ (2) $E \to T$

(3) $T \to T * F$ (4) $T \to F$

(5) $F \to (E)$ (6) $F \to \mathrm{id}$

si 移进当前输入符号和状态*i*

rj 按第*j*个产生式进行归约

acc 接受

状态	动作 action						转移 goto		
	id	+	*	(	)	$	*E*	*T*	*F*
0	*s*5			*s*4			1	2	3
1		*s*6				*acc*			
2		*r*2	*s*7		*r*2	*r*2			
3		*r*4	*r*4		*r*4	*r*4			
4	*s*5			*s*4			8	2	3
5		*r*6	*r*6		*r*6	*r*6			
6	*s*5			*s*4				9	3

LR分析算法：举例

例 (1) $E \rightarrow E + T$ (2) $E \rightarrow T$

(3) $T \rightarrow T * F$ (4) $T \rightarrow F$

(5) $F \rightarrow (E)$ (6) $F \rightarrow$ id

si 移进当前输入符号和状态*i*

rj 按第*j*个产生式进行归约

acc 接受

状态	action						goto		
	id	+	*	(	)	$	*E*	*T*	*F*
0	*s*5			*s*4			1	2	3
1		*s*6				*acc*			
2		*r*2	*s*7		*r*2	*r*2			
3		*r*4	*r*4		*r*4	*r*4			
4	*s*5			*s*4			8	2	3
5		*r*6	*r*6		*r*6	*r*6			
6	*s*5			*s*4				9	3

LR分析算法：举例

栈	输　入	动　作
0	id * id + id $	

状态	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3

si 移进当前输入符号和状态*i*

rj 按第*j*个产生式进行归约

acc 接受

LR分析算法：举例

栈	输　入	动　作
0	id * id + id $	移进 (查action表)

状态	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3

si 移进当前输入符号和状态*i*

rj 按第*j*个产生式进行归约

acc 接受

LR分析算法：举例

栈	输　入	动　作
0	id * id + id $	移进
0 id 5	* id + id $	

状态	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3

si 移进当前输入符号和状态*i*

rj 按第*j*个产生式进行归约

acc 接受

LR分析算法：举例

栈	输　入	动　作
0	id * id + id $	移进
0 id 5	* id + id $	按$F \rightarrow$ id归约

1. 查action[5, *]=>归约
2. 执行归约($F \rightarrow \alpha$):
- 从栈中弹出$|\alpha|$个<状态, 符号>对
- 查goto[0, F] =>3
- 将(F, 3)压入栈

状态	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3

si 移进当前输入符号和状态*i*

rj 按第*j*个产生式进行归约

acc 接受

LR分析算法：举例

栈	输　入	动　作
0	id * id + id $	移进
0 id 5	* id + id $	按$F \rightarrow$ id归约
0 *F* 3	* id + id $	

状态	action						goto		
	id	+	*	(	)	$	*E*	*T*	*F*
0	*s*5			*s*4			1	2	3
1		*s*6				*acc*			
2		*r*2	*s*7		*r*2	*r*2			
3		*r*4	*r*4		*r*4	*r*4			
4	*s*5			*s*4			8	2	3
5		*r*6	*r*6		*r*6	*r*6			
6	*s*5			*s*4				9	3

si 移进当前输入符号和状态*i*

rj 按第*j*个产生式进行归约

acc 接受

LR分析算法：举例

栈	输　入	动　作
0	id * id + id $	移进
0 id 5	* id + id $	按$F \rightarrow$ id归约
0 *F* 3	* id + id $	按$T \rightarrow F$归约

状态	action						goto		
	id	+	*	(	)	$	*E*	*T*	*F*
0	*s*5			*s*4			1	2	3
1		*s*6				*acc*			
2		*r*2	*s*7		*r*2	*r*2			
3		*r*4	*r*4		*r*4	*r*4			
4	*s*5			*s*4			8	2	3
5		*r*6	*r*6		*r*6	*r*6			
6	*s*5			*s*4				9	3

si 移进当前输入符号和状态*i*
rj 按第*j*个产生式进行归约
acc 接受

LR分析算法：举例

栈	输　入	动　作
0	id * id + id $	移进
0 id 5	* id + id $	按$F \rightarrow$ id归约
0 F 3	* id + id $	按$T \rightarrow F$归约
0 T 2	* id + id $	

状态	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3

si 移进当前输入符号和状态*i*

rj 按第*j*个产生式进行归约

acc 接受

LR分析算法：举例

栈	输　入	动　作
0	id * id + id $	移进
0 id 5	* id + id $	按$F \rightarrow$ id归约
0 F 3	* id + id $	按$T \rightarrow F$归约
0 T 2	* id + id $	移进

状态	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3

si 移进当前输入符号和状态*i*

rj 按第*j*个产生式进行归约

acc 接受

LR分析算法：举例

栈	输　入	动　作
0	id * id + id $	移进
0 id 5	* id + id $	按$F \rightarrow$ id归约
0 F 3	* id + id $	按$T \rightarrow F$归约
0 T 2	* id + id $	移进
0 T 2 * 7	id + id $	

状态	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3

si 移进当前输入符号和状态*i*

rj 按第*j*个产生式进行归约

acc 接受

LR分析算法：举例

栈	输　入	动　作
0	id * id + id $	移进
0 id 5	* id + id $	按$F \rightarrow$ id归约
0 F 3	* id + id $	按$T \rightarrow F$归约
0 T 2	* id + id $	移进
0 T 2 * 7	id + id $	移进

状态	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3

si 移进当前输入符号和状态*i*

rj 按第*j*个产生式进行归约

acc 接受

LR分析算法：举例

栈	输　入	动　作
0	id * id + id $	移进
0 id 5	* id + id $	按$F \rightarrow$ id归约
0 F 3	* id + id $	按$T \rightarrow F$归约
0 T 2	* id + id $	移进
0 T 2 * 7	id + id $	移进
0 T 2 * 7 id 5	+ id $	

状态	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3

si 移进当前输入符号和状态*i*

rj 按第*j*个产生式进行归约

acc 接受

LR分析算法：举例

栈	输　入	动　作
0	id * id + id $	移进
0 id 5	* id + id $	按$F \rightarrow$ id归约
0 F 3	* id + id $	按$T \rightarrow F$归约
0 T 2	* id + id $	移进
0 T 2 * 7	id + id $	移进
0 T 2 * 7 id 5	+ id $	按$F \rightarrow$ id归约

状态	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3

si 移进当前输入符号和状态*i*

rj 按第*j*个产生式进行归约

acc 接受

LR分析算法：举例

栈	输　入	动　作
0	id * id + id $	移进
0 id 5	* id + id $	按$F \rightarrow$ id归约
0 F 3	* id + id $	按$T \rightarrow F$归约
0 T 2	* id + id $	移进
0 T 2 * 7	id + id $	移进
0 T 2 * 7 id 5	+ id $	按$F \rightarrow$ id归约
0 T 2 * 7 F 10	+ id $	

状态	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3

si 移进当前输入符号和状态*i*

rj 按第*j*个产生式进行归约

acc 接受

LR分析算法：举例

栈	输　入	动　作
0	id * id + id $	移进
0 id 5	* id + id $	按$F \rightarrow$ id归约
0 F 3	* id + id $	按$T \rightarrow F$归约
0 T 2	* id + id $	移进
0 T 2 * 7	id + id $	移进
0 T 2 * 7 id 5	+ id $	按$F \rightarrow$ id归约
0 T 2 * 7 F 10	+ id $	按$T \rightarrow T*F$归约

状态	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3

si 移进当前输入符号和状态*i*

rj 按第*j*个产生式进行归约

acc 接受

LR分析算法：举例

栈	输　入	动　作
0	id * id + id $	移进
0 id 5	* id + id $	按$F \rightarrow$ id归约
0 F 3	* id + id $	按$T \rightarrow F$归约
0 T 2	* id + id $	移进
0 T 2 * 7	id + id $	移进
0 T 2 * 7 id 5	+ id $	按$F \rightarrow$ id归约
0 T 2 * 7 F 10	+ id $	按$T \rightarrow T*F$归约
. . .	. . .	. . .

状态	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3

si 移进当前输入符号和状态*i*

rj 按第*j*个产生式进行归约

acc 接受

LR分析算法：举例

栈	输　入	动　作
0	id * id + id $	移进
0 id 5	* id + id $	按$F \rightarrow$ id归约
0 F 3	* id + id $	按$T \rightarrow F$归约
0 T 2	* id + id $	移进
0 T 2 * 7	id + id $	移进
0 T 2 * 7 id 5	+ id $	按$F \rightarrow$ id归约
0 T 2 * 7 F 10	+ id $	按$T \rightarrow T*F$归约
. . .	. . .	. . .
0 E 1	$	

状态	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3

si 移进当前输入符号和状态*i*

rj 按第*j*个产生式进行归约

acc 接受

LR分析算法：举例

栈	输　入	动　作
0	id * id + id $	移进
0 id 5	* id + id $	按$F \rightarrow$ id归约
0 F 3	* id + id $	按$T \rightarrow F$归约
0 T 2	* id + id $	移进
0 T 2 * 7	id + id $	移进
0 T 2 * 7 id 5	+ id $	按$F \rightarrow$ id归约
0 T 2 * 7 F 10	+ id $	按$T \rightarrow T*F$归约
. . .	. . .	. . .
0 E 1	$	接受

状态	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3

si 移进当前输入符号和状态*i*

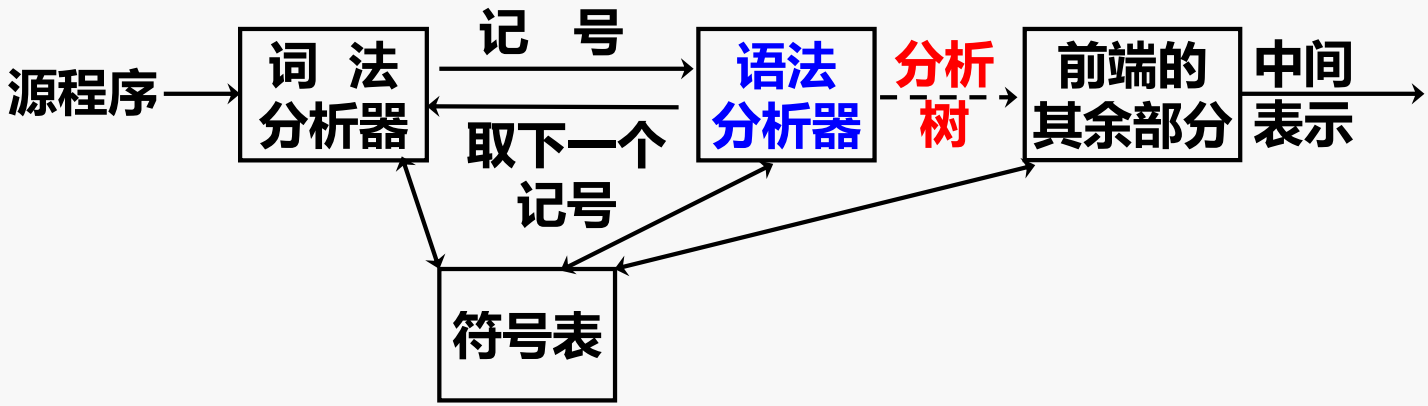
rj 按第*j*个产生式进行归约

acc 接受

LR语法分析器

- **关键在于构造LR分析表**
  - 计算所有可能的状态
    - 每一个状态描述了语法分析过程中所处的位置
    - 可确定正在分析的产生式集合
    - 可确定句柄形成的中间步骤
  - 明确状态之前的跳转关系
  - 明确状态与输入之间对应的移进或者归约操作

本节提纲



- **LR(k)分析技术**
  - LR分析器的简单模型
    - action, goto函数
  - 简单的LR方法（简称SLR）
    - 活前缀，识别活前缀的DFA/NFA，SLR算法
  - 规范的LR方法
  - 向前看的LR方法（简称LALR）

句柄(Handles)

句型的句柄（可归约串）

- 该句型中和某产生式右部匹配的子串，并且
- 把它归约成该产生式左部的非终结符，代表了最右推导的逆过程的一步

$S \to aABe$

$A \to Abc \mid b$

$B \to d$

$S \Rightarrow_{rm} aABe \Rightarrow_{rm} aAde \Rightarrow_{rm} aAbcde \Rightarrow_{rm} abbcde$

- 句柄的右边仅含终结符
- 如果文法二义，那么句柄可能不唯一

LR语法分析器的格局

- LR语法分析的每一步都形成一个格局config

$$(s_0X_1s_1X_2s_2\ldots X_ms_m,\quad a_ia_{i+1}\ldots a_n\$)$$

栈的内容 **尚未处理的输入**

  - 代表最右句型$X_1X_2\ldots X_ma_ia_{i+1}\ldots a_n$
  - $X_1X_2\ldots X_m$是最右句型的一个前缀

LR语法分析器的格局

- LR语法分析的每一步都形成一个格局config

  $(s_0X_1s_1X_2s_2\ldots X_ms_m,\ \ a_ia_{i+1}\ldots a_n\$)$

  栈的内容 **尚未处理的输入**

  - 代表最右句型$X_1X_2\ldots X_ma_ia_{i+1}\ldots a_n$
  - $X_1X_2\ldots X_m$是最右句型的一个前缀
  - 每一个前缀都对应一个状态，因此，找出所有可能在栈里出现的前缀，就可以确定所有的状态

LR语法分析器的格局

- LR语法分析的每一步都形成一个格局config

  $(s_0X_1s_1X_2s_2\ldots X_ms_m,\ \ a_ia_{i+1}\ldots a_n\$)$

  栈的内容 **尚未处理的输入**

  - 代表最右句型$X_1X_2\ldots X_ma_ia_{i+1}\ldots a_n$
  - $X_1X_2\ldots X_m$是最右句型的一个前缀
  - 每一个前缀都对应一个状态，因此，找出所有可能在栈里出现的前缀，就可以确定所有的状态
  - 状态之间的转换 <==> 前缀之间的转换

LR语法分析器的格局

- **LR语法分析的每一步都形成一个格局config**

  $(s_0X_1s_1X_2s_2\ldots X_ms_m,\ a_ia_{i+1}\ldots a_n\$)$

  栈的内容 **尚未处理的输入**

  - 代表最右句型$X_1X_2\ldots X_ma_ia_{i+1}\ldots a_n$
  - $X_1X_2\ldots X_m$是最右句型的一个前缀
  - 每一个前缀都对应一个状态，因此，找出所有可能在栈里出现的前缀，就可以确定所有的状态
  - 状态之间的转换 <==> 前缀之间的转换
  - 在栈顶为s，下一个字符为a的格局下，前缀为p
    - 何时移进？当p包含句柄的一部分且存在p′= pa
    - 何时归约？当p包含整个句柄时

活前缀

$S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

栈中可能出现的串：

a
ab
aA
aAb
aAbc
aAd
aAB
aABe
S

活前缀：
最右句型的前缀，该前缀不超过最右句柄的右端

$$S \Rightarrow^*_{rm} \gamma A w \Rightarrow_{rm} \gamma \beta w$$

$\gamma\beta$的任何前缀（包括ε和$\gamma\beta$本身）都是一个活前缀。

活前缀与句柄的关系

$S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

栈中可能出现的串：

a
ab ← 出现句柄（对应$A \rightarrow b$）
aA
aAb
$aAbc$ ← 出现句柄（对应$A \rightarrow Abc$）
aAd ← 出现句柄（对应$B \rightarrow d$）
aAB
$aABe$ ← 出现句柄（对应$S \rightarrow aABe$）
S

- 活前缀已含有句柄，表明产生式A→β的右部β已出现在栈顶。

活前缀与句柄的关系

$S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

栈中可能出现的串：

a
ab
aA ← 出现产生式$A \rightarrow Abc$右端的一部分，期望从输入串中看到bc
aAb ← 出现产生式A→Abc右端的一部分，期望从输入串中看到c
$aAbc$
aAd
aAB ← 出现产生式S→aABe的右端一部分，期望从输入串中看到e
$aABe$
S

- 活前缀已含有句柄，表明产生式A→β的右部β已出现在栈顶。
- 活前缀只含句柄的一部分符号如β_1表明$A \rightarrow \beta_1\beta_2$的右部子串$\beta_1$已出现在栈顶，当前期待从输入串中看到$\beta_2$推出的符号。

LR分析: 基本概念

- **活前缀或可行前缀（viable prefix）：**
  - 最右句型的前缀，该前缀不超过最右句柄的右端

$$S \Rightarrow^*_{rm} \gamma A w \Rightarrow_{rm} \gamma \beta w$$

  - $\gamma\beta$的任何前缀（包括ε和$\gamma\beta$本身）都是活前缀
  - 都出现在栈顶

LR分析方法的特点

- 栈中的文法符号总是形成一个活前缀
- 分析表的转移函数本质上是识别活前缀的DFA

下表蓝色部分构成识别活前缀DFA的状态转换表

状态	动作						转移		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3

LR分析方法的特点

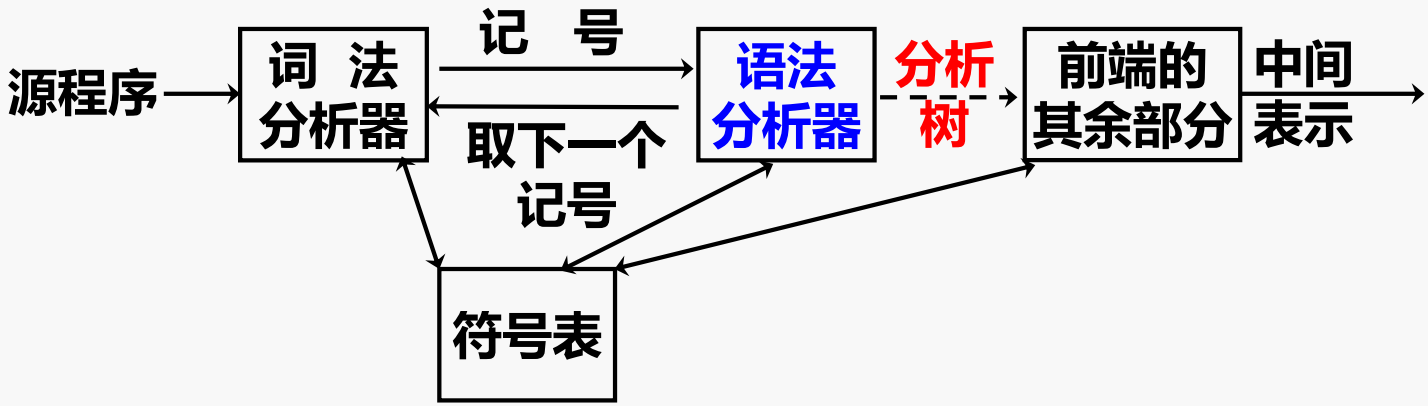
- 栈中的文法符号总是形成一个活前缀
- 分析表的转移函数本质上是识别活前缀的DFA
- 栈顶的状态符号包含确定句柄所需的一切信息

栈	输　入	动　作
0	id * id + id $	移进
…	…	…
0 *T* 2 * 7	id + id $	移进
0 *T* 2 * 7 id 5	+ id $	按*F* → id归约
0 *T* 2 * 7 *F* 10	+ id $	按*T* → *T***F*归约

LR分析方法的特点

- 栈中的文法符号总是形成一个活前缀
- 分析表的转移函数本质上是识别活前缀的DFA
- 栈顶的状态符号包含确定句柄所需的一切信息
- 是已知的最一般的无回溯的移进-归约方法
- 能分析的文法类是预测分析法能分析的文法类的真超集
- 能及时发现语法错误
- 手工构造分析表的工作量太大

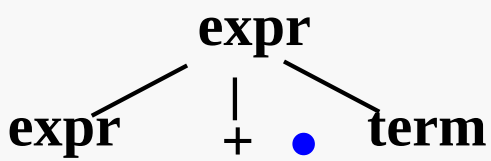
本节提纲



- **LR(k)分析技术**
  - LR分析器的简单模型
    - action, goto函数
  - 简单的LR方法（简称SLR）
    - 活前缀，识别活前缀的DFA/NFA，SLR算法
  - 规范的LR方法
  - 向前看的LR方法（简称LALR）

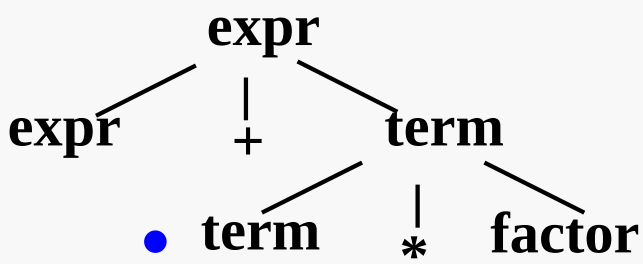
SLR分析表的构造

- SLR (Simple LR)
- LR(0)项目（简称项目）
  - 在右部的某个地方加点的产生式
  - 加点的目的是用来表示分析过程中的状态



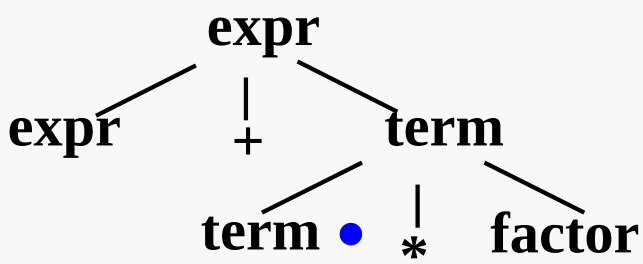
SLR分析表的构造

- SLR (Simple LR)
- LR(0)项目（简称项目）
  - 在右部的某个地方加点的产生式
  - 加点的目的是用来表示分析过程中的状态



SLR分析表的构造

- SLR (Simple LR)
- LR(0)项目（简称项目）
  - 在右部的某个地方加点的产生式
  - 加点的目的是用来表示分析过程中的状态



SLR分析表的构造

- SLR (Simple LR)
- LR(0)项目（简称项目）
  - 在右部的某个地方加点的产生式
  - 加点的目的是用来表示分析过程中的状态
- 例 $A \rightarrow XYZ$对应有四个项目

  $A \rightarrow \cdot XYZ$

  $A \rightarrow X \cdot YZ$

  $A \rightarrow XY \cdot Z$

  $A \rightarrow XYZ \cdot$
- 例 $A \rightarrow \varepsilon$只有一个项目和它对应

  $A \rightarrow \cdot$

项代表了一个可能的前缀

点的左边代表历史信息，点的右边代表展望信息。

SLR分析表的构造

- 从文法构造识别活前缀的DFA
- 从上述DFA构造分析表

构造识别活前缀的DFA

1. 拓（增）广文法（augmented grammar)

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid \mathrm{id}$$

构造识别活前缀的DFA

1. 拓（增）广文法（augmented grammar)

$E' \to E$

$E \to E + T \mid T$

$T \to T * F \mid F$

$F \to (E) \mid \mathrm{id}$

当且仅当分析器使用$E' \to E$归约时，宣告分析成功

构造识别活前缀的DFA

2. 构造LR(0)项目集规范族

项集族是若干可能前缀的集合，对应DFA的状态

I_0:

$E' \rightarrow \cdot E$

构造识别活前缀的DFA

2. 构造LR(0)项目集规范族

I_0:

$E' \rightarrow \cdot E$

$E \rightarrow \cdot E + T$

$E \rightarrow \cdot T$

求项目集的闭包closure(I)
闭包函数closure(I)
1、I的每个项目均加入closure(I)
2、如果A→α·Bβ在 closure(I)中，且B→γ是产生式，那么如果项目B→ ·γ还不在closure(I)中的话，那么把它加入。

构造识别活前缀的DFA

2. 构造LR(0)项目集规范族

I_0:

$E' \rightarrow \cdot E$

$E \rightarrow \cdot E + T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

求项目集的闭包closure(I)
闭包函数closure(I)
1、I的每个项目均加入closure(I)
2、如果A→α·Bβ在 closure(I)中，且B→γ是产生式，那么如果项目B→ ·γ还不在closure(I)中的话，那么把它加入。

构造识别活前缀的DFA

2. 构造LR(0)项目集规范族

I_0:

$E' \rightarrow \cdot E$

$E \rightarrow \cdot E + T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

$F \rightarrow (E)$

$F \rightarrow \mathbf{id}$

求项目集的闭包closure(I)
闭包函数closure(I)
1、I的每个项目均加入closure(I)
2、如果A→α·Bβ在 closure(I)中，且B→γ是产生式，那么如果项目B→ ·γ还不在closure(I)中的话，那么把它加入。

构造识别活前缀的DFA

2. 构造LR(0)项目集规范族

I_0:

$E' \to \cdot E$

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot \mathbf{id}$

核心项目:初始项目($E' \to \cdot E$)或者点不在最左边的项

非核心项目：不是初始项，且点在最左边

可以通过对核心项目求闭包来获得
为节省存储空间，可省去

构造识别活前缀的DFA

2. 构造LR(0)项目集规范族

I_0:

$E' \to \cdot E$

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

E

I_1:

$E' \to E \cdot$

$E \to E \cdot + T$

$I_1 := goto(I_0, E)$

求项目集I和文法符号X的I′ = goto(I, X)
(当输入为X时离开I状态后的转换)
1、I′ = ∅
2、对于I中的每一项A→α·Xβ,
I′ = I′ ∪ closure(A→αX·β)

构造识别活前缀的DFA

2. 构造LR(0)项目集规范族

I_0:

$E' \to \cdot E$

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot \mathrm{id}$

$\xrightarrow{E}$ I_1:

$E' \to E \cdot$

$E \to E \cdot + T$

$\xrightarrow{T}$ I_2:

$E \to T \cdot$

$T \to T \cdot * F$

构造识别活前缀的DFA

2. 构造LR(0)项目集规范族

I_0:

$E' \to \cdot E$

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot \mathrm{id}$

E → I_1:

$E' \to E \cdot$

$E \to E \cdot + T$

T → I_2:

$E \to T \cdot$

$T \to T \cdot * F$

F → I_3:

$T \to F \cdot$

构造识别活前缀的DFA

2. 构造LR(0)项目集规范族

I_0:

$E' \to \cdot E$

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot \mathbf{id}$

$\xrightarrow{(}$

I_4:

$F \to (\cdot E)$

构造识别活前缀的DFA

2. 构造LR(0)项目集规范族

I_0:

$E' \to \cdot E$

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot \mathrm{id}$

(

I_4:

$F \to (\cdot E)$

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

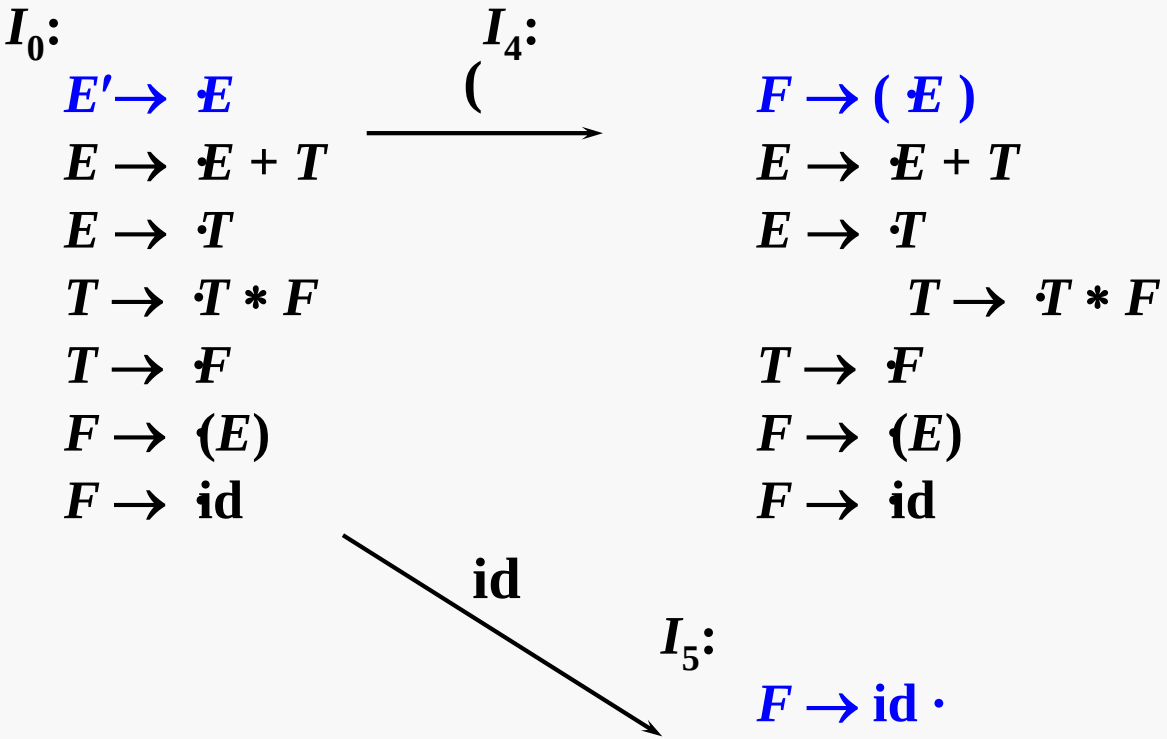
$T \to \cdot F$

$F \to \cdot (E)$

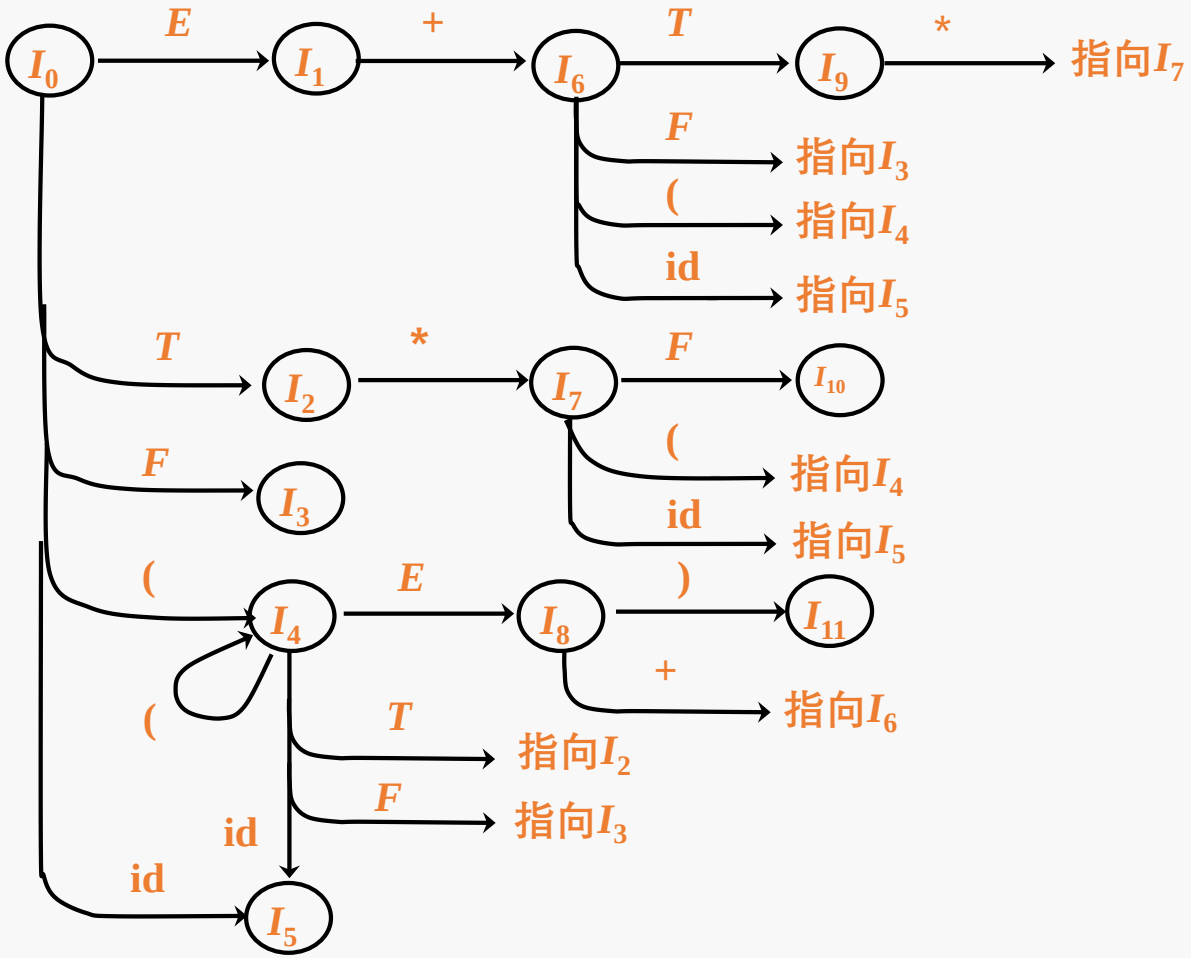
$F \to \cdot \mathrm{id}$

构造识别活前缀的DFA

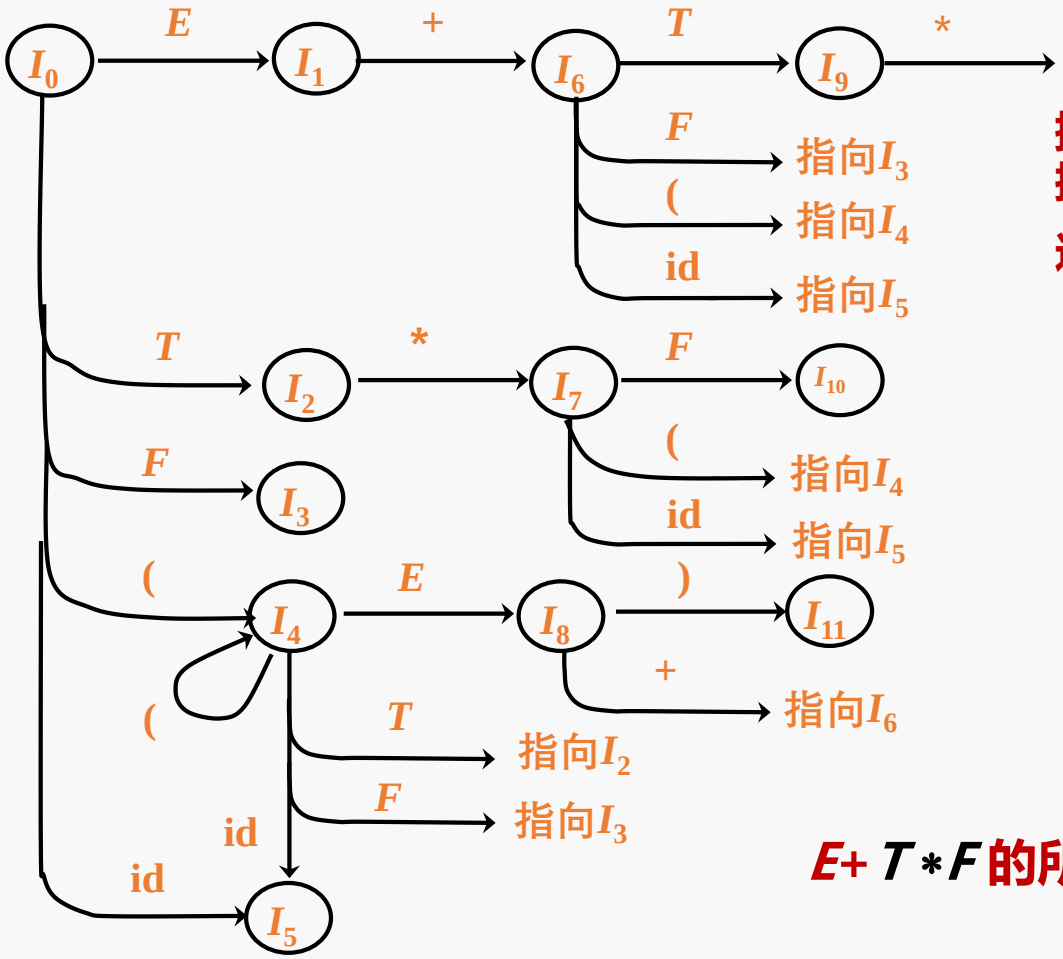
2. 构造LR(0)项目集规范族



构造识别活前缀的DFA



构造识别活前缀的DFA



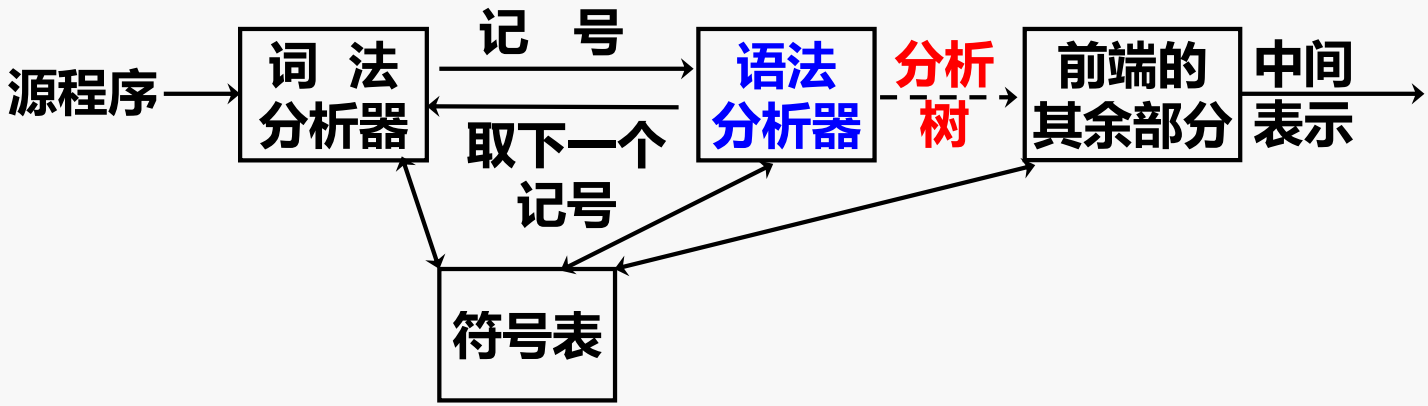
把所有状态都作为接受状态

这是一个DFA

$$E' \Rightarrow E$$
$$\Rightarrow E+T$$
$$\Rightarrow E+T*F$$
$$\Rightarrow E+T*\text{id}$$
$$\Rightarrow E+T*F*\text{id}$$

$E+T*F$的所有前缀都可接受

本节提纲



- **LR(k)分析技术**
  - LR分析器的简单模型
    - action, goto函数
  - 简单的LR方法（简称SLR）
    - 活前缀，识别活前缀的DFA/NFA，SLR算法
  - 规范的LR方法
  - 向前看的LR方法（简称LALR）

SLR分析表的构造

- 从文法构造识别活前缀的DFA
- 从上述DFA构造分析表

从DFA构造SLR分析表

- 状态 i 从 I_i 构造，它的 $action$ 函数如下确定：
  - 如果 $[A \to \alpha \cdot a\beta]$ 在 I_i 中，并且 $goto(I_i, a) = I_j$，那么置 $action[i, a]$ 为 sj
  - 如果 $[A \to \alpha \cdot]$ 在 I_i 中，那么对 $FOLLOW(A)$ 中的所有 a，置 $action[i, a]$ 为 rj，j 是产生式 $A \to \alpha$ 的编号
  - 如果 $[S' \to S \cdot]$ 在 I_i 中，那么置 $action[i, \$]$ 为接受 acc
  - 上面的 a 是终结符

- **如果出现动作冲突，那么该文法就不是SLR(1)文法**

从DFA构造SLR分析表

- 状态 i 从 I_i 构造，它的*action*函数如下确定：
  - 此处省略，参见上页
- 使用下面规则构造状态i的*goto*函数：
  - 对所有的非终结符A，如果$goto(I_i, A) = I_j$，那么$goto[i, A] = j$

从DFA构造SLR分析表

- 状态 i 从 I_i 构造，它的*action*函数如下确定：
  - 此处省略，参见上页
- 使用下面规则构造状态i的*goto*函数：
  - 此处省略，参见上页
- 分析器的初始状态是包含$[S' \rightarrow \cdot S]$的项目集对应的状态

不能由上面两步定义的条目都置为error

SLR分析器

例 (1) $E \to E + T$ (2) $E \to T$

(3) $T \to T * F$ (4) $T \to F$

(5) $F \to (E)$ (6) $F \to$ id

si 移进当前输入符号和状态*i*

rj 按第*j*个产生式进行归约

acc 接受

状态	动作 action						转移 goto		
	id	+	*	(	)	$	*E*	*T*	*F*
0	*s*5			*s*4			1	2	3
1		*s*6				*acc*			
2		*r*2	*s*7		*r*2	*r*2			
3		*r*4	*r*4		*r*4	*r*4			
4	*s*5			*s*4			8	2	3
5		*r*6	*r*6		*r*6	*r*6			
6	*s*5			*s*4				9	3

SLR(1)文法

- 一个上下文无关文法G，通过上述算法构造出SLR语法分析表，且表项中没有移进/归约或者归约/归约冲突，那么G就是SLR(1)文法。
- 1代表了当看到某个产生式右部时，只需要再向前看1个符号就可决定是否用该式进行归约。
- 通常可以省略1，写作SLR文法

SLR分析器——解释

- 例　I_2:

  (2)$E \to T \cdot$

  (3)$T \to T \cdot * F$

  - 归约：因为FOLLOW(E) = {$, +,)},

    所以 $action[2, \$]=action[2, +]=action[2,)]=r2$

  - 移进：因为圆点在中间，且点后面是终结符，

    所以，$action[2, *] = s7$

活前缀的概念-revisit

- LR(0)自动机刻画了可能出现在文法符号栈中的所有串；
- 栈中的内容一定是某个最右句型的前缀；
- 但是不是所有前缀都会出现在栈中。

$$E \Rightarrow^*_{rm} F * \mathbf{id} \Rightarrow_{rm} (E) * \mathbf{id}$$

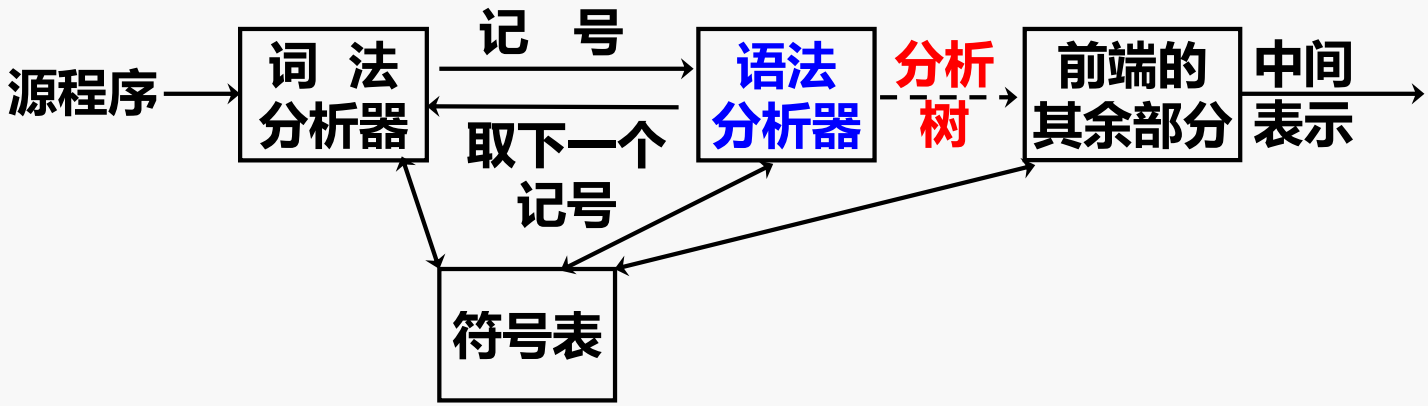
活前缀的概念-revisit

- LR(0)自动机刻画了可能出现在文法符号栈中的所有串；
- 栈中的内容一定是某个最右句型的前缀；
- 但是不是所有前缀都会出现在栈中。

$$E \Rightarrow^*_{rm} F * \mathbf{id} \Rightarrow_{rm} (E) * \mathbf{id}$$

- 栈中只能出现(, (*E*, (*E*)，而不会出现(*E*)*
  - 因为看到*时，(*E*)是句柄，会被归约成为*F*

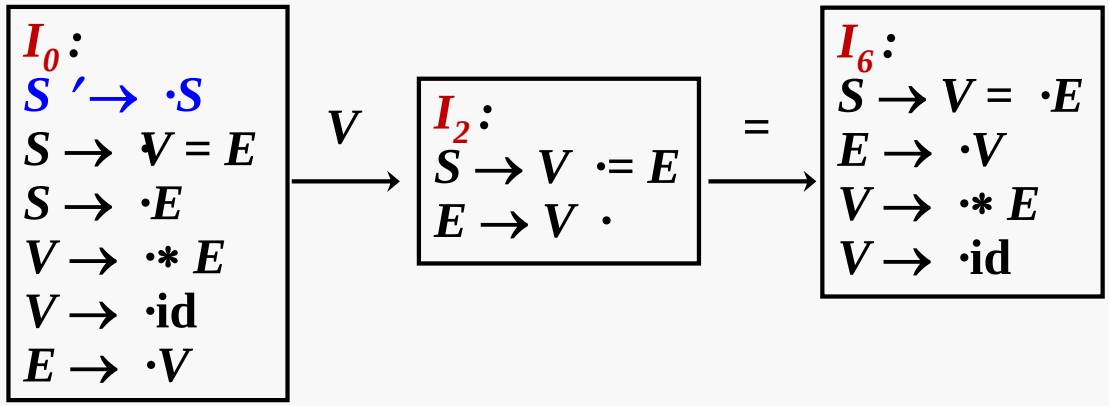
本节提纲



- **LR(k)分析技术**
  - LR分析器的简单模型
    - action, goto函数
  - 简单的LR方法（简称SLR）
    - 活前缀，识别活前缀的DFA/NFA，SLR算法
  - 规范的LR方法
  - 向前看的LR方法（简称LALR）

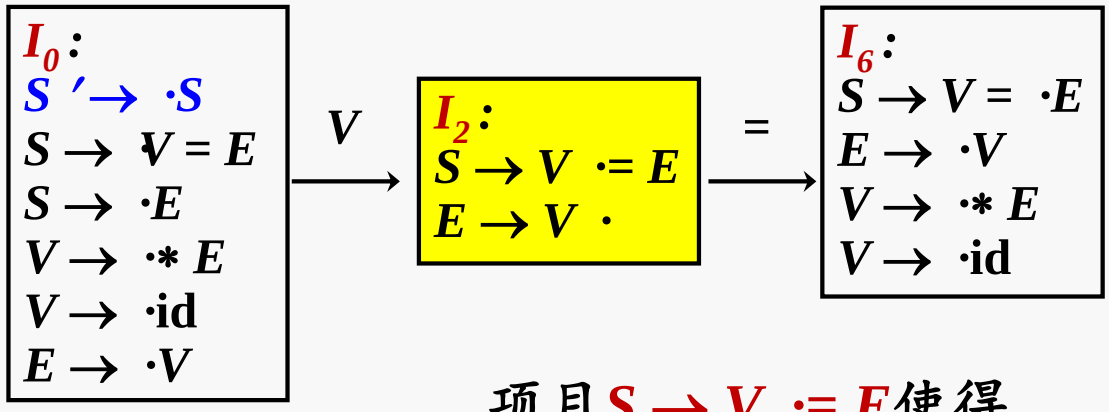
SLR(1)文法的描述能力有限

$S \to V = E$
$S \to E$
$V \to * E$
$V \to \mathrm{id}$
$E \to V$



SLR(1)文法的描述能力有限

$S \to V = E$
$S \to E$
$V \to * E$
$V \to \text{id}$
$E \to V$

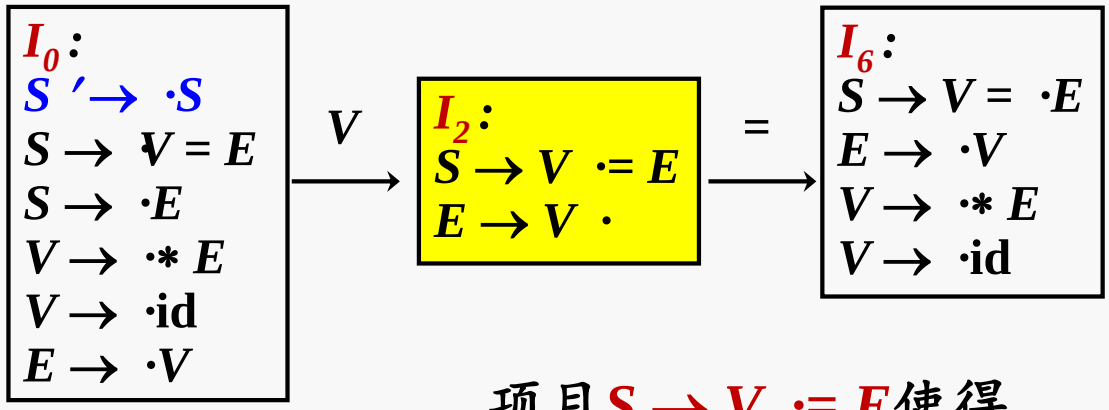


项目$S \to V \cdot = E$使得
action[2, =] = s6

项目$E \to V$ 使得
action[2, =] = r5
因为Follow(E)={=, $}

SLR(1)文法的描述能力有限

$S \to V = E$
$S \to E$
$V \to * E$
$V \to \text{id}$
$E \to V$



项目 $S \to V \cdot = E$ 使得
action[2, =] = s6

项目 $E \to V$ 使得
action[2, =] = r5
因为Follow(E)={=, $}

产生移进-归约冲突，但该文法不是二义的。

规范的LR分析

- 目标：在识别活前缀DFA的状态中，增加信息，排除一些不正确的归约操作
- 方法：添加了前向搜索符
  - 一个项目$A\rightarrow\alpha\beta$，如果最终用这个产生式进行归约之后，期望看见的符号是a，则这个加点项的前向搜索符是a。
  - 上述项目可以写成：$A\rightarrow\alpha\beta,\ a$
- 与SLR(1)分析的区别
  - 项目集的定义发生了改变：LR(0) => LR(1)
  - closure(I) 和GOTO函数需要修改

规范的LR分析

- 目标：在识别活前缀DFA的**状态**中，**增加信息**，**排除一些**不正确的**归约**操作
- 方法：添加了前向搜索符
  - 一个项目$A\rightarrow\alpha\beta$，如果最终用这个产生式进行归约之后，期望看见的符号是a，则这个加点项的前向搜索符是a。
  - 上述项目可以写成：$A\rightarrow\alpha\beta,\ a$
- 与SLR(1)分析的区别
  - 项目集的定义发生了改变：LR(0) => LR(1)
  - closure(I) 和GOTO函数需要修改

规范的LR分析

- **目标：在识别活前缀DFA的状态中，增加信息，排除一些不正确的归约操作**
- **方法：添加了前向搜索符**
  - 一个项目$A\rightarrow\alpha\beta$，如果最终用这个产生式进行归约之后，期望看见的符号是a，则这个加点项的前向搜索符是a。
  - 上述项目可以写成：$A\rightarrow\alpha\beta,\ a$
- **与SLR(1)分析的区别**
  - 项目集的定义发生了改变：LR(0) => LR(1)
  - closure(I) 和GOTO函数需要修改

规范的LR分析

- **LR(1)项目：**

$$[A \rightarrow \alpha \cdot \beta, a]$$

  - 当项目由两个分量组成，第一分量为SLR中的项，第二分量为搜索符（向前看符号）
  - LR(1)中的1代表了搜索符a的长度

规范的LR分析

- **LR(1)项目：**

  $[A \rightarrow \alpha \cdot \beta, a]$

  - 当项目由两个分量组成，第一分量为SLR中的项，第二分量为搜索符（向前看符号）
  - LR(1)中的1代表了搜索符a的长度

- **使用注意事项：**
  - 当β不为空时，a不起作用
  - 当β为空时，如果下一个输入符号是a，将按照$A \rightarrow \alpha$进行归约
    - a的集合是FOLLOW(A)的子集

规范的LR分析

- **LR(1)项目：**

  $[A \to \alpha \cdot \beta, a]$

  - 当项目由两个分量组成，第一分量为SLR中的项，第二分量为搜索符（向前看符号）
  - LR(1)中的1代表了搜索符a的长度

- **LR(1)项目$[A \to \alpha \cdot \beta, a]$对活前缀$\gamma$有效：**

  - 如果存在着推导$S \Rightarrow^*_{rm} \delta A w \Rightarrow_{rm} \delta\alpha\beta w$，其中：
    - $\gamma = \delta\alpha$;
    - a是w的第一个符号，或者w是ε且a是$

规范的LR分析：举例

- 例 $S \to BB$

  $B \to bB \mid a$

LR(1)项目$[A \to \alpha \cdot \beta, a]$对活前缀$\gamma$有效：
存在着推导$S \Rightarrow^*_{rm} \delta Aw \Rightarrow_{rm} \delta\alpha\beta w$，其中：
$\gamma = \delta\alpha$;
a是w的第一个符号，或者w是ε且a是\$

从最右推导$S \Rightarrow^*_{rm} bbBba \Rightarrow_{rm} bbbBba$看出：

令$A = B,\ \alpha = b,\ \beta = B,\ \delta = bb,\ \gamma = \delta\alpha = bbb,\ w = ba$

$[B \to b \cdot B, b]$对活前缀$\gamma = bbb$是有效的

规范的LR分析：举例

- 例 $S \to BB$

  $B \to bB \mid a$

LR(1)项目$[A \to \alpha \cdot \beta, a]$对活前缀$\gamma$有效：
存在着推导$S \Rightarrow^*_{rm} \delta Aw \Rightarrow_{rm} \delta\alpha\beta w$，其中：
$\gamma = \delta\alpha$;
a是w的第一个符号，或者w是ε且a是\$

从最右推导$S \Rightarrow^*_{rm} bbBba \Rightarrow_{rm} bbbBba$看出：

令$A = B,\ \alpha = b,\ \beta = B,\ \delta = bb,\ \gamma = \delta\alpha = bbb,\ w = ba$

$[B \to b \cdot B, b]$对活前缀$\gamma = bbb$是有效的

规范的LR分析—步骤

- 构造LR(1)项目集规范族
  - 也就是构造识别活前缀的DFA
- 构造规范的LR分析表
  - 状态之间的转换关系

构造LR(1)项目集规范族

基础运算1：计算闭包CLOSURE(I)

- I中的任何项目都属于CLOSURE(I)
- 若有项目 $[A\rightarrow\alpha B\beta, a]$在CLOSURE(I)中，而$B\rightarrow\gamma$是文法中的产生式，$b$是FIRST($\beta a$)中的元素，则$[B\rightarrow\gamma, b]$也属于CLOSURE(I)

保证在用$B\rightarrow\gamma$进行归约后，
- 出现的输入字符b是句柄$\alpha B\beta$中B的后继符号
- 或者是$\alpha B\beta$归约为A后可能出现的终结符。

构造LR(1)项目集规范族

- **基础运算2：通过GOTO(I,X)算CLOSURE(J)**
  - 将J置为空集
  - 若有项目 $[A\rightarrow\alpha\cdot X\beta, a]$在I中，那么将项目 $[A\rightarrow\alpha X\cdot\beta, a]$放入J中
  - 计算并返回CLOSURE(J)

注意：GOTO(I,X)中的X可以是终结符或非终结符

构造LR(1)项目集规范族

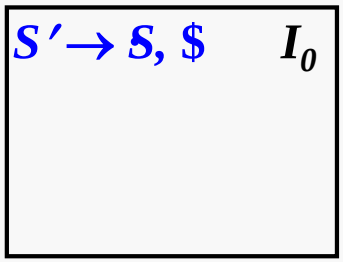
具体算法

- 初始项目集I_0：

  I_0 = CLOSURE([$S' \rightarrow S$, \$]) 将\$作为向前的搜索符

- 设C为最终返回的项目集族，初始为C={I_0}
- 重复以下步骤
  - 对C中的任意项目集I，重复
    - ✓对每一个文法符号X(终结符或非终结符)
      - 如果GOTO(I,X) ≠ ∅ 且 GOTO(I,X) ∉ C，那么将GOTO(I,X)放入C
      - 注：上述GOTO(I,X)是上一页ppt中计算闭包的GOTO
  - 当C中项目集不再增加为止

构造LR(1)项目集族：举例



步骤一：从初始项开始

构造LR(1)项目集族：举例

$S' \rightarrow S, \$$ I_0

$S \rightarrow BB, \$, b/a$

步骤二：计算非核心项目的第一个分量

构造LR(1)项目集族：举例

$S' \rightarrow S, \$$ I_0

$S \rightarrow BB, \$$

步骤三：通过FIRST(ε$) 计算非核心项目的第二个分量

构造LR(1)项目集族：举例

$S' \rightarrow S,\ \$$ I_0
$S \rightarrow BB,\ \$$
$B \rightarrow bB,\ b/a$
$B \rightarrow a,\ b/a$

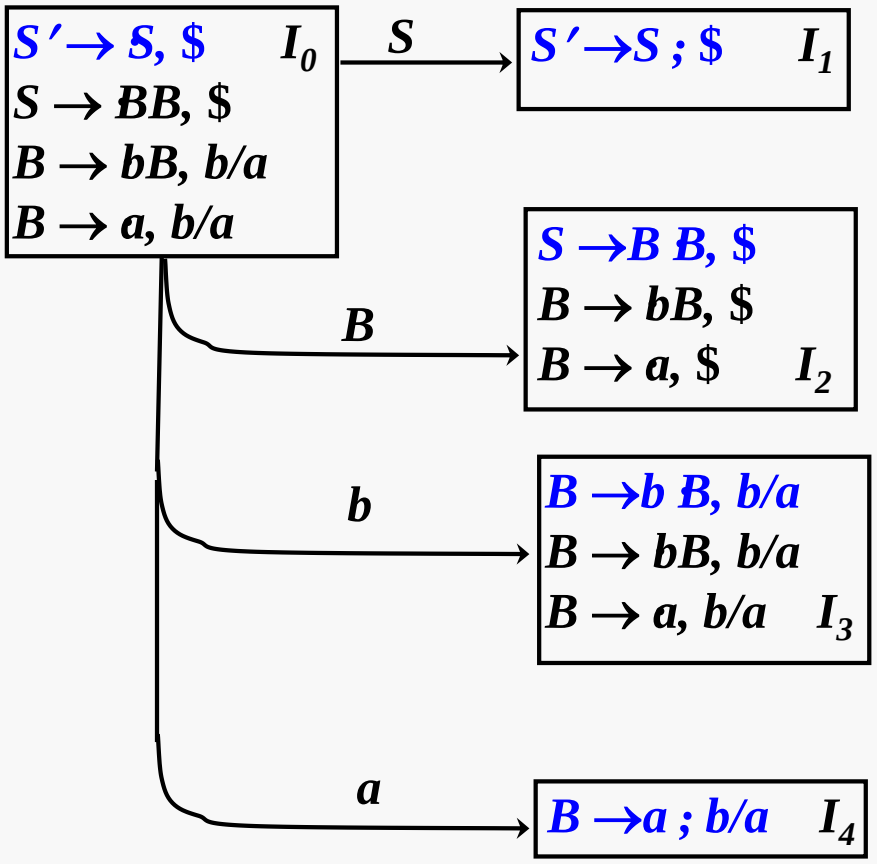
步骤二：计算非核心项目的第一个分量

构造LR(1)项目集族：举例

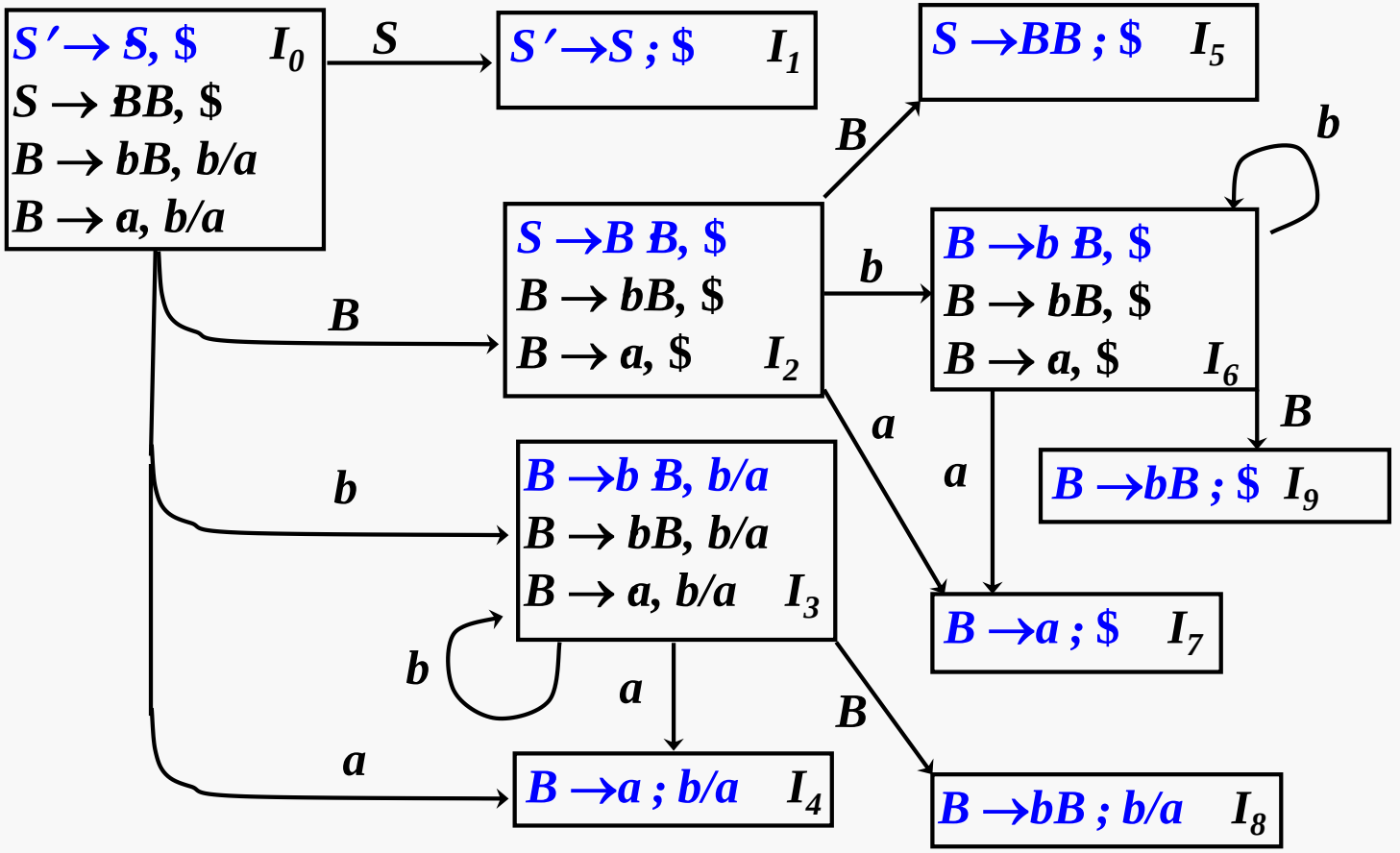
$S' \rightarrow S, \$$	I_0
$S \rightarrow BB, \$$	
$B \rightarrow bB, b/a$	
$B \rightarrow a, b/a$	

步骤三：通过FIRST(B$) 计算非核心项目的第二个分量

构造LR(1)项目集族：举例



构造LR(1)项目集族：举例



构造规范的LR分析表

- **构造识别拓广文法G′活前缀的DFA**
  - 基于LR(1)项目族来构造
- **状态*i*的*action*函数如下确定：**
  - 如果$[A \to \alpha \cdot a\beta, b]$在$I_i$中，且$goto(I_i, a) = I_j$，那么置$action[i, a]$为$sj$ (此时，不看b)
  - 如果$[A \to \alpha \cdot, a]$在I_i中，且$A \neq S'$，那么置$action[i, a]$为rj (此时，不再看FOLLOW(A))
  - 如果$[S' \to S \cdot, \$]$在$I_i$中，那么置$action[i, \$] = acc$

如果上述构造出现了冲突，那么文法就不是LR(1)的

构造规范的LR分析表

- 构造识别拓广文法G′活前缀的DFA
- 状态*i*的*action*函数如下确定：
  - 参见上页ppt
- 状态*i*的*goto*函数如下确定：
  - 如果$goto(I_i, A) = I_j$, 那么$goto[i, A] = j$

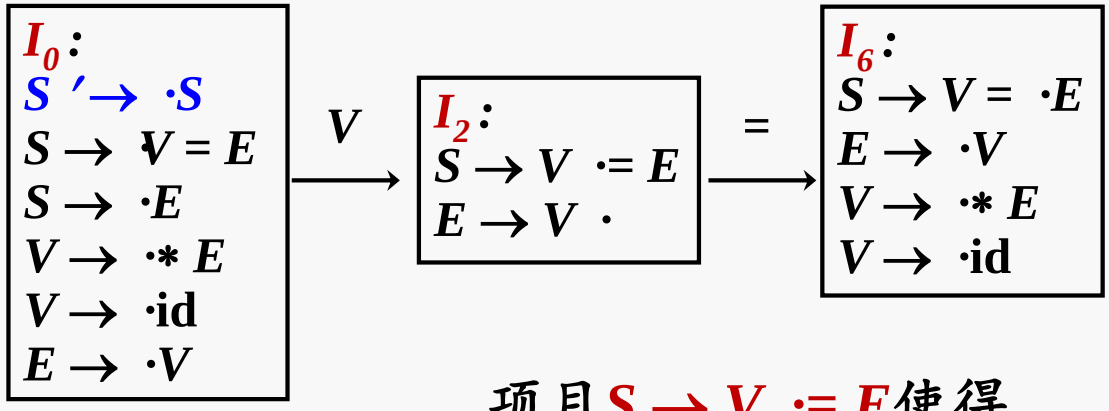
构造规范的LR分析表

- 构造识别拓广文法G′活前缀的DFA
- 状态*i*的*action*函数如下确定：
  - 参见上页ppt
- 状态*i*的*goto*函数如下确定：
  - 如果$goto(I_i, A) = I_j$, 那么$goto[i, A] = j$
- 分析器的初始状态是包含$[S' \rightarrow \cdot S, \$]$的项目集对应的状态

用上面规则未能定义的所有条目都置为**error**

非SLR(1)文法–revisit

$S \to V = E$
$S \to E$
$V \to * E$
$V \to \text{id}$
$E \to V$



项目$S \to V \cdot = E$使得
action[2, =] = s6

项目$E \to V$ 使得
action[2, =] = r5
因为Follow(E)={=, $}

产生移进-归约冲突，但该文法不是二义的。

非SLR(1)但是LR(1)文法

$S \rightarrow V = E$
$S \rightarrow E$
$V \rightarrow * E$
$V \rightarrow \text{id}$
$E \rightarrow V$

I_0 :
$S' \rightarrow \cdot S, \$$
$S \rightarrow \cdot V = E, \$$
$S \rightarrow \cdot E, \$$
$V \rightarrow \cdot * E, =$
$V \rightarrow \cdot \text{id}, =$
$E \rightarrow \cdot V, \$$
$V \rightarrow \cdot * E, \$$
$V \rightarrow \cdot \text{id}, \$$

非SLR(1)但是LR(1)文法

$S \to V = E$
$S \to E$
$V \to * E$
$V \to \text{id}$
$E \to V$

I_0:
$S' \to \cdot S, \$$
$S \to \cdot V = E, \$$
$S \to \cdot E, \$$
$V \to \cdot * E, =$
$V \to \cdot \text{id}, =$
$E \to \cdot V, \$$
$V \to \cdot * E, \$$
$V \to \cdot \text{id}, \$$

计算闭包：

定义里：　$[A \to \alpha B \beta, a]$

这里：　$[S' \to \varepsilon \cdot S \varepsilon, \$]$

FIRST(βa)

⇓

FIRST(ε$)={$}

非SLR(1)但是LR(1)文法

$S \to V = E$
$S \to E$
$V \to * E$
$V \to \text{id}$
$E \to V$

I_0:
$S' \to \cdot S, \$$
$S \to \cdot V = E, \$$
$S \to \cdot E, \$$
$V \to \cdot * E, =$
$V \to \cdot \text{id}, =$
$E \to \cdot V, \$$
$V \to \cdot * E, \$$
$V \to \cdot \text{id}, \$$

计算闭包：

定义里：　$[A \to \alpha B \beta, a]$　　FIRST(βa)

这里：　$[S' \to \varepsilon \cdot S \varepsilon, \$]$　　FIRST(ε$)={$}

非SLR(1)但是LR(1)文法

$S \to V = E$
$S \to E$
$V \to * E$
$V \to \text{id}$
$E \to V$

I_0:
$S' \to \cdot S, \$$
$S \to \cdot V = E, \$$
$S \to \cdot E, \$$
$V \to \cdot * E, =$
$V \to \cdot \text{id}, =$
$E \to \cdot V, \$$
$V \to \cdot * E, \$$
$V \to \cdot \text{id}, \$$

计算闭包：

定义里：　$[A \to \alpha \cdot B \beta, a]$

这里：　$[S \to \varepsilon \cdot V = E, \$]$

FIRST(βa)

FIRST(= E$)={=}

非SLR(1)但是LR(1)文法

$S \to V = E$
$S \to E$
$V \to * E$
$V \to \text{id}$
$E \to V$

I_0:
$S' \to \cdot S, \$$
$S \to \cdot V = E, \$$
$S \to \cdot E, \$$
$V \to \cdot * E, =$
$V \to \cdot \text{id}, =$
$E \to \cdot V, \$$
$V \to \cdot * E, \$$
$V \to \cdot \text{id}, \$$

计算闭包：

定义里：　$[A \to \alpha \cdot B \beta, a]$

这里：　$[S \to \varepsilon \cdot V = E, \$]$

FIRST(βa)

FIRST(= E$)={=}

非SLR(1)但是LR(1)文法

$S \to V = E$
$S \to E$
$V \to * E$
$V \to \text{id}$
$E \to V$

I_0:
$S' \to \cdot S, \$$
$S \to \cdot V = E, \$$
$S \to \cdot E, \$$
$V \to \cdot * E, =$
$V \to \cdot \text{id}, =$
$E \to \cdot V, \$$
$V \to \cdot * E, \$$
$V \to \cdot \text{id}, \$$

计算闭包：

定义里：　$[A \to \alpha \cdot B \beta, a]$

这里：　$[S \to \varepsilon \cdot V = E, \$]$

FIRST(βa)

FIRST(= E$)={=}

非SLR(1)但是LR(1)文法

$S \to V = E$
$S \to E$
$V \to * E$
$V \to \text{id}$
$E \to V$

I_0:
$S' \to \cdot S, \$$
$S \to \cdot V = E, \$$
$S \to \cdot E, \$$
$V \to \cdot * E, =$
$V \to \cdot \text{id}, =$
$E \to \cdot V, \$$
$V \to \cdot * E, \$$
$V \to \cdot \text{id}, \$$

计算闭包：

定义里：　$[A \to \alpha B \beta, a]$

这里：　$[S \to \varepsilon E \varepsilon, \$]$

FIRST(βa)

⇓

FIRST(ε$)={$}

非SLR(1)但是LR(1)文法

$S \to V = E$
$S \to E$
$V \to * E$
$V \to \text{id}$
$E \to V$

I_0 :
$S' \to \cdot S, \$$
$S \to \cdot V = E, \$$
$S \to \cdot E, \$$
$V \to \cdot * E, =$
$V \to \cdot \text{id}, =$
$E \to \cdot V, \$$
$V \to \cdot * E, \$$
$V \to \cdot \text{id}, \$$

非SLR(1)但是LR(1)文法

$S \to V = E$
$S \to E$
$V \to * E$
$V \to \text{id}$
$E \to V$

I_0:
$S' \to \cdot S, \$$
$S \to \cdot V = E, \$$
$S \to \cdot E, \$$
$V \to \cdot * E, =/\$$
$V \to \cdot \text{id}, =/\$$
$E \to \cdot V, \$$

可通过合并搜索符简化

非SLR(1)但是LR(1)文法

$S \to V = E$
$S \to E$
$V \to * E$
$V \to \text{id}$
$E \to V$

I_0:
$S' \to \cdot S, \$$
$S \to \cdot V = E, \$$
$S \to \cdot E, \$$
$V \to \cdot * E, =/\$$
$V \to \cdot \text{id}, =/\$$
$E \to \cdot V, \$$

V ⟶

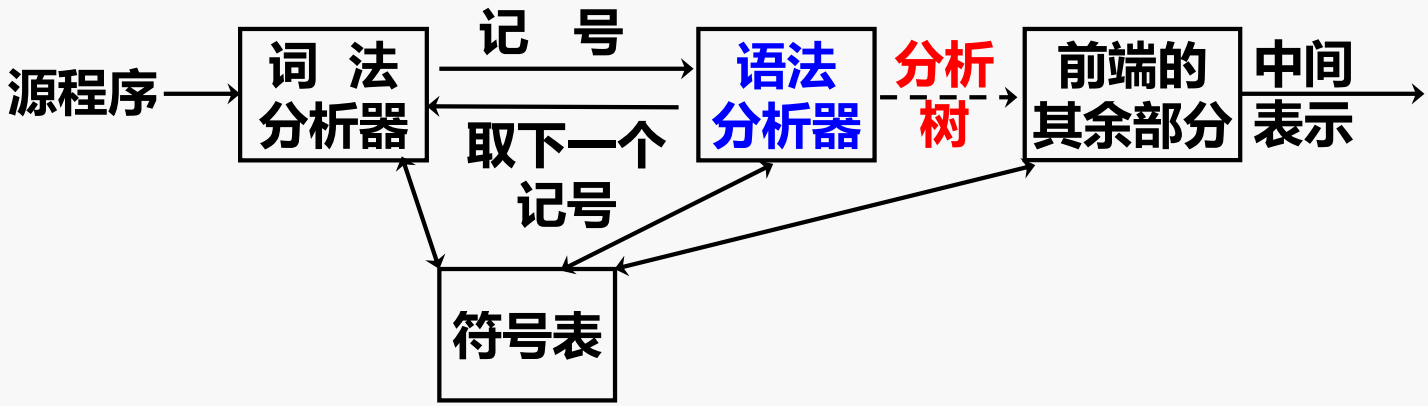
I_2:
$S \to V \cdot = E, \$$
$E \to V \cdot, \$$

项目[$S \to V \cdot = E, \$$]使得
action[2, =] = s6

项目[$E \to V \cdot ; \$$]使得
action[2, $] = r5
因为{$}是Follow(E)={=, $}的真子集

每一个SLR(1)文法都是LR(1)的

本节提纲



- **LR(k)分析技术**
  - LR分析器的简单模型
    - action, goto函数
  - 简单的LR方法（简称SLR）
    - 活前缀，识别活前缀的DFA/NFA，SLR算法
  - 规范的LR方法
  - 向前看的LR方法（简称LALR）

LALR分析方法

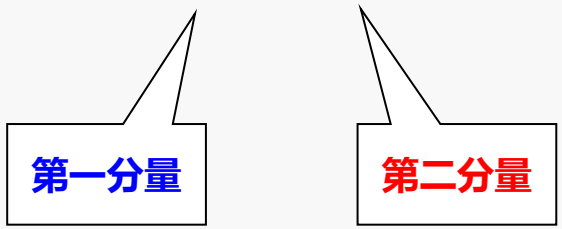
- **研究LALR的原因**

  规范LR分析表的状态数偏多

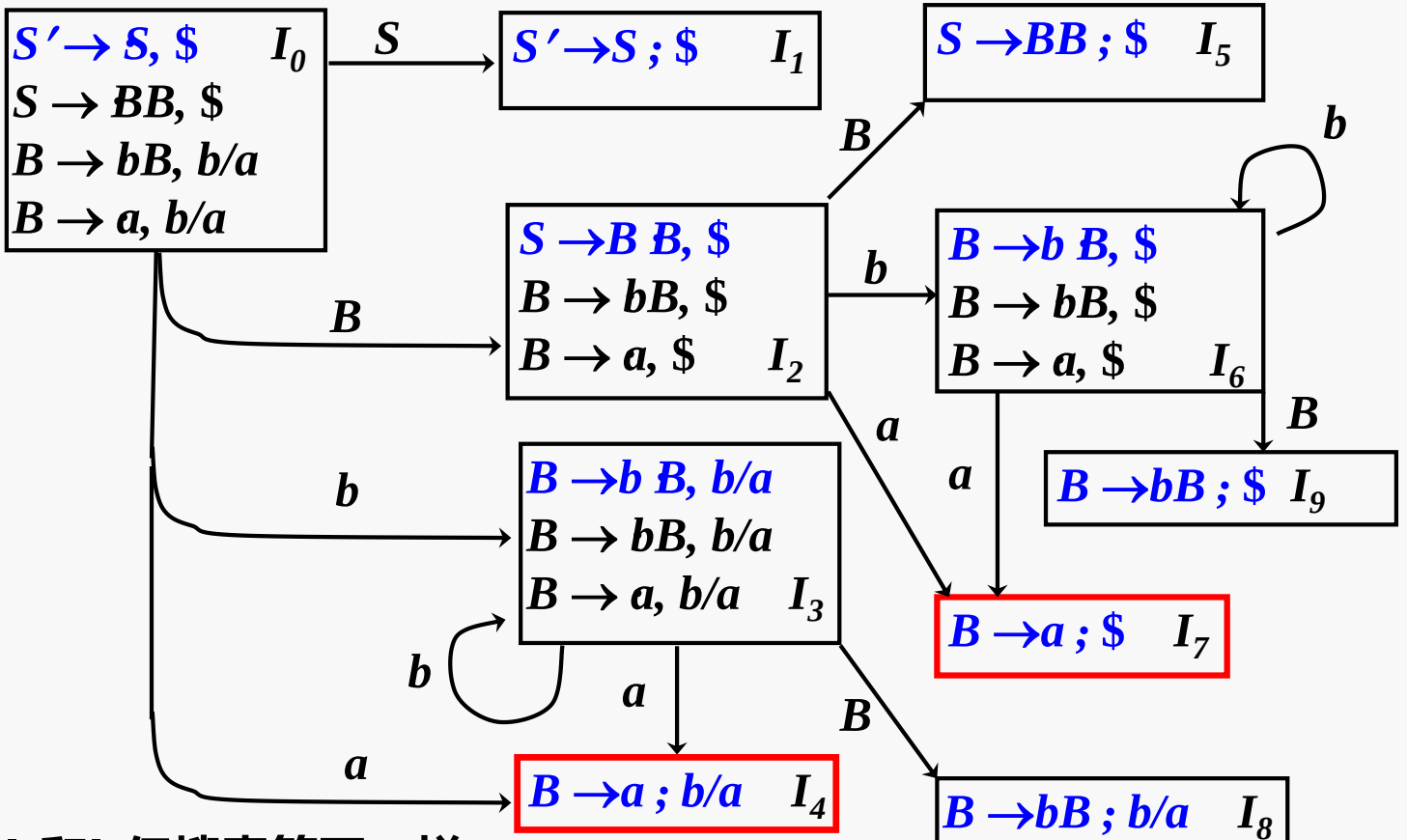
- **LALR特点**
  - LALR和SLR的分析表有同样多的状态，比规范LR分析表要小得多
  - LALR的能力介于SLR和规范LR之间
  - LALR的能力在很多情况下已经够用
- **LALR分析表构造方法**
  - 通过合并规范LR(1)项目集来得到

LALR分析方法

- 合并识别 LR(1)文法的活前缀的DFA中的相同核心项目集(同心项目集，注意：不是项)
- 同心的LR(1)项目集
  - 核心：项目集中第一分量的集合
  - 略去搜索符后它们是相同的集合
  - 例： [B → ḃB, $] 与 [B → ḃB, b/a]

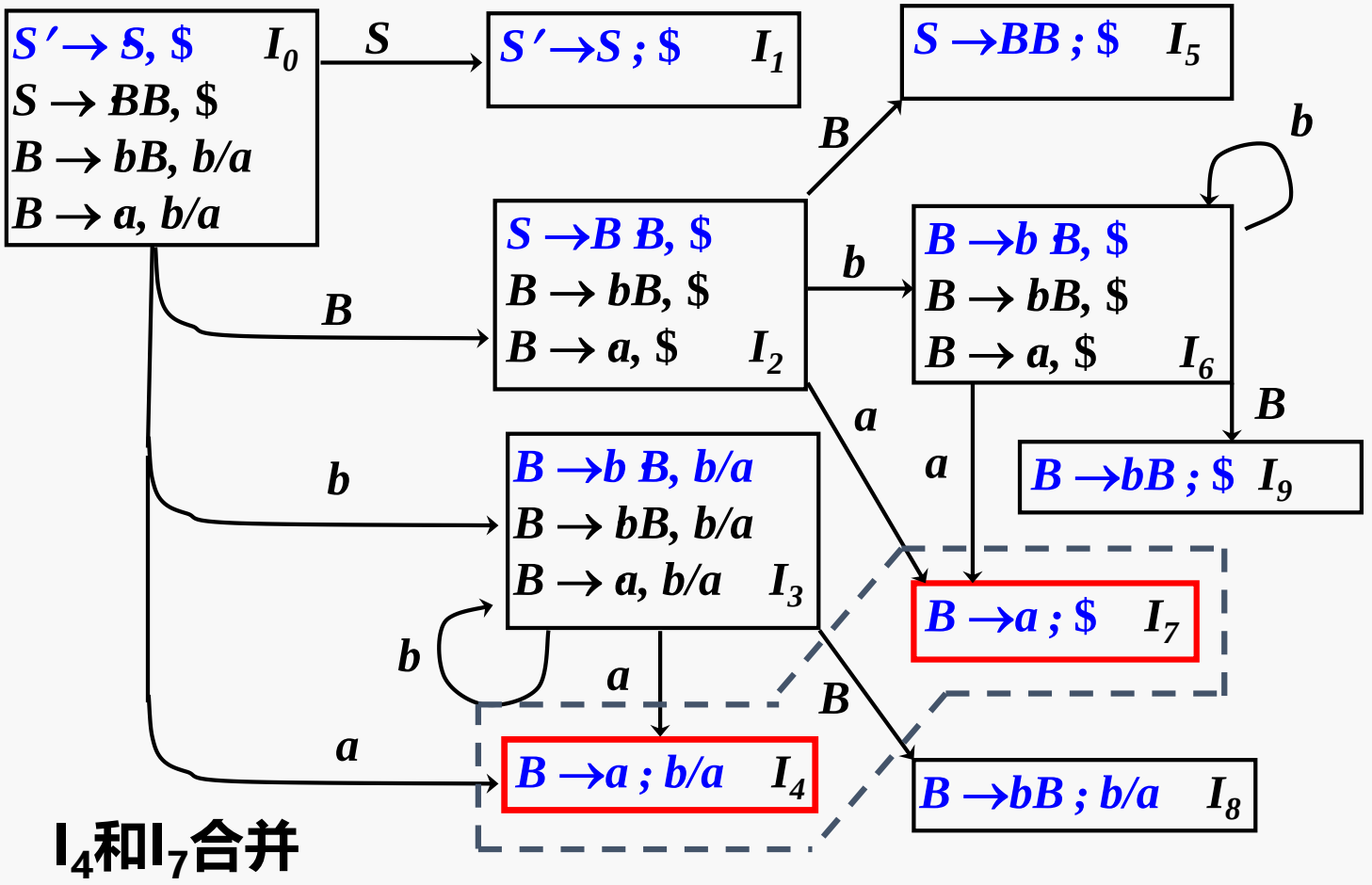


识别活前缀的DFA



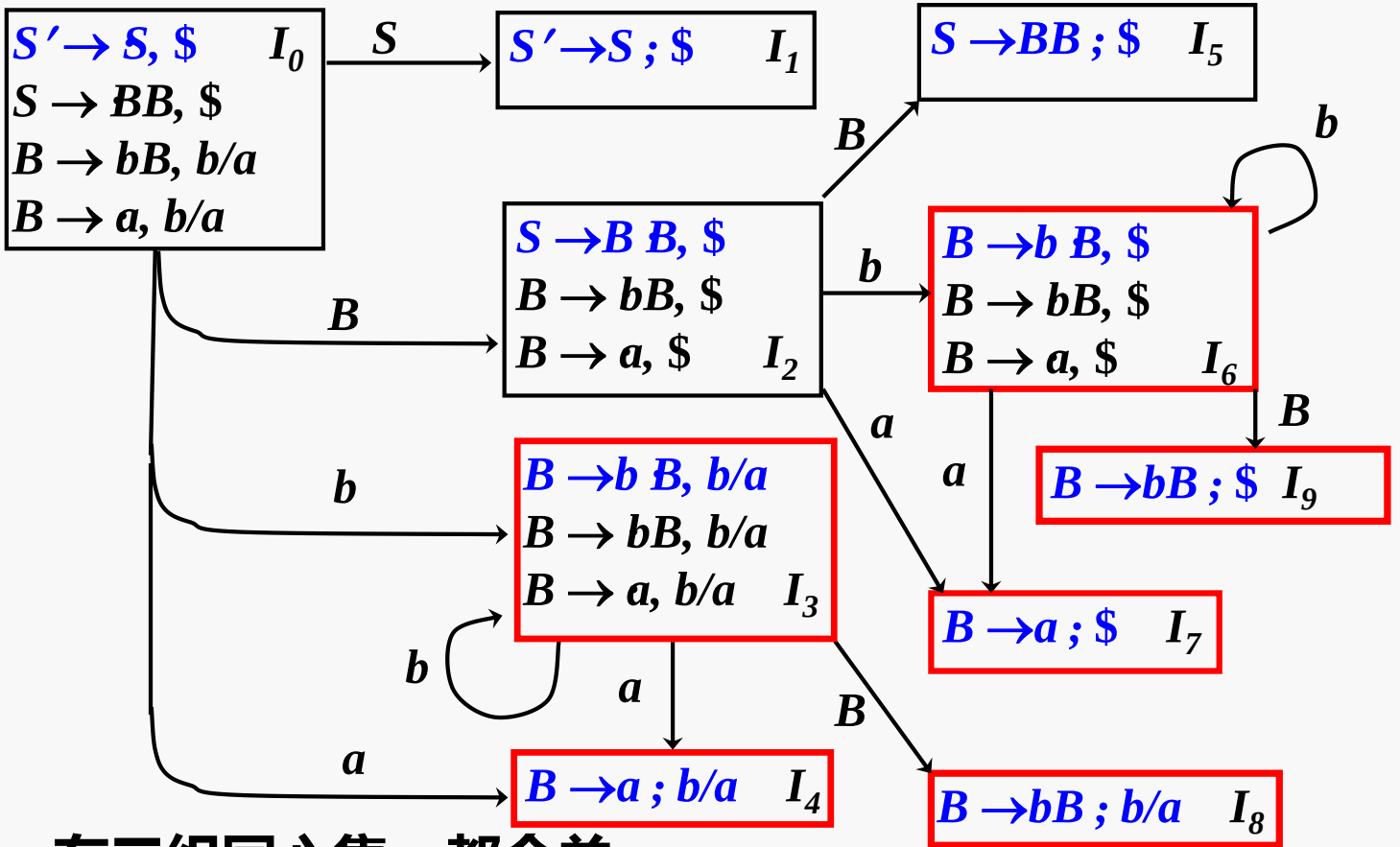
I_4和I_7仅搜索符不一样

合并同心项目集



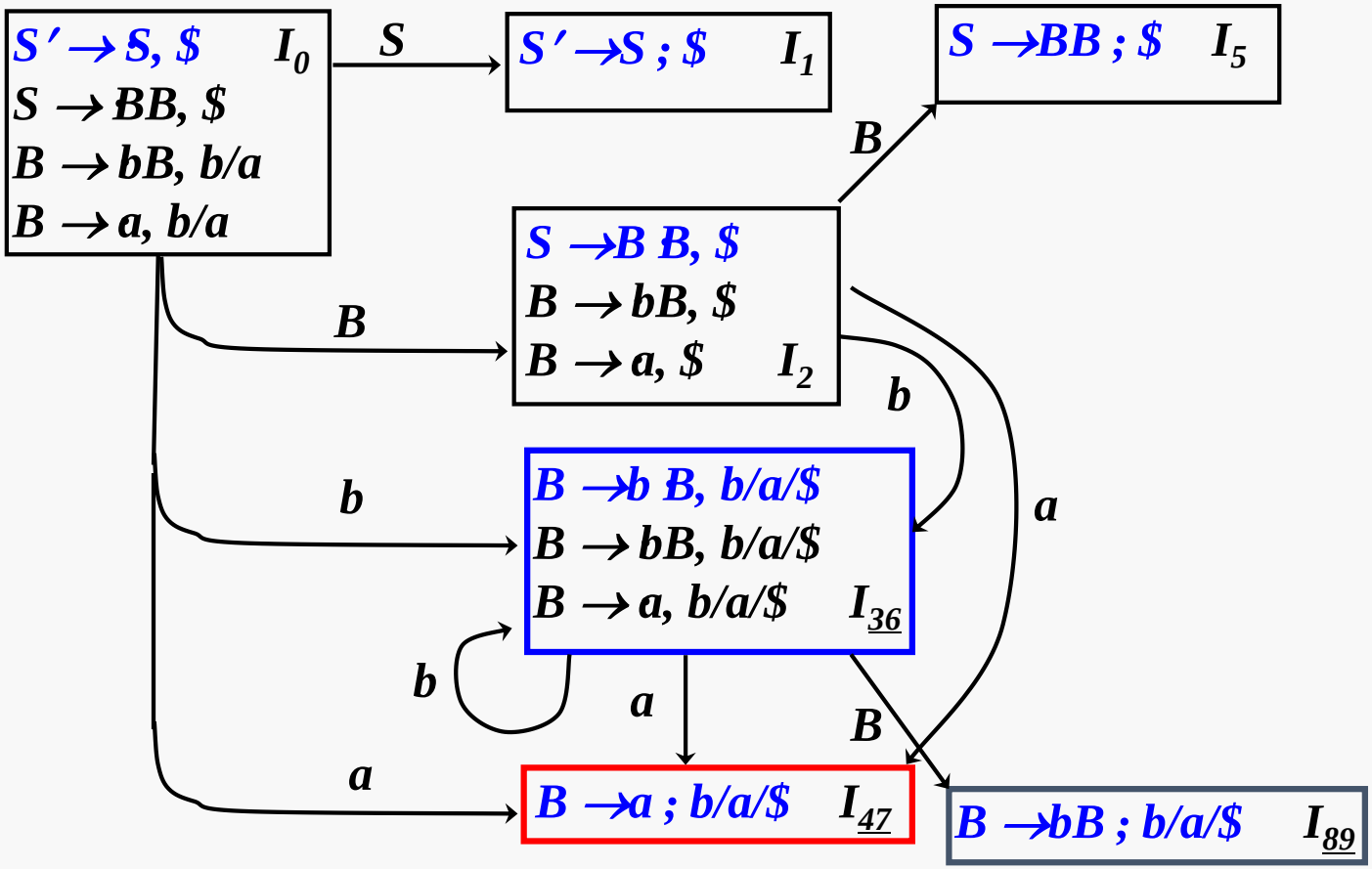
I_4和I_7合并

合并同心项目集

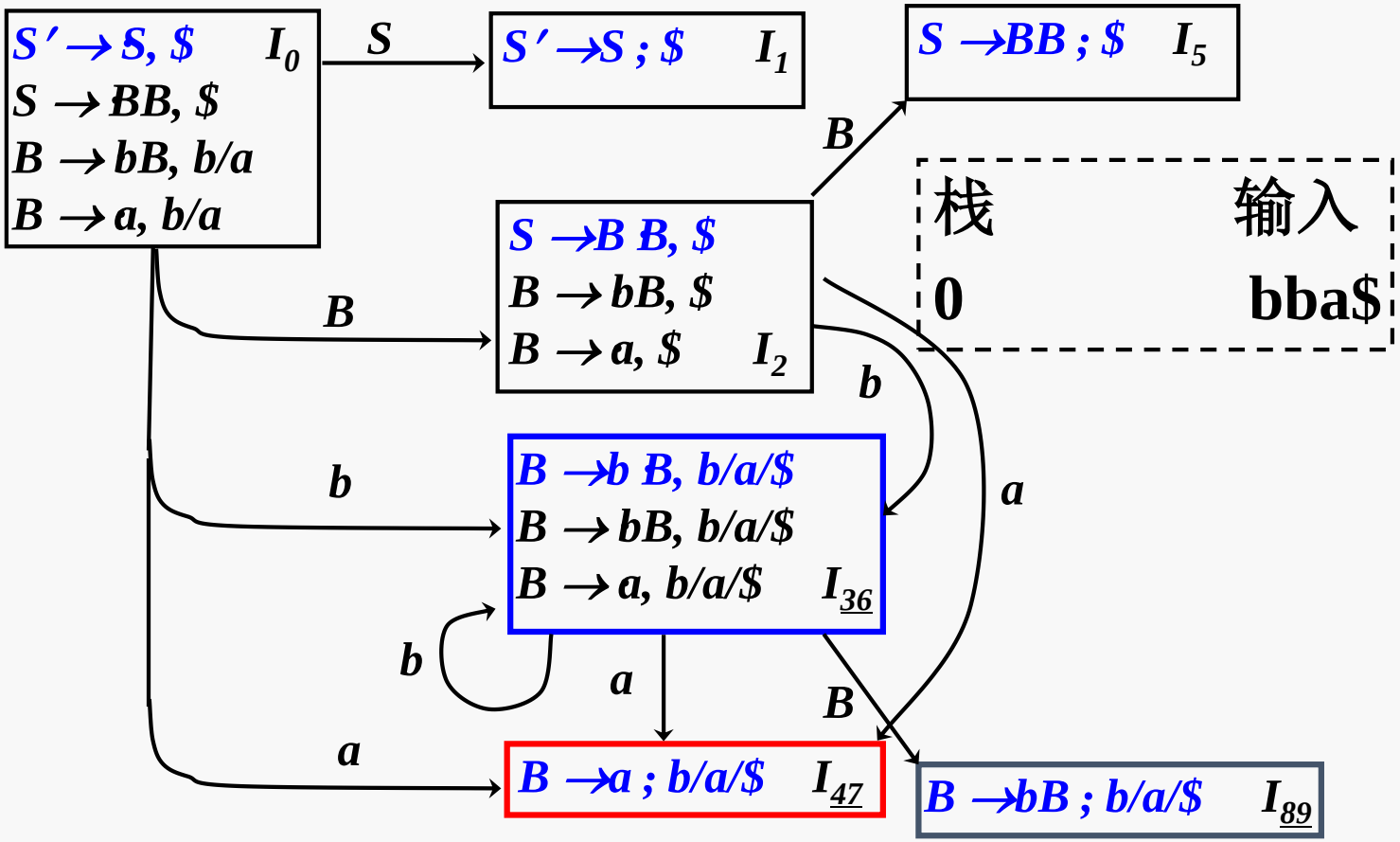


有三组同心集，都合并

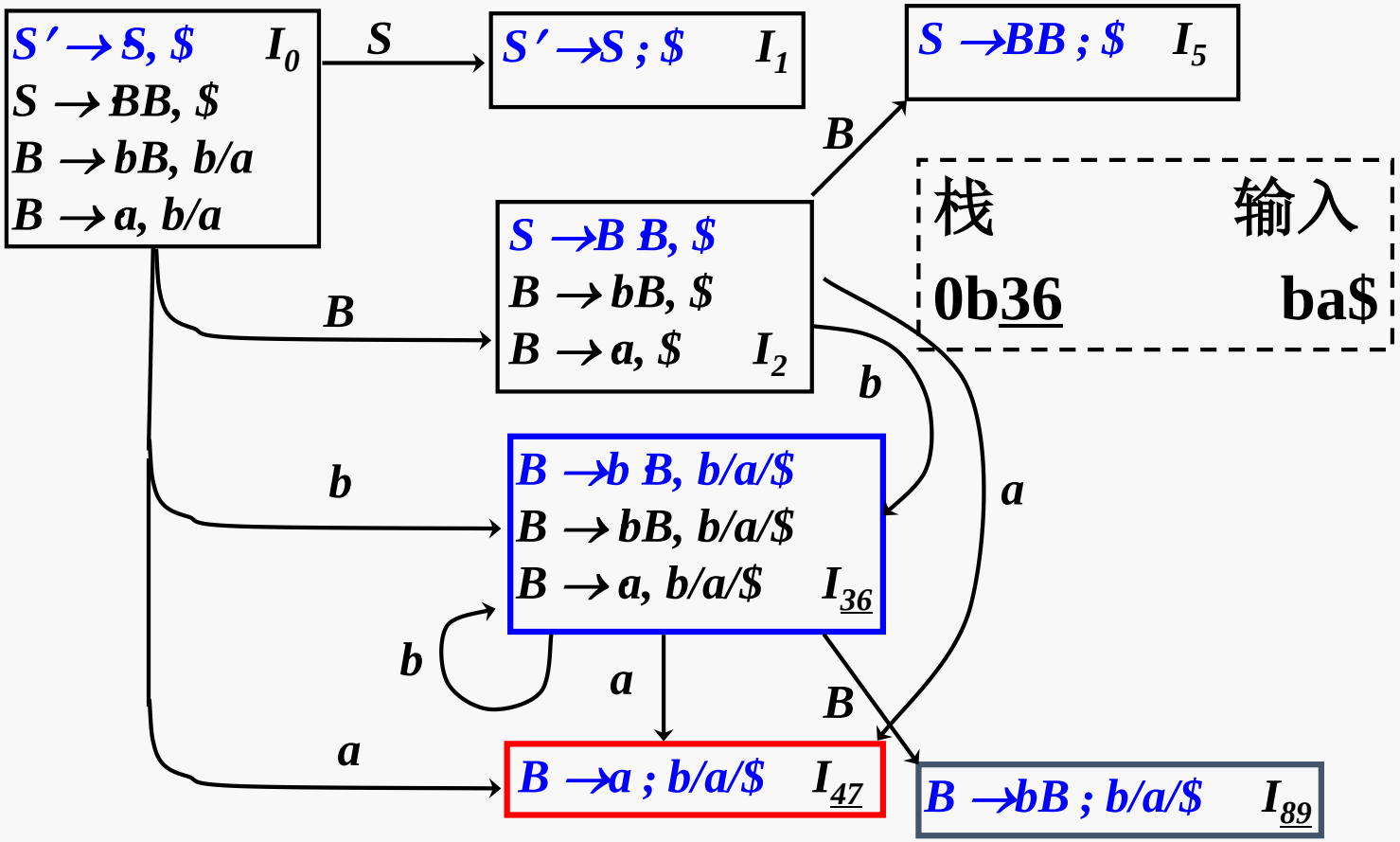
合并同心项目集


S′→S, $ I0
S →BB, $
B →bB, b/a
B →a, b/a
S
S′→S ; $ I1
S →BB ; $ I5
B
S →B B, $
B →bB, $
B →a, $ I2
B
b
a
b
B →b B, b/a/$
B →bB, b/a/$
B →a, b/a/$ I36
b
a
B
a
B →a ; b/a/$ I47
B →bB ; b/a/$ I89

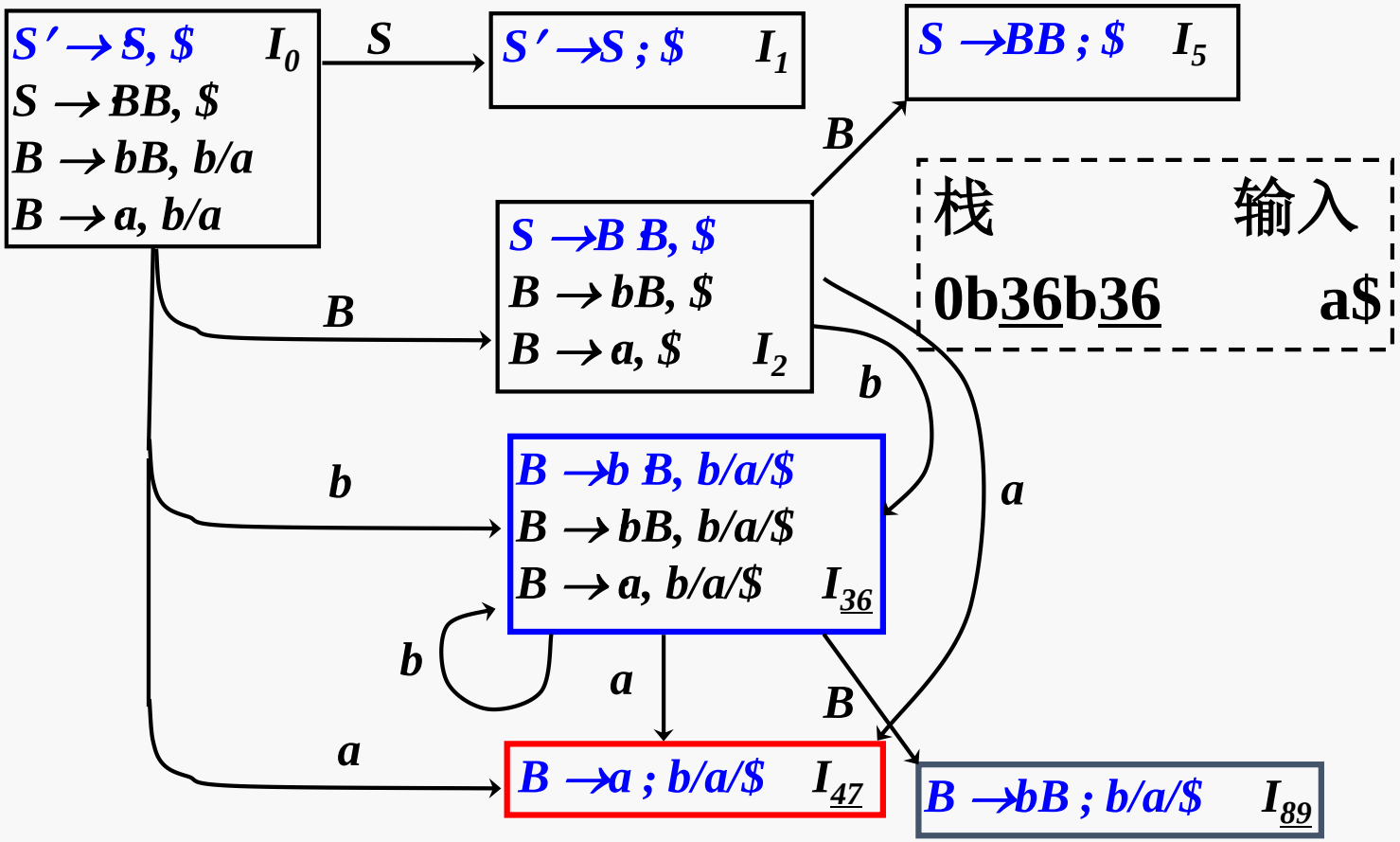
LALR分析：举例


S′ → S, $ I0
S → BB, $
B → bB, b/a
B → a, b/a
S
S′ → S ; $ I1
S → BB ; $ I5
B
B
S → B B, $
B → bB, $
B → a, $ I2
b
b
B → b B, b/a/$
B → bB, b/a/$
B → a, b/a/$ I36
a
b
a
B
a
B → a ; b/a/$ I47
B → bB ; b/a/$ I89
栈 输入
0 bba$

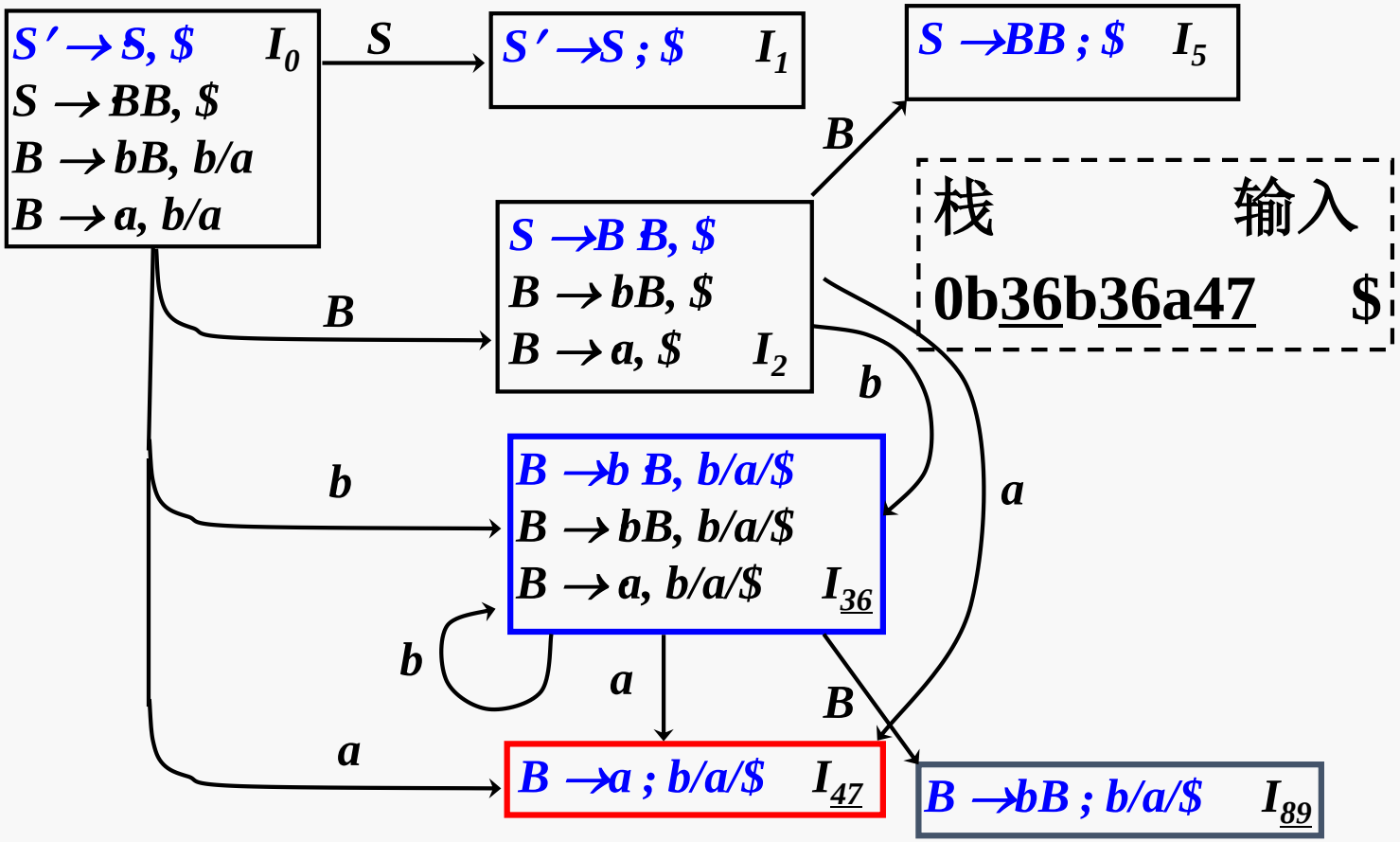
LALR分析：举例


S′ → S, $ I0
S → BB, $
B → bB, b/a
B → a, b/a
S
S′ → S ; $ I1
S → BB ; $ I5
B
B
S → B B, $
B → bB, $
B → a, $ I2
b
b
B → b B, b/a/$
B → bB, b/a/$
B → a, b/a/$ I36
a
b
a
B
a
B → a ; b/a/$ I47
B → bB ; b/a/$ I89
栈 输入
0b36 ba$

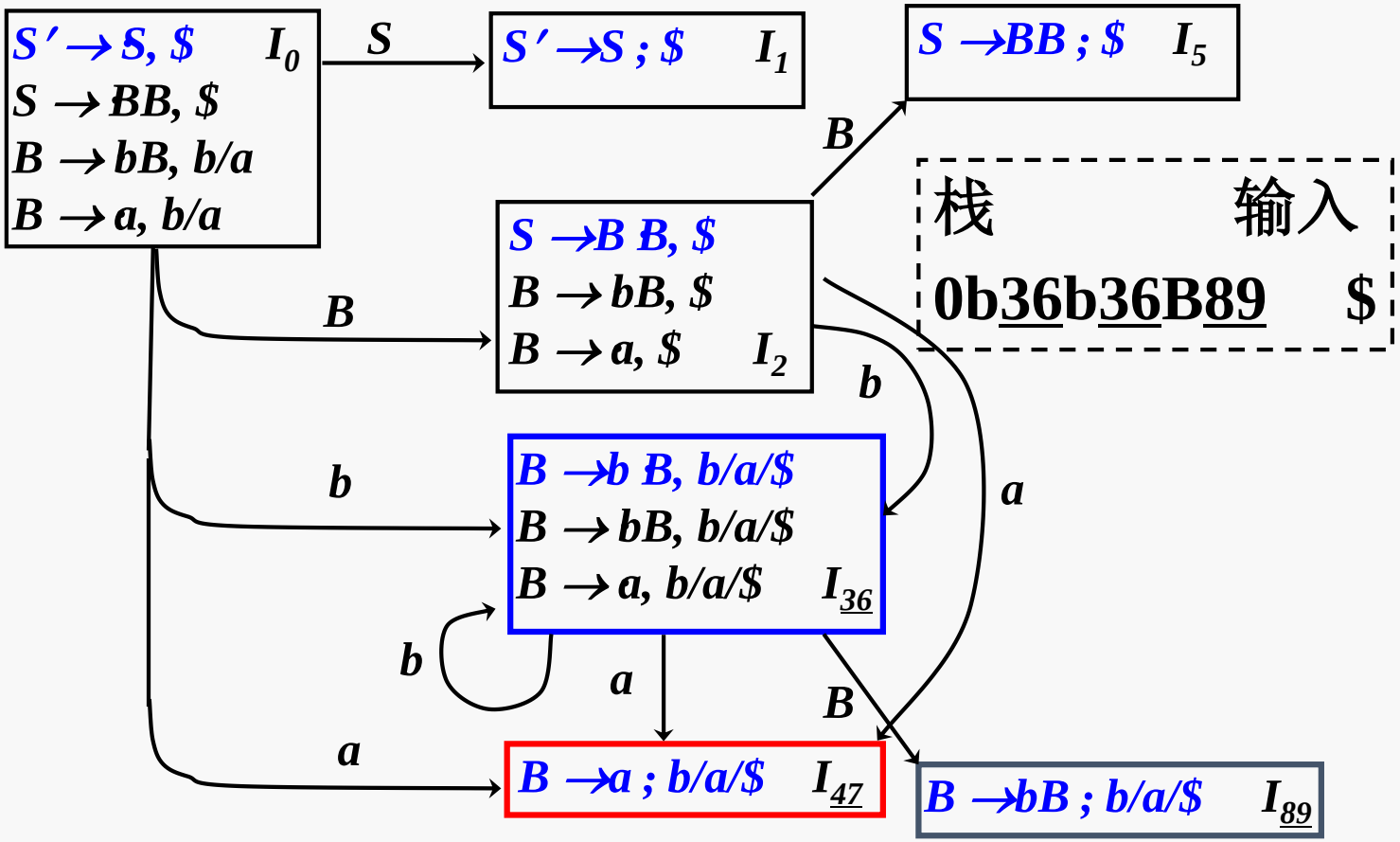
LALR分析：举例


S′ → S, $ I0
S → BB, $
B → bB, b/a
B → a, b/a
S
S′ → S ; $ I1
S → BB ; $ I5
B
S → B B, $
B → bB, $
B → a, $ I2
B
b
a
B → b B, b/a/$
B → bB, b/a/$
B → a, b/a/$ I36
b
a
B
a
B → a ; b/a/$ I47
B → bB ; b/a/$ I89
栈 输入
0b36b36 a$

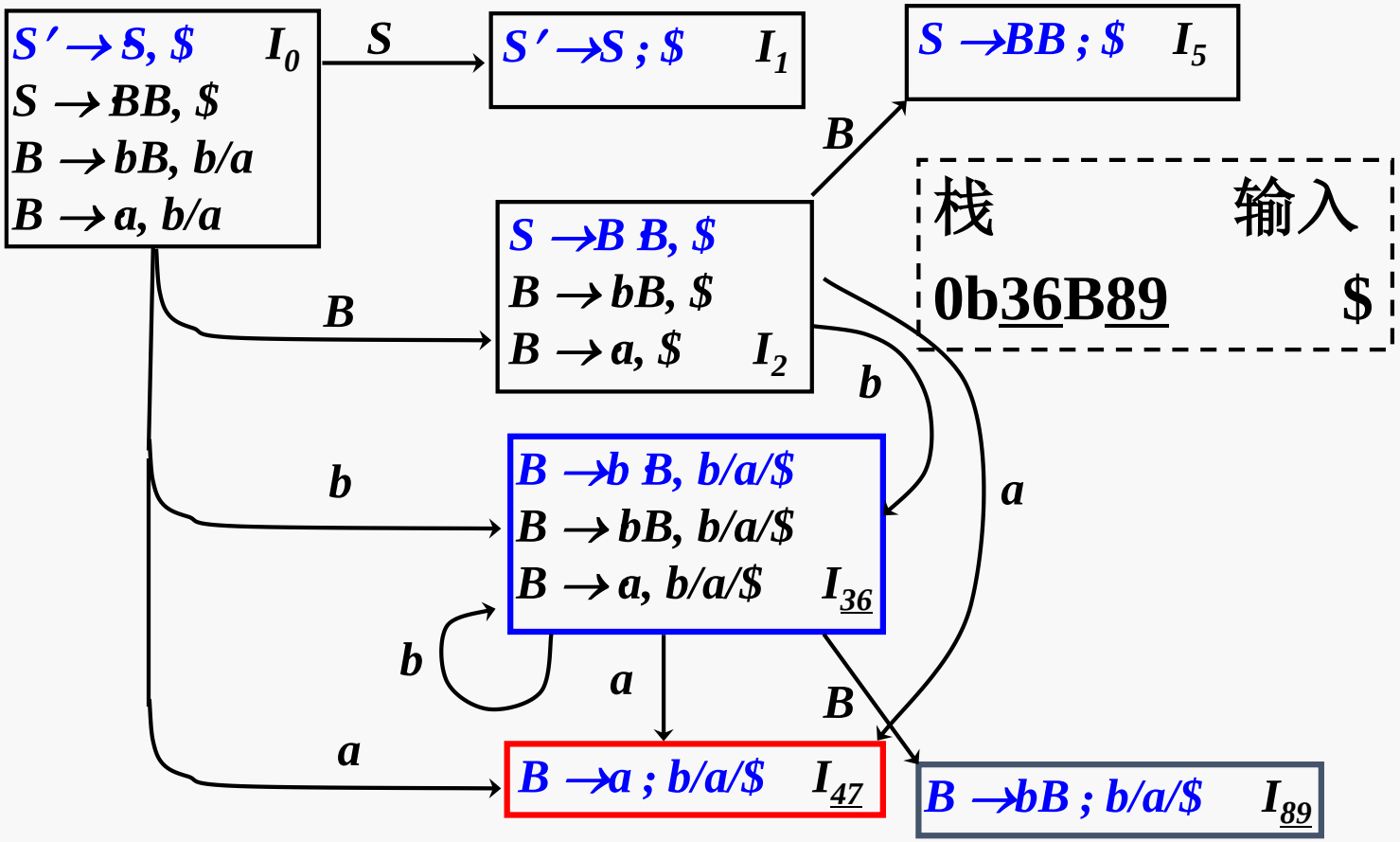
LALR分析：举例


S′→S, $ I0
S→BB, $
B→bB, b/a
B→a, b/a
S
S′→S ; $ I1
S→BB ; $ I5
B
S→B B, $
B→bB, $
B→a, $ I2
B
b
a
B→b B, b/a/$
B→bB, b/a/$
B→a, b/a/$ I36
b
a
B
a
B→a ; b/a/$ I47
B→bB ; b/a/$ I89
栈 输入
0b36b36a47 $

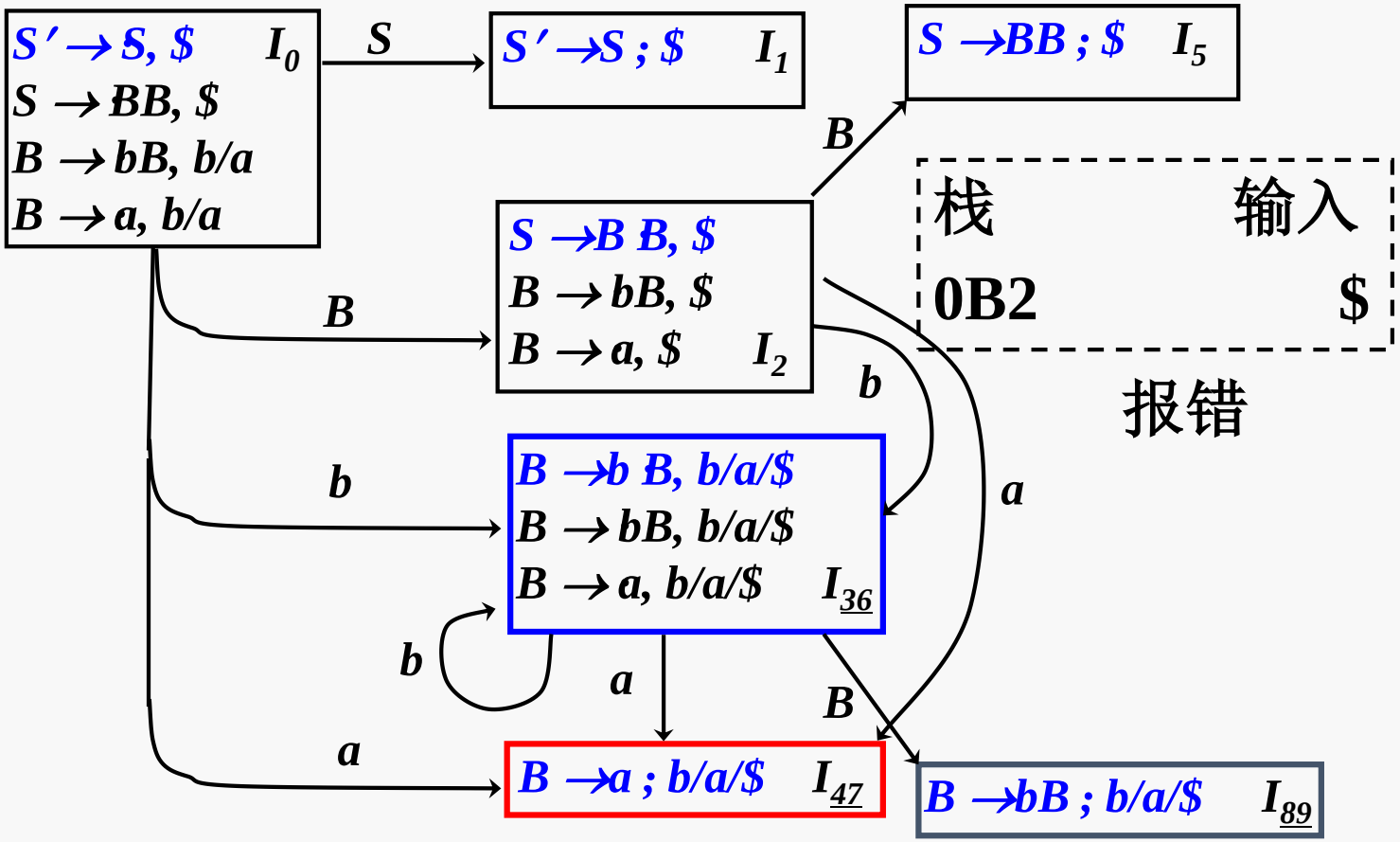
LALR分析：举例


S′→S, $ I0
S→BB, $
B→bB, b/a
B→a, b/a
S
S′→S ; $ I1
S→BB ; $ I5
B
S→B B, $
B→bB, $
B→a, $ I2
B
b
a
B→b B, b/a/$
B→bB, b/a/$
B→a, b/a/$ I36
b
a
B
a
B→a ; b/a/$ I47
B→bB ; b/a/$ I89
栈 输入
0b36b36B89 $

LALR分析：举例


S′ → S, $ I0
S → BB, $
B → bB, b/a
B → a, b/a
S
S′ → S ; $ I1
S → BB ; $ I5
B
S → B B, $
B → bB, $
B → a, $ I2
B
b
a
B → b B, b/a/$
B → bB, b/a/$
B → a, b/a/$ I36
b
a
B
a
B → a ; b/a/$ I47
B → bB ; b/a/$ I89
栈 输入
0b36B89 $

LALR分析：举例


S′ → S, $ I0
S → BB, $
B → bB, b/a
B → a, b/a
S
S′ → S ; $ I1
S → BB ; $ I5
B
S → B B, $
B → bB, $
B → a, $ I2
B
b
a
B → b B, b/a/$
B → bB, b/a/$
B → a, b/a/$ I36
b
a
B
a
B → a ; b/a/$ I47
B → bB ; b/a/$ I89
栈 输入
0B2 $
报错

LALR分析方法

2、构造LALR(1)分析表

- 构造LR(1)项目集规范族$C = \{I_0, I_1, \ldots, I_n\}$
- 构造LALR(1)项目集规范族$C' = \{J_0, J_1, \ldots, J_k\}$，其中任意项目集$J_i = I_n \cup I_m \cup \ldots \cup I_t$
  - $I_n, I_m, \ldots, I_t \in C$且具有共同的核心
- 按构造规范LR(1)分析表的方式构造分析表

如没有语法分析动作冲突，那么给定文法就是**LALR(1)文法**

LALR分析方法

- 合并同心项目集可能会引起冲突
  - 同心集的合并不会引起新的移进–归约冲突

项目集1	项目集2
$[A\rightarrow\alpha\ ;a]$	$[B\rightarrow\beta a\gamma,b]$
$[B\rightarrow\beta a\gamma,c]$	$[A\rightarrow\alpha\ ;d]$
...	...

如果有移进归约冲突，则合并前就有冲突

LALR分析方法

- 合并同心项目集可能会引起冲突
  - 同心集的合并不会引起新的移进–归约冲突
  - 同心集的合并有可能产生新的归约–归约冲突

$S' \to S$

$S \to aAd \mid bBd \mid aBe \mid bAe$

$A \to c$

$B \to c$

对ac有效的项目集

$A \to c \ ; d$

$B \to c \ ; e$

对bc有效的项目集

$A \to c \ ; e$

$B \to c \ ; d$

合并同心集后

$A \to c \ ; d/e$

$B \to c \ ; d/e$

该文法是LR(1)的
但不是LALR(1)的

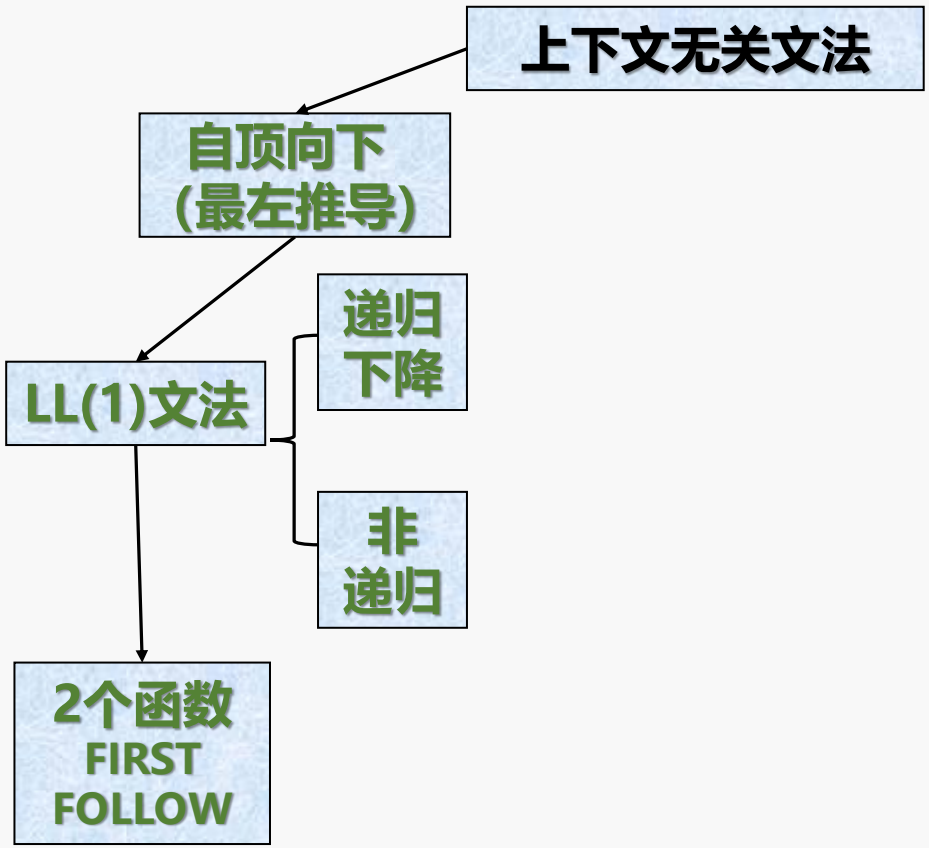
LR分析法总结

		SLR	LALR	LR(1)
初始状态		$[S' \rightarrow S]$	$[S' \rightarrow S, \$]$	$[S' \rightarrow S, \$]$
项目集		LR(0) CLOSURE(I)	合并LR(1)项目集族的同心项目集	LR(1), CLOSURE(I) 搜索符考虑FISRT(βa)
动作	移进	$[A \rightarrow \alpha a\beta] \in I_i$ $GOTO(I_i, a) = I_j$ $ACTION[i, a] = sj$	与LR(1) 一致	$[A \rightarrow \alpha a\beta, b] \in I_i$ $GOTO(I_i, a) = I_j$ $ACTION[i, a] = sj$
	归约	$[A \rightarrow \alpha] \in I_i, A \neq S'$ $a \in FOLLOW(A)$ $ACTION[i, a] = rj$	与LR(1) 一致	$[A \rightarrow \alpha, a] \in I_i$ $A \neq S'$ $ACTION[i, a] = rj$
	接受	$[S' \rightarrow S \cdot] \in I_i$ $ACTION[i, \$] = acc$	与LR(1) 一致	$[S' \rightarrow S \cdot, \$] \in I_i$ $ACTION[i, \$] = acc$
	出错	空白条目	与LR(1) 一致	空白条目
GOTO		$GOTO(I_i, A) = I_j$ $GOTO[i, A] = j$	与LR(1) 一致	$GOTO(I_i, A) = I_j$ $GOTO[i, A] = j$
状态量		少(几百)	与SLR一样	多(几千)

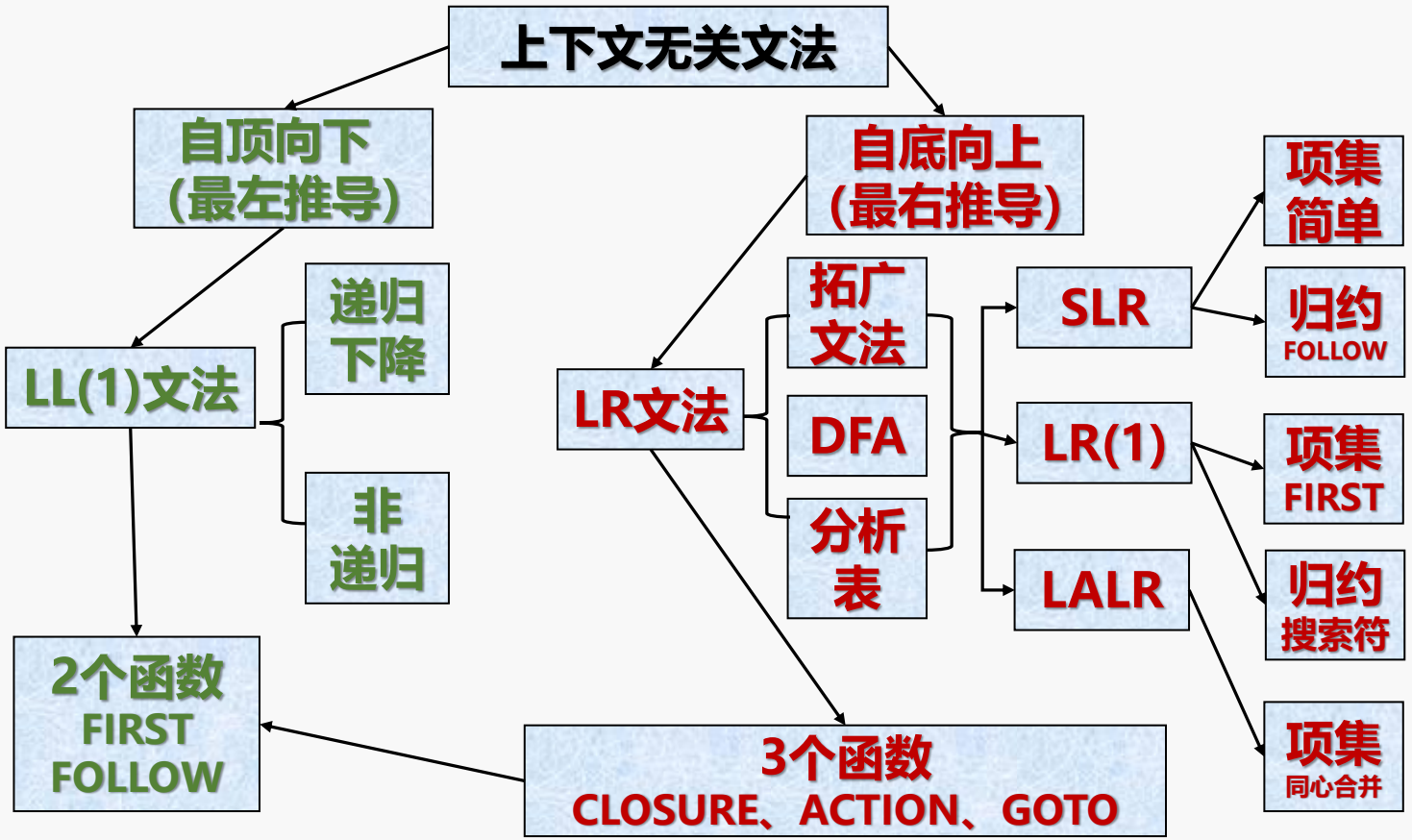
LR和LL分析方法的比较

	LR(1)方 法	LL(1)方 法
建立分析树	自 底 而 上	自 顶 而 下
归约or推导	规 范 归 约	最 左 推 导
决定使用产生式的时机	看见产生式整个右部推出的串后(句柄)	看见产生式推出的第一个终结符后
对文法的限制	无	无左递归、无公共左因子
分析表	状态×文法符号，大	非终结符×终结符，小
分析栈	状态栈，信息更多	文法符号栈
确定句柄	根据栈顶状态和下一个符号便可以确定句柄和归约所用产生式	无句柄概念
语法错误	决不会将出错点后的符号移入分析栈	和LR一样，决不会读过出错点而不报错

语法分析技术总结



语法分析技术总结


上下文无关文法
自顶向下
(最左推导)
自底向上
(最右推导)
LL(1)文法
递归
下降
非
递归
LR文法
拓广
文法
DFA
分析
表
SLR
LR(1)
LALR
项集
简单
归约
FOLLOW
项集
FIRST
归约
搜索符
项集
同心合并
2个函数
FIRST
FOLLOW
3个函数
CLOSURE、ACTION、GOTO

《编译原理和技术》

语法分析 V

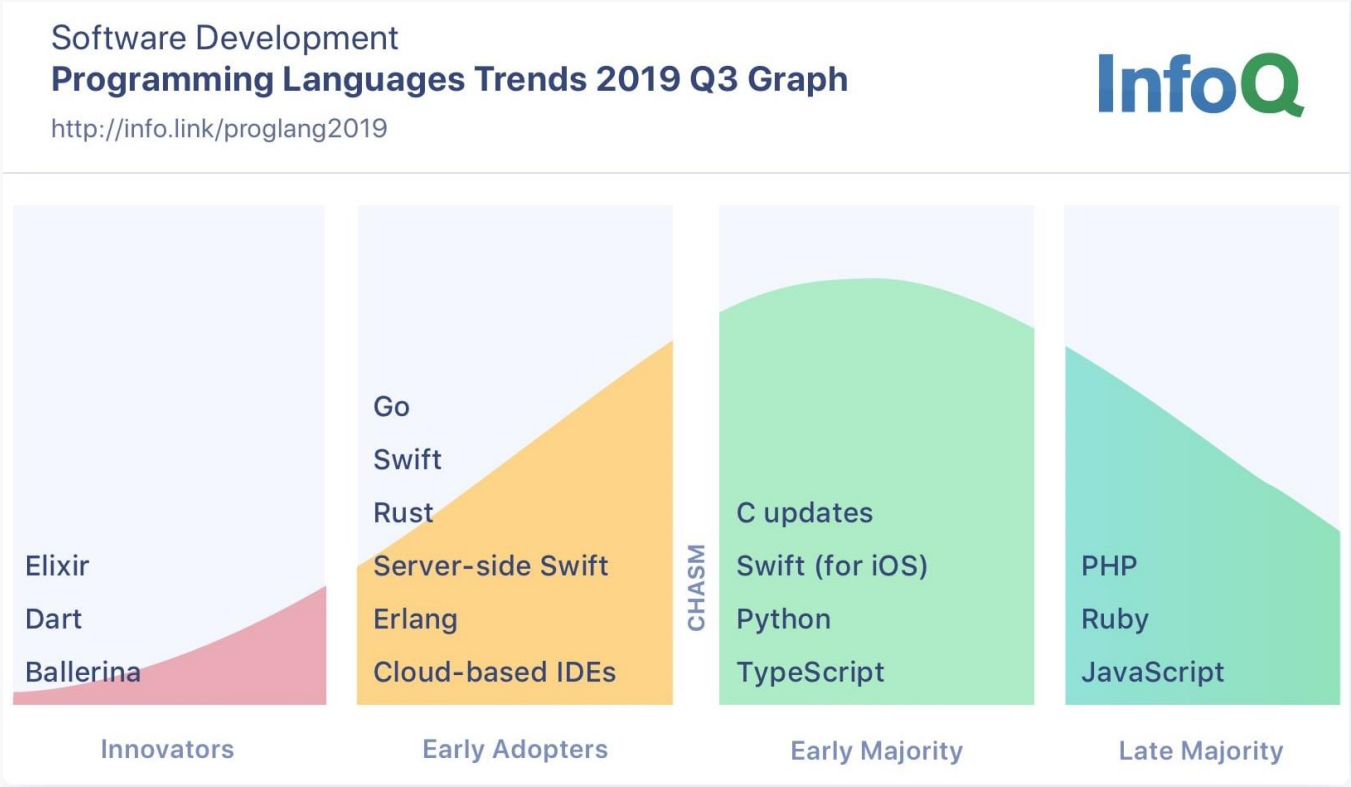
谢谢！

《编译原理和技术》

中间代码表示

中科大计算机学院
李 诚
2022-09-28

为什么需要中间代码表示?

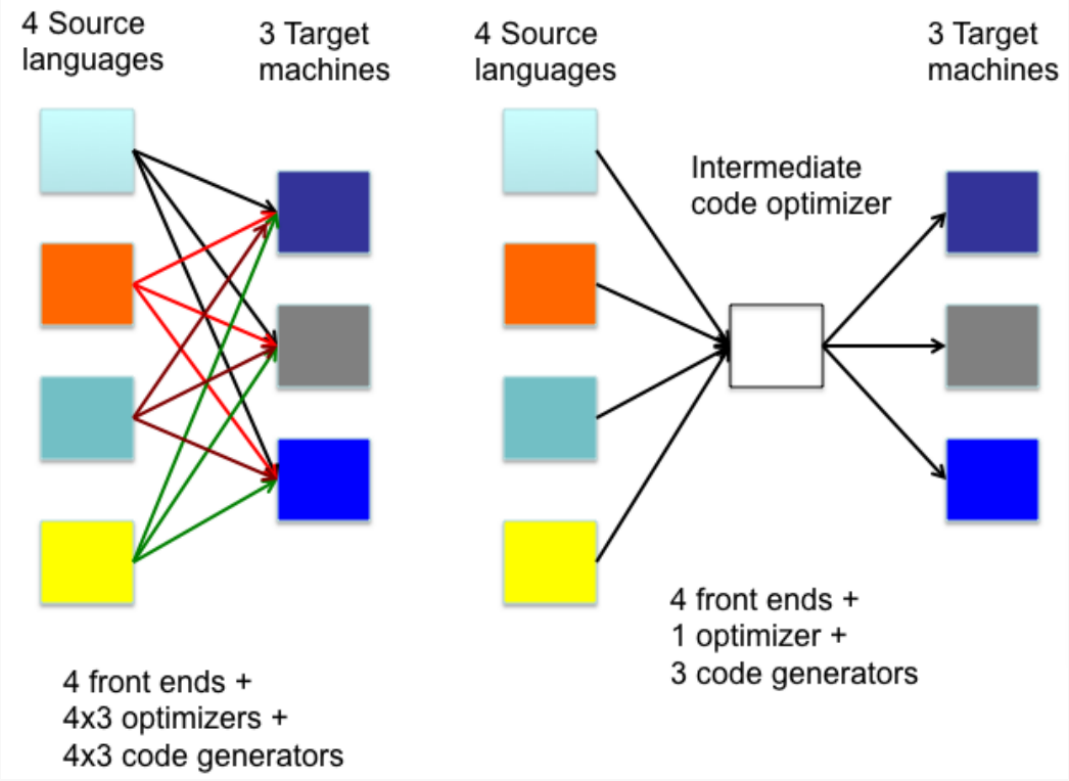


为什么需要中间代码表示?

分类	名称	版本	扩展	初始年份
CISC	x86	16, 32, 64 (16→32→64)	x87, IA-32, MMX, 3DNow!, SSE, SSE2, PAE, x86-64, SSE3, SSSE3, SSE4, BMI, AVX, AES, FMA, XOP, F16C	1978
RISC	MIPS	32	MDMX, MIPS-3D	1981
VLIW	Elbrus	64	Just-in-time dynamic translation: x87, IA-32, MMX, SSE, SSE2, x86-64, SSE3, AVX	2014

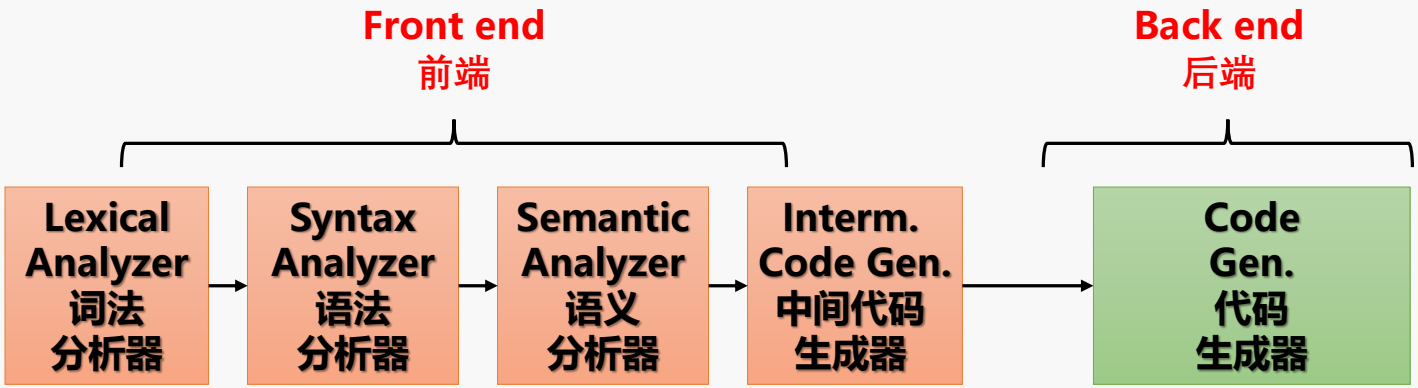
指令集体系结构(ISA)的发展

为什么需要中间代码表示?



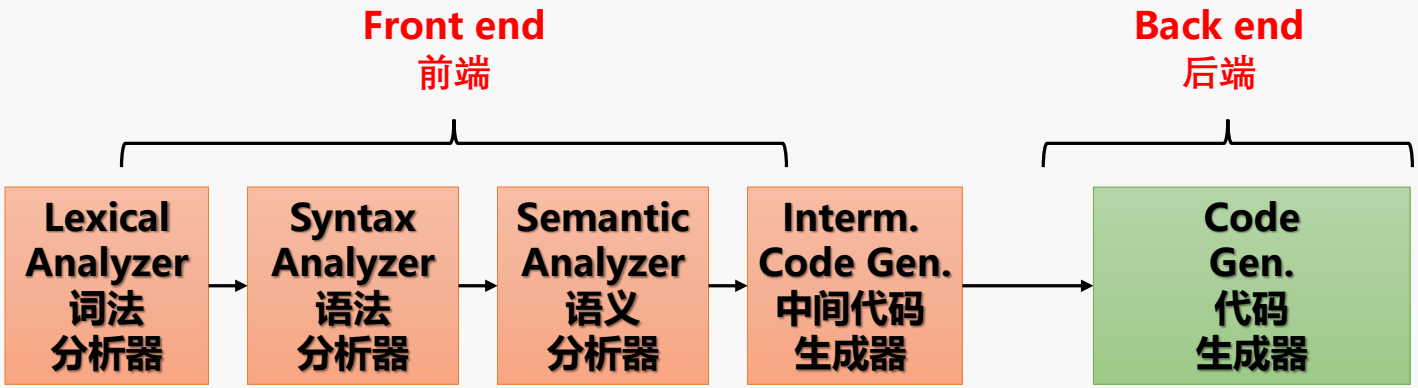
实践过程中，推陈出新的语言、不断涌现的指令集、开发成本之间的权衡

为什么需要中间代码表示



- 前端与后端分离
  - 不同的源语言、不同的机器可以得到不同的编译优化组合
  - 前端只关注和分析与源语言相关的细节，与目标机器无关

为什么需要中间代码表示



- **前端与后端分离**
  - 为新机器构建编译器，只需要设计从中间代码到新的目标机器代码的编译器 (前端独立)
- **中间代码优化与源语言和目标机器均无关**

中间表示有哪些类型?

- 简而言之，编译器任何完整的中间输出都是中间代码表示形式
- 常见类型有:
  - 后缀表示
  - 语法树或DAG图
  - 三地址码(TAC)
  - 静态单赋值形式(SSA)

重点关注
LLVM IR是TAC类型

后缀表示

*uop*是一元运算符

$$E \rightarrow E\ opE \mid uopE \mid (E) \mid \mathrm{id} \mid \mathrm{num}$$

表达式E	后缀式E'
id	id
num	num
$E_1\ op\ E_2$	$E_1'\ E_2'\ op$
$uopE$	$E'uop$
(E)	E'

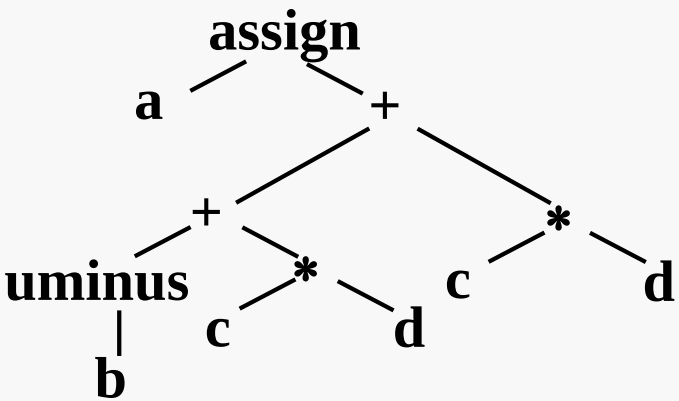
后缀表示

- 后缀表示不需要括号
  - (8 – 5) + 2 的后缀表示是8 5 –2 +
- 后缀表示的最大优点是便于计算机处理表达式

计算栈	输入串
	8 5 –2 +
8	5 –2 +
8 5	–2 +
3	2 +
3 2	+
5	

图形表示

- 语法树是一种图形化的中间表示

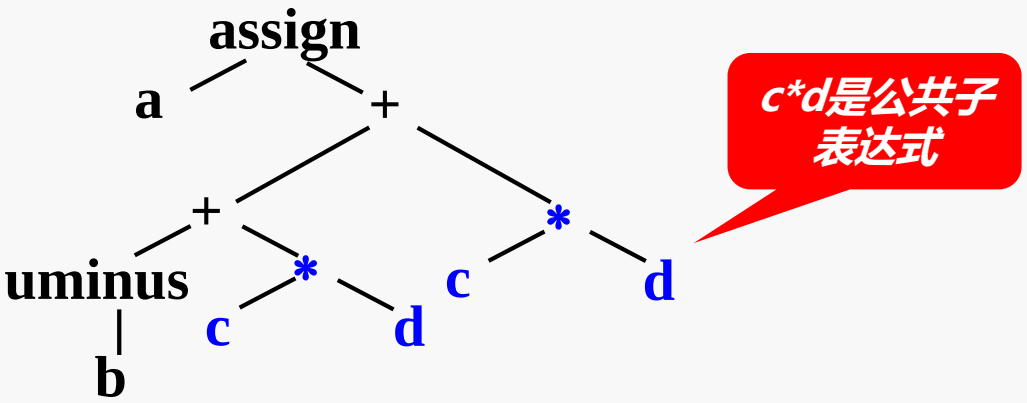


(a) 语法树

a = (–b + c*d) + c*d的图形表示

图形表示

- 语法树是一种图形化的中间表示

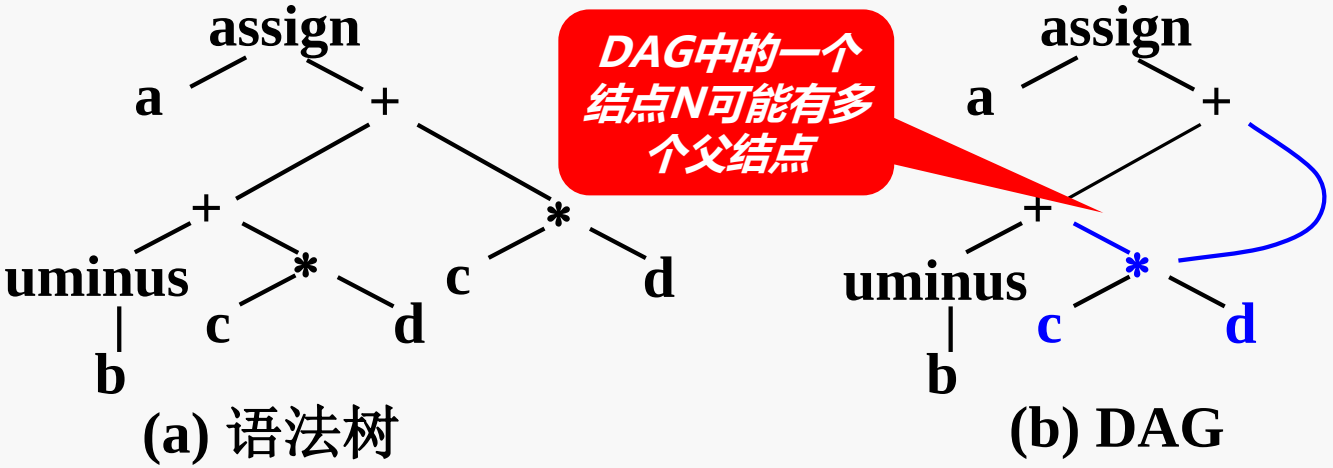


(a) 语法树

a = (–b + c*d) + c*d的图形表示

图形表示

- 语法树是一种图形化的中间表示
- 有向无环图(Directed Acyclic Graph, DAG)也是一种中间表示



a = (–b + c*d) + c*d的图形表示

三地址代码

- **三地址代码 (Three-Address Code，TAC)**

一般形式：$x = y\ op\ z$

- 最多一个算符
- 最多三个计算分量
- 每一个分量代表一个地址，因此三地址

- **例 表达式$x + y * z$翻译成的三地址语句序列**

$t_1 = y * z$

$t_2 = x + t_1$

三地址代码

- 三地址代码是语法树或DAG的一种线性表示
  - 例　　a = (–b + c*d) + c*d

语法树的代码

$t_1 = -b$

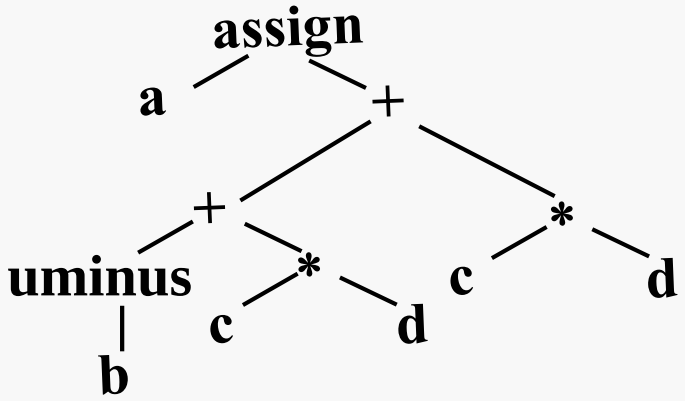
$t_2 = c * d$

$t_3 = t_1 + t_2$

$t_4 = c * d$

$t_5 = t_3 + t_4$

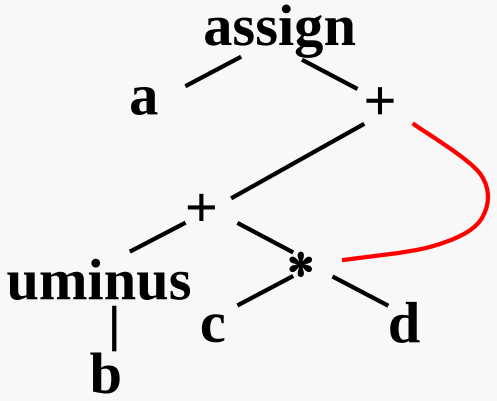
$a = t_5$



三地址代码

- 三地址代码是语法树或DAG的一种线性表示
  - 例　　a = (–b + c*d) + c*d

语法树的代码	DAG的代码
$t_1 = -b$	$t_1 = -b$
$t_2 = c * d$	$t_2 = c * d$
$t_3 = t_1 + t_2$	$t_3 = t_1 + t_2$
$t_4 = c * d$	$t_4 = t_3 + t_2$
$t_5 = t_3 + t_4$	$a = t_4$
$a = t_5$	



三地址代码

常用的三地址语句

- 运算/赋值语句　　***x = y op z***，　***x = op y***，　***x = y***
- 无条件转移　　**goto L**
- 条件转移1　　if *x* goto L, if False *x* goto L
- 条件转移2　　**if *x relop y* goto L**

三地址代码

常用的三地址语句

- 过程调用
  - **param** x_1 //设置参数
  - param x_2
  - …
  - param x_n
  - **call p , *n*** //调用子过程p，n为参数个数
- 过程返回　　**return *y***
- 索引赋值　　***x* = *y*[*i*]** 和 ***x*[*i*] = *y***
  - 注意：i表示距离y处i个内存单元
- 地址和指针赋值 ***x* = &*y*，*x* = **y*** 和 ****x* = *y***

三地址代码翻译：举例

- 考虑语句，令数组a的每个元素占8存储单元
  - do i = i + 1; while (a[i] < v);

```
L:  t1 = i + 1
    i  = t1
    t2 = i * 8
    t3 = a[ t2 ]
    if t3 < v goto L
```

符号标号

```
100:  t1 = i + 1
101:  i  = t1
102:  t2 = i * 8
103:  t3 = a[ t2 ]
104:  if t3 < v goto 100
```

位置标号

三地址代码的实现方式

- 三地址代码只说明了指令的组成部分，我们还需关注其在编译器中的具体数据结构实现
- 常见的实现方式有三种：
  - 四元式：(op, arg1, arg2, result)
  - 三元式：(op, arg1, arg2)
  - 间接三元式：(三元式的指针表)

三地址代码的实现方式

- **四元式(Quadruple)**
  - 包括4个字段：　　op　arg_1　arg_2　$result$
  - op: 运算符的内部编码
  - arg_1　arg_2　$result$是地址
  - 例如：x = y + z 的四元式　+　　y　　z　　x

三地址代码的实现方式

四元式(Quadruple)

- 例： a = b * – c + b * – c

```
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
 a = t5
```

	op	arg_1	arg_2	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

缺点：临时变量太多，增加时间和空间成本

三地址代码的实现方式

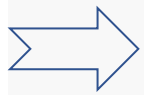
- **三元式(Triple)**
  - 格式： op arg_1 arg_2
  - 将存储结果的临时变量隐藏起来，用指令所代表的的位置来表示其运算结果
  - $x = y\ op\ z$ 将被拆分为(?是编号)
    - $?$ op y z
    - x $=$ $(?)$

三地址代码的实现方式

三元式(Triple)

- 例： a = b * – c + b * – c

	op	arg_1	arg_2	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		



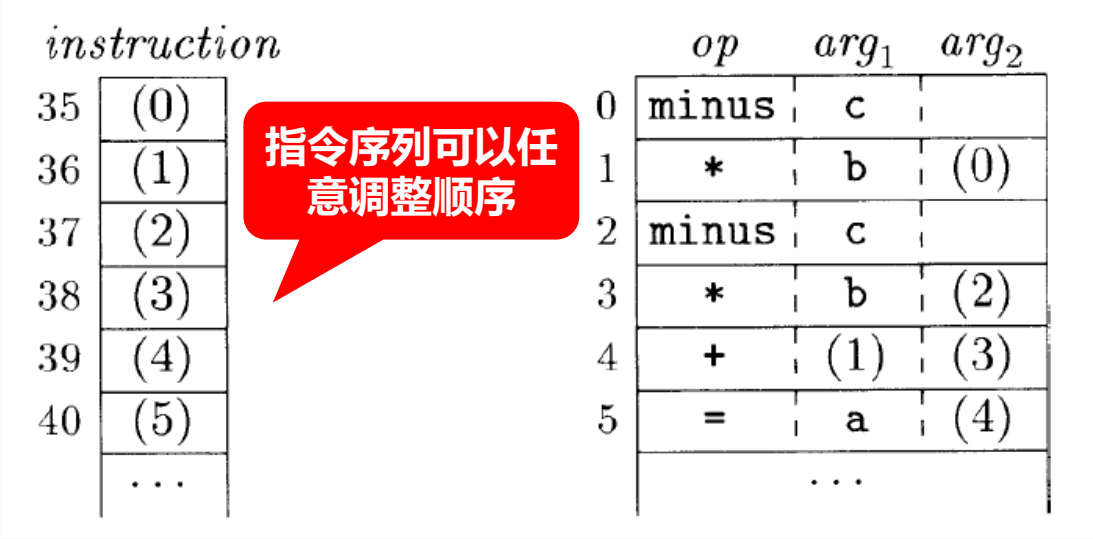
	op	arg_1	arg_2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
		...	

缺点：隐式的临时变量，代码位置调整会造成引用该位置的代码也要修改。

三地址代码的实现方式

间接三元式(Indirect triple)

- 例： a = b * – c + b * – c



优势：比四元式空间开销小，比三元式更灵活

三地址代码的实现方式总结

四元式	按编号次序计算	计算结果存于result	方便移动，计算次序容易调整	大量引入临时变量
三元式	按编号次序计算	由编号代表	不方便移动	在代码生成时进行临时变量的分配
间接三元式	按指令列表次序计算	由编号代表	方便移动，计算次序容易调整	在代码生成时进行临时变量的分配

静态单赋值形式

- 一种便于某些代码优化的中间表示
- 和三地址代码的主要区别
  - 所有赋值指令都是对不同名字的变量的赋值

三地址代码	静态单赋值形式
$p = a + b$	$p_1 = a + b$
$q = p - c$	$q_1 = p_1 - c$
$p = q * d$	$p_2 = q_1 * d$
$p = e - p$	$p_3 = e - p_2$
$q = p + q$	$q_2 = p_3 + q_1$

SSA由Barry K. Rosen、Mark N. Wegman和F. Kenneth Zadeck于1988年提出

静态单赋值形式

- 一种便于某些代码优化的中间表示
- 和三地址代码的主要区别
  - 所有赋值指令都是对不同名字的变量的赋值
  - 同一个变量在不同控制流路径上都被定值

```
if (flag) x = –1; else x = 1;
y = x * a;
```

改成

```
if (flag) x1 = –1; else x2 = 1;
x3 = φ(x1, x2);      //由flag的值决定用x1还是x2
y = x3 * a;
```

举例

快速排序程序片段如下

i = m –1; j = n; v = a[n];

```
while (1) {
 do i = i +1; while(a[i]<v);
 do j =j –1;while (a[j]>v);
 if (i >= j) break;
 x=a[i]; a[i]=a[j]; a[j]=x;
}
x=a[i]; a[i]=a[n]; a[n]=x;
```

```
(1) i := m –1
(2) j := n
(3) t1 := 4 * n
(4) v := a[t1]
(5) i := i + 1
(6) t2 := 4 * i
(7) t3 := a[t2]
(8) if t3 < v goto   (5)
(9)   j := j –1
(10) t4 := 4 * j
(11) t5 := a[t4]
(12) if t5 > v goto (9)
(13) if i >= j goto (23)
(14)  t6 := 4 * i
(15)  x := a[t6]
(16)  t7 := 4 * i
(17)  t8 := 4 * j
(18)  t9 := a[t8]
(19)  a[t7] := t9
(20)  t10 := 4 * j
(21)  a[t10] := x
(22) goto (5)
(23)  t11 := 4 * i
(24)  x := a[t11]
(25)  t12 := 4 * i
(26)  t13 := 4 * n
(27)  t14 := a[t13]
(28)  a[t12] := t14
(29)  t15 := 4 * n
(30)  a[t15] := x
```

基本块(Basic block)

- 连续的三地址指令序列，控制流从它的开始进入，并从它的末尾离开，中间没有停止或分支的可能性（末尾除外）
- 流图(flow graph)

  用有向边表示基本块之间的控制流信息，基本块作为结点

基本块划分算法

- **输入：三地址指令序列**
- **输出：基本块列表**
- **算法：**
  - 首先确定基本块的第一个指令，即首指令(leader)
    - 指令序列的第一条三地址指令是一个首指令
    - 任意转移指令的目标指令是一个首指令
    - 紧跟一个转移指令的指令是一个首指令
  - 然后，每个首指令对应的基本块包括了从它自己开始，直到下一个首指令(不含)或指令序列结尾之间的所有指令

举例

(1) i := m –1
(2) j := n
(3) t1 := 4 * n
(4) v := a[t1]
(5) i := i + 1
(6) t2 := 4 * i
(7) t3 := a[t2]
(8) if t3 < v goto (5)
(9) j := j –1
(10) t4 := 4 * j
(11) t5 := a[t4]
(12) if t5 > v goto (9)
(13) if i >= j goto (23)
(14) t6 := 4 * i
(15) x := a[t6]
(16) t7 := 4 * i
(17) t8 := 4 * j
(18) t9 := a[t8]
(19) a[t7] := t9
(20) t10 := 4 * j
(21) a[t10] := x
(22) goto (5)
(23) t11 := 4 * i
(24) x := a[t11]
(25) t12 := 4 * i
(26) t13 := 4 * n
(27) t14 := a[t13]
(28) a[t12] := t14
(29) t15 := 4 * n
(30) a[t15] := x

举例——首指令

(1) i := m –1
(2) j := n
(3) t1 := 4 * n
(4) v := a[t1]
(5) i := i + 1
(6) t2 := 4 * i
(7) t3 := a[t2]
(8) if t3 < v goto (5)
(9) j := j –1
(10) t4 := 4 * j
(11) t5 := a[t4]
(12) if t5 > v goto (9)
(13) if i >= j goto (23)
(14) t6 := 4 * i
(15) x := a[t6]
(16) t7 := 4 * i
(17) t8 := 4 * j
(18) t9 := a[t8]
(19) a[t7] := t9
(20) t10 := 4 * j
(21) a[t10] := x
(22) goto (5)
(23) t11 := 4 * i
(24) x := a[t11]
(25) t12 := 4 * i
(26) t13 := 4 * n
(27) t14 := a[t13]
(28) a[t12] := t14
(29) t15 := 4 * n
(30) a[t15] := x

举例——基本块

B_1
(1) i := m –1
(2) j := n
(3) t1 := 4 * n
(4) v := a[t1]

B_2
(5) i := i + 1
(6) t2 := 4 * i
(7) t3 := a[t2]
(8) if t3 < v goto (5)

B_3
(9) j := j –1
(10) t4 := 4 * j
(11) t5 := a[t4]
(12) if t5 > v goto (9)

B_4
(13) if i >= j goto (23)

B_5
(14) t6 := 4 * i
(15) x := a[t6]
(16) t7 := 4 * i
(17) t8 := 4 * j
(18) t9 := a[t8]
(19) a[t7] := t9
(20) t10 := 4 * j
(21) a[t10] := x
(22) goto (5)

B_6
(23) t11 := 4 * i
(24) x := a[t11]
(25) t12 := 4 * i
(26) t13 := 4 * n
(27) t14 := a[t13]
(28) a[t12] := t14
(29) t15 := 4 * n
(30) a[t15] := x

流图 (Flow graph)

- 流图的结点是一些基本块
- 从基本块*B*到基本块*C*之间有一条边，当且仅当*C*的第一个指令可能紧跟在*B*的最后一条指令之后执行
  - *B*是*C*的前驱(predecessor)
  - *C*是*B*的后继(successor)

流图 (Flow graph)

- **流图的结点是一些基本块**
- **从基本块*B*到基本块*C*之间有一条边，当且仅当*C*的第一个指令可能紧跟在*B*的最后一条指令之后执行，判定方法如下：**
  - 有一个从*B*的结尾跳转到*C*的开头的跳转指令
  - 参考原来三地址指令序列中的顺序，*C*紧跟在*B*之后，且*B*的结尾没有无条件跳转指令

举例——流图

B_1
(1) i := m –1
(2) j := n
(3) t1 := 4 * n
(4) v := a[t1]

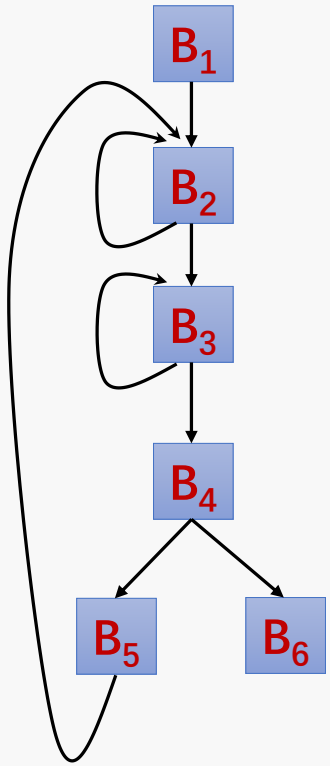
B_2
(5) i := i + 1
(6) t2 := 4 * i
(7) t3 := a[t2]
(8) if t3 < v goto (5)

B_3
(9) j := j –1
(10) t4 := 4 * j
(11) t5 := a[t4]
(12) if t5 > v goto (9)

B_4
(13) if i >= j goto (23)

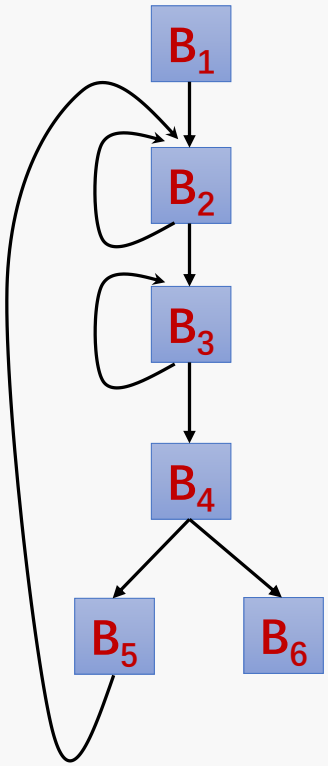
B_5
(14) t6 := 4 * i
(15) x := a[t6]
(16) t7 := 4 * i
(17) t8 := 4 * j
(18) t9 := a[t8]
(19) a[t7] := t9

B_6
(20) t10 := 4 * j
(21) a[t10] := x
(22) goto (5)
(23) t11 := 4 * i
(24) x := a[t11]
(25) t12 := 4 * i
(26) t13 := 4 * n
(27) t14 := a[t13]
(28) a[t12] := t14
(29) t15 := 4 * n
(30) a[t15] := x



循环

- **流图中的一个结点集合L是一个循环，如果它满足：**
  - 该集合有唯一的入口结点
  - 任意结点都有一个到达入口结点的非空路径，且该路径全部在L中
- **不包含其他循环的循环叫做内循环**
- **右图中的循环**
  - B_2自身
  - B_3自身
  - $\{B_2, B_3, B_4, B_5\}$



《编译原理和技术》

中间代码表示

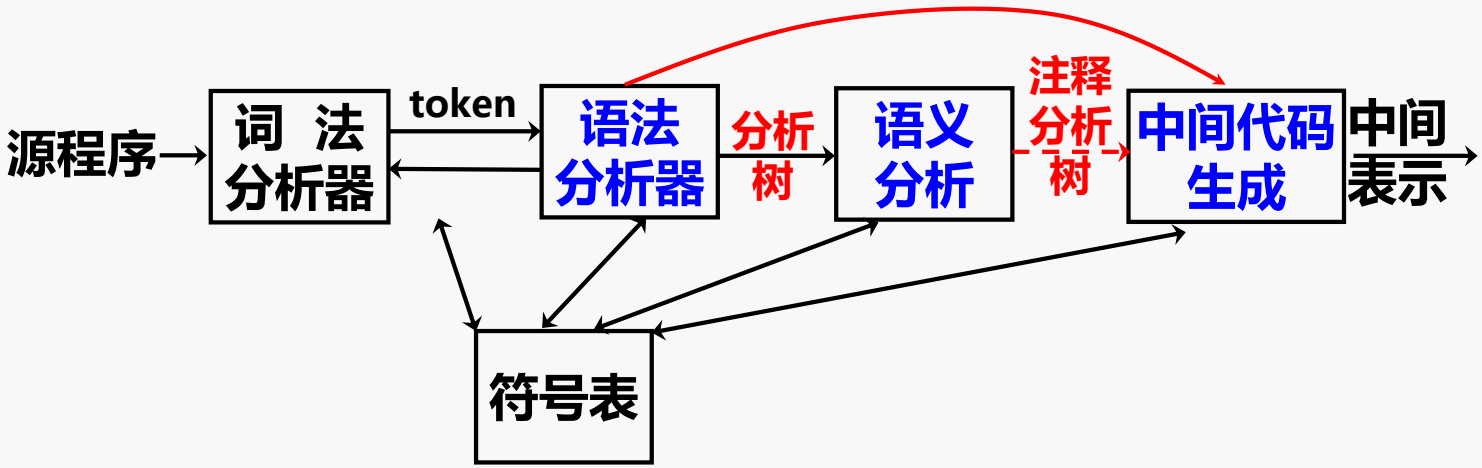
谢谢！

《编译原理和技术》

语法制导翻译 I

中科大计算机学院
李 诚
2022-09-28

本节提纲



- 语法制导翻译简介
- 语法制导定义
  - 属性、属性文法、综合属性及继承属性
  - 属性依赖图与计算次序

语法制导翻译简介

- **编译程序的目标：将源程序翻译成为语义等价的目标程序。**
  - 源程序与目标程序具有不同的语法结构，表达的结果却是相同的。
- **语法制导翻译**
  - 使用上下文无关文法(CFG)来引导对语言的翻译，是一种面向文法的翻译技术

语法制导翻译技术的应用

- **中间代码生成：将源程序翻译为中间代码**
  - 复杂性介于源语言和机器语言的一种表示形式
  - 便于生成目标代码、机器无关优化、移植

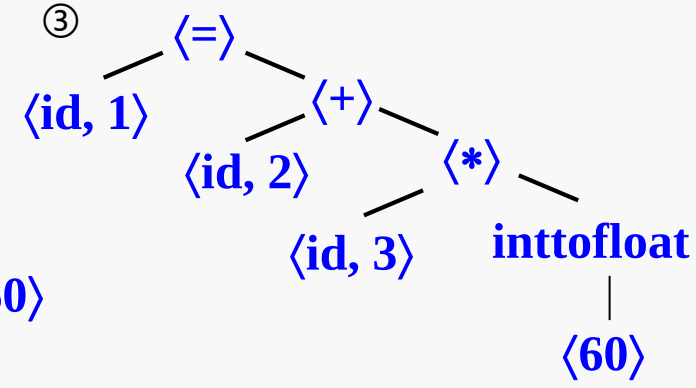
①

position = initial + rate * 60

②

⟨id, 1⟩ ⟨=⟩ ⟨id, 2⟩ ⟨+⟩ ⟨id, 3⟩ ⟨*⟩ ⟨60⟩

③



④

t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3

语法制导翻译技术的应用

- **语义分析：对结构上正确的源程序进行上下文有关性质的审查**
  - 例：每个算符是否具有语言规范允许的运算对象

```
一个C程序片断
int arr[2], b;
b = arr * 10;
```

语法制导翻译技术的应用

- **语义分析：对结构上正确的源程序进行上下文有关性质的审查**
  - 例：每个算符是否具有语言规范允许的运算对象
  - 例：数组访问越界

```
一个C程序片断
int a[10];
a[10] = 5;
```

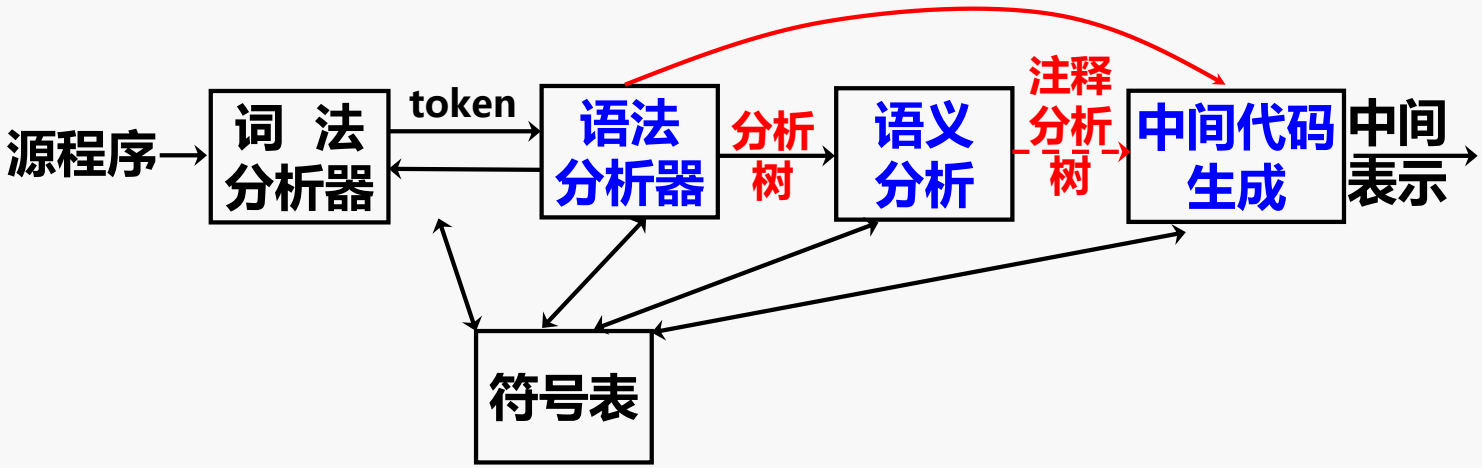
语法制导翻译技术的应用

- **语义分析：对结构上正确的源程序进行上下文有关性质的审查**
  - 例：每个算符是否具有语言规范允许的运算对象
  - 例：数组访问越界
  - 例：类型强制转换

语法制导翻译面临的问题

- **如何表示语义信息?**
  - 为CFG中的文法符号设置语义属性，用来表示语法成分对应的语义信息
- **如何计算语义属性?**
  - 文法符号的语义属性值是用与文法符号所在产生式（语法规则）相关联的语义规则来计算的
  - 对于给定的输入串x，构建x的语法分析树，并利用与产生式相关联的语义规则来计算分析树中各结点对应的语义属性值

本节提纲



- 语法制导翻译简介
- 语法制导定义
  - 属性、属性文法、综合属性及继承属性
  - 属性依赖图与计算次序

语法制导的定义

语法制导定义 (Syntax-Directed Definition, SDD)

- 基础的上下文无关文法
- 每个文法符号有一组属性
- 每个文法产生式$A \to \alpha$有一组形式为

  $b=f(c_1, c_2, \ldots, c_k)$的语义规则，其中$f$是函数

  b和$c_1, c_2, \ldots, c_k$是该产生式文法符号的属性

语法制导的定义

语法制导定义 (Syntax-Directed Definition, SDD)

- 基础的上下文无关文法
- 每个文法符号有一组属性
- 每个文法产生式$A \to \alpha$有一组形式为

  $b=f(c_1, c_2, \ldots, c_k)$的语义规则，其中$f$是函数

  b和$c_1, c_2, \ldots, c_k$是该产生式文法符号的属性
- 综合属性(synthesized attribute)：如果b是A的属性，$c_1, c_2, \ldots, c_k$是产生式右部文法符号的属性或A的其它属性
- 继承属性(inherited attribute)：如果b是右部某文法符号X的属性

终结符只能有综合属性，属性值无需计算，由词法分析给定

综合属性：举例

结尾标记

产 生 式	语 义 规 则
$L \rightarrow E$ n	$print\ (E.val)$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ digit	$F.val =$ digit.$lexval$

带副作用的规则（虚拟属性）

对E加下标以区分不同的属性值

词法分析给定

属性的计算

- 思考：如何将语义规则所引起的属性计算与语法分析结合起来?

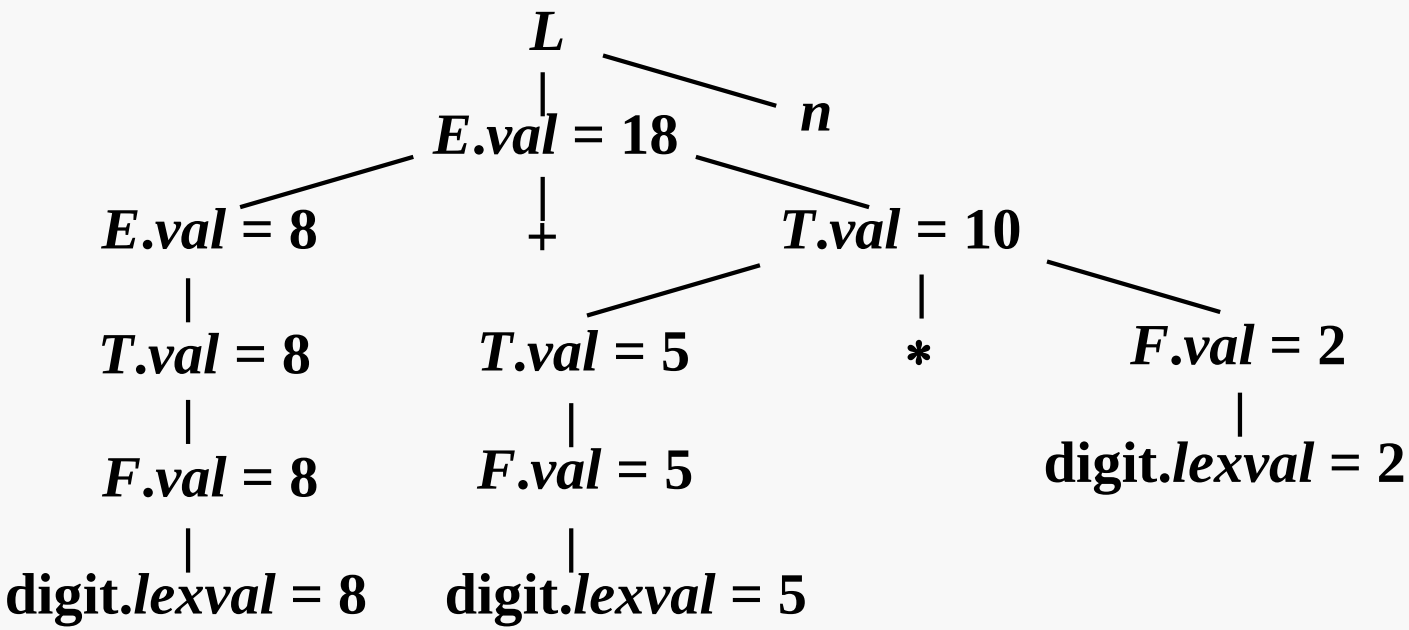
属性的计算

- 思考：如何将语义规则所引起的属性计算与语法分析结合起来?
- 语法分析的过程显式或隐式地构造了分析树(parse tree)
- 因此，可以先构造分析树，在分析树上，对每一个节点上的属性值进行求值
  - 注释分析树(Annotated parse tree)

注释分析树

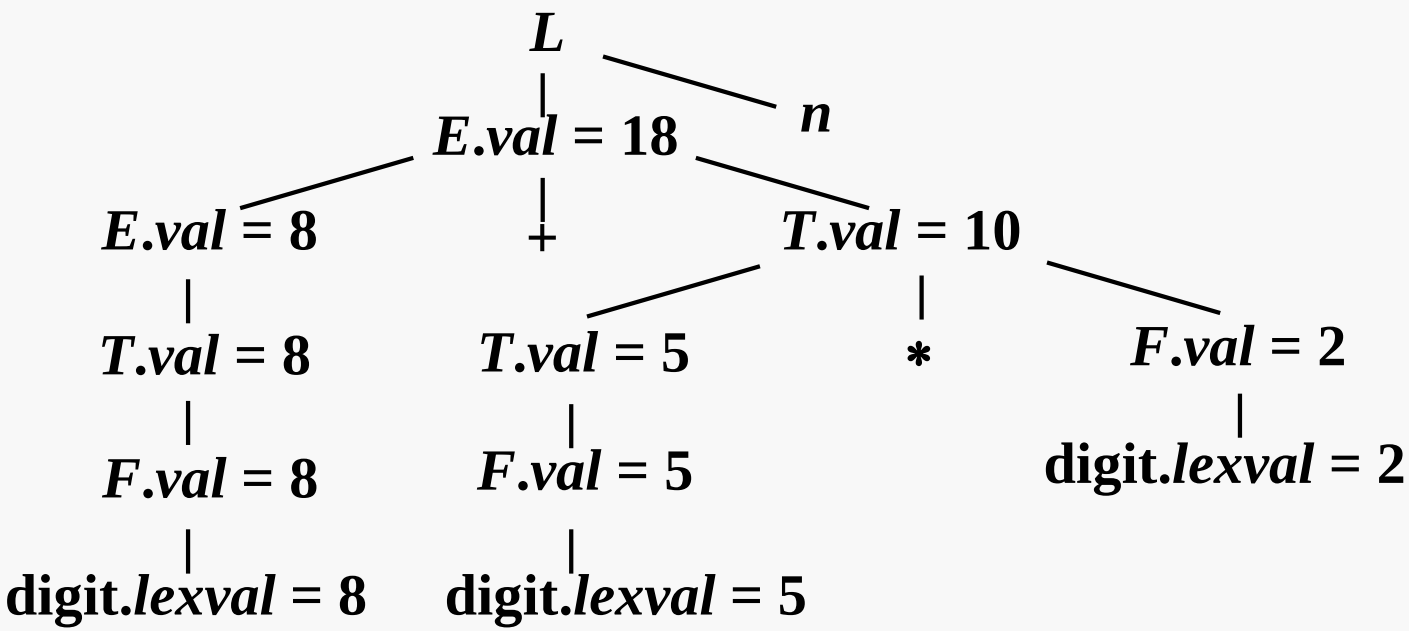
- **定义：结点的属性值都标注出来的分析树**

  8+5*2 n的注释分析树



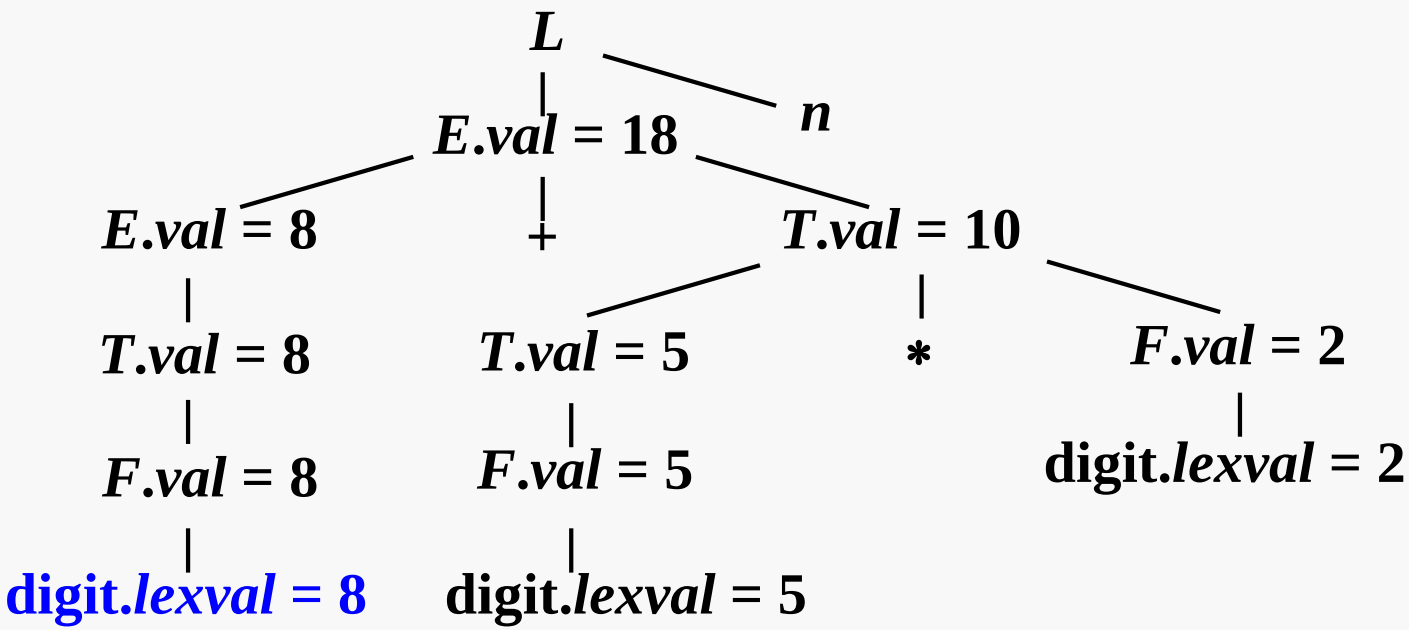
注释分析树+综合属性计算

- 各结点综合属性的计算可以自底向上地完成



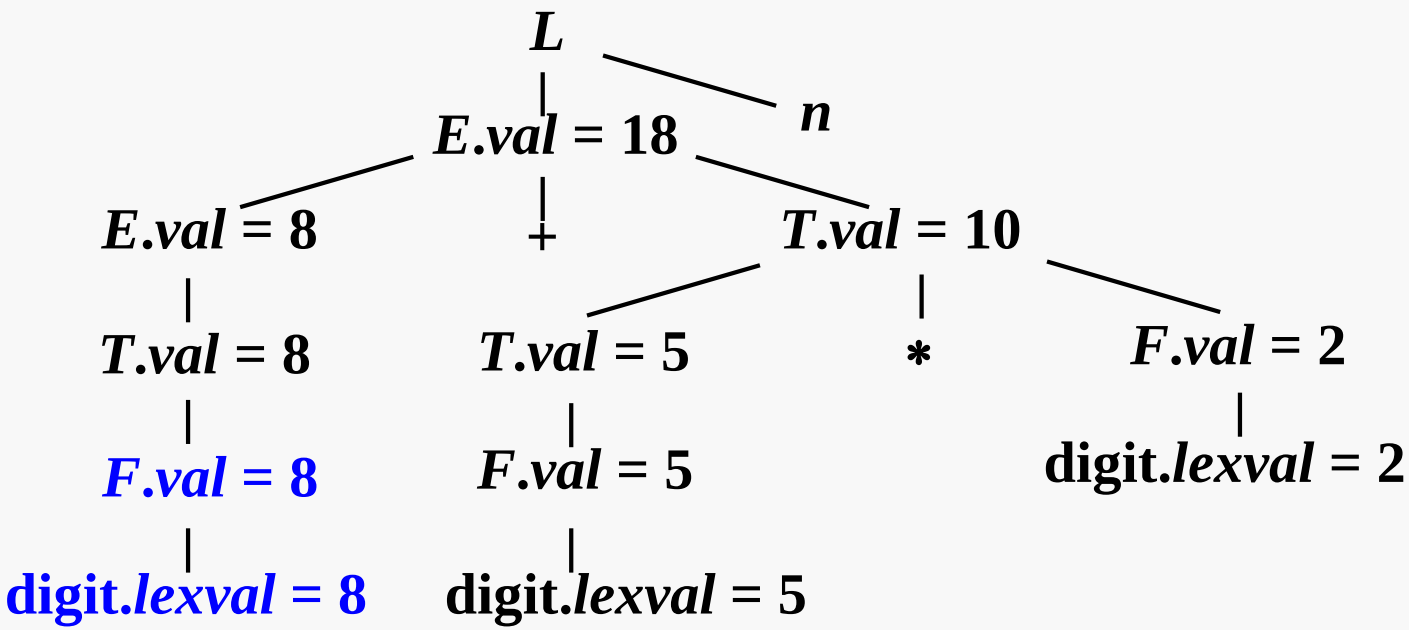
注释分析树+综合属性计算

- 各结点综合属性的计算可以自底向上地完成



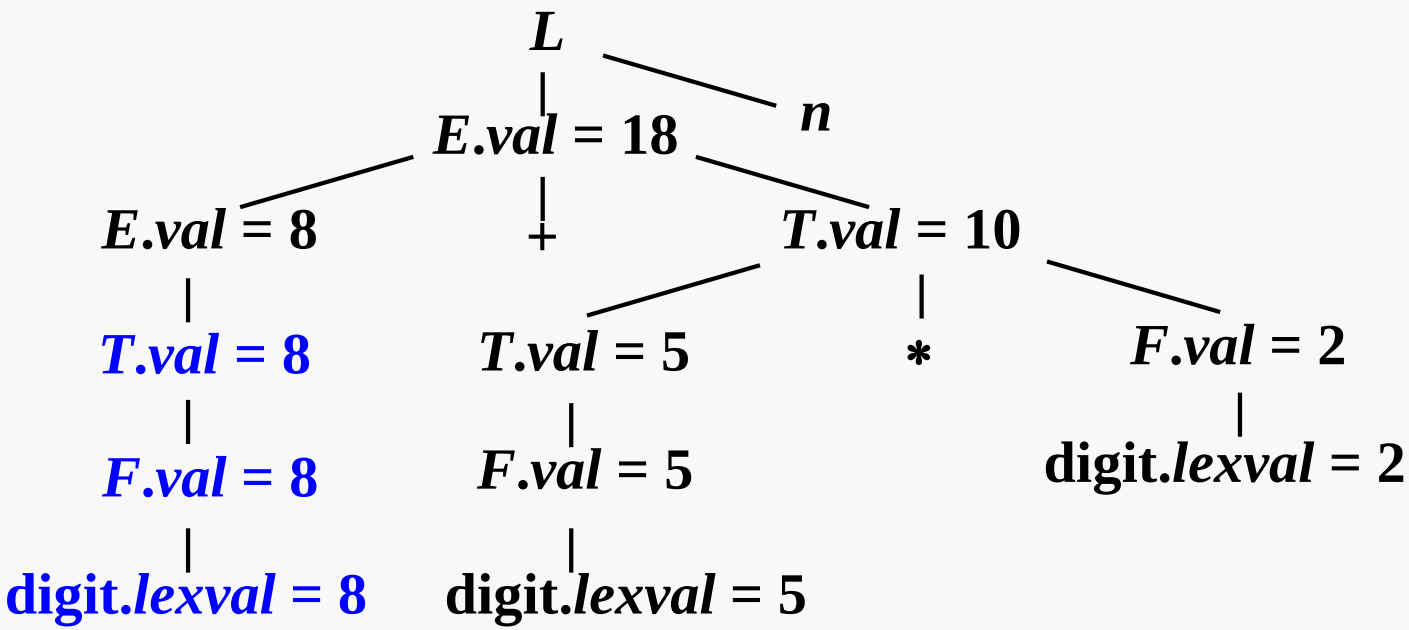
注释分析树+综合属性计算

- 各结点综合属性的计算可以自底向上地完成



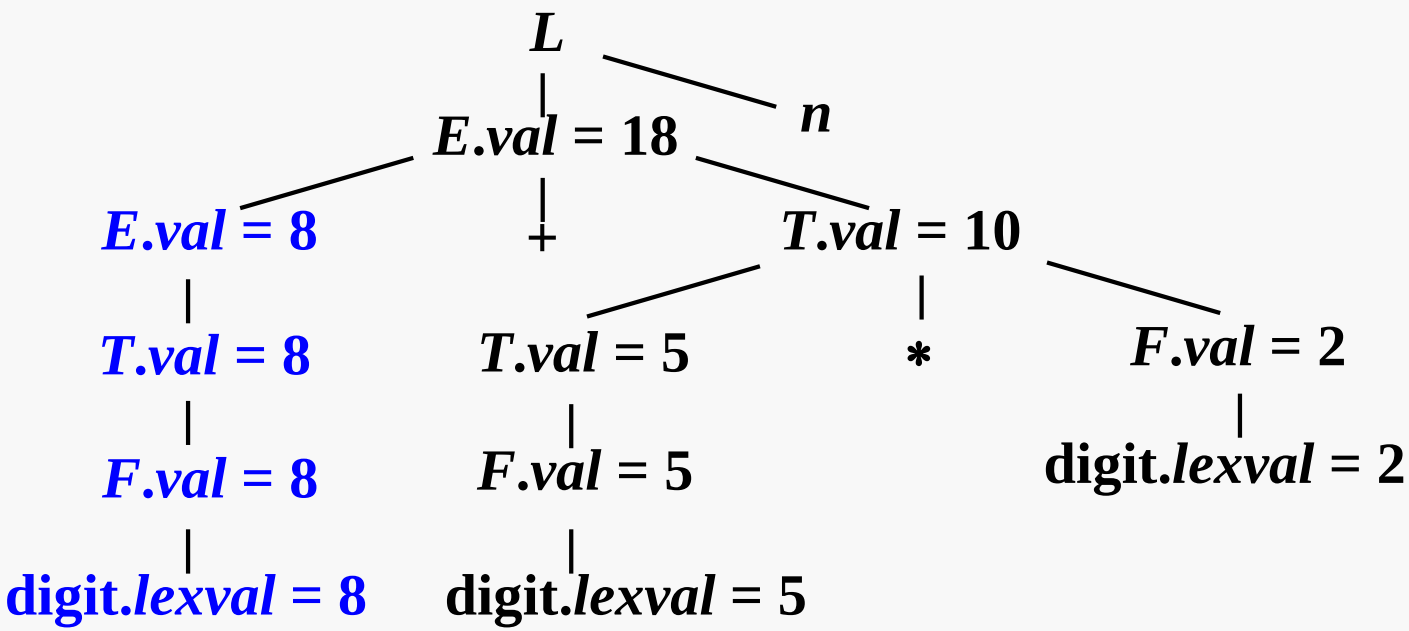
注释分析树+综合属性计算

- 各结点综合属性的计算可以自底向上地完成



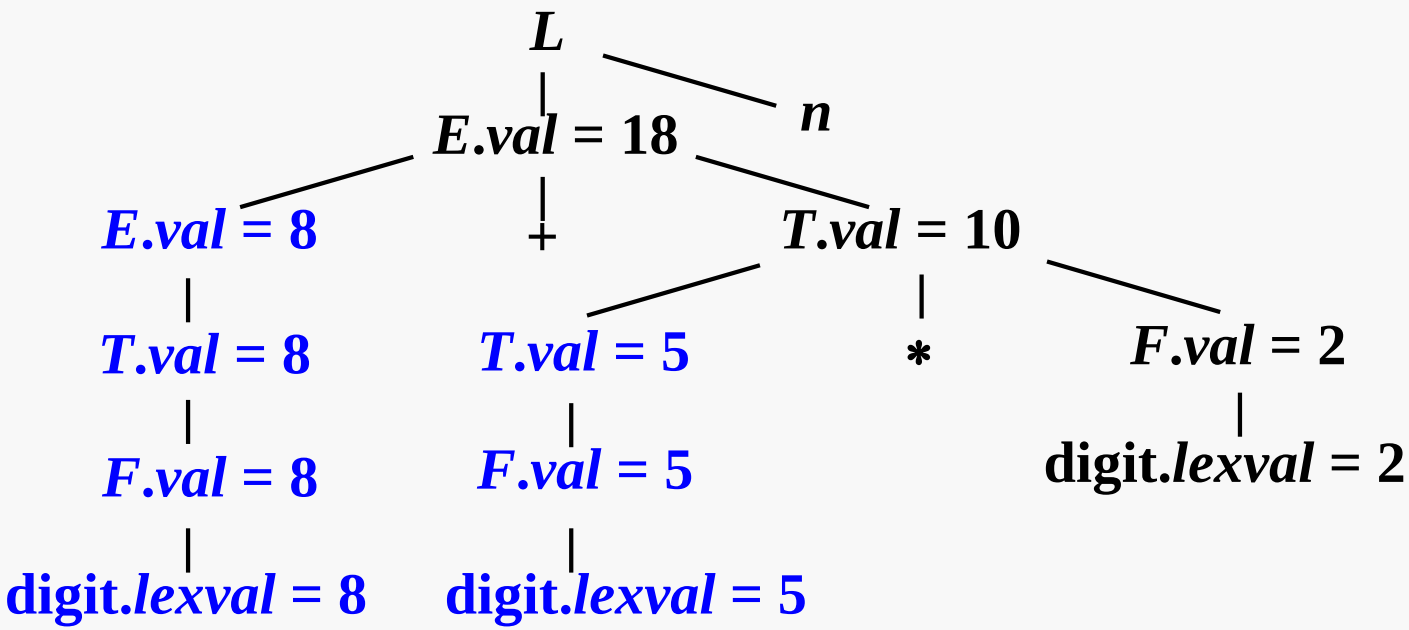
注释分析树+综合属性计算

- 各结点综合属性的计算可以自底向上地完成



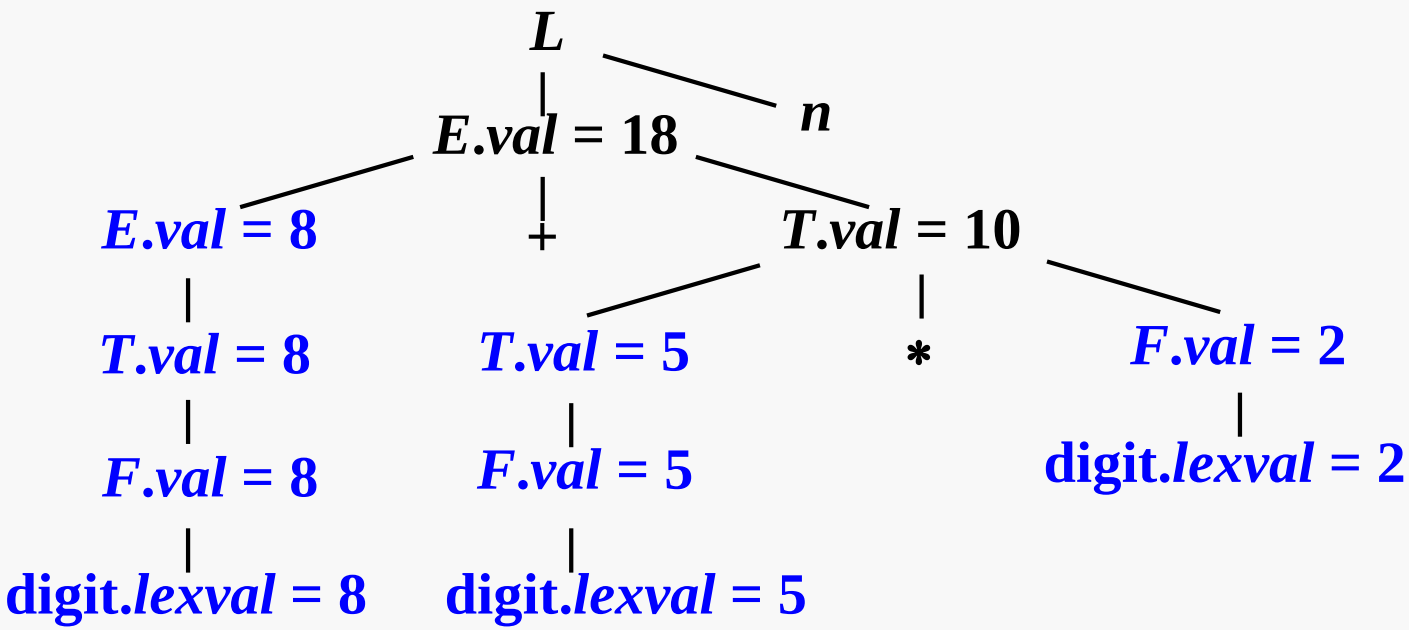
注释分析树+综合属性计算

- 各结点综合属性的计算可以自底向上地完成



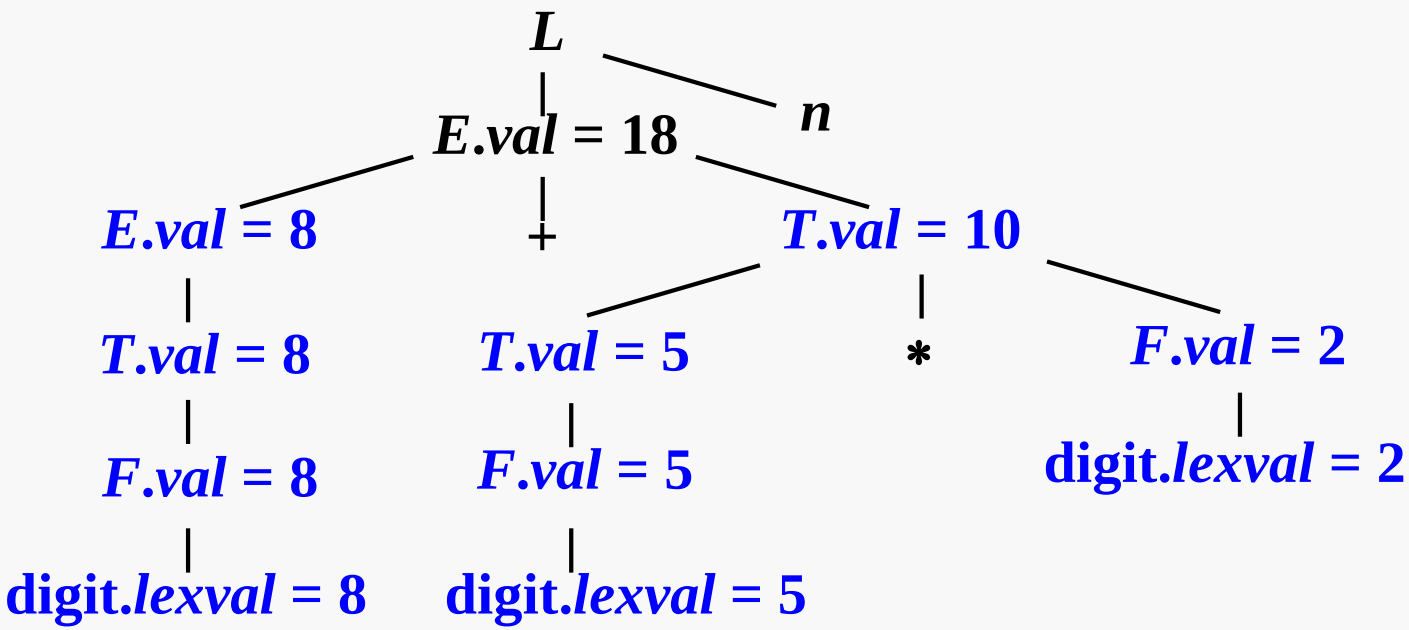
注释分析树+综合属性计算

- 各结点综合属性的计算可以自底向上地完成



注释分析树+综合属性计算

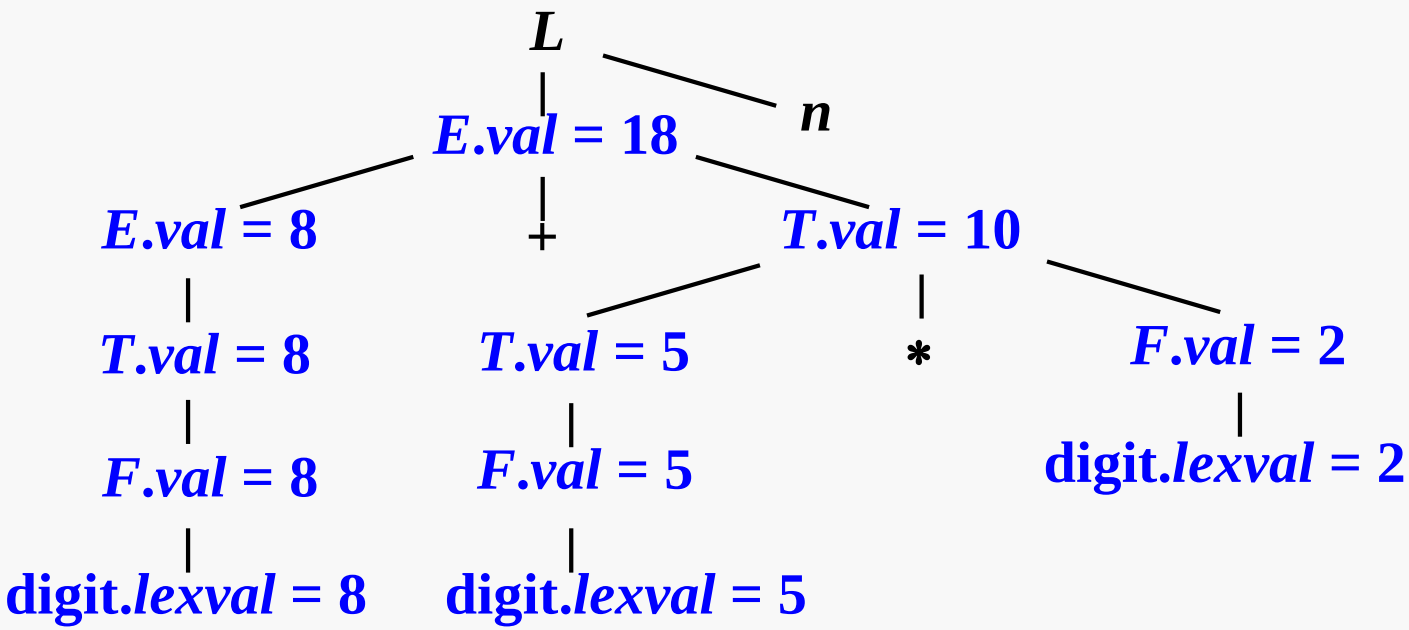
- 各结点综合属性的计算可以自底向上地完成



注释分析树+综合属性计算

- 各结点综合属性的计算可以自底向上地完成

综合属性的计算可以和LR分析器一起自然地实现。



例1

- 已知如下文法：

  $E \rightarrow E - T \mid T$

  $T \rightarrow num \mid num.num$

  - 写一个语法制导定义，来确定减法表达式的类型

例1

- 已知如下文法：

  $E \rightarrow E - T \mid T$

  $T \rightarrow num \mid num.num$

  - 写一个语法制导定义，来确定减法表达式的类型

- 设E和T有综合属性type，num的综合属性为integer，num.num的综合属性为float

产生式	语义规则
$E \rightarrow E_1 - T$	E.type = if (E_1.type == T.type) T.type else float
$E \rightarrow T$	E.type = T.type
$T \rightarrow num$	T.type = integer
$T \rightarrow num.num$	T.type = float

综合属性的局限

- 考虑消除左递归的算术表达式文法

产 生 式	语 义 规 则
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow$ digit	$F.val =$ digit.$lexval$

消除左递归

产 生 式
$T \rightarrow FT'$
$T' \rightarrow *FT_1'$
$T' \rightarrow \varepsilon$
$F \rightarrow$ digit

- 思考是否可以直接依赖综合属性val来计算算术表达式的值?

综合属性的局限

- 考虑消除左递归的算术表达式文法

产 生 式	语 义 规 则
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow$ digit	$F.val =$ digit.$lexval$

消除左递归

产 生 式
$T \rightarrow FT'$
$T' \rightarrow *FT_1'$
$T' \rightarrow \varepsilon$
$F \rightarrow$ digit

- **答案是否定的**，因为T对应的项中，第一个运算分量是F，而运算符和第二个运算分量在T'中

语法制导的定义

语法制导定义 (Syntax-Directed Definition, SDD)

- 基础的上下文无关文法
- 每个文法符号有一组属性
- 每个文法产生式$A \to \alpha$有一组形式为

  $b=f(c_1, c_2, \ldots, c_k)$的语义规则，其中$f$是函数

  b和$c_1, c_2, \ldots, c_k$是该产生式文法符号的属性
- 综合属性(synthesized attribute)：如果b是A的属性，$c_1, c_2, \ldots, c_k$是产生式右部文法符号的属性或A的其它属性
- 继承属性(inherited attribute)：如果b是右部某文法符号X的属性，$c_1, c_2, \ldots, c_k$是A和产生式右部文法符号的属性

借助继承属性

- 考虑消除左递归的算术表达式文法
- 为T'引入继承属性*inh*
  - 该属性继承了对应的*号的左运算分量

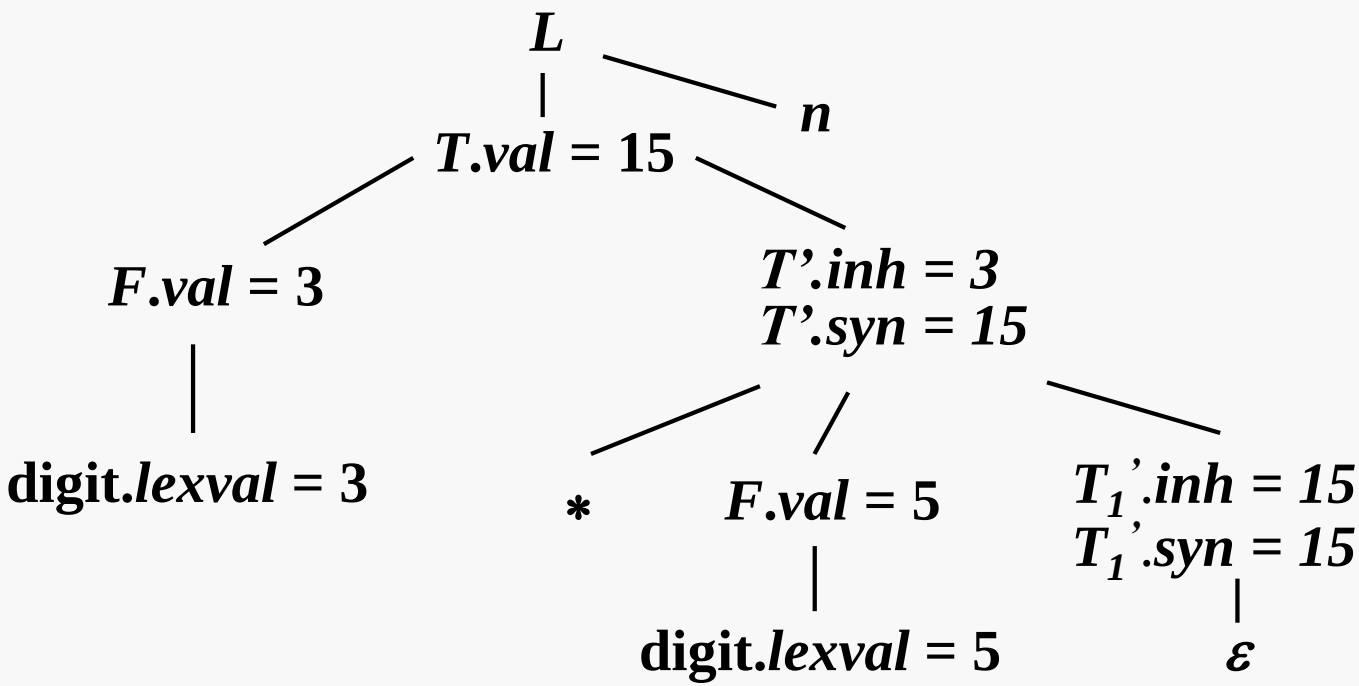
产 生 式	语 义 规 则
$T \rightarrow FT'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \rightarrow *FT_1'$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
$F \rightarrow$ digit	$F.val$ = digit.$lexval$

红色代表通过计算分量的值传递开来

蓝色代表最终结果返回

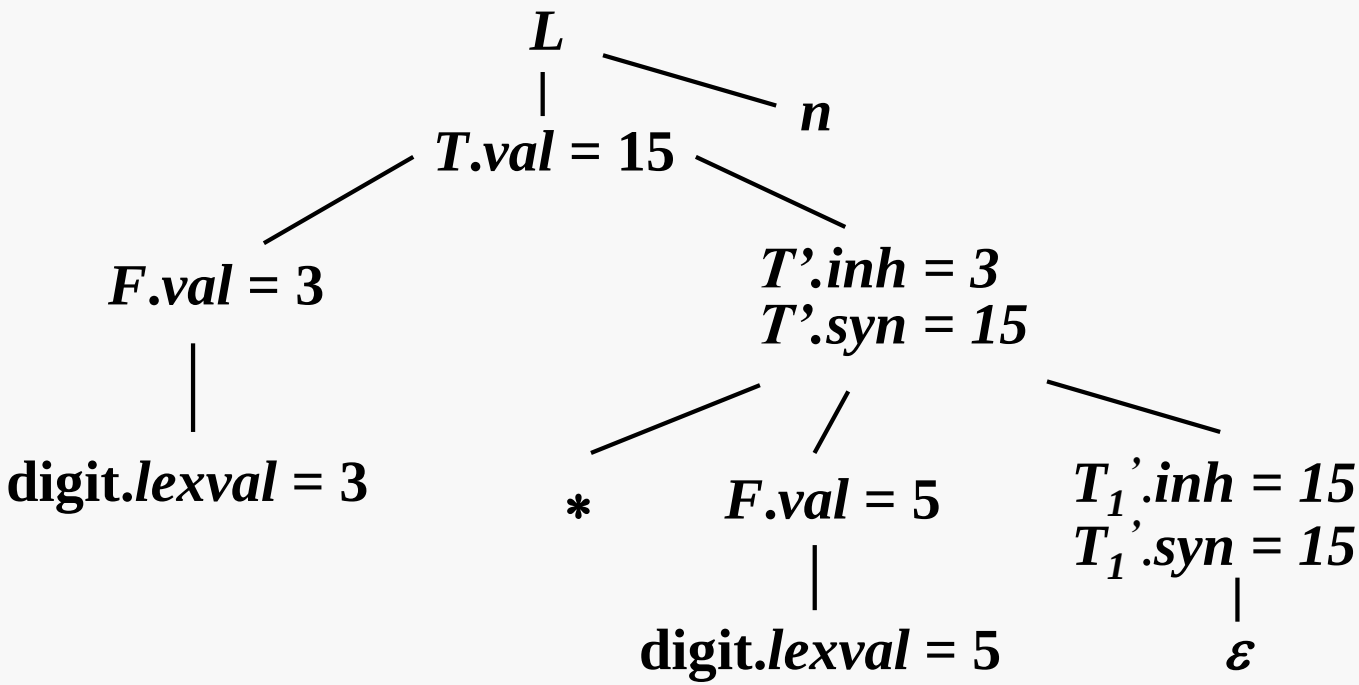
注释分析树+继承属性计算

- 3*5的注释分析树



注释分析树+继承属性计算

- 3*5的注释分析树
- 显然自底向上的计算方式是不合适的



SDD的求值顺序

- **SDD为CFG中的文法符号设置语义属性。**
  - 对于给定的输入串x，应用语义规则计算分析树中各结点对应的属性值

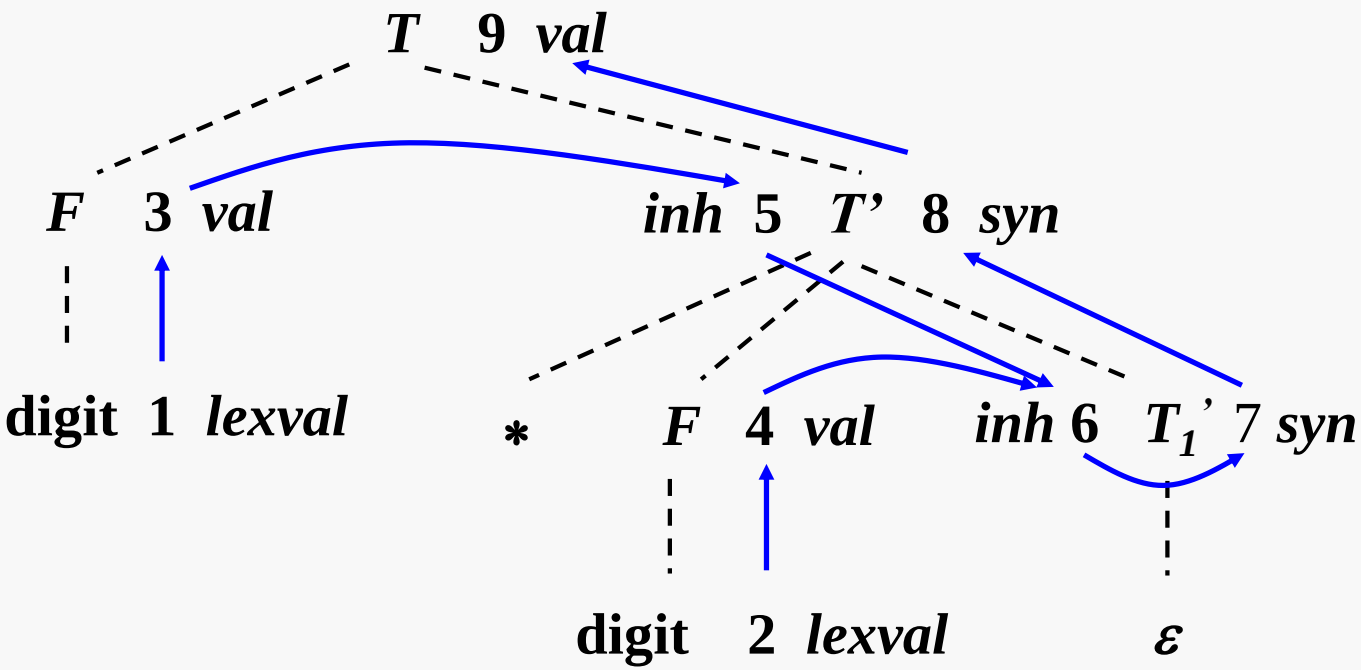
- **按照什么顺序计算属性值?**
  - 语义规则建立了属性之间的依赖关系，在对语法分析树节点的一个属性求值之前，必须首先求出这个属性值所依赖的所有属性值

属性依赖图

- **依赖图(dependency graph)是一个描述了分析树中结点属性间依赖关系的有向图**
  - 属性值为点(vertex)：分析树中每个标号为X的结点的每个属性a都对应着依赖图中的一个结点
  - 属性依赖关系为边(edge)：如果属性$X.a$的值依赖于属性$Y.b$的值，则依赖图中有一条从$Y.b$的结点指向$X.a$的结点的有向边

属性依赖图

- 以消除左递归的算术表达式文法和3*5为例

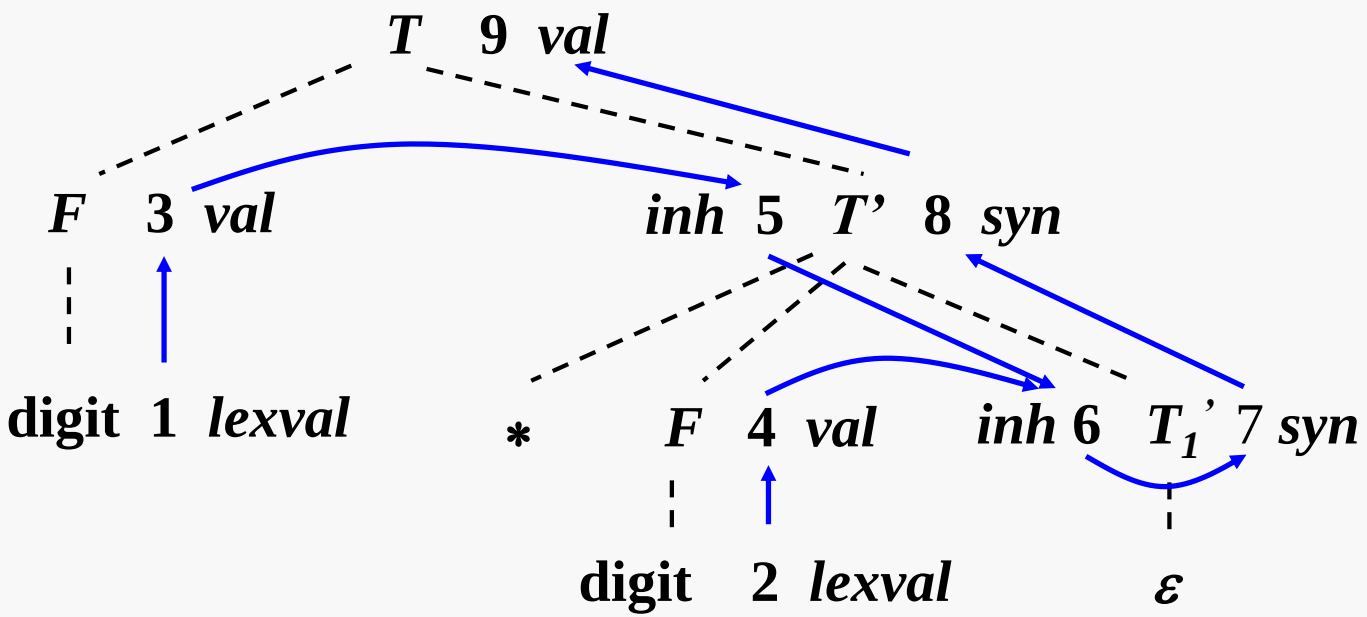


属性的计算次序

- **可行的求值顺序是满足下列条件的结点序列$N_1, N_2, \ldots, N_k$：**
  - 如果依赖图中有一条从结点N_i到N_j的边($N_i \rightarrow N_j$)，那么，在节点序列中，N_i排在N_j前面
  - 该排序称为这个图的拓扑排序(topological sort)

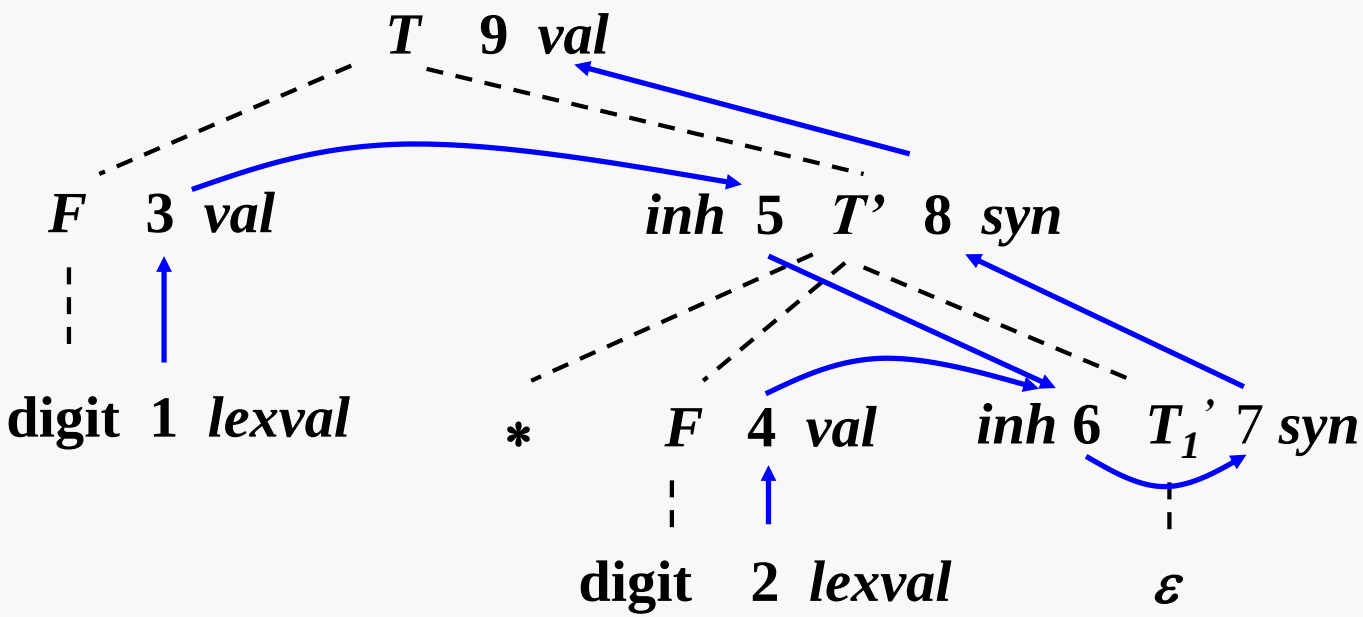
属性计算次序：举例

- 构造输入的分析树，构造属性依赖图，对结点进行拓扑排序，按拓扑排序的次序计算属性
  - 可行排序一：1, 2, 3, 4, 5, 6, 7, 8, 9
  - 可行排序二：2, 4, 1, 3, 5, 6, 7, 8, 9



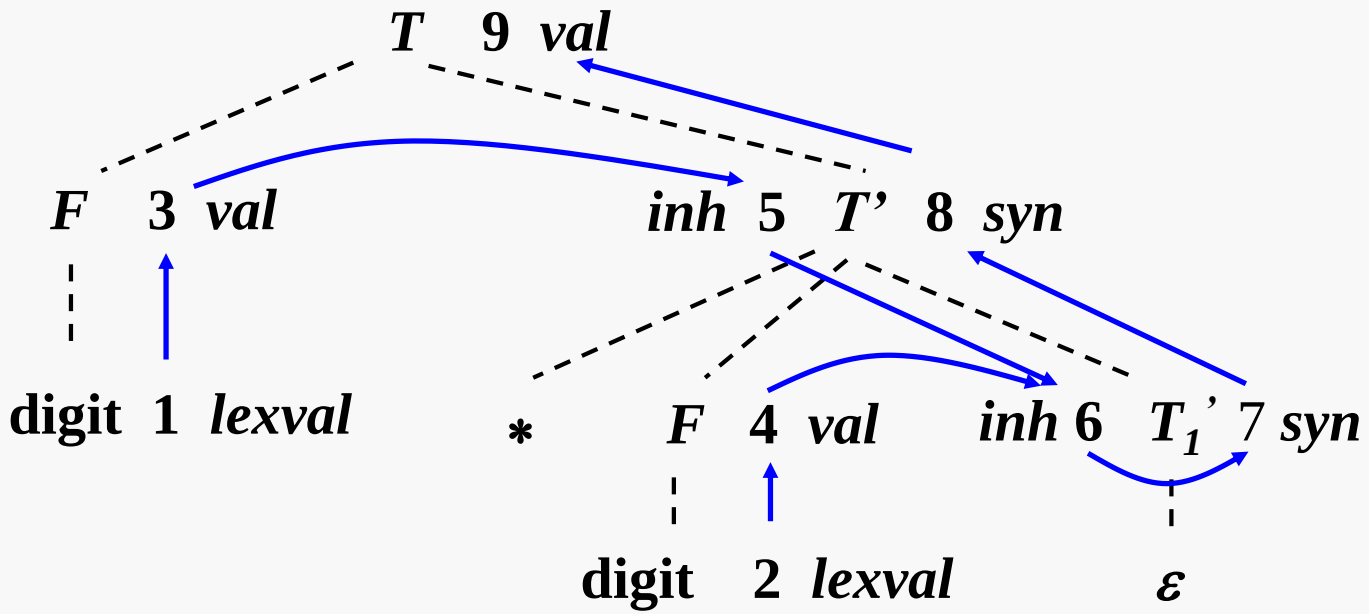
属性计算次序：举例

- 构造输入的分析树，构造属性依赖图，对结点进行拓扑排序，按拓扑排序的次序计算属性
  - 可行排序一：1, 2, 3, 4, 5, 6, 7, 8, 9
  - 可行排序二：2, 4, 1, 3, 5, 6, 7, 8, 9



属性计算次序：举例

- 构造输入的分析树，构造属性依赖图，对结点进行拓扑排序，按拓扑排序的次序计算属性
  - 可行排序一：1, 2, 3, 4, 5, 6, 7, 8, 9
  - 可行排序二：2, 4, 1, 3, 5, 6, 7, 8, 9



属性计算的问题

- 依赖于拓扑排序
- 思考：在有向图中，什么时候拓扑排序不存在?

属性计算的问题

- 依赖于拓扑排序
- 思考：在有向图中，什么时候拓扑排序不存在?
  - 当图中出现环的时候
  - SDD的属性之间存在循环依赖关系

属性计算的问题

- **依赖于拓扑排序**
- **思考：在有向图中，什么时候拓扑排序不存在?**
  - 当图中出现环的时候
  - SDD的属性之间存在循环依赖关系
- **解决方案：**
  - 使用某些特定类型的依赖图不存在环的SDD
    - S属性的SDD和L属性的SDD

《编译原理和技术》

语法制导翻译 I

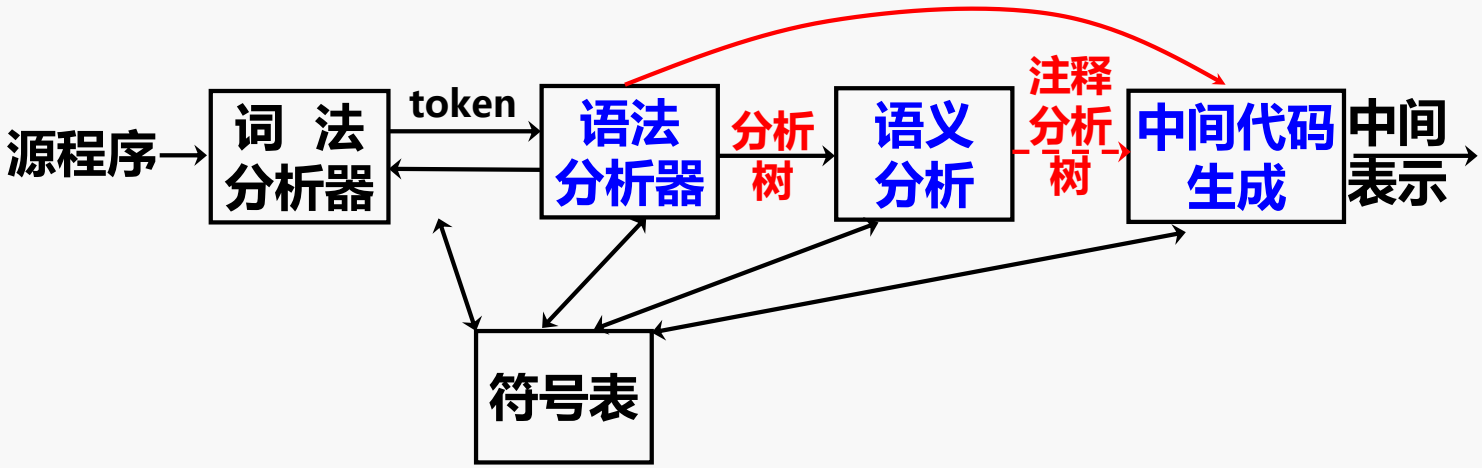
谢谢！

《编译原理和技术》

语法制导翻译 II

中科大计算机学院
李 诚
2022-10-08/10

本节提纲



- S属性的定义
- L属性的定义
- 语法制导定义的应用

S属性的定义(S-SDD)

- **仅仅使用综合属性的语法制导定义称为S属性的SDD，或S-属性定义、S-SDD**
  - 如果一个SDD是S属性的，可以按照语法分析树节点的任何自底向上顺序来计算它的各个属性值
  - S-属性定义可在自底向上的语法分析过程中实现
    - 例如：LR分析器

S属性的定义(S-SDD)

- 仅仅使用综合属性的语法制导定义称为S属性的SDD，或S-属性定义、S-SDD

产 生 式	语 义 规 则
$L \rightarrow E$ n	*print* (*E.val*)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ digit	$F.val =$ digit.*lexval*

例1

- **下面是产生字母表Σ = {0, 1, 2}上数字串的一个文法：**

  S → D S D | 2

  D → 0 | 1

  - 写一个语法制导定义，判断它接受的句子是否为回文数

例1

- **下面是产生字母表Σ = {0, 1, 2}上数字串的一个文法：**

  S → D S D | 2

  D → 0 | 1

  - 写一个语法制导定义，判断它接受的句子是否为回文数

$S' \rightarrow S$	print(S.*val*)
$S \rightarrow D_1\ S_1\ D_2$	S.*val* = (D_1.*val* == D_2.*val*) and S_1.*val*
$S \rightarrow 2$	S.*val* = *true*
$D \rightarrow 0$	D.*val* = 0
$D \rightarrow 1$	D.*val* = 1

例2

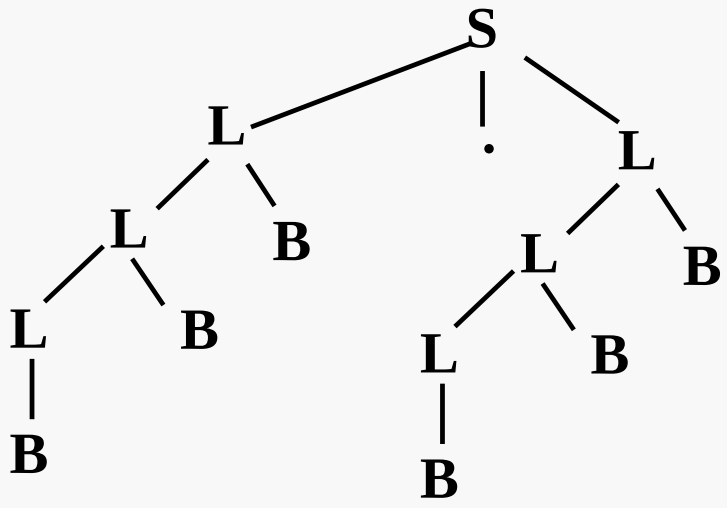
- 为下面文法写一个语法制导的定义，用S的综合属性*val*给出下面文法中S产生的二进制数的值。例如，输入101.101时，S.*val* = 5.625（可以修改文法）

  若按$2^2+0+2^0+2^{-1}+0+2^{-3}$来计算，需要继承属性来确定每个B离开小数点的距离

$S \rightarrow L . L \mid L$

$L \rightarrow L B \mid B$

$B \rightarrow 0 \mid 1$



例2

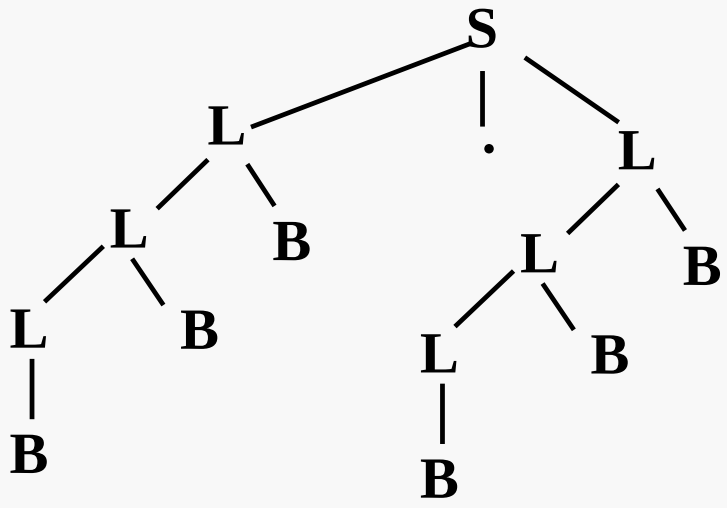
- 为下面文法写一个语法制导的定义，用S的综合属性*val*给出下面文法中S产生的二进制数的值。例如，输入101.101时，S.*val* = 5.625（可以修改文法）

  若小数点左边按$(1 \times 2 + 0) \times 2 + 1$计算。该办法不能直接用于小数点右边，右边计算需改成$((1 \times 2 + 0) \times 2 + 1)/2^3$，这时需要综合属性来统计B的个数

$S \rightarrow L \,.\, L \mid L$

$L \rightarrow L\ B \mid B$

$B \rightarrow 0 \mid 1$



例2

- 为下面文法写一个语法制导的定义，用S的综合属性*val*给出下面文法中S产生的二进制数的值。例如，输入101.101时，S.*val* = 5.625（可以修改文法）

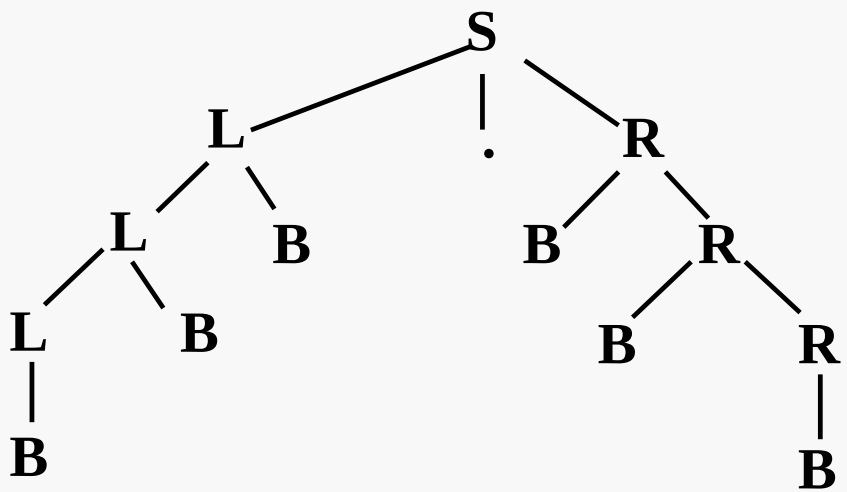
  更清楚的办法是将文法改成下面的形式

$S \rightarrow L . R \mid L$

$L \rightarrow L B \mid B$

$R \rightarrow B R \mid B$

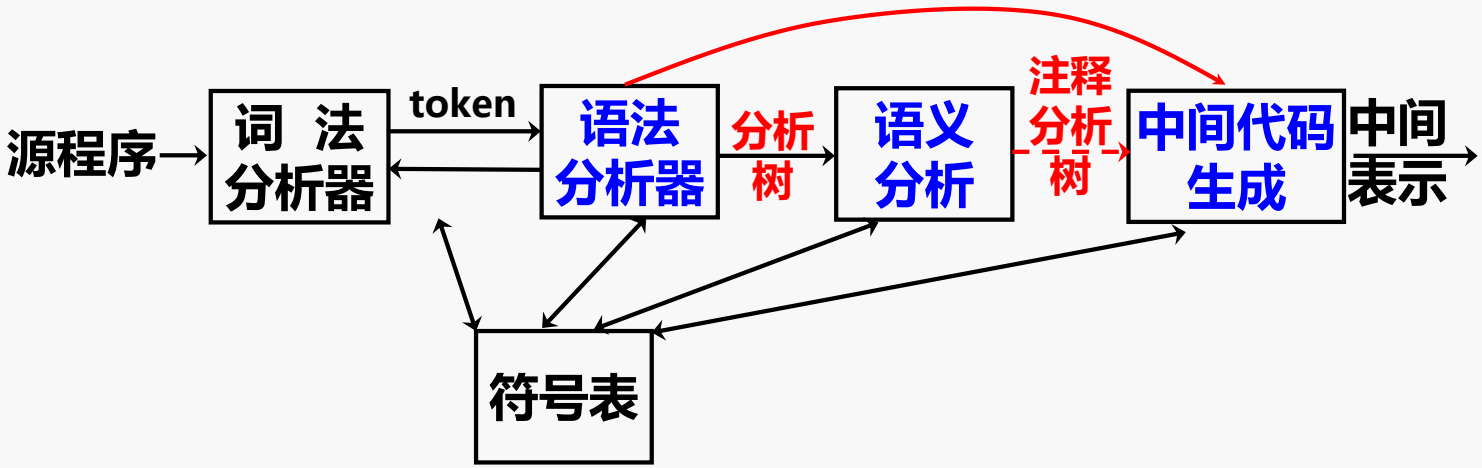
$B \rightarrow 0 \mid 1$



例2

$S \rightarrow L\ .\ R$	$S.\ val = L.\ val + R.\ val$
$S \rightarrow L$	$S.\ val = L.\ val$
$L \rightarrow L_1\ B$	$L.\ val = L_1.\ val \times 2 + B.\ val$
$L \rightarrow B$	$L.\ val = B.\ val$
$R \rightarrow B\ R_1$	$R.\ val = R_1.\ val\ /\ 2 + B.\ val\ /\ 2$
$R \rightarrow B$	$R.\ val = B.\ val\ /\ 2$
$B \rightarrow 0$	$B.\ val = 0$
$B \rightarrow 1$	$B.\ val = 1$

本节提纲



- S属性的定义
- L属性的定义
- 语法制导定义的应用

L属性的定义

- **L-属性定义(也称为L属性的SDD或L-SDD)的直观含义：**
  - 在一个产生式所关联的各属性之间，依赖图的边可以从左到右，但不能从右到左(因此称为L属性的，L是Left的首字母)
  - 可以在LR分析器或者LL分析器中实现
  - 更加一般化

L属性的定义

- **任意产生式$A \rightarrow X_1X_2...X_n$，其右部符号$X_i(1 \le i \le n)$的继承属性仅依赖于下列属性：**
  - A的继承属性
    - 如果依赖A的综合属性，由于A的综合属性可能依赖X_i的属性，包括X_i的综合属性和继承属性，因此可能形成环路
  - 产生式中X_i左边的符号$X_1,X_2,...,X_{i-1}$的属性
  - X_i本身的属性，但X_i的全部属性不能在依赖图中形成环路

L属性的定义

例：*L-SDD*

	产生式	语义规则
(1)	$T \rightarrow F\ T'$	$T'.inh = F.val$
		$T.val = T'.syn$
(2)	$T' \rightarrow *\ F\ T_1'$	$T_1'.inh = T'.inh \times F.val$
		$T'.syn = T_1'.syn$
(3)	$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
(4)	$F \rightarrow$ digit	$F.val =$ digit.*lexval*

继承属性

综合属性

L属性的定义

非*L*属性的*SDD*

➢例

产生式	语义规则
(1) $A \rightarrow LM$	$L.i = l(A.i)$ $M.i = m(L.s)$ $A.s = f(M.s)$
(2) $A \rightarrow QR$	$R.i = r(A.i)$ $Q.i = q(R.s)$ $A.s = f(Q.s)$

继承属性

综合属性

L属性的定义：举例

int id, id, id
标识符声明

产　生　式	语 义 规 则
$D \rightarrow TL$	$L.in = T.type$
$T \rightarrow$ int	$T.type = integer$
$T \rightarrow$ real	$T.type = real$
$L \rightarrow L_1$, id	$L_1.in = L.in$; $addType(\text{id}.entry, L.in)$
$L \rightarrow$ id	$addType(\text{id}.entry, L.in)$

- *type* – *T*的综合属性
- *in* – *L*的继承属性，把声明的类型传递给标识符列表
- *addType*– 把类型加到符号表中的标识符条目里(副作用)

属性文法vs.副作用

- 一个没有副作用的SDD称为属性文法
  - 属性文法增加了语义规则描述的复杂度
  - 如：符号表必须作为属性传递
  - 为了简单起见，我们可以把符号表作为全局变量，通过副作用函数读取或者添加标识符

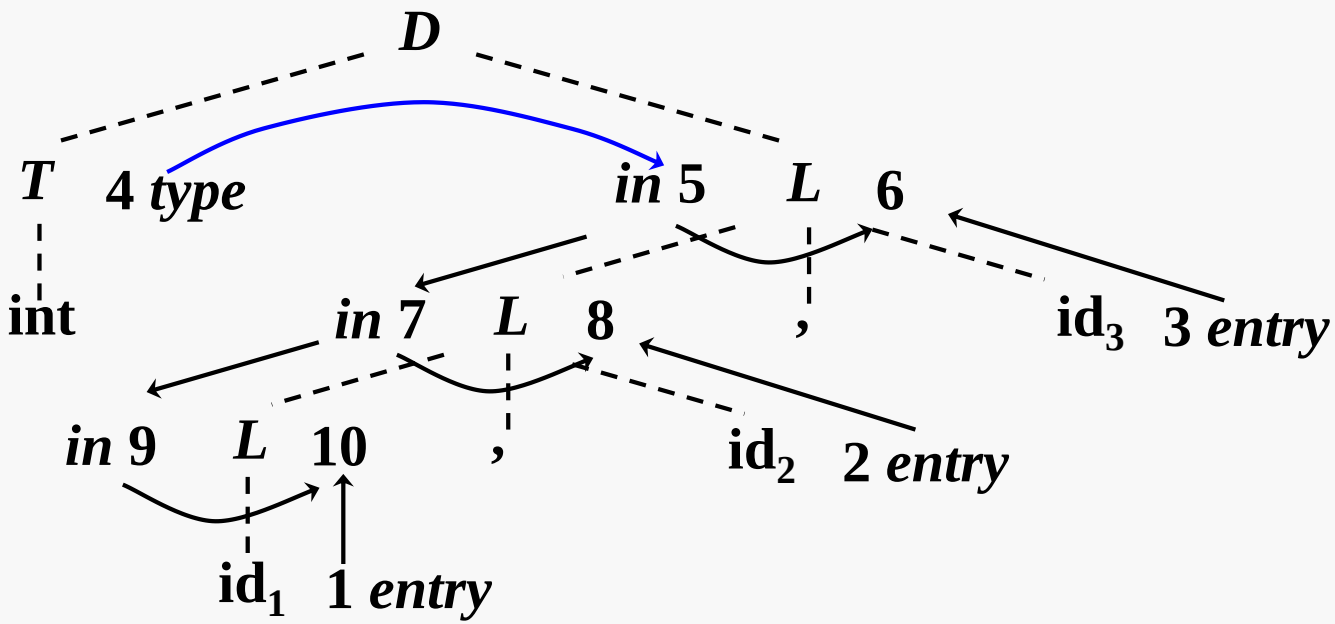
属性文法vs.副作用

- **一个没有副作用的SDD称为属性文法**
  - 属性文法增加了语义规则描述的复杂度
  - 如：符号表必须作为属性传递
  - 为了简单起见，我们可以把符号表作为全局变量，通过副作用函数读取或者添加标识符
- **受控的副作用**
  - 不会对属性求值产生约束，即可以按照任何拓扑顺序求值，不会影响最终结果
  - 或者对求值过程添加约束

L属性的定义：举例

- 例 int id_1, id_2, id_3的分析树（虚线）的依赖图（实线）

  $D \rightarrow TL$　　$L.in = T.type$

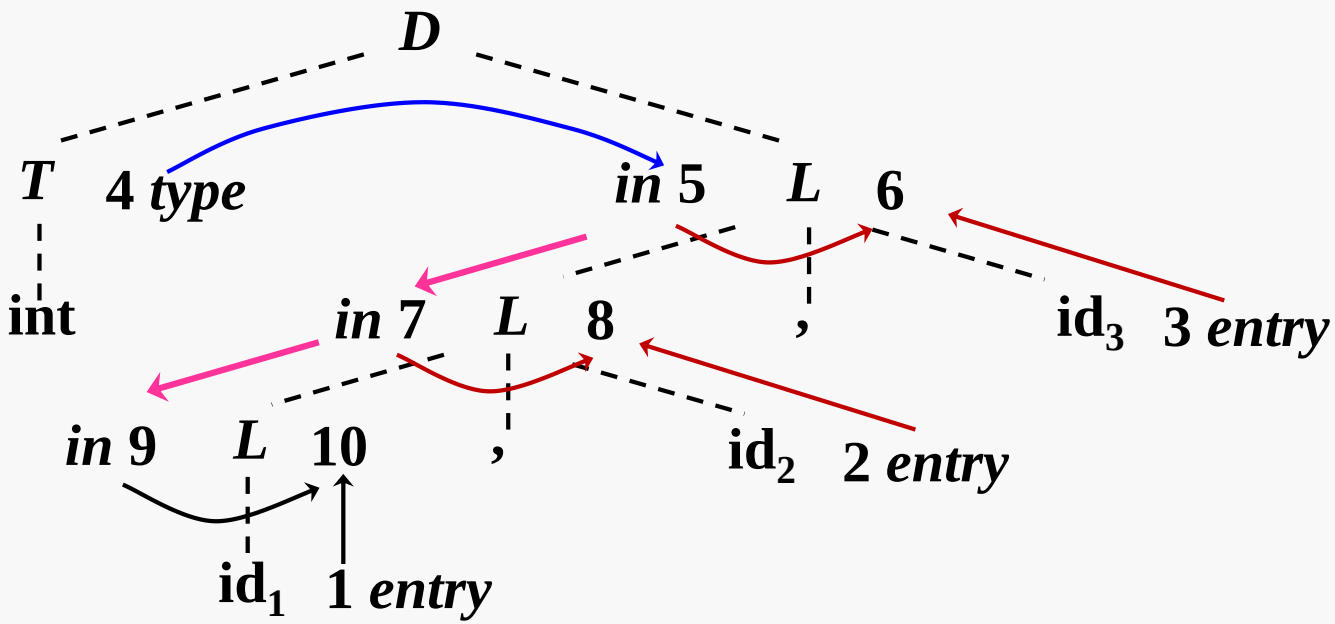


L属性的定义：举例

- 例 int id_1, id_2, id_3的分析树（虚线）的依赖图（实线）

  $L \rightarrow L_1$, id　$L_1.in = L.in$;

  addType (id.*entry*, *L.in*)

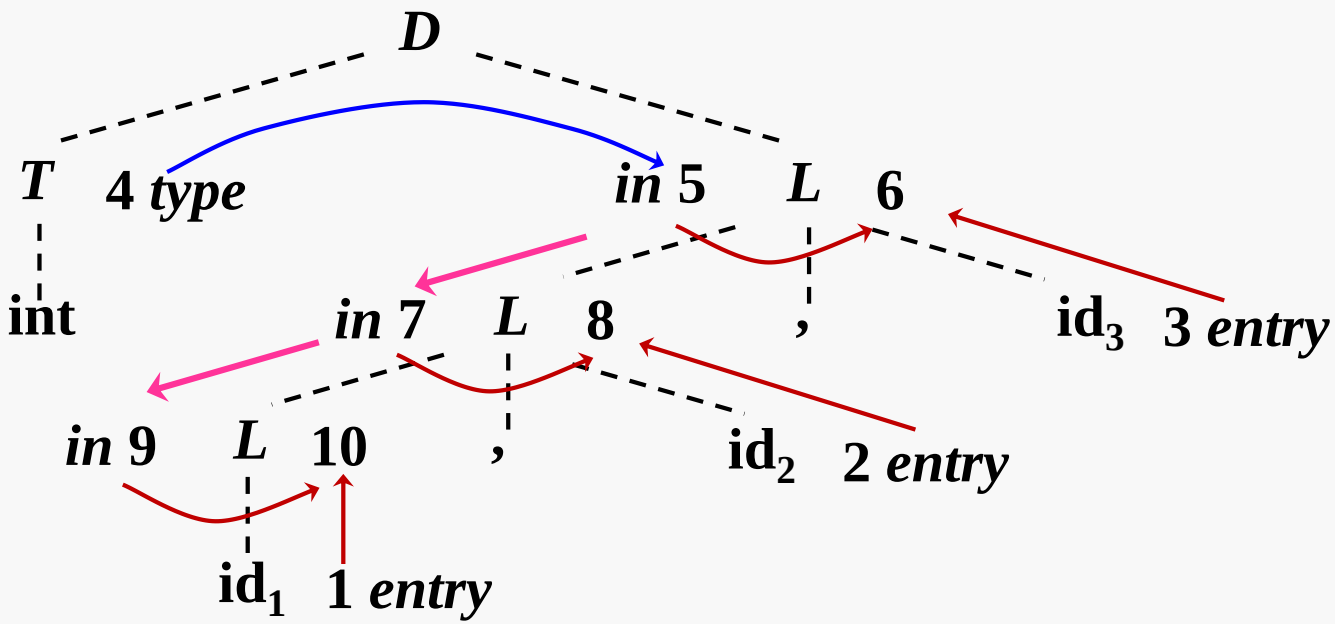


L属性的定义：举例

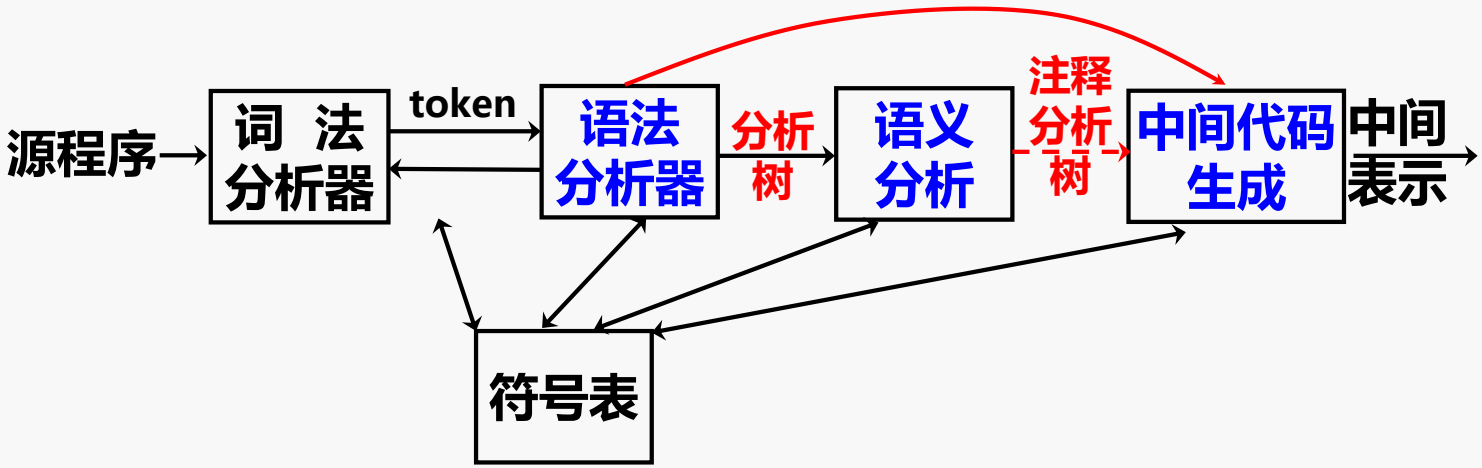
- 例 int id_1, id_2, id_3的分析树（虚线）的依赖图（实线）

  $L \rightarrow$ id

  addType (id.*entry*, *L.in*)



本节提纲



- S属性的定义
- L属性的定义
- **语法制导定义的应用**

语法制导定义的应用

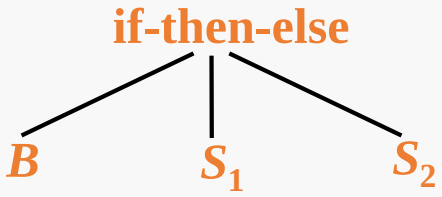
- **抽象语法树的构造**
  - S属性定义的方法
  - L属性定义的方法
- **类型检查(下一章)**
- **中间代码生成(下一章)**

抽象语法树

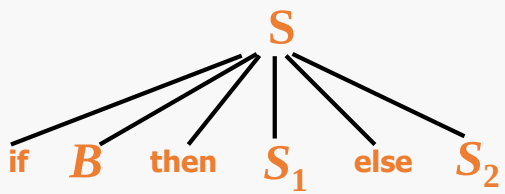
抽象语法树(Abstract syntax tree, AST)

- 简称语法树，是分析树的浓缩表示：算符和关键字是作为内部结点。
- 语法制导翻译可基于分析树，也可基于语法树

S→ if B then S1 else S2



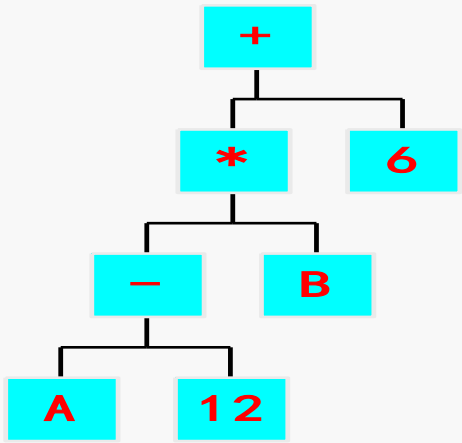
语法树



分析树

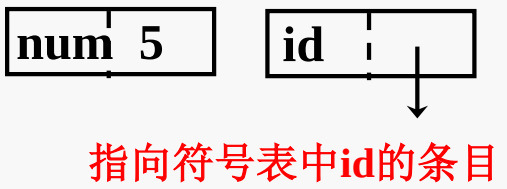
抽象语法树

- 例：表达式 (A - 12) * B + 6 的语法树。



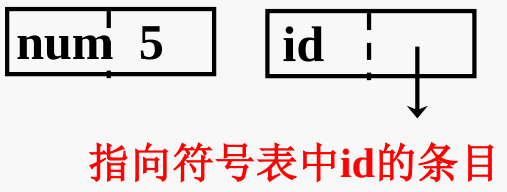
建立算符表达式的语法树

- 对**基本运算对象结点**
  - 一个域存放运算对象类别
  - 另一个域存放其值（也可用其他域保存其他属性或者指向该属性值的指针）

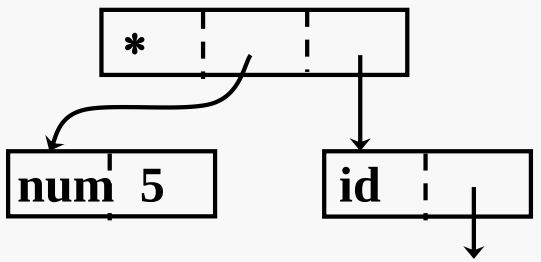


建立算符表达式的语法树

- 对**基本运算对象结点**
  - 一个域存放运算对象类别
  - 另一个域存放其值（也可用其他域保存其他属性或者指向该属性值的指针）



- 对**算符结点**
  - 一个域存放算符并作为该结点的标记
  - 其余两个域存放指向运算对象的指针。



建立算符表达式的语法树

- **mknode (op, left, right)**
  - 建立一个运算符号结点，标号是op，两个域left和right分别指向左子树和右子树。
- **mkleaf (id, entry)**
  - 建立一个标识符结点，标号为id，一个域entry指向标识符在符号表中的入口。
- **mkleaf (num, val)**
  - 建立一个数结点，标号为num，一个域val用于存放数的值。

构造抽象语法树的S属性语法制导定义

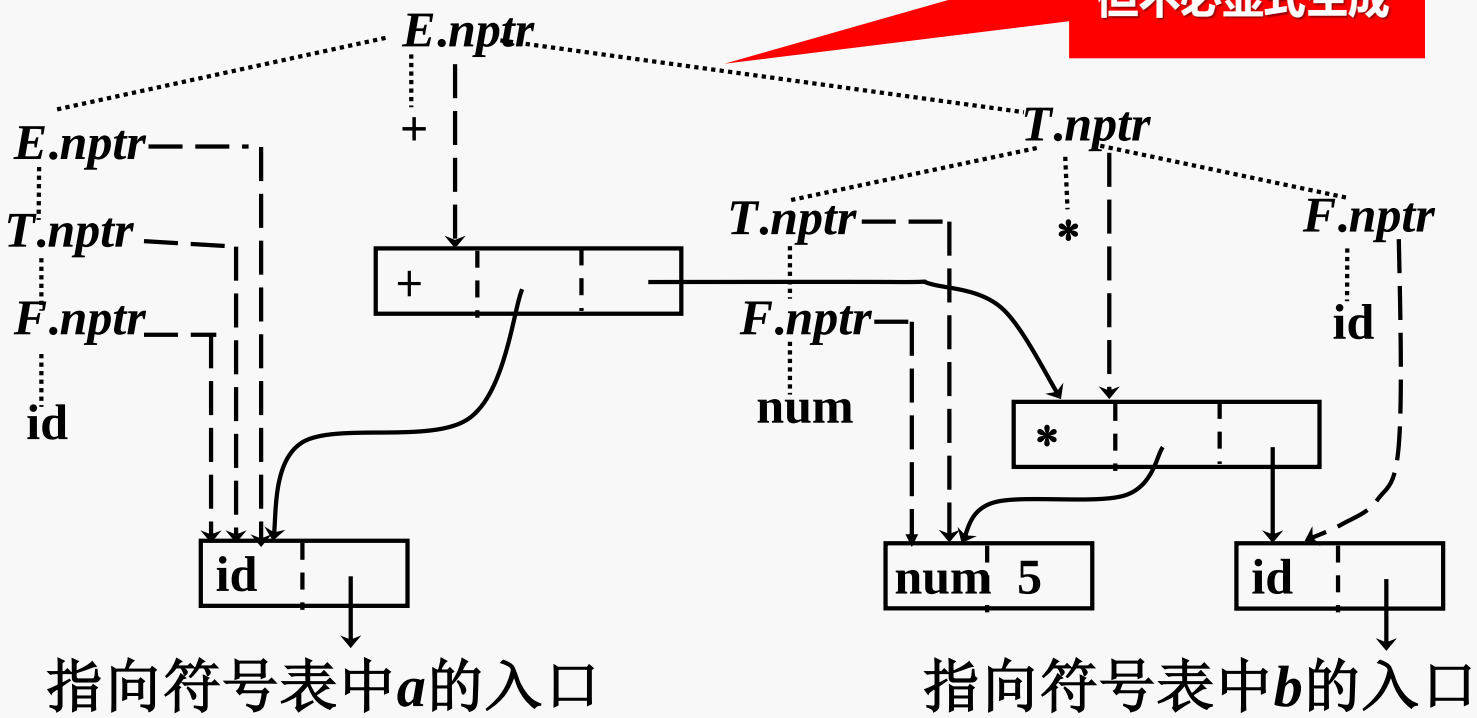
- 以算数表达式为例
  - nptr综合属性：文法符号对应的抽象语法树结点

产　生　式	语　义　规　则
$E \rightarrow E_1 + T$	$E.nptr = mkNode(\text{‘+’}, E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr = T.nptr$
$T \rightarrow T_1 * F$	$T.nptr = mkNode(\text{‘*’}, T_1.nptr, F.nptr)$
$T \rightarrow F$	$T.nptr = F.nptr$
$F \rightarrow (E)$	$F.nptr = E.nptr$
$F \rightarrow$ id	$F.nptr = mkLeaf$ (id, id.*entry*)
$F \rightarrow$ num	$F.nptr = mkLeaf$ (num, num.*val*)

S属性定义的语法树构造

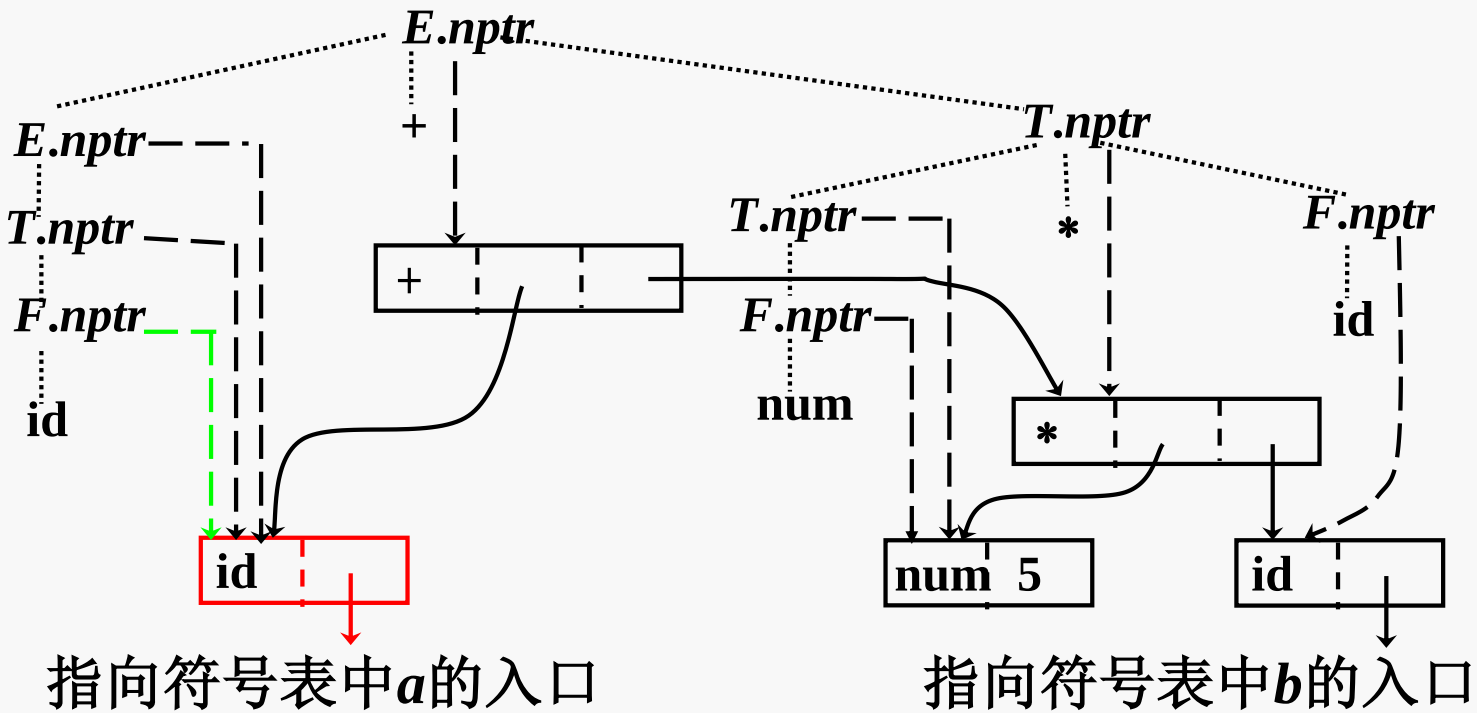
a+5*b的语法树的构造

不带箭头的虚线
代表语法分析树，
但不必显式生成



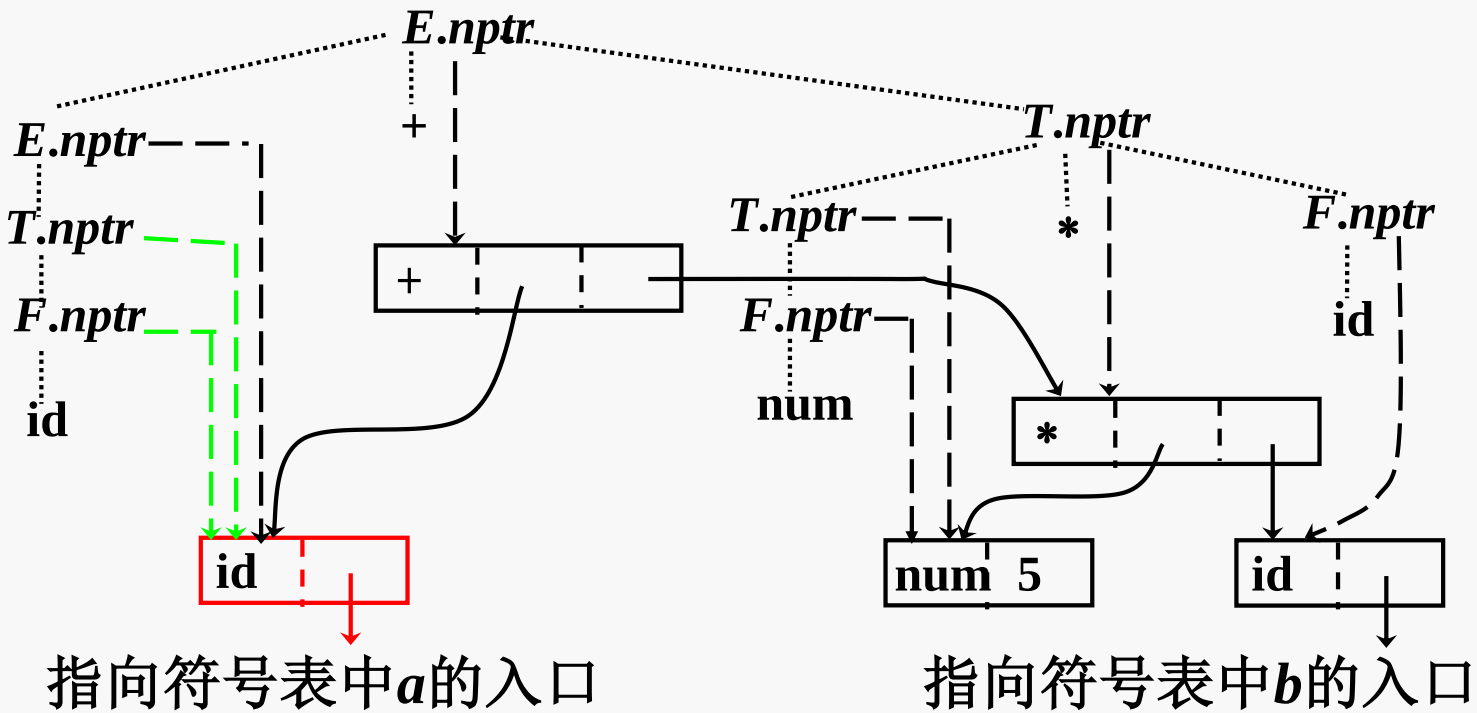
S属性定义的语法树构造

a+5*b的语法树的构造



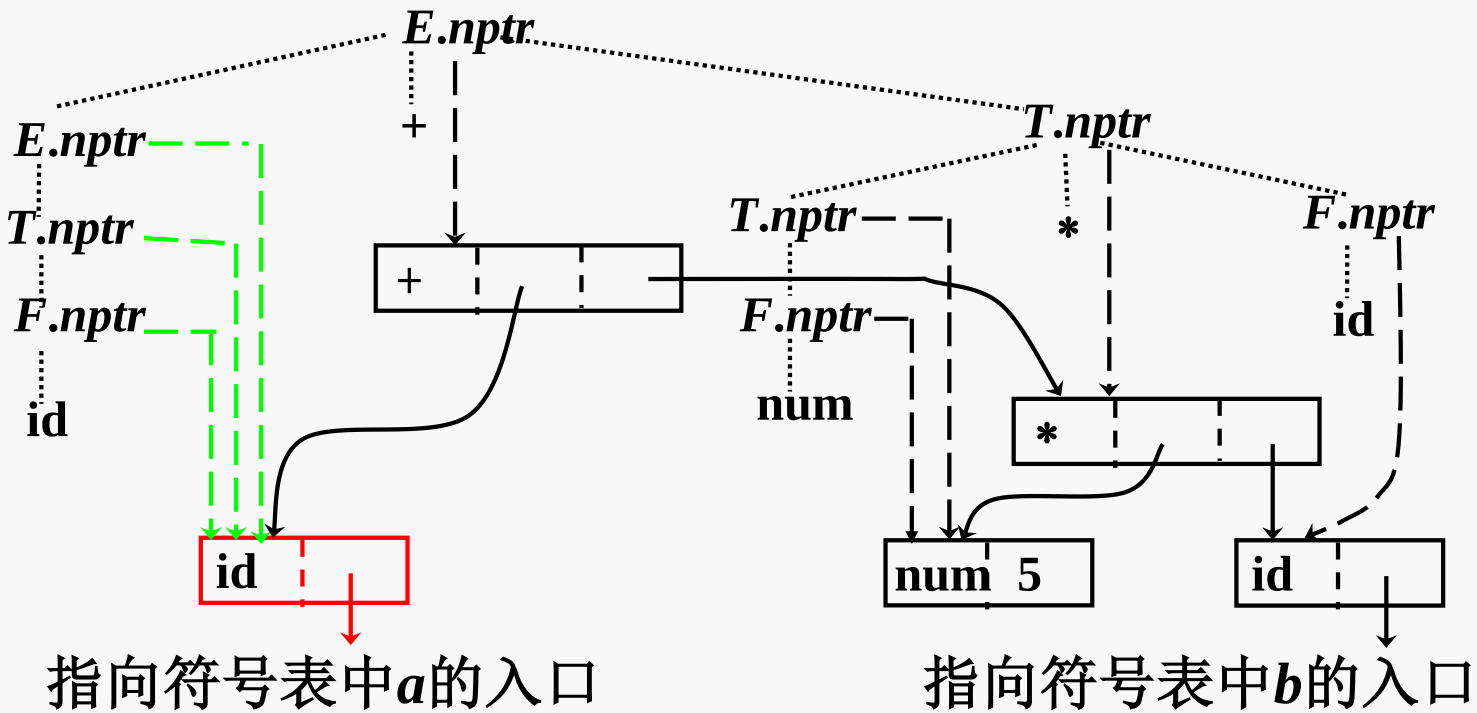
S属性定义的语法树构造

a+5*b的语法树的构造



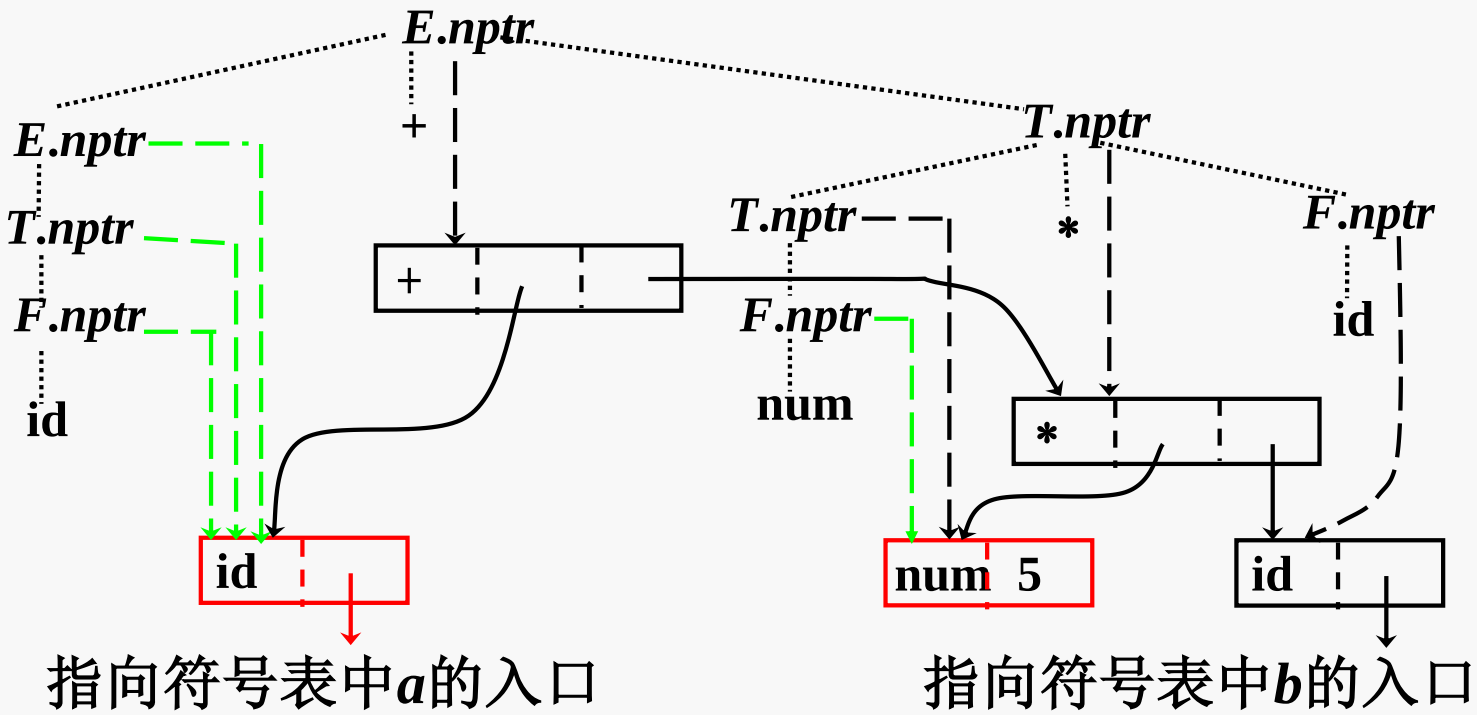
S属性定义的语法树构造

a+5*b的语法树的构造



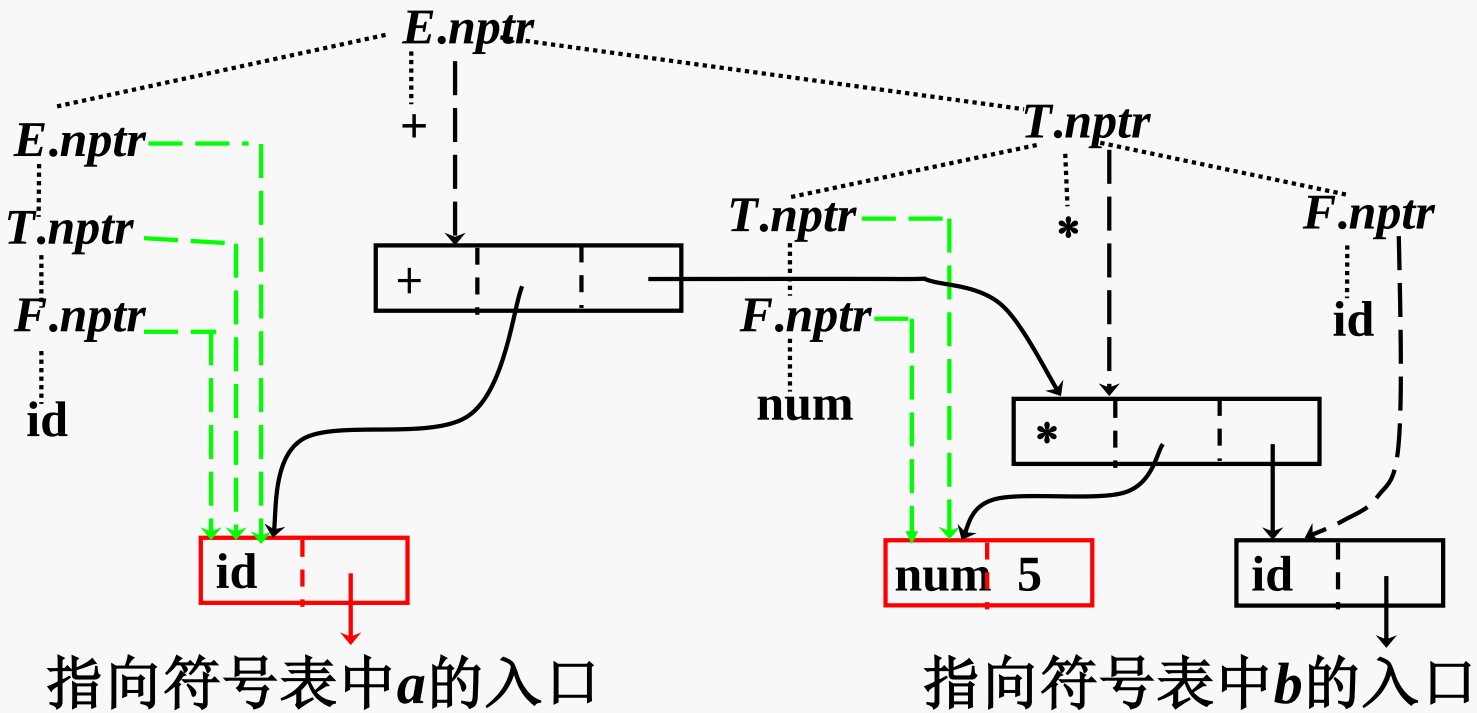
S属性定义的语法树构造

a+5*b的语法树的构造



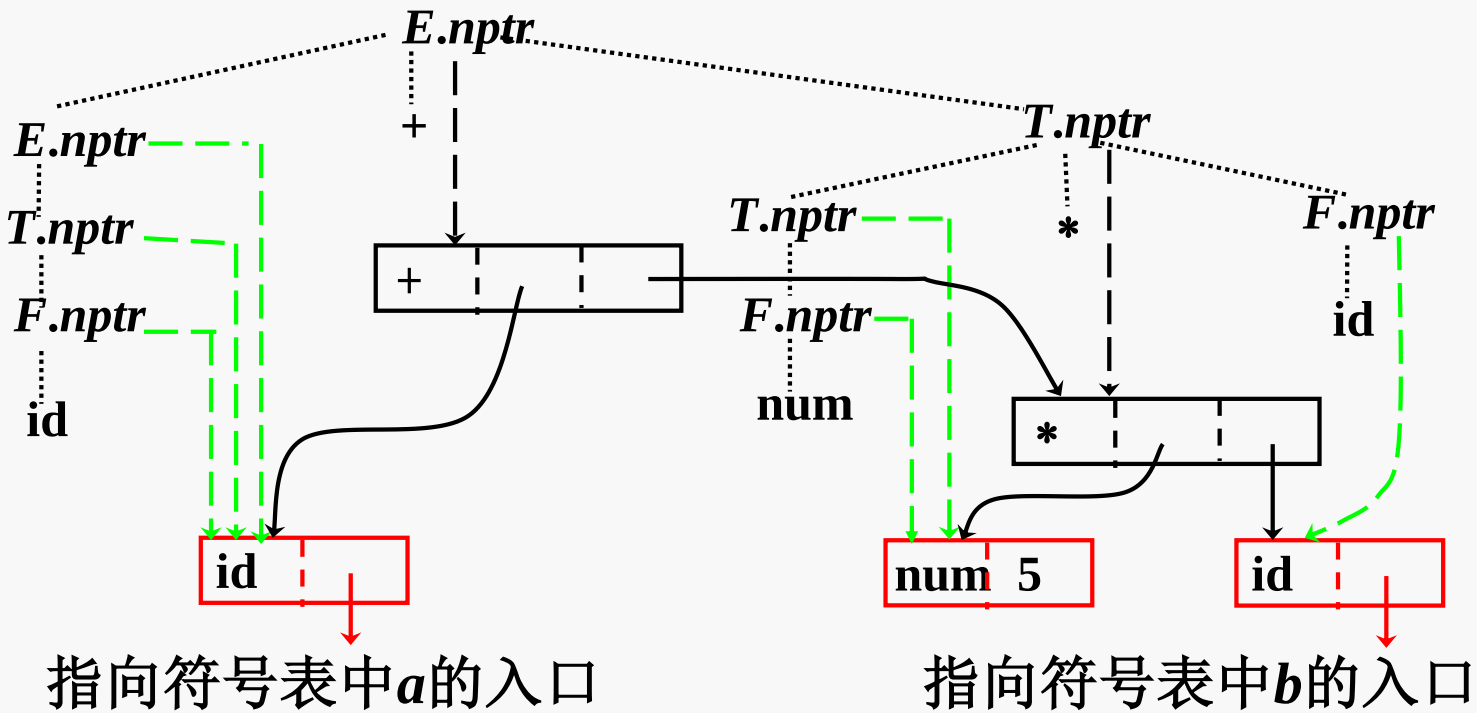
S属性定义的语法树构造

a+5*b的语法树的构造



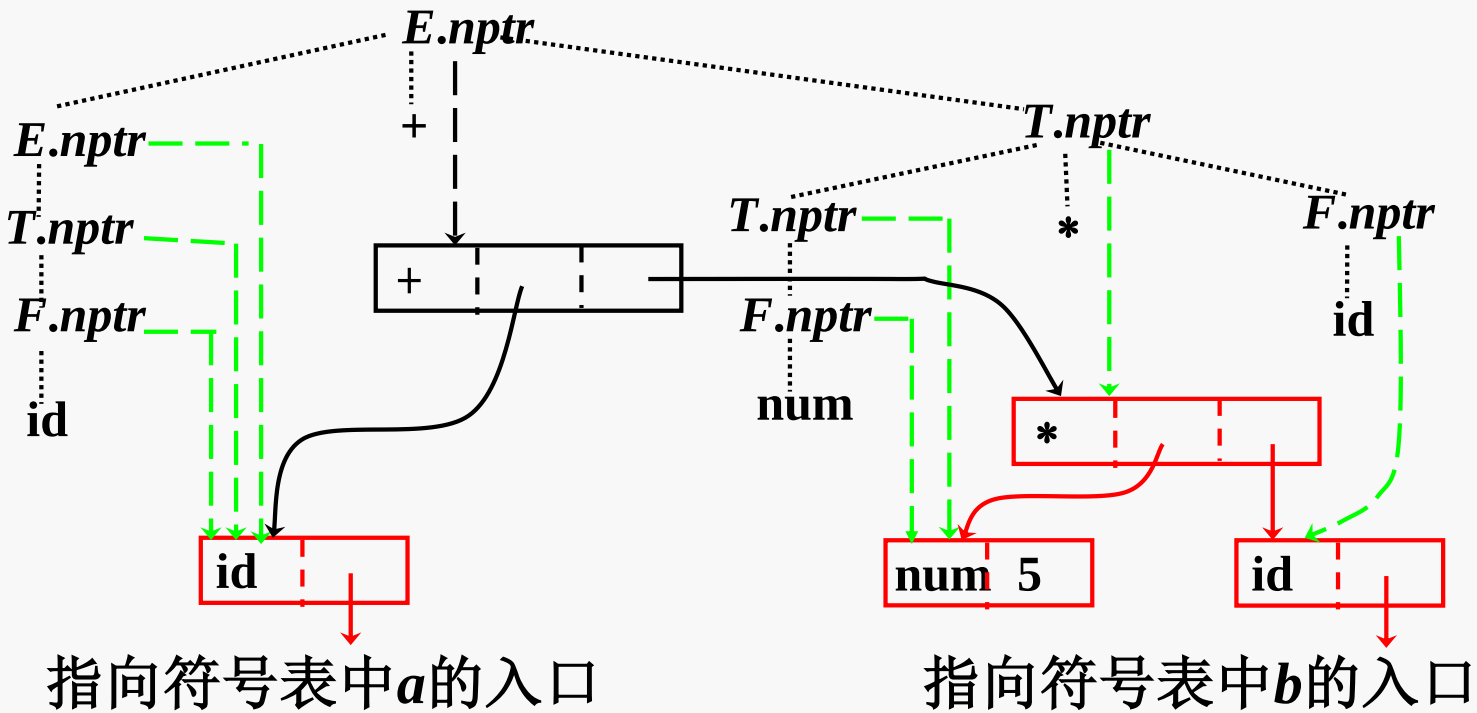
S属性定义的语法树构造

a+5*b的语法树的构造



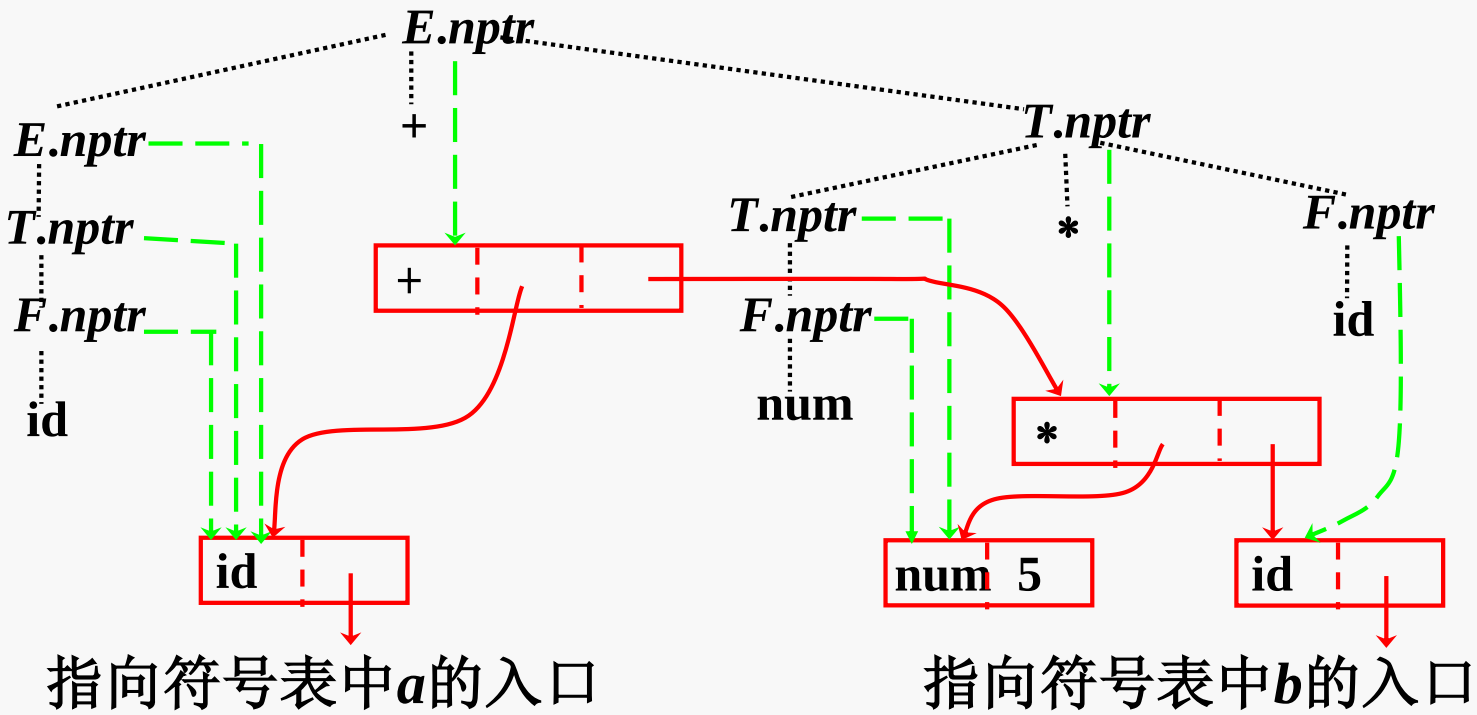
S属性定义的语法树构造

a+5*b的语法树的构造



S属性定义的语法树构造

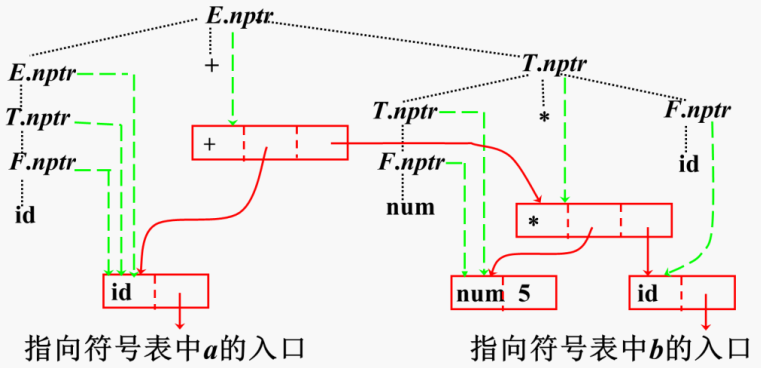
a+5*b的语法树的构造



S属性定义的语法树构造

a+5**b*的语法树构造步骤

1) p1:=mkleaf(id,entry a);
2) p2:=mkleaf(num, 5);
3) p3:=mkleaf(id, entry b)
4) p4:=mknode('*', p2, p3)
5) p5:=mknode('+', p1, p4)



$p_1, p_2, \ldots, p_5$是指向结点的指针，*entry a* 和 *entry c* 分别指向符号表中标识符*a*和*c*的指针。

S属性的SDD—小结

- 每个属性都是综合属性
- 在依赖图中，总是通过子结点的属性值来计算父结点的属性值。

S属性的SDD—小结

- **每个属性都是综合属性**
- **在依赖图中，总是通过子结点的属性值来计算父结点的属性值。**
  - 特别适合与自底向上的语法分析过程一起计算
    - 在用产生式归约时，即在构造分析树的中间节点时，计算相关属性（此时其子结点的属性必然已经计算完）

S属性的SDD—小结

- **每个属性都是综合属性**
- **在依赖图中，总是通过子结点的属性值来计算父结点的属性值。**
  - 特别适合与自底向上的语法分析过程一起计算
    - 在用产生式归约时，即在构造分析树的中间节点时，计算相关属性（此时其子结点的属性必然已经计算完）
  - 也可以与自顶向下的语法分析过程一起计算
    - 递归下降分析中，可以在过程A()的最后一步计算A的属性(此时，A调用的其他子结点过程已处理完)

语法制导定义的应用

- **抽象语法树的构造**
  - S属性定义的方法
  - L属性定义的方法
- **类型检查(下一章)**
- **中间代码生成(下一章)**

左递归的消除引起继承属性

- 考虑以下左递归文法

产　生　式	语　义　规　则
$E \rightarrow E_1 + T$	$E.nptr = mkNode(\text{‘+’}, E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr = T.nptr$
$T \rightarrow T_1 * F$	$T.nptr = mkNode(\text{‘*’}, T_1.nptr, F.nptr)$
$T \rightarrow F$	$T.nptr = F.nptr$
$F \rightarrow (E)$	$F.nptr = E.nptr$
$F \rightarrow \text{id}$	$F.nptr = mkLeaf\ (\text{id}, \text{id}.entry)$
$F \rightarrow \text{num}$	$F.nptr = mkLeaf\ (\text{num}, \text{num}.val)$

左递归的消除引起继承属性

- 首先消除左递归

$E \rightarrow E_1 + T$
$E \rightarrow T$
$T \rightarrow T_1 * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow$ id
$F \rightarrow$ num

$T + T + T + \ldots$

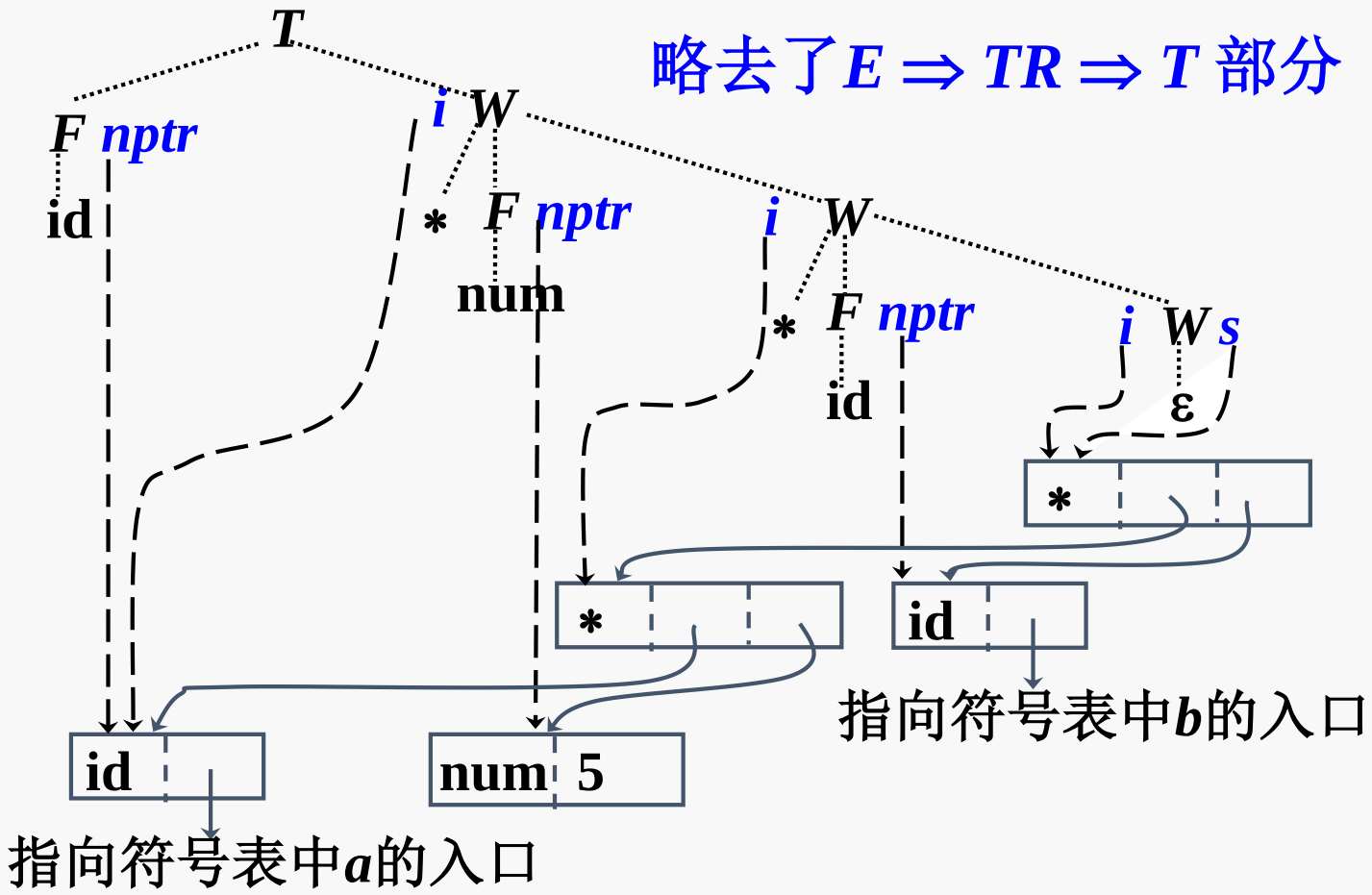
$E \rightarrow TR$
$R \rightarrow +TR_1$
$R \rightarrow \varepsilon$
$T \rightarrow FW$
$W \rightarrow *FW_1$
$W \rightarrow \varepsilon$

F产生式部分不再给出

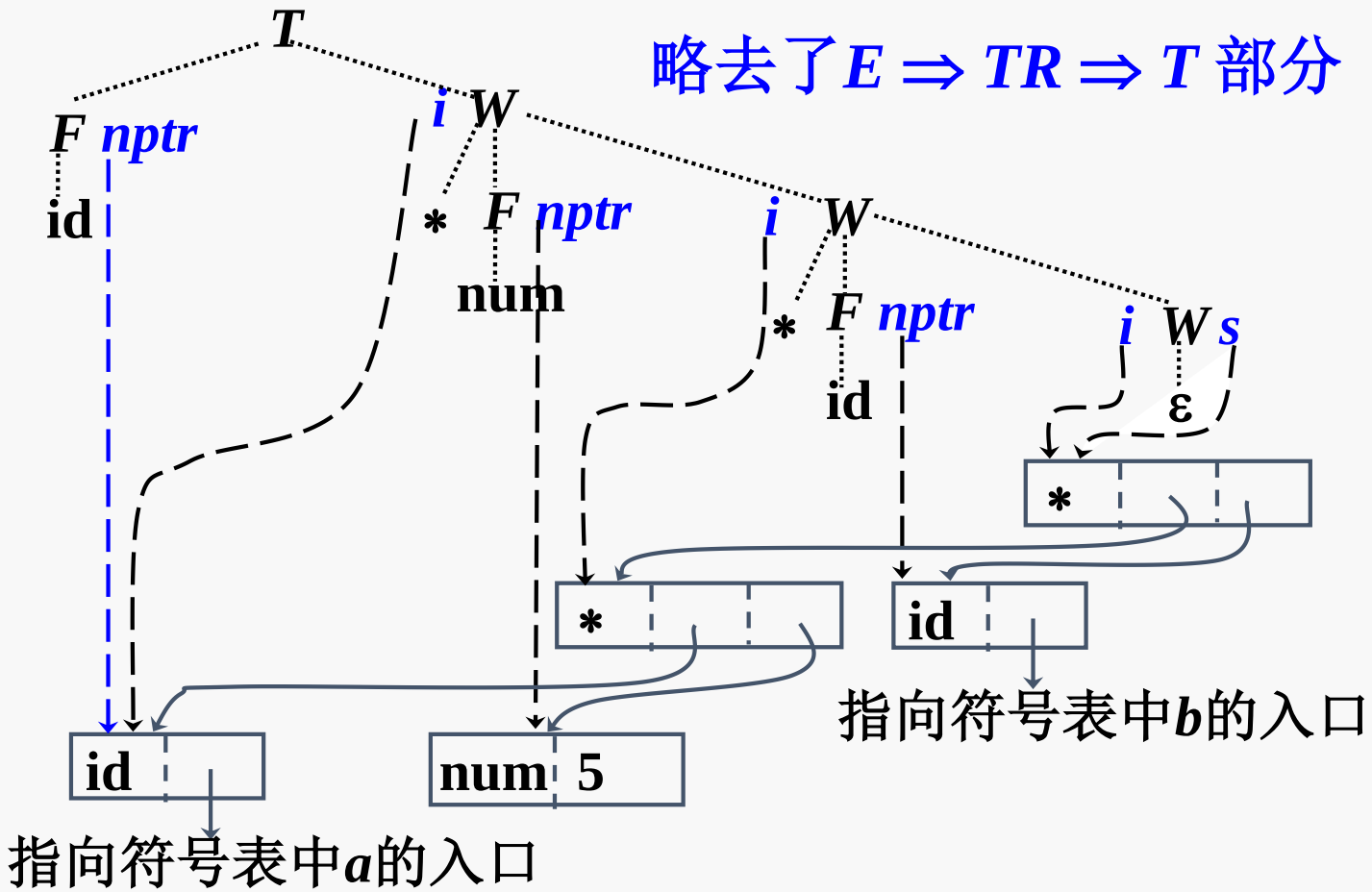
L属性定义的语法树构造

产　生　式	语　义　规　则
$T \rightarrow FW$	$W.i = F.nptr$ $T.nptr = W.s$
$W \rightarrow *FW_1$	$W_1.i = mkNode$ (‘*’, $W.i$, $F.nptr$) $W.s = W_1.s$
$W \rightarrow \varepsilon$	$W.s = W.i$
$F \rightarrow$ id	$F.nptr = mkLeaf$ (id, id.*entry*)
$F \rightarrow$ num	$F.nptr = mkLeaf$ (num, num.*val*)

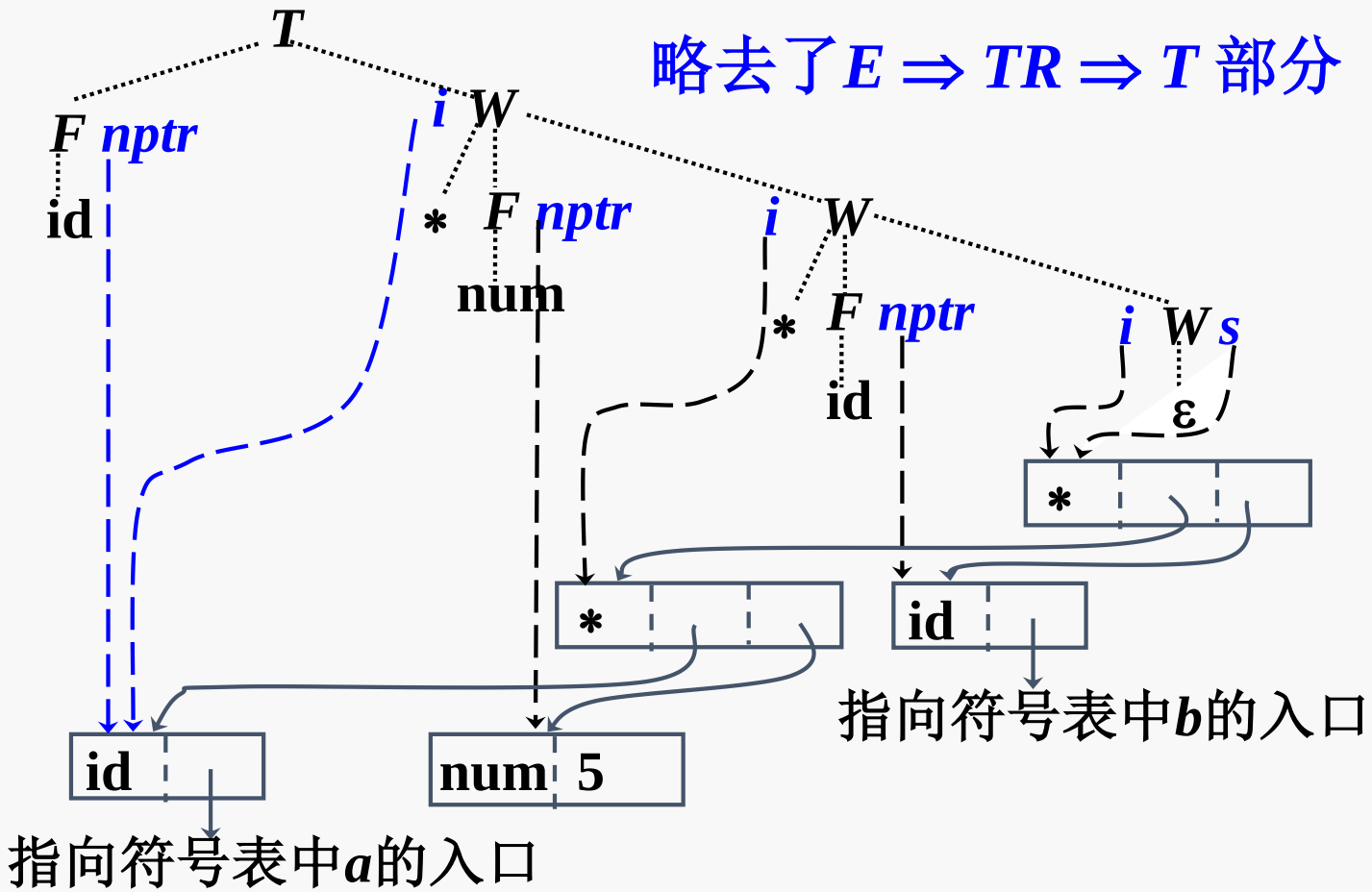
L属性定义的语法树构造



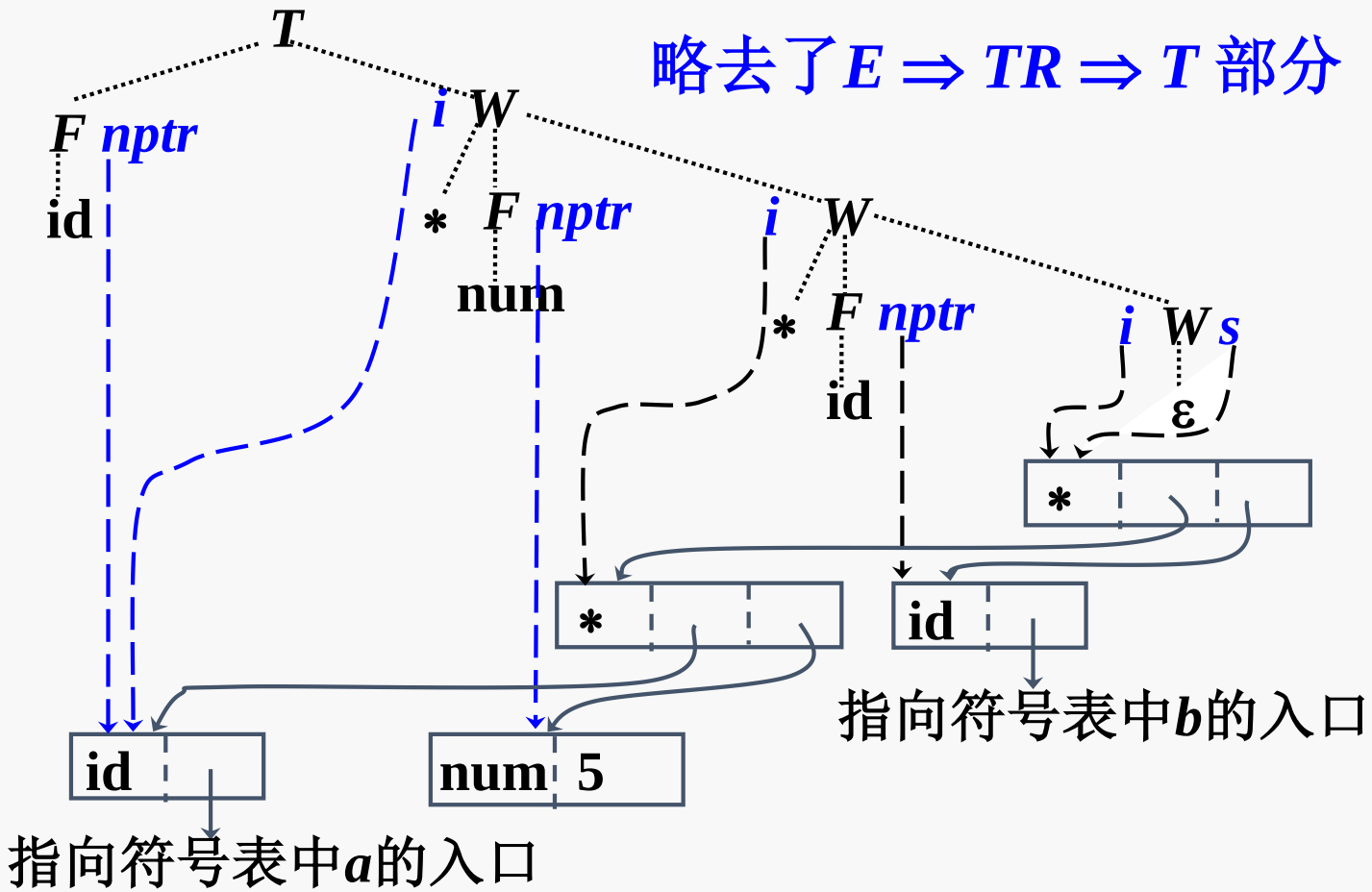
L属性定义的语法树构造



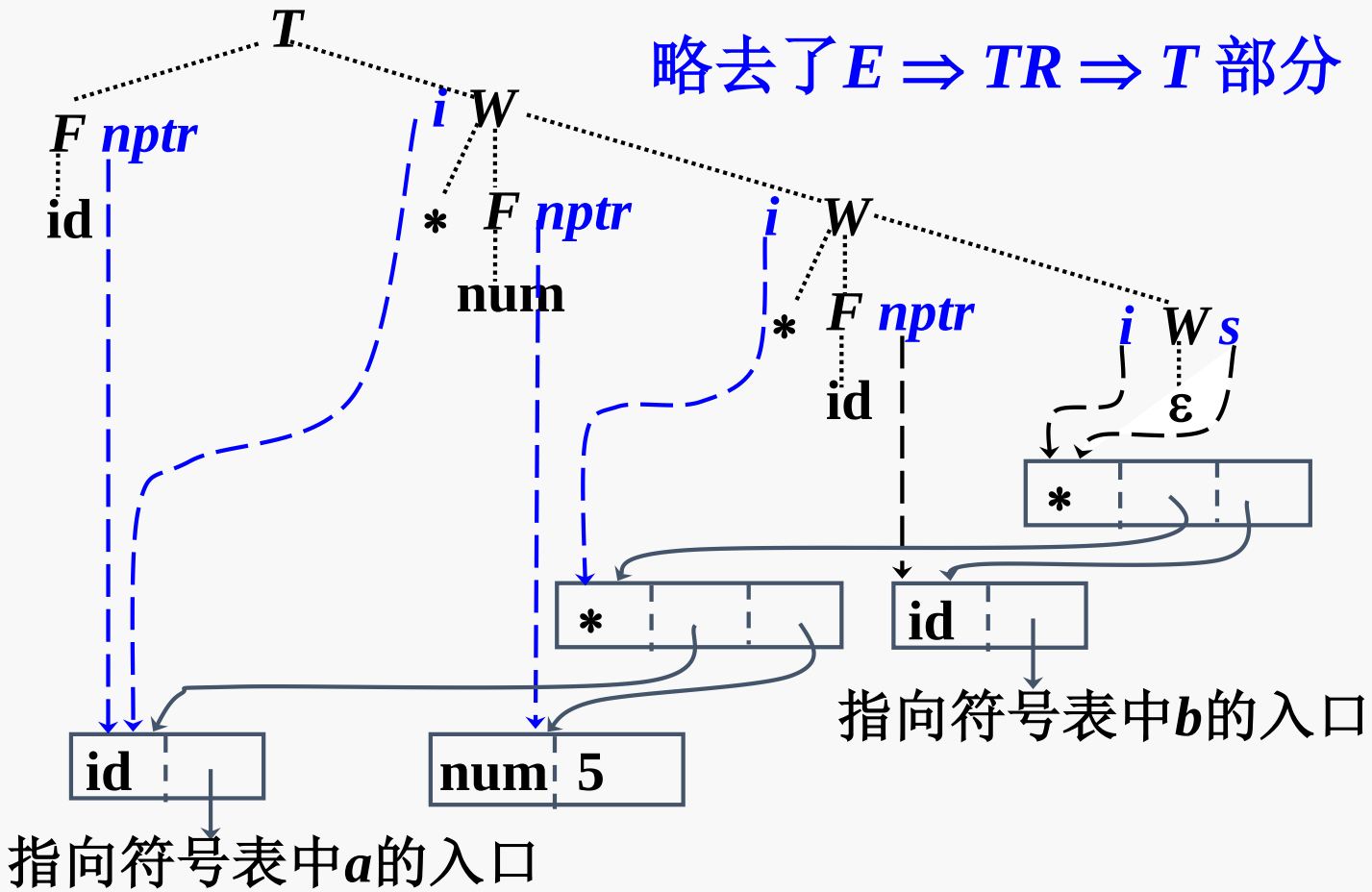
L属性定义的语法树构造



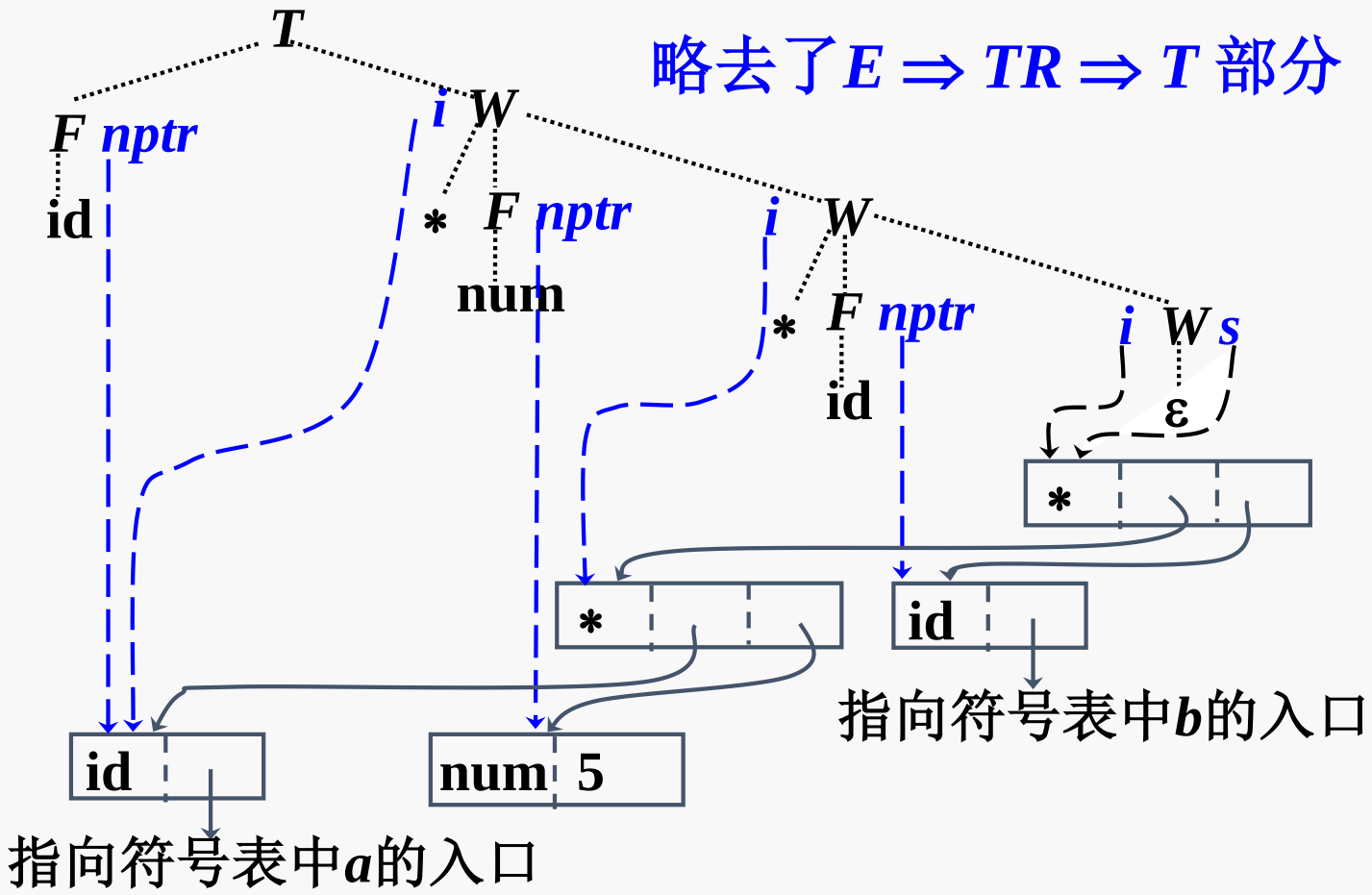
L属性定义的语法树构造



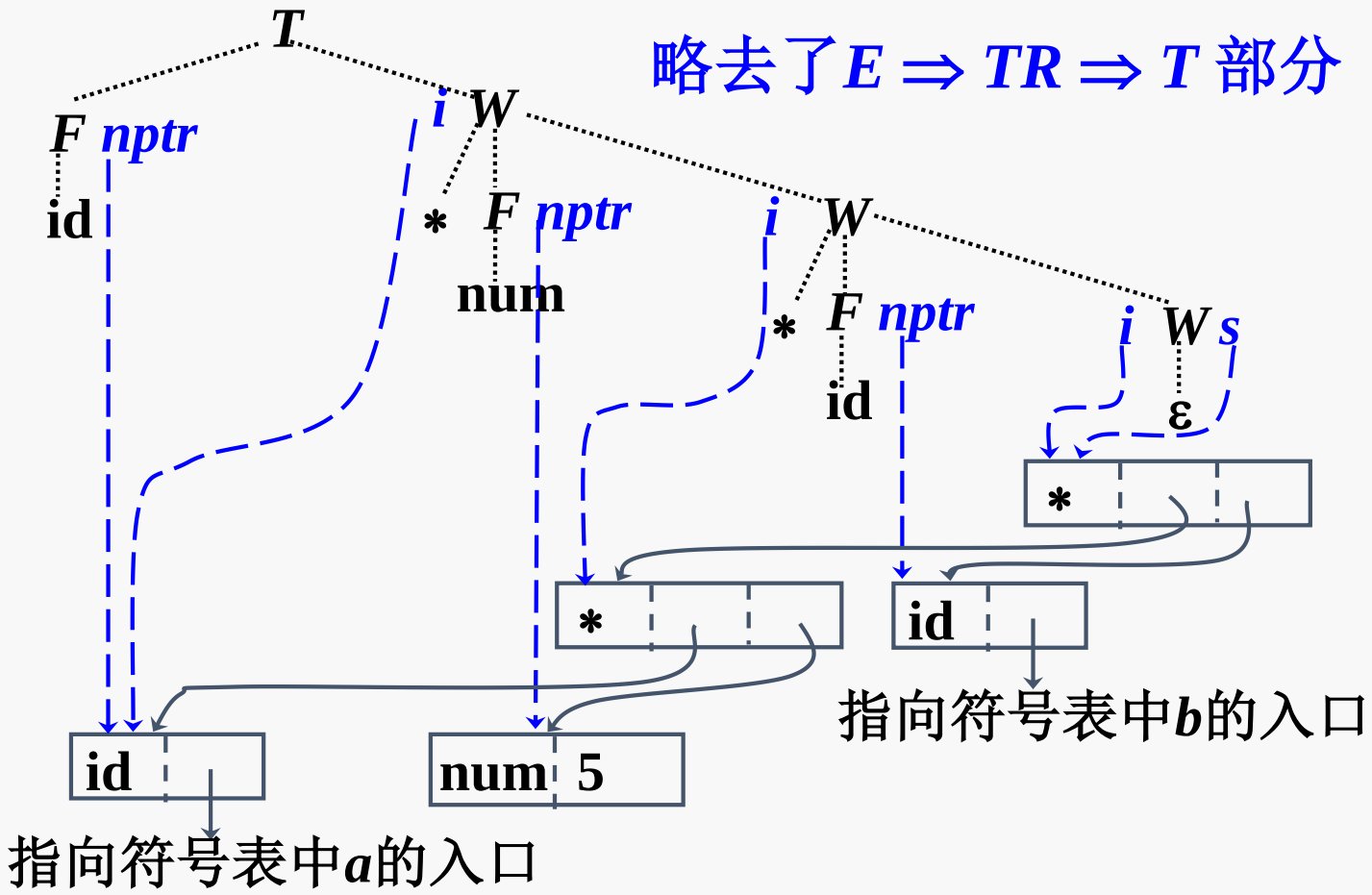
L属性定义的语法树构造



L属性定义的语法树构造



L属性定义的语法树构造



S-SDD和L-SDD对比

- **最后抽象语法树都是一样的**
- **但是，分析树却是不同的**
  - S-SDD的分析树与抽象语法树比较接近
  - L-SDD的分析树与抽象语法树结构不同

《编译原理和技术》

语法制导翻译 II

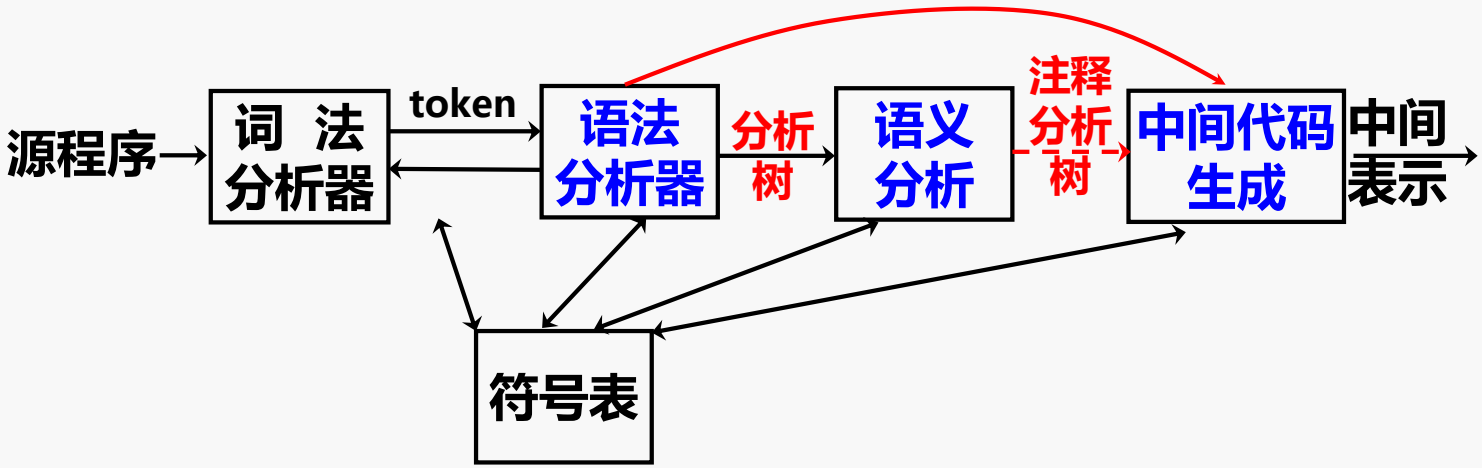
谢谢！

《编译原理和技术》

语法制导翻译 III

中科大计算机学院
李 诚
2022-10-10

本节提纲



- 从语法制导定义到翻译方案
- S属性定义的SDT
  - 实现方式1：先建树，后计算
  - 实现方式2：边分析，边计算

语法制导翻译方案

- **语法制导翻译方案(SDT)是在产生式右部中嵌入了程序片段(称为语义动作)的CFG**
- **SDT可以看作是SDD的具体实施方案**
  - 通过建立语法分析树的方案
  - 在语法分析过程中，边分析边计算的方案
    - 与LR或者LL分析方法结合

将S-SDD转换为SDT

- **将一个S-SDD转换为SDT的方法：**
  - 将每个语义动作都放在产生式的最后
  - 称为“后缀翻译方案”

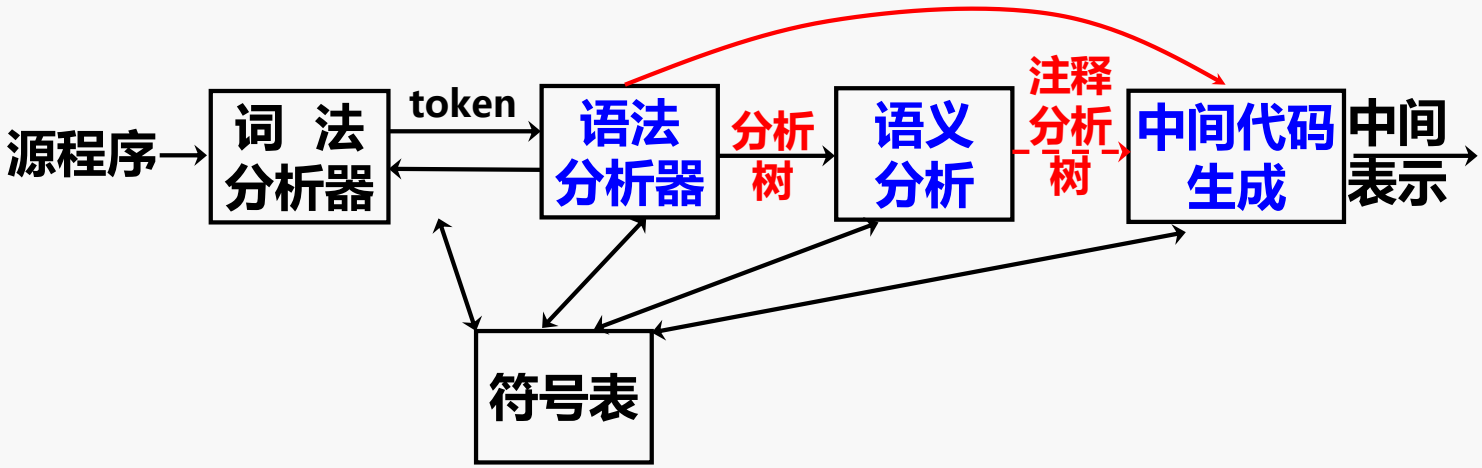
S-SDD

产生式	语义规则
(1) $L \rightarrow E$ n	$L.val = E.val$
(2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
(3) $E \rightarrow T$	$E.val = T.val$
(4) $T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
(5) $T \rightarrow F$	$T.val = F.val$
(6) $F \rightarrow (E)$	$F.val = E.val$
(7) $F \rightarrow$ digit	$F.val =$ digit.$lexval$

SDT

(1) $L \rightarrow E$ n { $L.val = E.val$}
(2) $E \rightarrow E_1 + T$ {$E.val = E_1.val + T.val$}
(3) $E \rightarrow T$ { $E.val = T.val$}
(4) $T \rightarrow T_1 * F$ { $T.val = T_1.val \times F.val$}
(5) $T \rightarrow F$ { $T.val = F.val$}
(6) $F \rightarrow (E)$ { $F.val = E.val$}
(7) $F \rightarrow$ digit { $F.val =$ digit.$lexval$}

本节提纲



- 从语法制导定义到翻译方案
- S属性定义的SDT
  - 实现方式1：先建树，后计算
  - 实现方式2：边分析，边计算

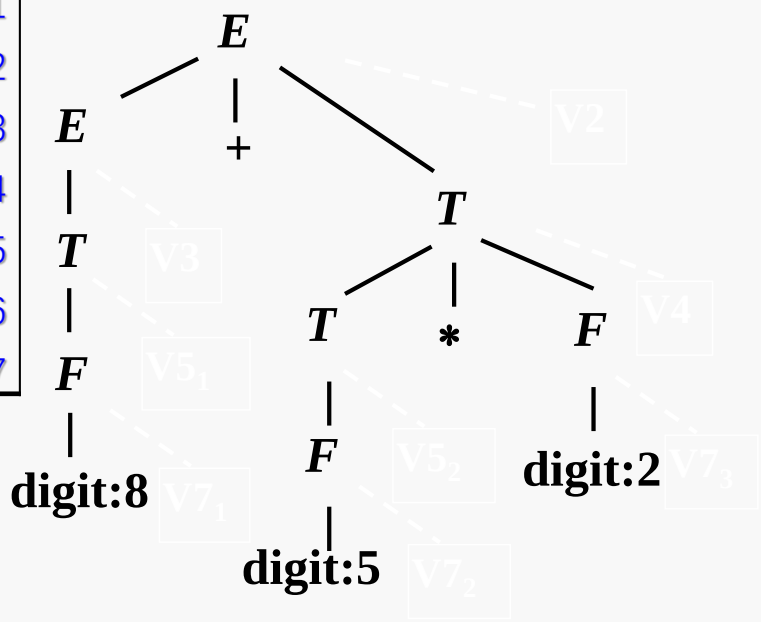
S-属性定义的SDT实现-1

- **基于分析树的语法制导翻译方案**
  - 建立语法分析树
  - 将语义动作看作是虚拟结点
  - 从左到右、深度优先地遍历分析树，在访问虚拟结点时执行相应的动作

S-属性定义的SDT实现-1

- 基于分析树的语法制导翻译方案

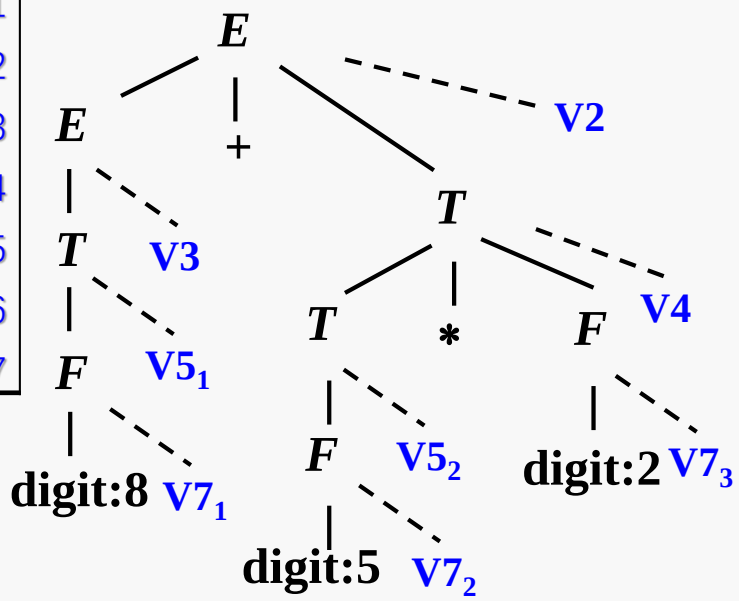
$L \rightarrow E$ n	{*print* (*E.val*) }	V1
$E \rightarrow E_1 + T$	{*E.val* = E_1 .*val* + *T.val* }	V2
$E \rightarrow T$	{*E.val* = *T.val* }	V3
$T \rightarrow T_1 * F$	{*T.val* = T_1.*val* * *F.val* }	V4
$T \rightarrow F$	{*T.val* = *F.val* }	V5
$F \rightarrow (E)$	{*F.val* = *E.val* }	V6
$F \rightarrow$ digit	{*F.val* = digit.*lexval*}	V7



S-属性定义的SDT实现-1

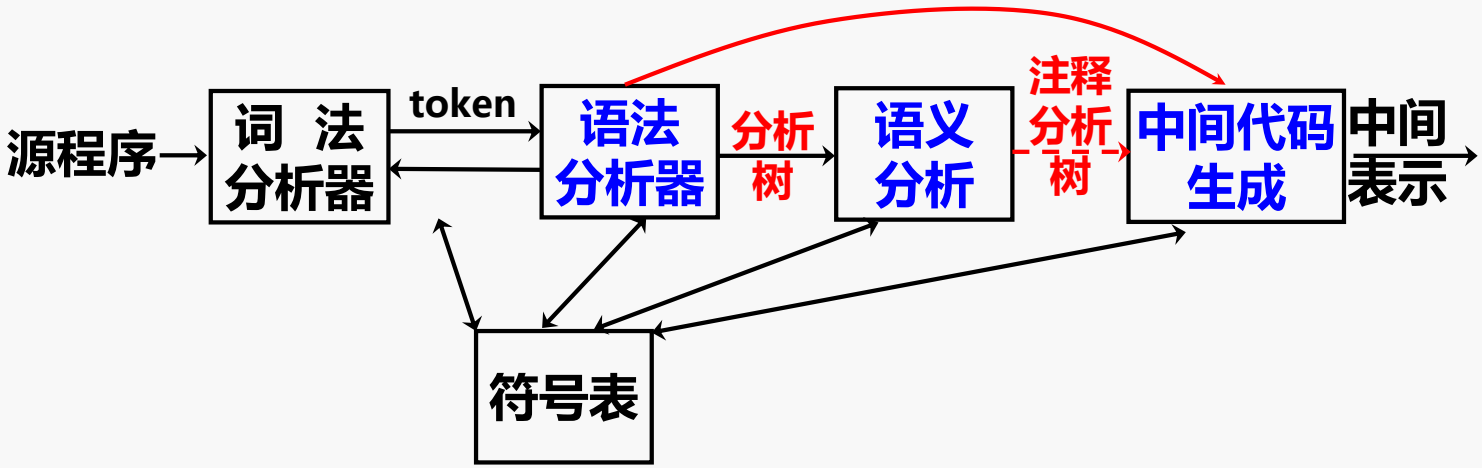
基于分析树的语法制导翻译方案

$L \rightarrow E$ n	{*print* (*E.val*) }	V1
$E \rightarrow E_1 + T$	{*E.val* = E_1 .*val* + *T.val* }	V2
$E \rightarrow T$	{*E.val* = *T.val* }	V3
$T \rightarrow T_1 * F$	{*T.val* = T_1.*val* * *F.val* }	V4
$T \rightarrow F$	{*T.val* = *F.val* }	V5
$F \rightarrow (E)$	{*F.val* = *E.val* }	V6
$F \rightarrow$ digit	{*F.val* = digit.*lexval*}	V7



- 语句8+5*2的分析树如右
- 深度优先可知动作执行顺序
  - $V7_1$, $V5_1$,V3,$V7_2$,$V5_2$,$V7_3$,V4,V2

本节提纲



- 从语法制导定义到翻译方案
- S属性定义的SDT
  - 实现方式1：先建树，后计算
  - 实现方式2：边分析，边计算

S-属性定义的SDT实现-2

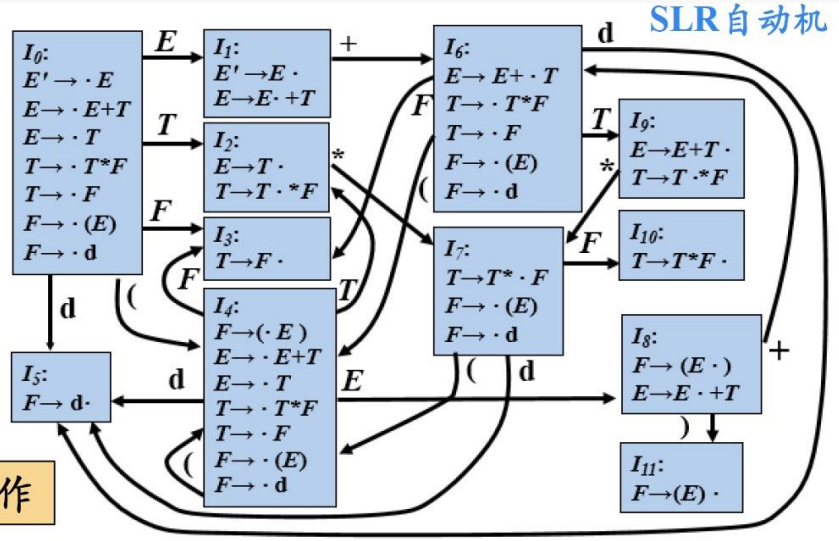
- 综合属性可通过自底向上的LR方法来计算
- 当归约发生时执行相应的语义动作

➢例

S-SDD

产生式	语义规则
(1) $L \to E$ n	$L.val = E.val$
(2) $E \to E_1 + T$	$E.val = E_1.val + T.val$
(3) $E \to T$	$E.val = T.val$
(4) $T \to T_1 * F$	$T.val = T_1.val \times F.val$
(5) $T \to F$	$T.val = F.val$
(6) $F \to (E)$	$F.val = E.val$
(7) $F \to$ digit	$F.val =$ digit.*lexval*

当归约发生时执行相应的语义动作

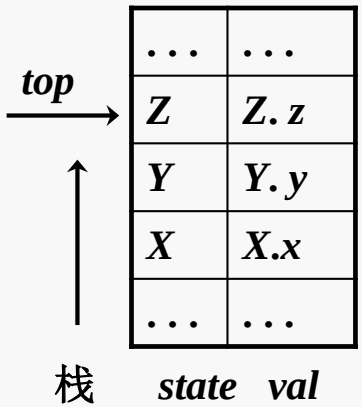


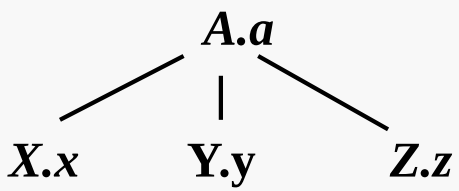
S-属性定义的SDT实现-2

- 可以通过扩展的LR语法分析栈来实现
  - 在分析栈中使用一个附加的域来存放综合属性值。若支持多个属性，那么可以在栈中存放指针
  - 每一个栈元素包含状态、文法符号、综合属性三个域
    - 也可以将分析栈看成三个平行的栈，分别是状态栈、文法符号栈、综合属性栈，分开看的理由是，入栈出栈并不完全同步
  - 语义动作将修改为对栈中文法符号属性的计算

S-属性定义的SDT实现-2

- 可以通过扩展的LR语法分析栈来实现
  - 考虑产生式 $A \rightarrow XYZ$

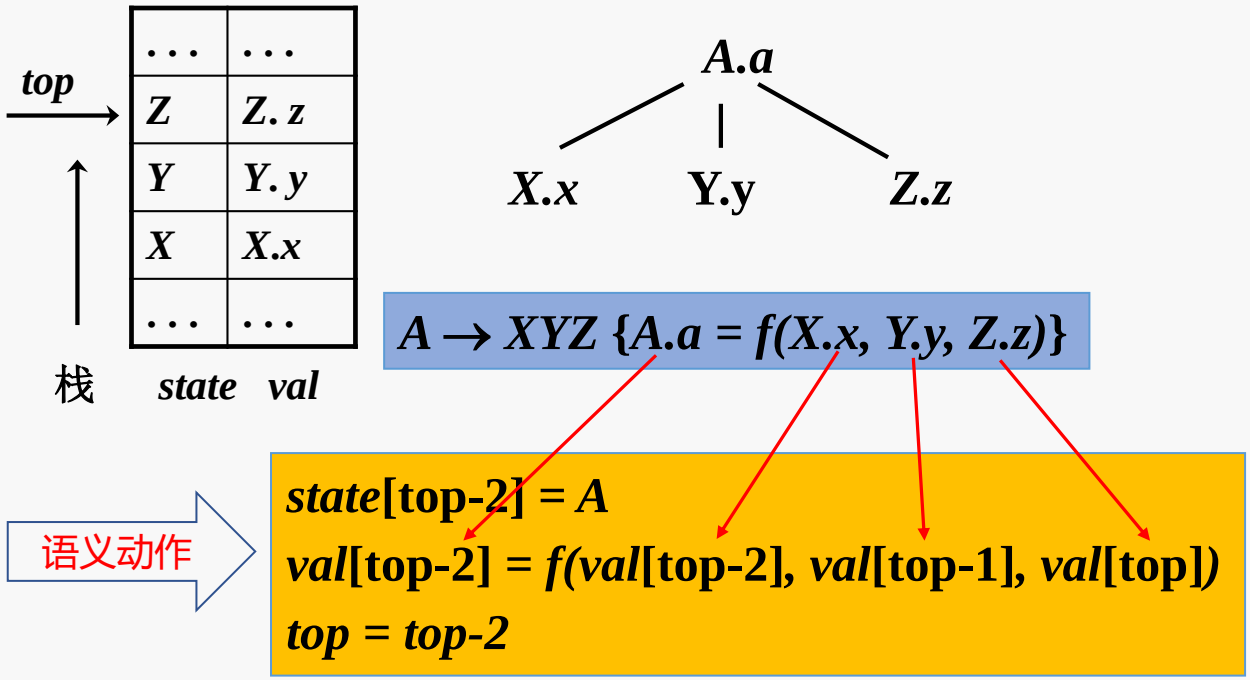




$A \rightarrow XYZ \{A.a = f(X.x, Y.y, Z.z)\}$

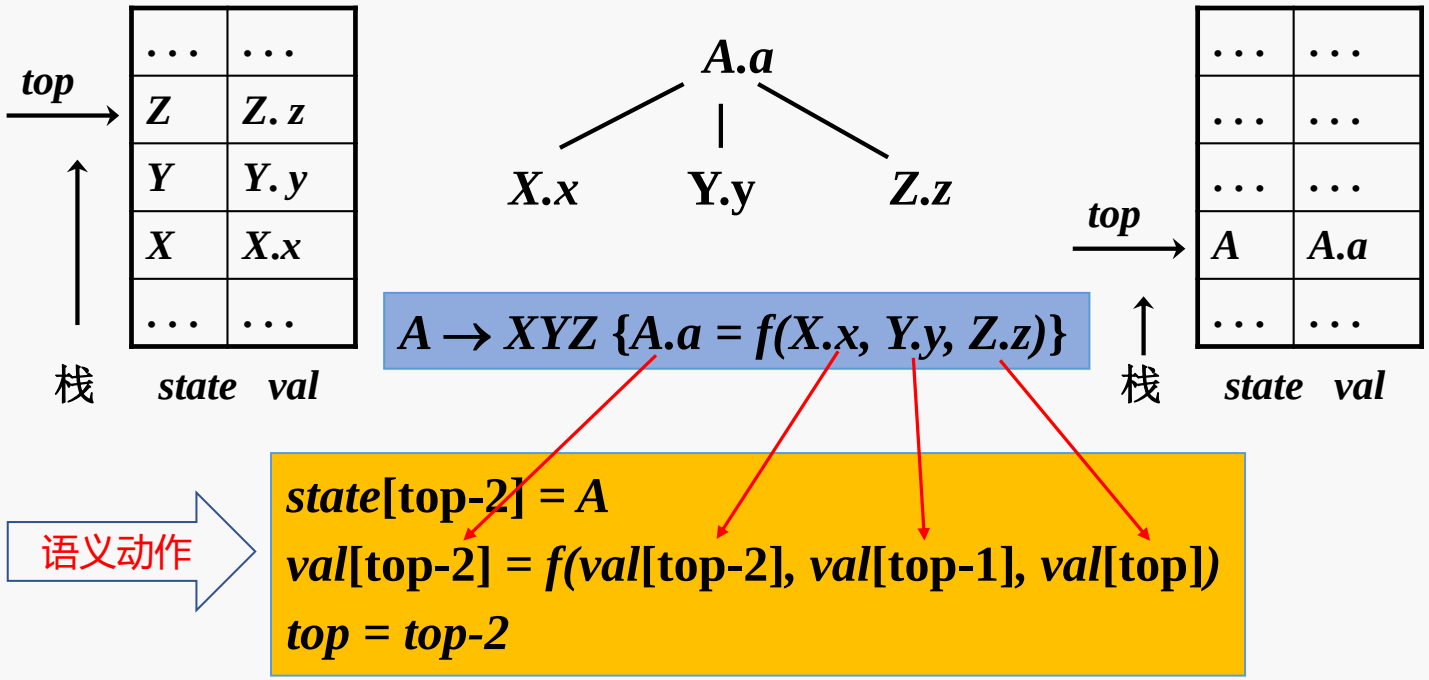
S-属性定义的SDT实现-2

- 可以通过扩展的LR语法分析栈来实现
  - 考虑产生式 $A \rightarrow XYZ$



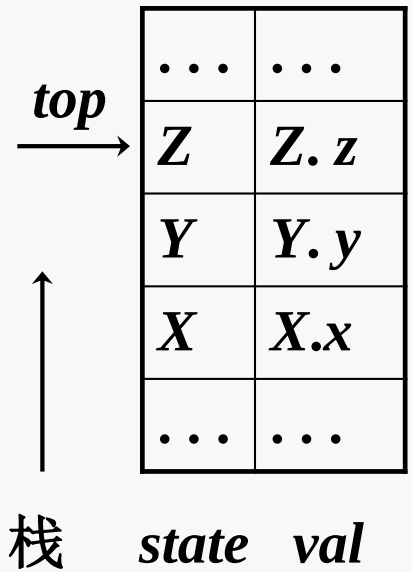
S-属性定义的SDT实现-2

- 可以通过扩展的LR语法分析栈来实现
  - 考虑产生式 $A \rightarrow XYZ$



SLR分析栈中实现计算器

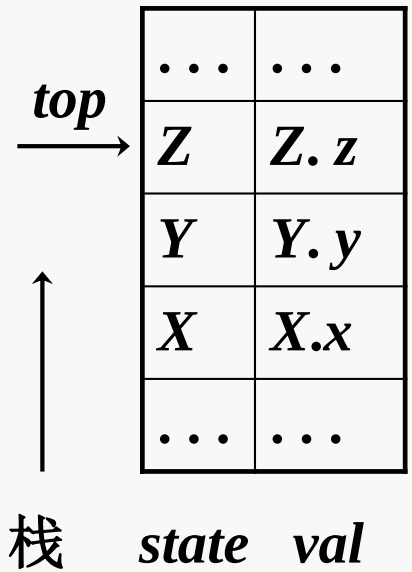
- 简单计算器的语法制导定义改成栈操作代码



产 生 式	语 义 规 则
$L \rightarrow E$ n	*print* (*E.val*)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	*E.val* = *T.val*
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	*T.val* = *F.val*
$F \rightarrow (E)$	*F.val* = *E.val*
$F \rightarrow$ digit	*F.val* = digit.*lexval*

SLR分析栈中实现计算器

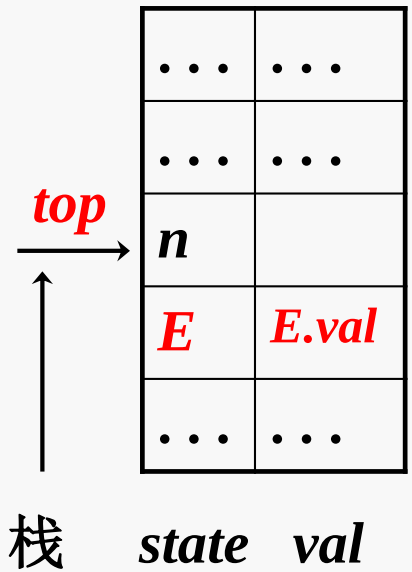
- 简单计算器的语法制导定义改成栈操作代码



产 生 式	代 码 段
$L \rightarrow E$ n	*print* (*E.val*)
$E \rightarrow E_1 + T$	*E.val* = E_1*.val* + *T.val*
$E \rightarrow T$	*E.val* = *T.val*
$T \rightarrow T_1 * F$	*T.val* = T_1*.val* * *F.val*
$T \rightarrow F$	*T.val* = *F.val*
$F \rightarrow (E)$	*F.val* = *E.val*
$F \rightarrow$ digit	*F.val* = digit.*lexval*

SLR分析栈中实现计算器

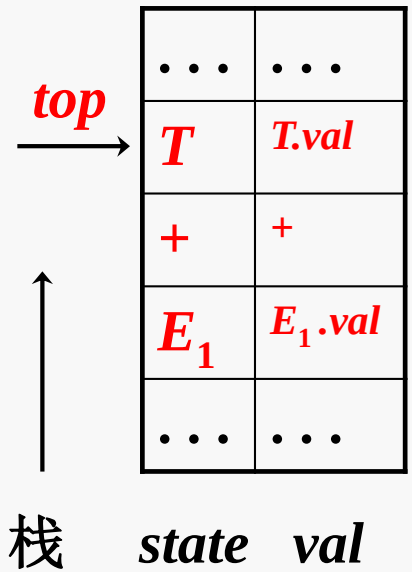
- 简单计算器的语法制导定义改成栈操作代码



产 生 式	代 码 段
$L \rightarrow E$ n	*print* (*val* [*top*–1])
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow$ digit	$F.val$ = digit.*lexval*

SLR分析栈中实现计算器

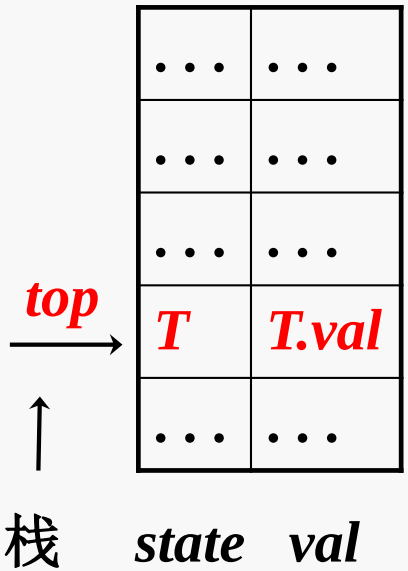
- 简单计算器的语法制导定义改成栈操作代码



产 生 式	代 码 段
$L \rightarrow E$ n	*print* (*val* [*top*–1])
$E \rightarrow E_1 + T$	*val* [*top* –2] = *val* [*top* –2]+*val* [*top*]
$E \rightarrow T$	*E.val* = *T.val*
$T \rightarrow T_1 * F$	*T.val* = T_1*.val* * *F.val*
$T \rightarrow F$	*T.val* = *F.val*
$F \rightarrow (E)$	*F.val* = *E.val*
$F \rightarrow$ digit	*F.val* = digit.*lexval*

SLR分析栈中实现计算器

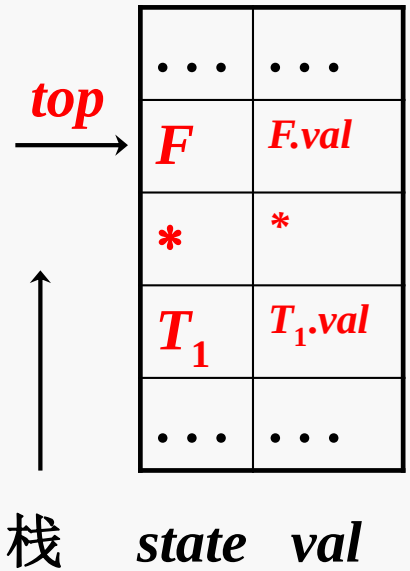
- 简单计算器的语法制导定义改成栈操作代码



产 生 式	代 码 段
$L \rightarrow E$ n	*print* (*val* [*top*–1])
$E \rightarrow E_1 + T$	*val* [*top* –2] = *val* [*top* –2]+*val* [*top*]
$E \rightarrow T$	值不变，无动作
$T \rightarrow T_1 * F$	*T.val* = T_1.*val* * *F.val*
$T \rightarrow F$	*T.val* = *F.val*
$F \rightarrow (E)$	*F.val* = *E.val*
$F \rightarrow$ digit	*F.val* = digit.*lexval*

SLR分析栈中实现计算器

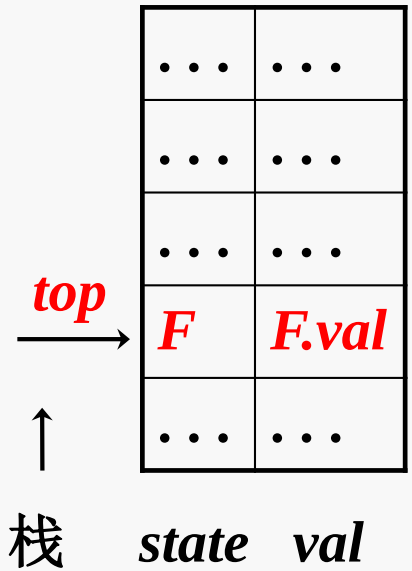
- 简单计算器的语法制导定义改成栈操作代码



产 生 式	代 码 段
$L \rightarrow E$ n	*print* (*val* [*top*–1])
$E \rightarrow E_1 + T$	*val* [*top* –2] = *val* [*top* –2]+*val* [*top*]
$E \rightarrow T$	
$T \rightarrow T_1 * F$	*val* [*top* –2] = *val* [*top* –2]×*val* [*top*]
$T \rightarrow F$	*T.val* = *F.val*
$F \rightarrow (E)$	*F.val* = *E.val*
$F \rightarrow$ digit	*F.val* = digit.*lexval*

SLR分析栈中实现计算器

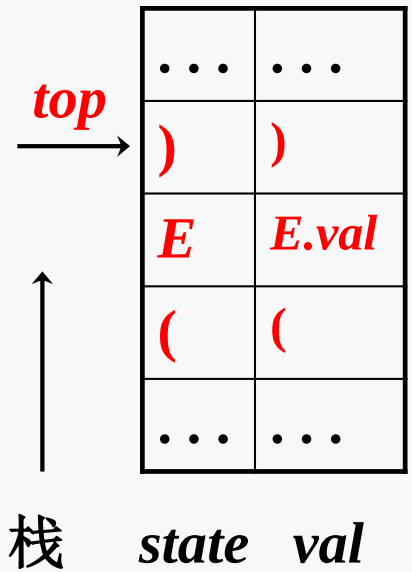
- 简单计算器的语法制导定义改成栈操作代码



产 生 式	代 码 段
$L \to E$ n	*print* (*val* [*top*–1])
$E \to E_1 + T$	*val* [*top* –2] = *val* [*top* –2]+*val* [*top*]
$E \to T$	值不变，无动作
$T \to T_1 * F$	*val* [*top* –2] = *val* [*top* –2]×*val* [*top*]
$T \to F$	值不变，无动作
$F \to (E)$	*F.val* = *E.val*
$F \to$ digit	*F.val* = digit.*lexval*

SLR分析栈中实现计算器

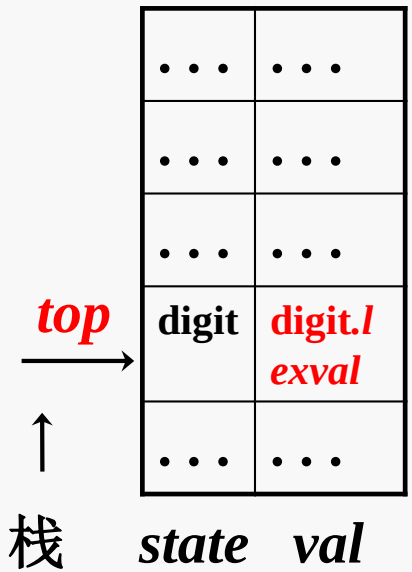
- 简单计算器的语法制导定义改成栈操作代码



产 生 式	代 码 段
$L \rightarrow E$ n	*print* (*val* [*top*–1])
$E \rightarrow E_1 + T$	*val* [*top* –2] = *val* [*top* –2]+*val* [*top*]
$E \rightarrow T$	值不变，无动作
$T \rightarrow T_1 * F$	*val* [*top* –2] = *val* [*top* –2]×*val* [*top*]
$T \rightarrow F$	值不变，无动作
$F \rightarrow (E)$	*val* [top –2] =*val* [top –1]
$F \rightarrow$ digit	*F.val* = digit.*lexval*

SLR分析栈中实现计算器

- 简单计算器的语法制导定义改成栈操作代码



产 生 式	代 码 段
$L \rightarrow E$ n	*print* (*val* [*top*–1])
$E \rightarrow E_1 + T$	*val* [*top* –2] = *val* [*top* –2]+*val* [*top*]
$E \rightarrow T$	值不变，无动作
$T \rightarrow T_1 * F$	*val* [*top* –2] = *val* [*top* –2]×*val* [*top*]
$T \rightarrow F$	值不变，无动作
$F \rightarrow (E)$	*val* [top –2] =*val* [top –1]
$F \rightarrow$ digit	值不变，无动作

总结

- 采用自底向上分析，例如LR分析，首先给出S-属性定义，然后，把S-属性定义变成可执行的代码段，放到产生式尾部，这就构成了翻译程序。
- 随着语法分析的进行，归约前调用相应的语义子程序，完成翻译的任务。

《编译原理和技术》

语法制导翻译 III

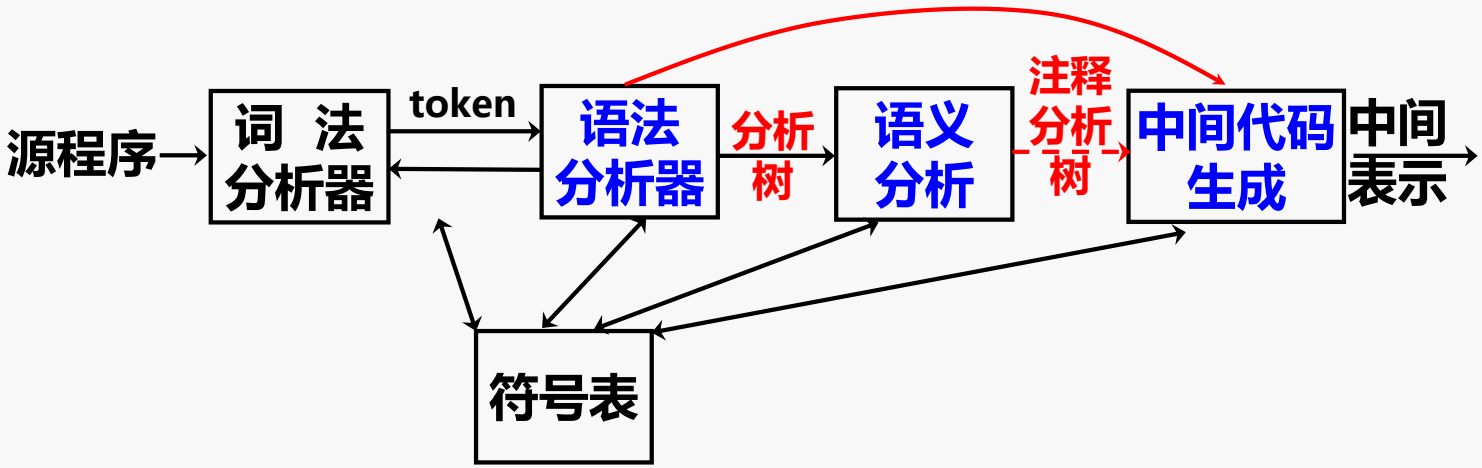
谢谢！

《编译原理和技术》

语法制导翻译 IV

中科大计算机学院
李 诚
2022-10-10

本节提纲



- L属性定义的SDT
- 实现方式1：与递归下降分析结合
- 实现方式2：与LR分析结合

*L*属性定义的SDT

- 如果每个产生式$A \to X_1 \ldots X_{j-1} X_j \ldots X_n$的每条语义规则计算的属性是$A$的综合属性；或者是$X_j$的继承属性，但它仅依赖：
  - 该产生式中X_j左边符号$X_1, X_2, \ldots, X_{j-1}$的属性；
  - A的继承属性
  - X_j的其他属性，且不能成环

- *S*属性定义属于*L*属性定义

*L*属性定义的SDT

- 变量类型声明的语法制导定义是一个*L-SDD*

产　生　式	语 义 规 则
$D \rightarrow TL$	*L.in* = *T.type*
$T \rightarrow$ int	*T. type* = *integer*
$T \rightarrow$ real	*T. type* = *real*
$L \rightarrow L_1$, id	L_1.*in* = *L.in*; *addType*(id.*entry*, *L.in*)
$L \rightarrow$ id	*addType*(id.*entry*, *L.in*)

- 后缀SDT在这里并不适用

L属性定义的SDT

- 消除左递归的算术表达式语法制导定义是L-SDD

产　生　式	语 义 规 则
$T \rightarrow FT'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \rightarrow *FT_1'$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
$F \rightarrow$ digit	$F.val =$ digit.$lexval$

- 后缀SDT在这里并不适用

将L-SDD转换为SDT

将L-SDD转换为SDT的规则

- 将计算一个产生式左部符号的综合属性的动作放置在这个产生式右部的最右端
- 将计算某个非终结符号A的继承属性的动作插入到产生式右部中紧靠在A的本次出现之前的位置上
  - 多个继承属性，要考虑次序，防止形成环

将L-SDD转换为SDT-举例

L-SDD

产 生 式	语 义 规 则
$D \rightarrow TL$	$L.in = T.type$
$T \rightarrow$ int	$T.type = integer$
$T \rightarrow$ real	$T.type = real$
$L \rightarrow L_1$, id	$L_1.in = L.in$; $addType(\text{id}.entry, L.in)$
$L \rightarrow$ id	$addType(\text{id}.entry, L.in)$

将L-SDD转换为SDT-举例

L-SDD

产 生 式	语 义 规 则
$D \rightarrow TL$	$L.in = T.type$
$T \rightarrow$ int	$T.type = integer$
$T \rightarrow$ real	$T.type = real$
$L \rightarrow L_1$, id	$L_1.in = L.in$; $addType(\text{id}.entry, L.in)$
$L \rightarrow$ id	$addType(\text{id}.entry, L.in)$

L-SDT

$D \rightarrow T \quad \{L.in = T.type\}\ L$

$T \rightarrow$ int $\quad \{T.type = integer\}$

$T \rightarrow$ real $\{T.type = real\}$

$L \rightarrow \{L_1.in = L.in\}\ L_1$, id $\{addtype(\text{id}.entry, L.in)\}$

$L \rightarrow$ id $\quad \{addtype(\text{id}.entry, L.in)\}$

将L-SDD转换为SDT-举例

L-SDD

产 生 式	语 义 规 则
$T \rightarrow FT'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \rightarrow *FT_1'$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
$F \rightarrow$ digit	$F.val =$ digit.*lexval*

将L-SDD转换为SDT-举例

L-SDD

产 生 式	语 义 规 则
$T \rightarrow FT'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \rightarrow *FT_1'$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
$T' \rightarrow \varepsilon$	$T'.syn = T'.inh$
$F \rightarrow$ digit	$F.val =$ digit.$lexval$

L-SDT

$T \rightarrow F\ \{T'.inh = F.val\}\ T'\ \{T.val = T'.syn\}$
$T' \rightarrow *F\ \{T_1'.inh = T'.inh \times F.val\}\ T_1'\ \{T'.syn = T_1'.syn\}$
$T' \rightarrow \varepsilon\ \{T'.syn = T'.inh\}$
$F \rightarrow$ digit $\{F.val =$ digit.$lexval\}$

L属性定义的SDT-举例

- 例 把有加和减的中缀表达式翻译成后缀表达式输出
  如果输入是8+5 –2，则输出是8 5 + 2 –，设计SDT

$E \rightarrow T\ R$

$R \rightarrow \text{addop}\ T\ R_1 \mid \varepsilon$

$T \rightarrow \text{num}$

L属性定义的SDT-举例

- 例 把有加和减的中缀表达式翻译成后缀表达式输出
  如果输入是8+5 –2，则输出是8 5 + 2 –，设计SDT

$E \rightarrow T\ R$

$R \rightarrow$ addop T {*print* (addop.*lexeme*)} $R_1 \mid \varepsilon$

$T \rightarrow$ num {*print* (num.*val*)}

$E \Rightarrow T\ R \Rightarrow$ num {*print* (8)} R

$\Rightarrow$ num{*print* (8)}addop T{*print* (+)}R

$\Rightarrow$ num{*print*(8)}addop num{*print*(5)}{*print* (+)}R

… {*print*(8)}{*print*(5)}{*print*(+)}addop T{*print*(–)}R

… {*print*(8)}{*print*(5)}{*print*(+)}{*print*(2)}{*print*(–)}

L属性定义的SDT-举例

- 例 把有加和减的中缀表达式翻译成后缀表达式输出
  如果输入是8+5 –2，则输出是8 5 + 2 –，设计SDT

$E \rightarrow T\ R$

$R \rightarrow$ addop $T\ R_1$ {*print* (addop.*lexeme*)} | ε

$T \rightarrow$ num {*print* (num.*val*)}

语义动作不能随意放置

L属性定义的SDT-举例

- 例 把有加和减的中缀表达式翻译成后缀表达式输出
  如果输入是8+5 –2，则输出是8 5 + 2 –，设计SDT

$E \rightarrow T\ R$

$R \rightarrow$ addop $T\ R_1$ {*print* (addop.*lexeme*)} | ε

$T \rightarrow$ num {*print* (num.*val*)}

语义动作不能随意放置

$E \Rightarrow T\ R \Rightarrow$ num {*print* (8)} R

$\Rightarrow$ num{*print* (8)}addop TR

$\Rightarrow$ num{*print*(8)}addop num{*print*(5)} R

… {*print*(8)}{*print*(5)} addop TR

… {*print*(8)}{*print*(5)}{*print*(2)}{*print*(–)}{*print*(+)}

L属性定义的SDT-举例

- 例 数学排版语言EQN，设计SDT，计算高度

E sub 1 .val

$S \rightarrow B$

$B \rightarrow B_1\ B_2$

$B \rightarrow B_1$ sub B_2

$B \rightarrow$ text

文字排版中的符号属性

S： *S.ht*，综合属性；待排公式的整体高度

B： *B.ps*，继承属性； 公式（文本）中字体的大小
B.ht，综合属性；公式排版高度

text： *text.h*，文本高度

文字排版中的符号属性

S： *S.ht*，综合属性；待排公式的整体高度

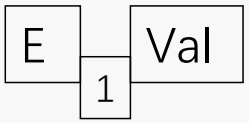
B： *B.ps*，继承属性； 公式（文本）中字体的大小
B.ht，综合属性；公式排版高度

text： *text.h*，文本高度

max(B_1, B_2)： 求两个排版公式的最大高度

shrink(*B*)： 将字体大小缩小为*B*的*30*％

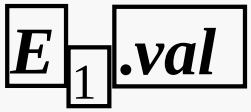
disp(B_1.*ht*，B_2.*ht*)：向下调整B_2的位置



L属性定义的SDT-举例

- 例 数学排版语言EQN（语法制导定义L-SDD）

E sub 1 .val



ps-point size (继承属性); ht-height(综合属性)

产 生 式	语 义 规 则
$S \rightarrow B$	$B.ps = 10;\ S.ht = B.ht$
$B \rightarrow B_1\ B_2$	$B_1.ps = B.ps;\ B_2.ps = B.ps;$ $B.ht = max(B_1.ht, B_2.ht)$
$B \rightarrow B_1$ sub B_2	$B_1.ps = B.ps;\ B_2.ps = shrink(B.ps);$ $B.ht = disp\ (B_1.ht, B_2.ht)$
$B \rightarrow$ text	$B.ht = \text{text}.h \times B.ps$

L属性定义的SDT-举例

- 例 数学排版语言EQN（翻译方案SDT）

$S \rightarrow \{B.ps = 10\}$
$B \quad \{S.ht = B.ht\}$

*B*继承属性的计算位于*B*的左边

L属性定义的SDT-举例

- 例 数学排版语言EQN（翻译方案SDT）

$S \rightarrow \{B.ps = 10\}$
$B \quad \{S.ht = B.ht\}$

*B*综合属性的计算放在右部末端

L属性定义的SDT-举例

- 例 数学排版语言EQN（翻译方案SDT）

$S \rightarrow$ $\{B.ps = 10\}$

$\quad B$ $\quad \{S.ht = B.ht\}$

$B \rightarrow$ $\quad \{B_1.ps = B.ps\}$

$\quad B_1$ $\quad \{B_2.ps = B.ps\}$

$\quad B_2$ $\quad \{B.ht = max(B_1.ht, B_2.ht)\}$

L属性定义的SDT-举例

- 例 数学排版语言EQN（翻译方案SDT）

$S \rightarrow$ $\{B.ps = 10\}$
B $\{S.ht = B.ht\}$

$B \rightarrow$ $\{B_1.ps = B.ps\}$
B_1 $\{B_2.ps = B.ps\}$
B_2 $\{B.ht = max(B_1.ht, B_2.ht)\}$

$B \rightarrow$ $\{B_1.ps = B.ps\}$
B_1
sub $\{B_2.ps = shrink(B.ps)\}$
B_2 $\{B.ht = disp(B_1.ht, B_2.ht)\}$

L属性定义的SDT-举例

- 例 数学排版语言EQN（翻译方案SDT）

$S \rightarrow$ { $B.ps = 10$ }
　　B { $S.ht = B.ht$ }

$B \rightarrow$ { $B_1.ps = B.ps$ }
　　B_1 { $B_2.ps = B.ps$ }
　　B_2 { $B.ht = max(B_1.ht, B_2.ht)$ }

$B \rightarrow$ { $B_1.ps = B.ps$ }
　　B_1
　　sub { $B_2.ps = shrink(B.ps)$ }
　　B_2 { $B.ht = disp\ (B_1.ht, B_2.ht)$ }

$B \rightarrow$ text { $B.ht = \text{text}.h \times B.ps$ }

L属性定义的SDT-举例

- **例 翻译while循环语句，生成代码**
  产生式 $S \rightarrow \text{while } (C)\ S_1$

L-SDD

$S \rightarrow \text{while } (C)\ S_1$	L1 = new(); L2 = new(); S_1.next = L1; C.false = S.next; C.true = L2; S.code = **label** || L1 || C.code || **label** || L2 || S_1.code

- *next:* 继承属性，语句结束后应跳转到的标号
- *true*、*false:* C为真/假时应该跳转到的标号
- *code:* 综合属性，表示代码

L属性定义的SDT-举例

- 根据语义动作的放置规则得到如下SDT：

L-SDD

$S \rightarrow$ while (C) S_1	L1 = new();	临时变量
	L2 = new();	
S_1的继承属性 →	S_1.next = L1;	
	C.false = S.next;	C的继承属性
	C.true = L2;	
S的综合属性 →	S.code = **label** || L1 || C.Ccode || **label** || L2 || S_1.code	

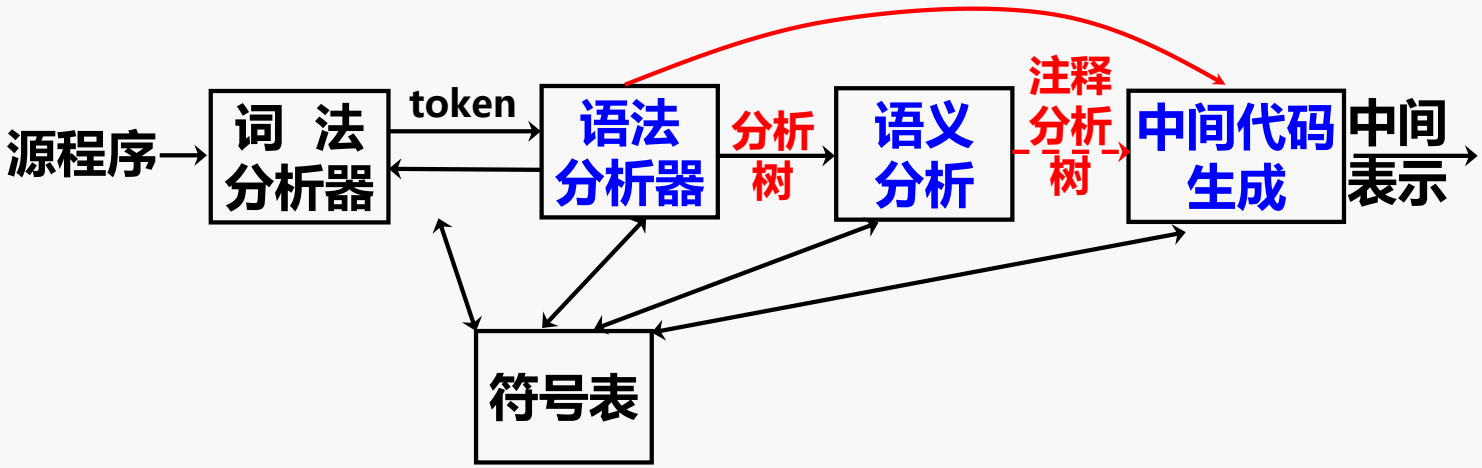
L-SDT

$S \rightarrow$ while (	{L1 = new(); L2 = new(); C.false = S.next; C.true = L2;}	
C)	{S_1.next = L1;}	
S_1	{S.code = **label** || L1 || C.Ccode || **label** || L2 || S_1.code}	

L属性定义的计算

- **结合SDT，考虑在语法分析过程中进行翻译**
  - 自顶向下计算
    - 递归下降分析器
  - 自底向上计算，考虑与LR分析器的结合
    - 删除翻译方案中嵌入的动作
    - 继承属性在分析栈中的计算

本节提纲



- L属性定义的SDT
- 实现方式1：与递归下降分析结合
- 实现方式2：与LR分析结合

L属性定义的自顶向下计算

- **递归下降翻译器的设计**
  - 为每个非终结符A构造一个函数
    - A的每个继承属性对应该函数的一个形参
    - 函数的返回值是A的综合属性值

L属性定义的自顶向下计算

递归下降翻译器的设计

- 为每个非终结符A构造一个函数
  - A的每个继承属性对应该函数的一个形参
  - 函数的返回值是A的综合属性值
- 在函数体中
  - 首先选择适当的A的产生式
  - 用局部变量保存产生式中文法符号的属性
  - 对产生式体中的终结符号，读入符号并获取其综合属性（由词法分析得到）
  - 对产生式体中的非终结符，调用相应函数，记录返回值

L属性定义计算-递归下降

$E \to T$ $\quad \{R.i = T.nptr\}$ $\qquad\qquad T + T + T + \ldots$

$\quad R$ $\quad \{E.nptr = R.s\}$

$R \to +$

$\quad T$ $\quad \{R_1.i = mkNode\ (\ '+',\ R.i,\ T.nptr)\}$

$\quad R_1$ $\quad \{R.s = R_1.s\}$

$R \to \varepsilon$ $\quad \{R.s = R.i\ \}$

$T \to F$ $\quad \{W.i = F.nptr\}$

$\quad W$ $\quad \{T.nptr = W.s\}$

$W \to *$

$\quad F$ $\quad \{W_1.i = mkNode\ (\ '*',\ W.i,\ F.nptr)\}$

$\quad W_1$ $\quad \{W.s = W_1.s\}$

$W \to \varepsilon$ $\quad \{W.s = W.i\ \}$

F 产生式部分不再给出

文法是LL(1)，所以适合自顶向下分析

L属性定义计算-递归下降

产生式$R \rightarrow +TR \mid \varepsilon$ 的递归下降分析过程

```
void R( ) {
    if (lookahead == '+' ) {
          match ( '+' );
          T( );
          R( );
    }
    else  /* 什么也不做 */
}
```

L属性定义计算-递归下降

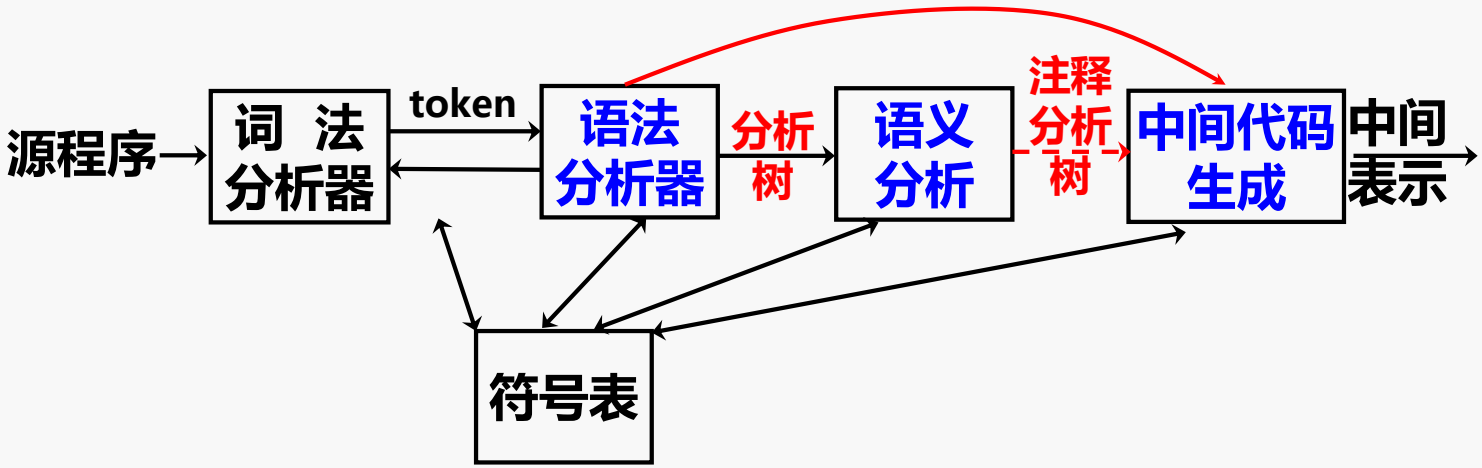
```
syntaxTreeNode* R (syntaxTreeNode* i) {
    syntaxTreeNode *nptr, *i1, *s1, *s;
    char addoplexeme;

    if (lookahead == '+' ) { /* 产生式 R → +T R  */
          addoplexeme = lexval;
          match('+' );
          nptr = T();
          i1 = mkNode(addoplexeme, i , nptr);
          s1 = R (i1);
          s = s1;
    }
    else  s = i;  /* 产生式 R → ε  */
    return s;
}
```

R : i, s

T : $nptr$

$+$: $addoplexeme$

本节提纲



- L属性定义的SDT
- 实现方式1：与递归下降分析结合
- **实现方式2：与LR分析结合**

L属性定义的自底向上计算

- **可否直接将带有继承属性的L-SDD直接与LR结合起来?**
- **考虑$A \to BC$产生式，假设$B.i = A.i$**
  - 当对B进行归约时，由于没有看到C，不能确定会使用该产生式，$A.i$无法得知，此时可以推迟计算
  - 等按照BC归约成为A之后，我们仍然不能确定包含了A的产生式，$A.i$仍无法得知，继续推迟计算
  - 最终退化为“先造分析树，后翻译”的策略
- **解决方案:**
  - 可将产生式中嵌入的动作删除，挪到产生式最右端

L属性定义的自底向上计算

- 将L-SDD转换为SDT
- 对于产生式$A \to \alpha \{a\} \beta$，a是语义动作
  - 引入新的非终结符M，代替$\{a\}$，形成$A \to \alpha M \beta$
  - 引入新的产生式 $M \to \varepsilon$
  - 修改a得到a’：
    - 将a需要的A或者α中的属性作为M的继承属性进行复制
    - 按照a中的方法计算各属性，将这些属性作为M的综合属性保存起来
  - 将$\{a'\}$与$M \to \varepsilon$关联起来

L属性定义的自底向上计算

- 假设LL文法对应的SDT有如下片段

$$A \rightarrow \{B.i = f(A.i)\}BC$$

- 修改后的SDT为

$$A \rightarrow M\ B\ C$$

$$M \rightarrow \varepsilon\ \{M.i = A.i;\ M.s = f(M.i)\}$$

分析栈的设计可以保证M的语义规则中可以使用A.i，因为A.i在M下方紧邻的位置。

L属性定义的自底向上计算

- 例：删除翻译方案中嵌入的动作

$E \rightarrow T\ R$

$R \rightarrow +\ T\ \{print\ (\text{‘+’})\}R_1 \mid -\ T\ \{print\ (\text{‘–’})\}R_1 \mid \varepsilon$

$T \rightarrow \text{num}\ \{print\ (\text{num}.val)\}$

这些动作的一个重要特点：
没有引用原来产生式文法符号的属性，即只涉及虚拟属性

L属性定义的自底向上计算

- 例：删除翻译方案中嵌入的动作

$E \to T\ R$

$R \to +\ T\ \{print\ (\text{'+'})\}R_1 \mid -\ T\ \{print\ (\text{'-'})\}R_1 \mid \varepsilon$

$T \to \text{num}\ \{print\ (\text{num}.val)\}$

加入产生ε的标记非终结符，让每个嵌入动作由不同标记非终结符M代表，并把该动作放在产生式$M \to \varepsilon$的右端

$E \to T\ R$

$R \to +\ T\ M\ R_1 \mid -\ T\ N\ R_1 \mid \varepsilon$

$T \to \text{num}\ \{print\ (\text{num}.val)\}$

$M \to \varepsilon\ \{print\ (\text{'+'})\}$

$N \to \varepsilon\ \{print\ (\text{'-'})\}$

这些动作的一个重要特点：
没有引用原来产生式文法符号的属性，即只涉及虚拟属性

L属性定义的自底向上计算

- 例 数学排版语言EQN

$S \rightarrow$ {$B.ps = 10$ }
B {$S.ht = B.ht$ }
$B \rightarrow$ {$B_1.ps = B.ps$ }
B_1 {$B_2.ps = B.ps$ }
B_2 {$B.ht = max(B_1.ht, B_2.ht)$ }
$B \rightarrow$ { $B_1.ps = B.ps$ }
B_1
sub { $B_2.ps = shrink(B.ps)$ }
B_2 {$B.ht = disp\ (B_1.ht, B_2.ht)$ }
$B \rightarrow$ text {$B.ht = \text{text}.h \times B.ps$ }

文字排版引入标记符号

为了自底向上的计算:

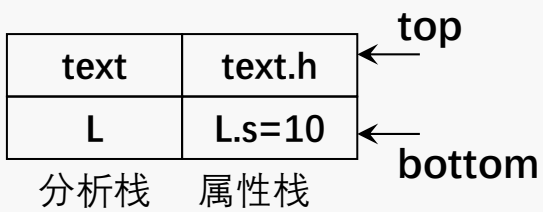
B→ text　　{ B.ht := text.h × B.ps }

必须确定继承属性B.ps的（“属性栈”）位置。为引入标记非终结符L、M和N及其属性，包括相应的空产生式和有关属性规则。这样B.ps即可在紧靠“句柄”text下方的位置上找到。(L的综合属性置为B.ps的初值)

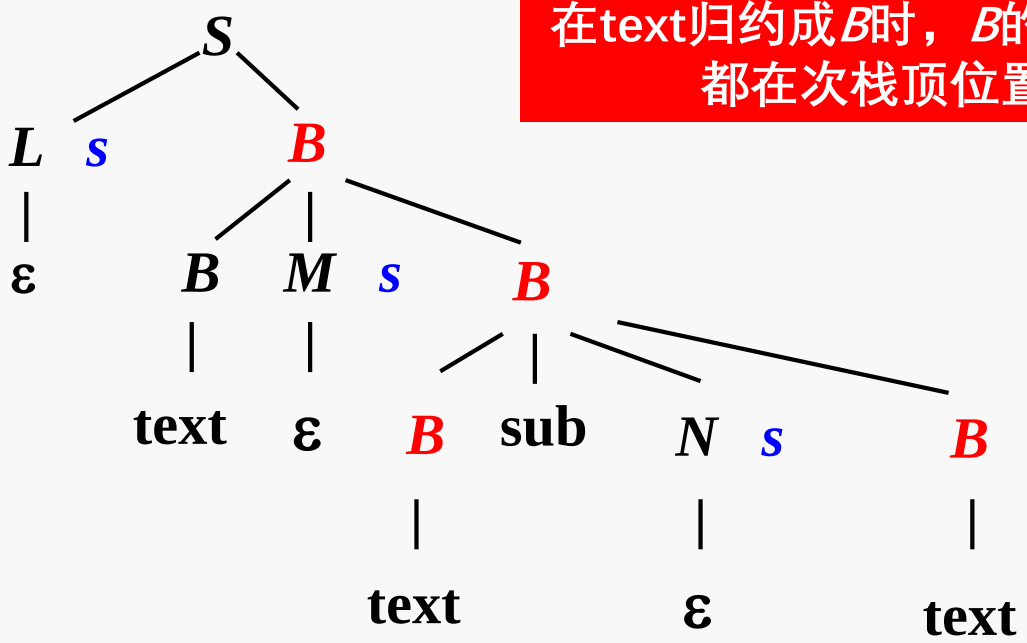
$S \rightarrow L\ B$

$B \rightarrow B_1 M\ B_2$

$B \rightarrow B_1\ \text{sub}\ N\ B_2$



L属性定义的自底向上计算



在text归约成*B*时，*B*的*ps*属性都在次栈顶位置

L属性定义的自底向上计算

产 生 式	语 义 规 则
$S \rightarrow LB$	$B.ps = L.s$; $S.ht = B.ht$
$L \rightarrow \varepsilon$	$L.s = 10$ 将$B.ps$存入栈中，便于引用
$B \rightarrow B_1\ MB_2$	$B_1.ps = B.ps$; $M.i = B.ps$; $B_2.ps = M.s$;$B.ht = max(B_1.ht, B_2.ht)$
$M \rightarrow \varepsilon$	$M.s = M.i$
$B \rightarrow B_1$ sub NB_2	$B_1.ps = B.ps$; $N.i = B.ps$; $B_2.ps = N.s$; $B.ht = disp\ (B_1.ht, B_2.ht)$
$N \rightarrow \varepsilon$	$N.s = shrink(N.i)$
$B \rightarrow$ text	$B.ht =$ text.$h \times B.ps$

L属性定义的自底向上计算

产 生 式	语 义 规 则
$S \rightarrow LB$	$B.ps = L.s$; $S.ht = B.ht$
$L \rightarrow \varepsilon$	$L.s = 10$ 将$B.ps$存入栈中，便于引用
$B \rightarrow B_1\ MB_2$	$B_1.ps = B.ps$; $M.i = B.ps$; $B_2.ps = M.s$;$B.ht = max(B_1.ht, B_2.ht)$
$M \rightarrow \varepsilon$	$M.s = M.i$ 单纯为了属性位置可预测
$B \rightarrow B_1$ sub NB_2	$B_1.ps = B.ps$; $N.i = B.ps$; $B_2.ps = N.s$; $B.ht = disp\ (B_1.ht, B_2.ht)$
$N \rightarrow \varepsilon$	$N.s = shrink(N.i)$
$B \rightarrow$ text	$B.ht =$ text.$h \times B.ps$

L属性定义的自底向上计算

产 生 式	语 义 规 则
$S \rightarrow LB$	$B.ps = L.s$; $S.ht = B.ht$
$L \rightarrow \varepsilon$	$L.s = 10$ 将$B.ps$存入栈中，便于引用
$B \rightarrow B_1\ MB_2$	$B_1.ps = B.ps$; $M.i = B.ps$; $B_2.ps = M.s$;$B.ht = max(B_1.ht, B_2.ht)$
$M \rightarrow \varepsilon$	$M.s = M.i$ 单纯为了属性位置可预测
$B \rightarrow B_1$ sub NB_2	$B_1.ps = B.ps$; $N.i = B.ps$; $B_2.ps = N.s$; $B.ht = disp\ (B_1.ht, B_2.ht)$
$N \rightarrow \varepsilon$	$N.s = shrink(N.i)$ 兼有计算功能
$B \rightarrow$ text	$B.ht$ = text.$h \times B.ps$

L属性定义的自底向上计算

产 生 式	语 义 规 则
$S \rightarrow LB$	$B.ps = L.s$; $S.ht = B.ht$
$L \rightarrow \varepsilon$	$L.s = 10$
$B \rightarrow B_1\ MB_2$	$B_1.ps = B.ps$; $M.i = B.ps$; $B_2.ps = M.s$; $B.ht = max(B_1.ht, B_2.ht)$
$M \rightarrow \varepsilon$	$M.s = M.i$
$B \rightarrow B_1$ sub NB_2	$B_1.ps = B.ps$; $N.i = B.ps$; $B_2.ps = N.s$; $B.ht = disp\ (B_1.ht, B_2.ht)$
$N \rightarrow \varepsilon$	$N.s = shrink(N.i)$
$B \rightarrow$ text	$B.ht =$ text.$h \times B.ps$

继承属性的值等于栈中某个综合属性的值，因此栈中只保存综合属性的值

L属性定义的自底向上计算

产 生 式	代 码 段
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$L.s = 10$
$B \rightarrow B_1\ MB_2$	$B_1.ps = B.ps;\ \ M.i = B.ps;$ $B_2.ps = M.s; B.ht = max(B_1.ht, B_2.ht)$
$M \rightarrow \varepsilon$	$M.s = M.i$
$B \rightarrow B_1$ sub NB_2	$B_1.ps = B.ps;\ \ N.i = B.ps;$ $B_2.ps = N.s;\ B.ht = disp\ (B_1.ht, B_2.ht)$
$N \rightarrow \varepsilon$	$N.s = shrink(N.i)$
$B \rightarrow$ text	$B.ht =$ text.$h \times B.ps$

B.ps= L.s; S.ht= B.ht

L属性定义的自底向上计算

产 生 式	代 码 段
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1\ MB_2$	$B_1.ps = B.ps;\ \ M.i = B.ps;$ $B_2.ps = M.s; B.ht = max(B_1.ht, B_2.ht)$
$M \rightarrow \varepsilon$	$M.s = M.i$
$B \rightarrow B_1$ sub NB_2	$B_1.ps = B.ps;\ \ N.i = B.ps;$ $B_2.ps = N.s;\ B.ht = disp\ (B_1.ht, B_2.ht)$
$N \rightarrow \varepsilon$	$N.s = shrink(N.i)$
$B \rightarrow$ text	$B.ht =$ text.$h \times B.ps$

L.s = 10

L属性定义的自底向上计算

产 生 式	代 码 段
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1\ MB_2$	$B_1.ps = B.ps;\ \ M.i = B.ps;$ $B_2.ps = M.s; B.ht = max(B_1.ht, B_2.ht)$
$M \rightarrow \varepsilon$	$M.s = M.i$
$B \rightarrow B_1$ sub NB_2	$B_1.ps = B.ps;\ \ N.i = B.ps;$ $B_2.ps = N.s;\ B.ht = disp\ (B_1.ht, B_2.ht)$
$N \rightarrow \varepsilon$	$N.s = shrink(N.i)$
$B \rightarrow$ text	$B.ht =$ text.$h \times B.ps$

归约时，弹出0个元素，随后将L压栈，因此top为top+1

L.s = 10

L属性定义的自底向上计算

产 生 式	代 码 段
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1\ MB_2$	$val[top-2] = max(val[top-2], val[top])$
$M \rightarrow \varepsilon$	$M.s = M.i$
$B \rightarrow B_1$ sub NB_2	$B_1.ps = B.ps;\ \ N.i = B.ps;$ $B_2.ps = N.s;\ B.ht = disp\ (B_1.ht, B_2.ht\ )$
$N \rightarrow \varepsilon$	$N.s = shrink(N.i)$
$B \rightarrow$ text	$B.ht =$ text.$h \times B.ps$

B1.ps= B.ps; M.i = B.ps;B2.ps= M.s;B.ht= max(B1.ht, B2.ht)

L属性定义的自底向上计算

产 生 式	代 码 段
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1\ MB_2$	$val[top-2] = max(val[top-2], val[top])$
$M \rightarrow \varepsilon$	$val[top+1] = val[top-1]$
$B \rightarrow B_1$ sub NB_2	$B_1.ps = B.ps;\ \ N.i = B.ps;$ $B_2.ps = N.s;\ B.ht = disp\ (B_1.ht, B_2.ht\ )$
$N \rightarrow \varepsilon$	$N.s = shrink(N.i)$
$B \rightarrow$ text	$B.ht = \text{text}.h \times B.ps$

M.i = B.ps; M.s = M.i

L属性定义的自底向上计算

产 生 式	代 码 段
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1\ MB_2$	$val[top-2] = max(val[top-2], val[top])$
$M \rightarrow \varepsilon$	$val[top+1] = val[top-1]$
$B \rightarrow B_1$ sub NB_2	$val[top-3] = disp\ (val[top-3], val[top]\ )$
$N \rightarrow \varepsilon$	$N.s = shrink(N.i)$
$B \rightarrow$ text	$B.ht =$ text.$h \times B.ps$

B1.ps=B.ps; N.i = B.ps;B2.ps= N.s; B.ht= disp(B1.ht,B2.ht)

L属性定义的自底向上计算

产 生 式	代 码 段
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1\ MB_2$	$val[top-2] = max(val[top-2], val[top])$
$M \rightarrow \varepsilon$	$val[top+1] = val[top-1]$
$B \rightarrow B_1$ sub NB_2	$val[top-3] = disp\ (val[top-3], val[top]\ )$
$N \rightarrow \varepsilon$	$val[top+1] = shrink(val[top-2])$
$B \rightarrow$ text	$B.ht = \text{text}.h \times B.ps$

N.i = B.ps; N.s = shrink(N.i)

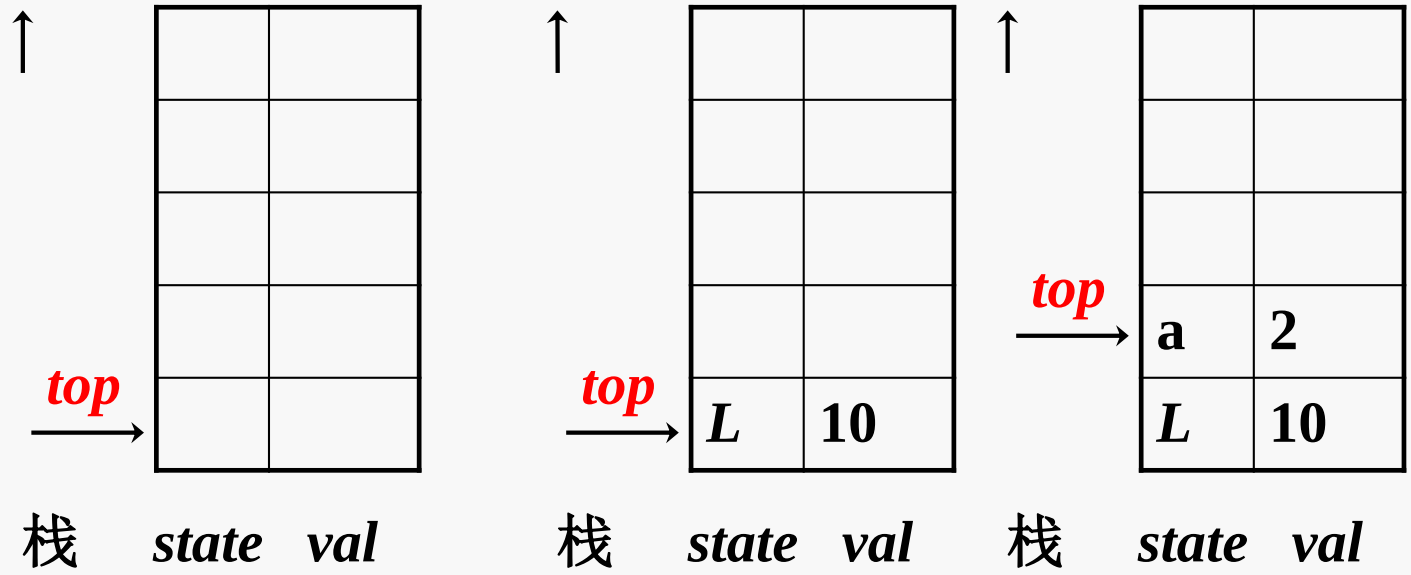
L属性定义的自底向上计算

产 生 式	代 码 段
$S \rightarrow LB$	$val[top-1] = val[top]$
$L \rightarrow \varepsilon$	$val[top+1] = 10$
$B \rightarrow B_1\ MB_2$	$val[top-2] = max(val[top-2], val[top])$
$M \rightarrow \varepsilon$	$val[top+1] = val[top-1]$
$B \rightarrow B_1$ sub NB_2	$val[top-3] = disp\ (val[top-3], val[top]\ )$
$N \rightarrow \varepsilon$	$val[top+1] = shrink(val[top-2])$
$B \rightarrow$ text	$val[top] = val[top] \times val[top-1]$

B.ht= text.h × B.ps

举例说明

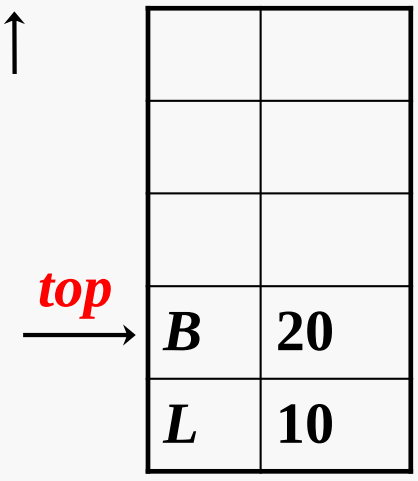
- 考虑字体大小10，基准高度2，输入串a sub 1



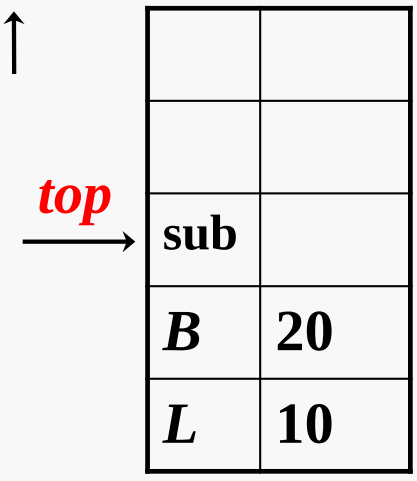
初始状态　　空归约　　移进

举例说明

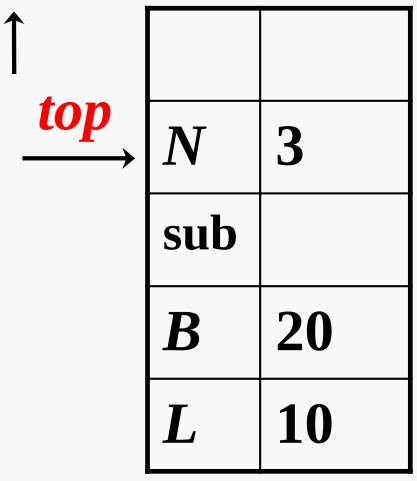
- 考虑字体大小10，基准高度2，输入串a sub 1



text归约



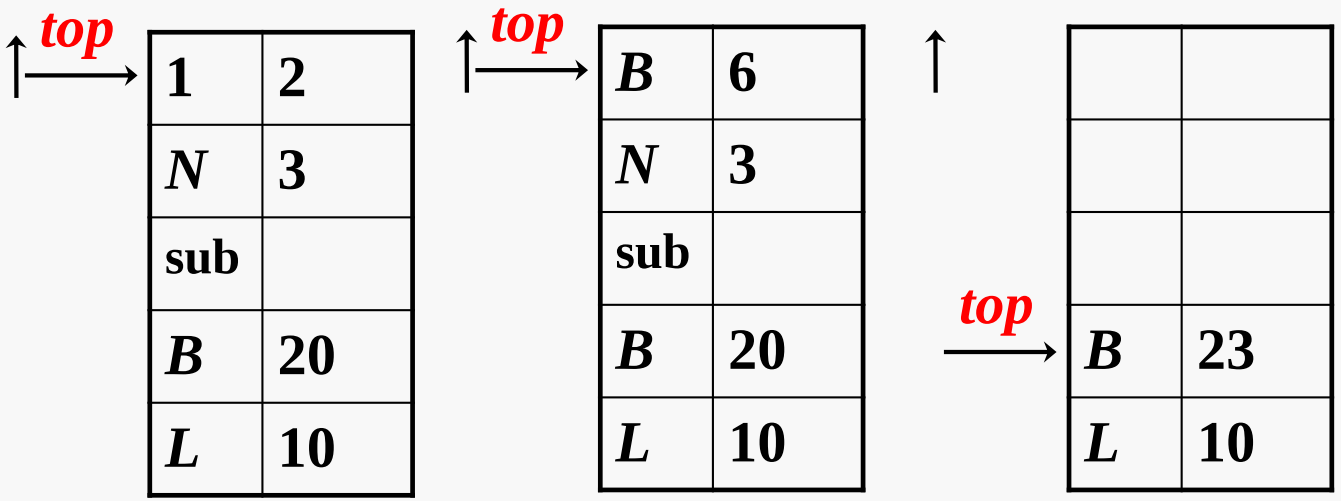
移进关键字



空归约，缩放字体

举例说明

- 考虑字体大小10，基准高度2，输入串a sub 1

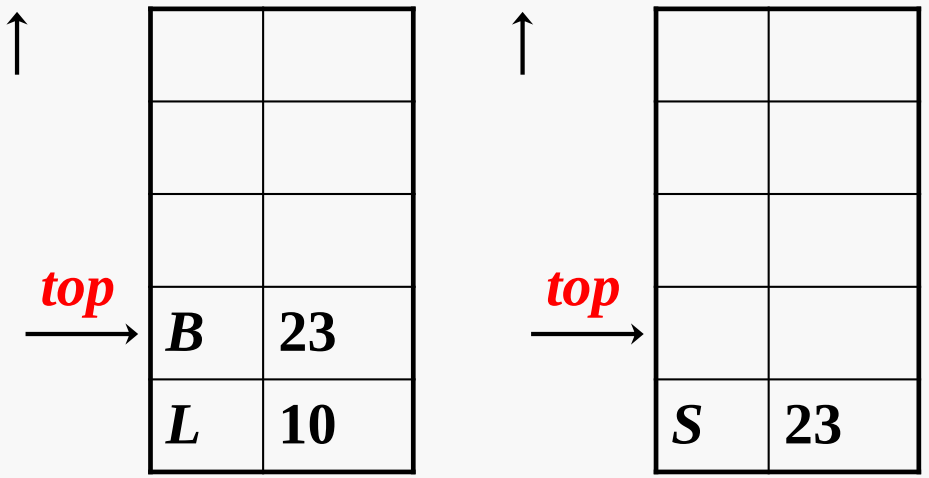


栈 *state* *val*　　栈 *state* *val*　　栈 *state* *val*

移进text　　text归约，使用缩放后的字体大小　　归约，重排高度
假设disp的结果为23

举例说明

- 考虑字体大小10，基准高度2，输入串a sub 1



本章要点

- 语义规则的两种描述方法：语法制导的定义和翻译方案
- 设计简单问题的语法制导定义和翻译方案，这是本章的重点和难点
- 语法制导定义和翻译方案的实现
  - 构造分析树
  - 与分析器结合

《编译原理和技术》

语法制导翻译 IV

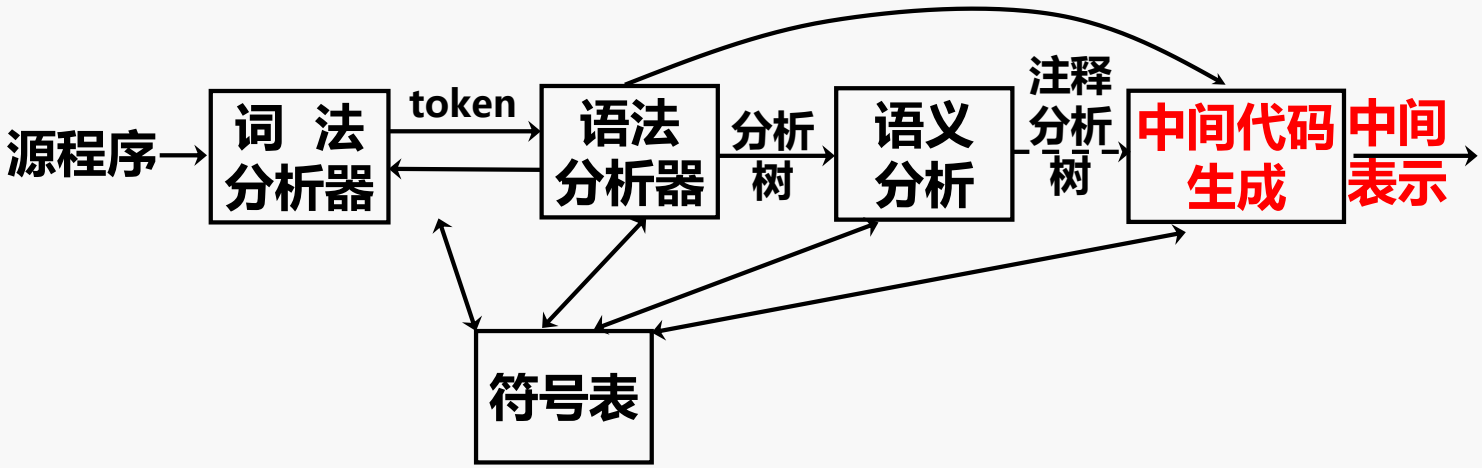
谢谢！

《编译原理和技术》

中间代码生成 I

中科大计算机学院
李 诚
2022-10-12/17

本节提纲



- 中间代码生成简介
- 表达式、switch语句、过程或函数的翻译

中间代码——三地址代码

常用的三地址语句

- 运算/赋值语句　　$x = y\ op\ z$，　$x = op\ y$，　$x = y$
- 无条件转移　　**goto L**
- 条件转移1　　if *x* goto L, if False *x* goto L
- 条件转移2　　**if *x relop y* goto L**

中间代码——三地址代码

常用的三地址语句

- 过程调用
  - **param x_1** //设置参数
  - param x_2
  - …
  - param x_n
  - **call p , *n*** //调用子过程p，n为参数个数
- 过程返回　　**return *y***
- 索引赋值　　***x* = *y*[*i*] 和 *x*[*i*] = *y***
  - 注意：i表示距离y处i个内存单元
- 地址和指针赋值　***x* = &*y*，*x* = **y* 和 **x* = *y***

中间代码生成

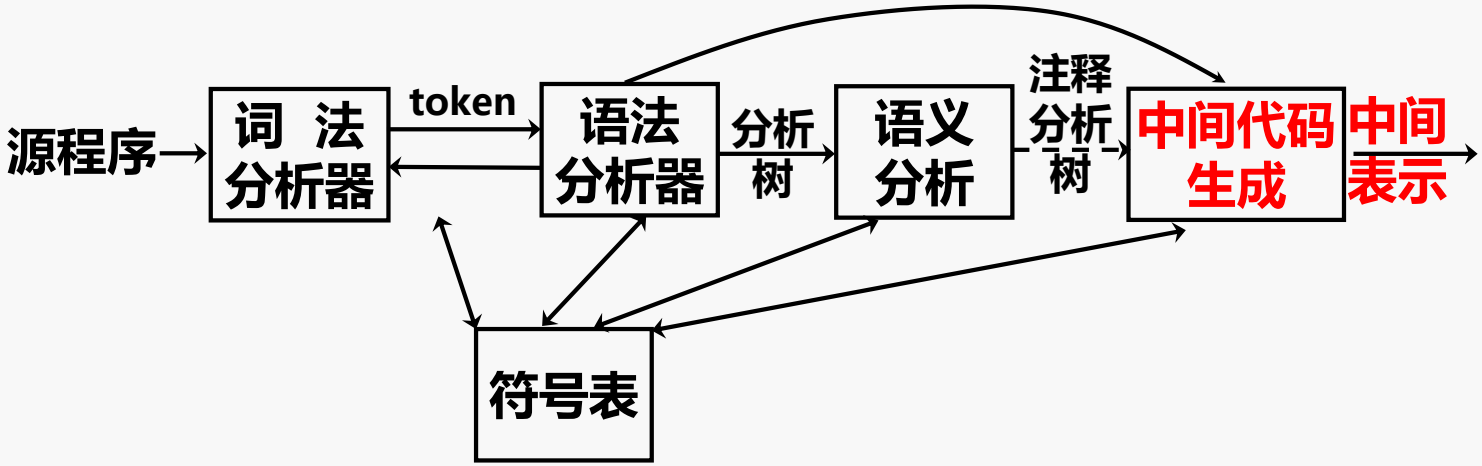
- 表达式的翻译
- switch语句的翻译
- 过程或函数的翻译
- 控制流语句的翻译
- 类型检查
- 声明语句的翻译
- 记录或结构体的翻译
- 数组寻址的翻译

包含布尔表达式计算，较复杂，因此，单独讲

涉及到类型表达式、静态相对地址分配组织，因此，单独设计一个章节来讲

核心技术：
语法制导翻译

本节提纲



- 中间代码生成简介
- 表达式、switch语句、过程或函数的翻译

表达式的中间代码生成

- **表达式文法**

$S \rightarrow \mathrm{id} := E \qquad E \rightarrow E_1 + E_2 \mid -E_1 \mid (E_1) \mid \mathrm{id}$

- **语法制导翻译方案**
  - 属性：*E.place* 存放结果的地址
  - 语义动作：从符号表中获取id的地址
    - *lookup(id.lexeme);*如果不存在，返回*nil*
  - 语义动作：产生临时变量
    - *newTemp();* 保存中间结果
  - 语义动作：输出翻译后的三地址指令
    - *gen(addr, op, arg1, arg2)*
    - 该动作将地址和运算符及临时变量拼接为字符串

表达式的中间代码翻译方案

$S \rightarrow$ id := E {p = *lookup*(id.*lexeme*);
if p != *nil* then
gen (p, ‘=’, E.*place*)
else *error* }

$E \rightarrow E_1 + E_2$
{E.*place* = *newTemp*();
gen (E.*place*, ‘=’, E_1.*place*, ‘+’, E_2.*place*) }

表达式的中间代码翻译方案

$E \rightarrow -E_1$ {*E.place = newTemp();*

gen (E.place, ‘=’, ‘uminus’, E_1*.place)* }

$E \rightarrow (E_1)$ {*E.place =* E_1*.place* }

$E \rightarrow$ id {*p = lookup*(id.*lexeme*);

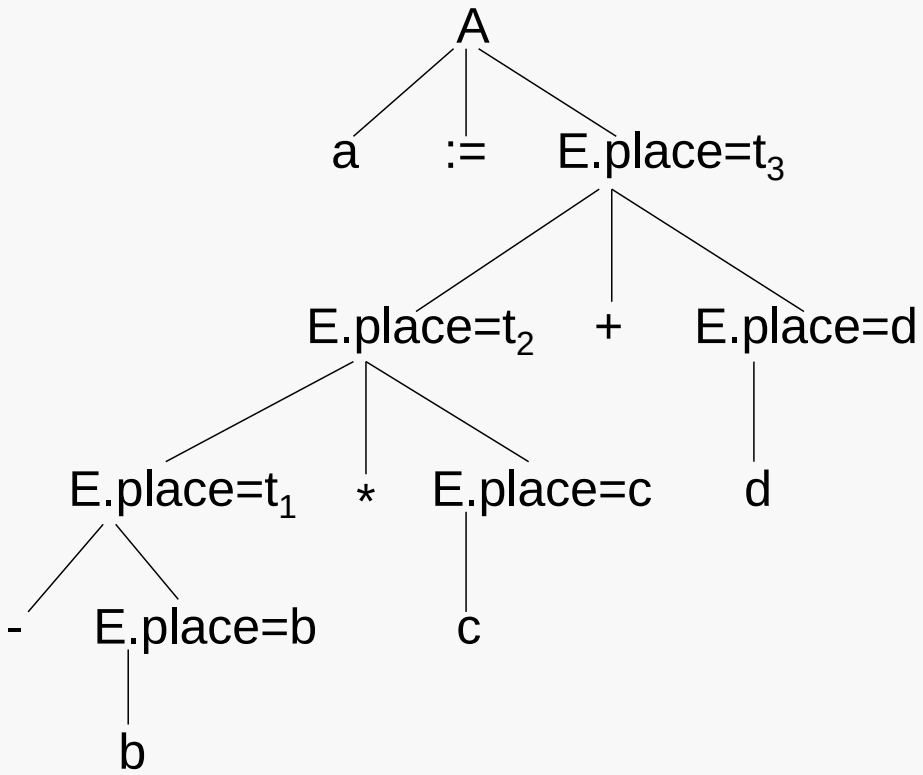
if *p* != *nil* then

E.place = p

else *error* }

举例：表达式翻译

- **a := -b*c+d的翻译**



TAC:

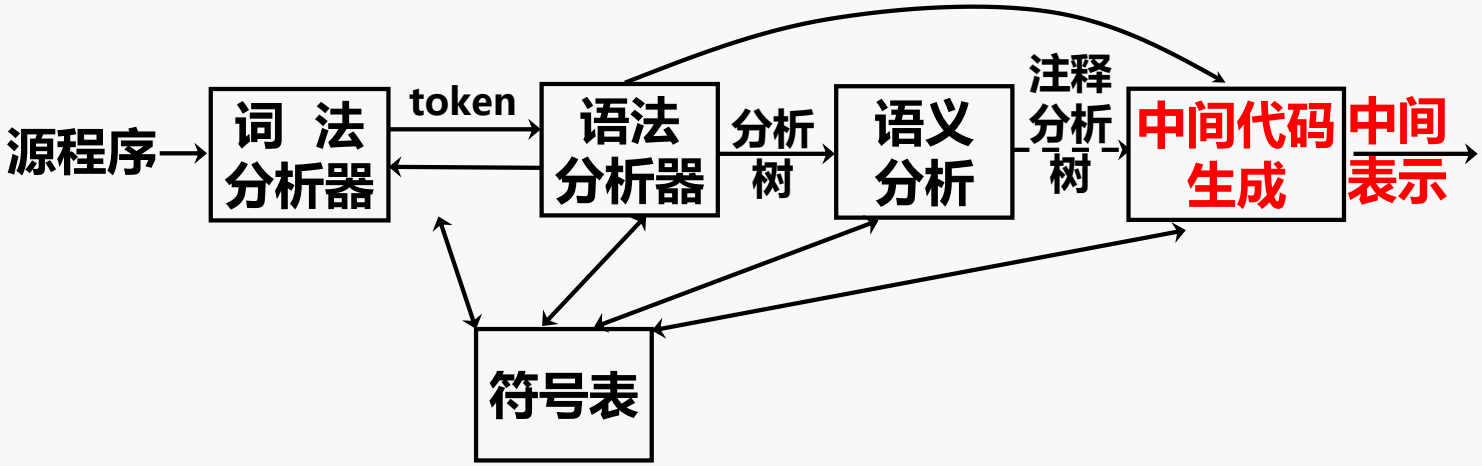
1) $t_1 := - b$

2) $t_2 := t_1 * c$

3) $t_3 := t_2 + d$

4) $a := t_3$

本节提纲



- 中间代码生成简介
- 表达式、switch语句、过程或函数的翻译

switch语句的文法

switch (*E*){ //E是一个选择表达式

case V_1: S_1// V是常量，S是语句

case V_2: S_2

. . .

case V_{n-1}: S_{n-1}

default: S_n

}

switch语句的文法

switch (*E*){ //E是一个选择表达式

case V_1: S_1// V是常量，S是语句

case V_2: S_2

. . .

case V_{n-1}: S_{n-1}

default: S_n

}

三地址代码形态：

- 计算E的值的代码
- S_j语句的代码
- 匹配E的值与V_j，并执行对应S_j的逻辑

switch语句的翻译

- 分支数较少时

```
        E的代码
        t = E.place
        if t != V1 goto L1
        S1的代码
        goto next
L1:     if t != V2 goto L2
        S2的代码
        goto next
L2:     . . .
        . . .
```

```
| Ln-2: if t != Vn-1 goto Ln-1
|       Sn -1的代码
|       goto next
| Ln-1: Sn的代码
| next:
```

switch语句的翻译

- 分支较多时，将分支测试代码集中在一起，便于生成较好的分支测试代码

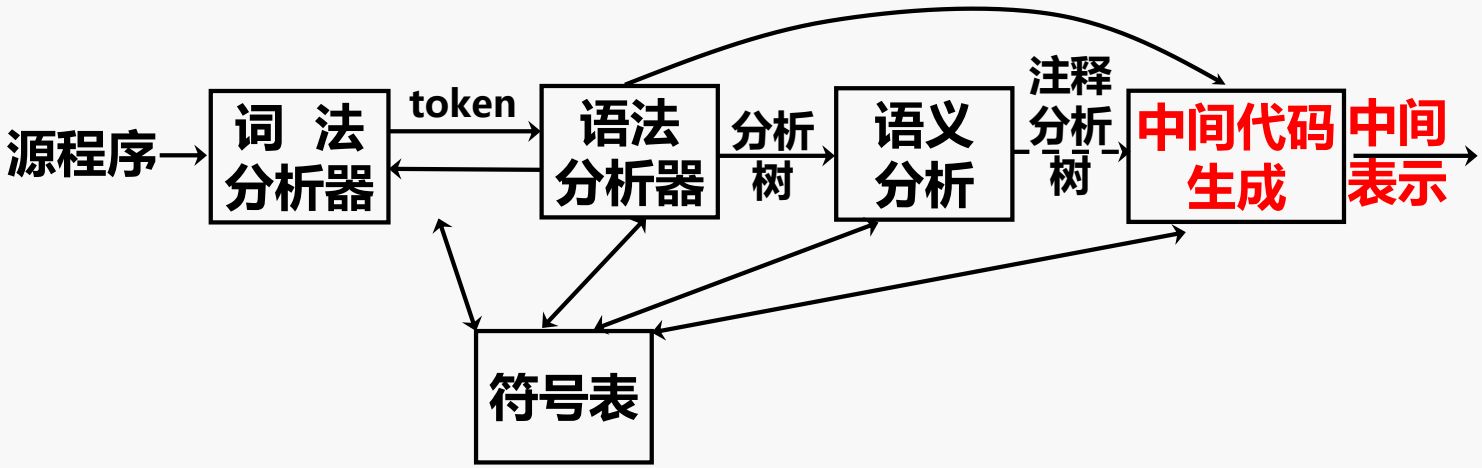
```
        t = E.place          | L_n:  S_n的代码
        goto test            |       goto next
L_1:    S_1的代码            |test: if t == V_1 goto L_1
        goto next            |       if t == V_2 goto L_2
L_2:    S_2的代码            |       . . .
        goto next            |       if t == V_{n-1} goto L_{n-1}
        . . .                |       goto L_n
L_{n-1}: S_{n-1}的代码       | next:
        goto next
```

switch语句的翻译

- 中间代码表示增加一种case语句，与之前的翻译等价，便于代码生成器对它进行特别处理

test: case t V_1 L_1
case t V_2 L_2
. . .
case t V_{n-1} L_{n-1}
case t t L_n
next:

本节提纲



- 中间代码生成简介
- 表达式、switch语句、**过程或函数的翻译**

过程调用的翻译

$S \rightarrow$ call id (*Elist*)

Elist $\rightarrow$ *Elist*, *E*

Elist $\rightarrow$ *E*

此处先忽略函数及过程的定义，会在后面的课程中讲解。

过程调用的翻译

- 过程调用id(E_1, E_2, …, E_n)的中间代码结构

E_1的代码
E_2的代码
. . .
E_n的代码
param E_1.*place*
param E_2.*place*
. . .
param E_n.*place*
call id.*place*, *n*

过程调用的翻译方案

为*Elist*设计一个综合属性paramlist，该列表记录函数调用的所有参数，且参数排列顺序与传参顺序一致

$Elist \rightarrow E$

{*Elist.paramlist = createEmptyList();*

Elist.push_back(E.place);}

过程调用的翻译方案

为*Elist*设计一个综合属性paramlist，该列表记录函数调用的所有参数，且参数排列顺序与传参顺序一致

$Elist \rightarrow E$

{*Elist.paramlist = createEmptyList();*

Elist.paramlist.push_back(E.place);}

$Elist \rightarrow Elist_1, E$

{$Elist.paramlist = Elist_1.paramlist$;

Elist.paramlist.push_back(E.place);}

过程调用的翻译方案

$S \rightarrow$ call id (*Elist*)

{for E_i in *Elist.paramlist*:

gen('param', E_i.*place*);

gen('call', id.*place*, *Elist.paramlist.size()*) ;}

《编译原理和技术》

中间代码生成 I

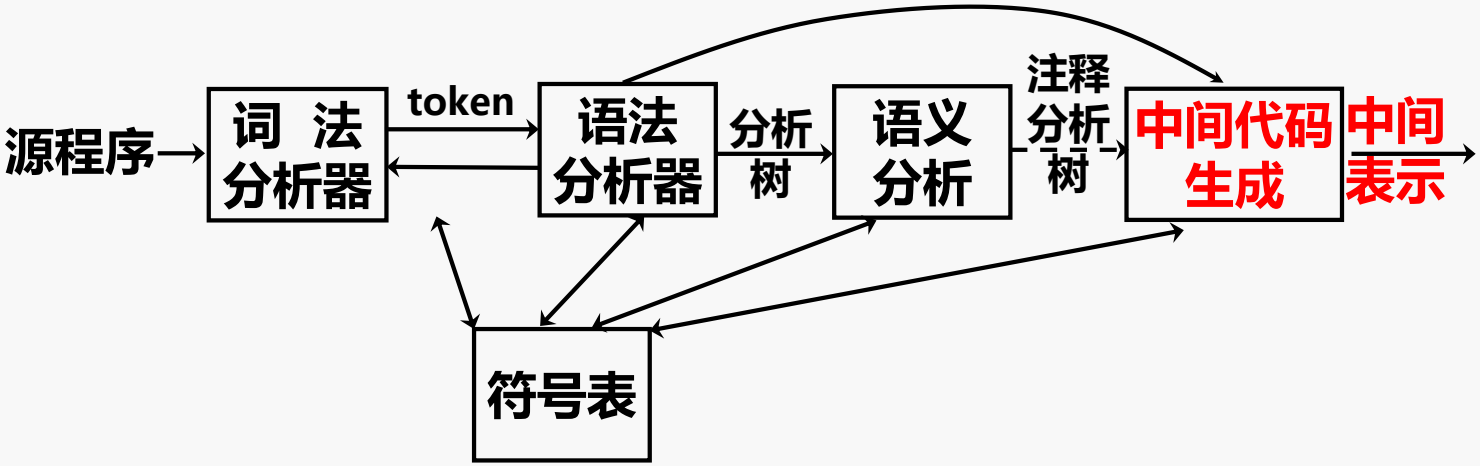
谢谢！

《编译原理和技术》

中间代码生成 II

中科大计算机学院
李 诚
2022-10-17

本节提纲



- 控制流语句文法
- 简单控制流语句的翻译
  - if, if-then-else, while, 顺序语句
- 布尔表达式的翻译

控制流语句的文法

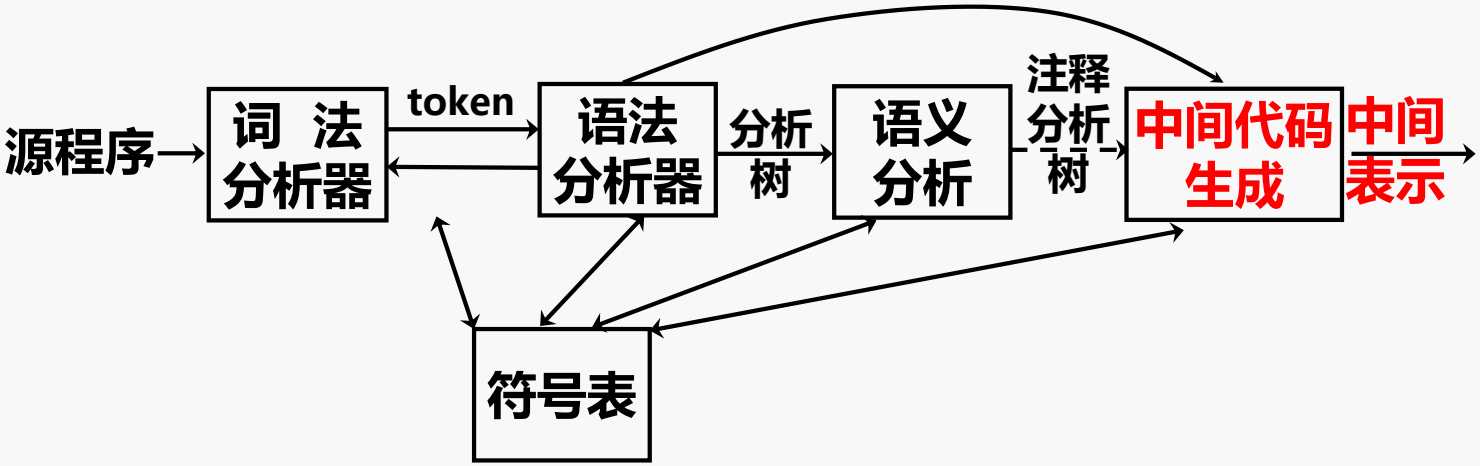
$S \rightarrow$ if B then S_1

| if B then S_1 else S_2

| while B do S_1

| S_1; S_2

本节提纲



- 控制流语句文法
- 简单控制流语句的翻译
  - if, if-then-else, while, 顺序语句
- 布尔表达式的翻译

if 语句中间代码生成的SDD

问题与对策

- 需要知道B为真或假时的跳转目标
- B、S_1分别会输出多少条指令是不确定的
- 引入标号：先确定标号，在目标确定时输出标号指令，可调用newLabel()产生新标号，每条语句有next 标号

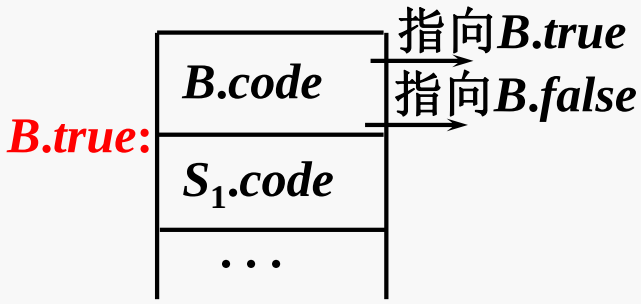
$S \rightarrow$ if B then S_1

{*B.true* = *newLabel*();

B.false = *S.next*;

S_1.*next* = *S.next*;

S.code = *B.code* || *gen*(*B.true*, ':') || S_1.*code* }



(a) if-then

if 语句中间代码生成的SDD

问题与对策

- 需要知道B为真或假时的跳转目标
- B、S_1分别会输出多少条指令是不确定的
- 引入标号：先确定标号，在目标确定时输出标号指令，可调用newLabel()产生新标号，每条语句有next 标号

$S \rightarrow$ if B then S_1

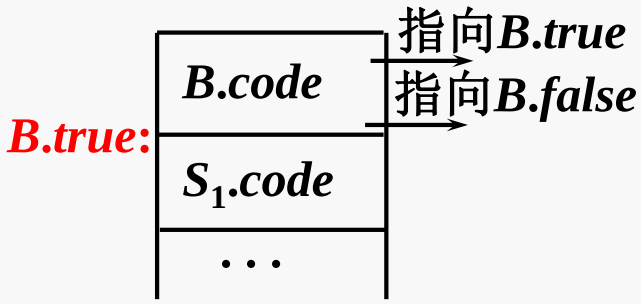
语义规则：

B.true = ***newLabel*()**;

B.false = ***S.next*;//继承属性**

S_1***.next*** = ***S.next*;**

S.code = ***B.code*** || ***gen*(*B.true*, ':')** || S_1***.code***



(a) if-then

if 语句中间代码生成的SDD

问题与对策

- 需要知道B为真或假时的跳转目标
- B、S_1分别会输出多少条指令是不
- 引入标号：先确定标号，在目标
  产生新标号，每条语句有next 标

newLabel()

- 标号指向S内部的三地址代码时需要调用newLabel
- 标号指向S外部的三地址代码时从S继承

$S \rightarrow$ **if** B **then** S_1

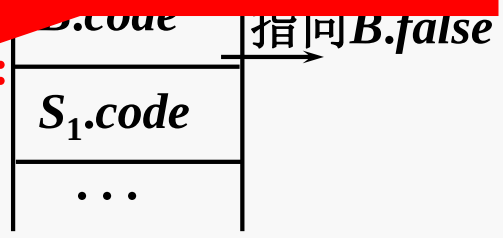
语义规则：

B.true = ***newLabel*****();**

B.false = ***S.next*****;//继承属性**

S_1***.next*** = ***S.next*****;**

S.code = ***B.code*** || ***gen*****(*B.true*, ':')** || S_1***.code***



(a) if-then

if 语句中间代码生成的SDD

问题与对策

- 需要知道B为真或假时的跳转目标
- B、S_1分别会输出多少条指令是不确定的
- 引入标号：先确定标号，在目标确定时输出标号指令，可调用newLabel()产生新标号，每条语句有next 标号

$S \rightarrow$ if B then S_1

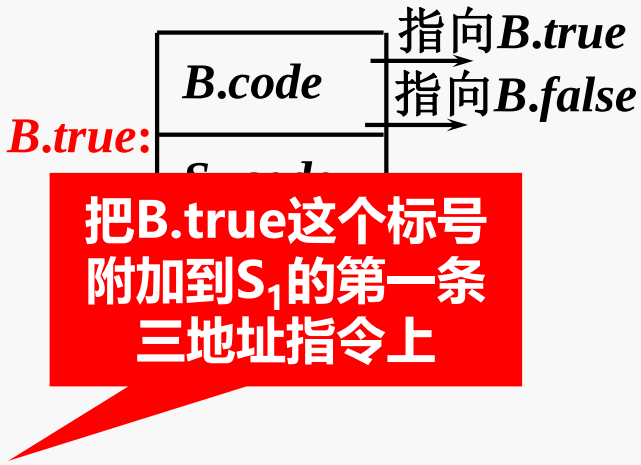
语义规则：

B.true = *newLabel*();

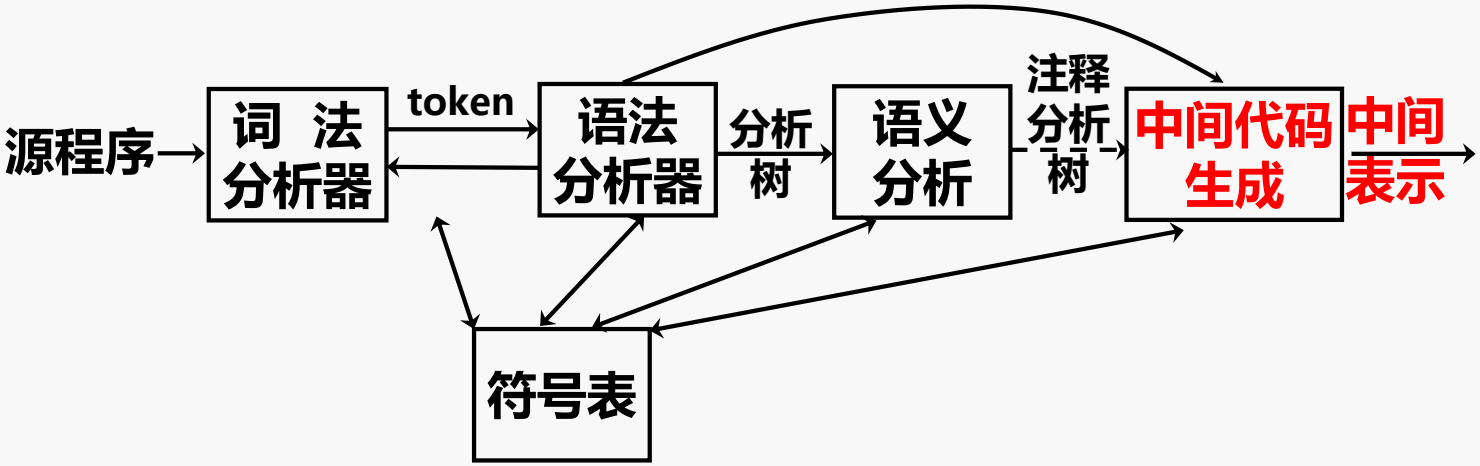
B.false = *S.next*;//**继承属性**

S_1.*next* = *S.next*;

S.code = *B.code* || *gen*(*B.true*, ':') || S_1.*code*



本节提纲



- 控制流语句文法
- 简单控制流语句的翻译
  - if, if-then-else, while, 顺序语句
- 布尔表达式的翻译

if 语句中间代码生成的SDD

- 考虑带有else的语句

$S \rightarrow$ if B then S_1 else S_2

语义动作：

$B.true = newLabel();$

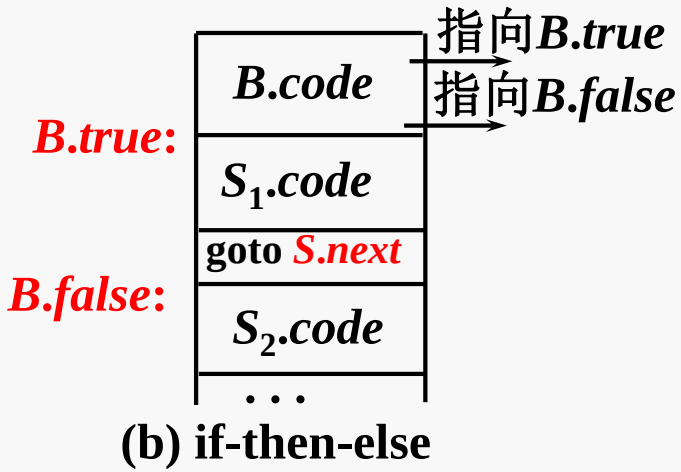
$B.false = newLabel();$

$S_1.next = S.next;$

$S_2.next = S.next;$

$S.code = B.code \,||\, gen(B.true, ':') \,||\, S_1.code \,||$

$gen('goto', S.next) \,||\, gen(B.false, ':') \,||\, S_2.code$



(b) if-then-else

if 语句中间代码生成的SDD

- 考虑带有else的语句

$S \rightarrow$ if B then S_1 else S_2

语义规则：

B.true = *newLabel*();

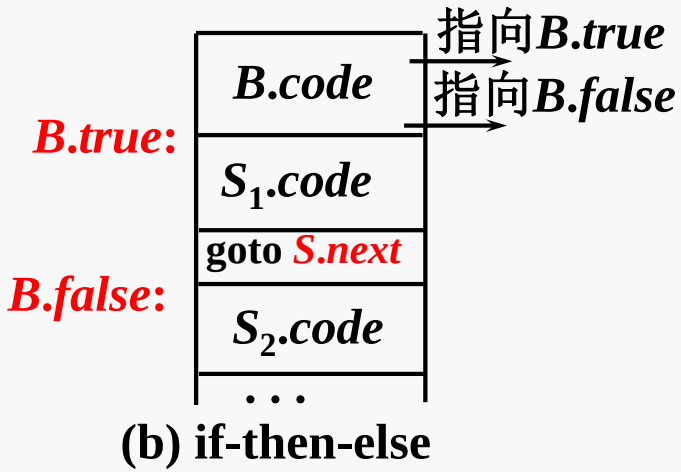
B.false = *newLabel*();

$S_1.next = S.next$;

$S_2.next = S.next$;

$S.code = B.code \,||\, gen(B.true, ':') \,||\, S_1.code \,||$

$gen(\text{'goto'}, S.next) \,||\, gen(B.false, ':') \,||\, S_2.code$



(b) if-then-else

if 语句中间代码生成的SDD

- 考虑带有else的语句

$S \rightarrow$ if B then S_1 else S_2

语义规则：

B.true = *newLabel*();

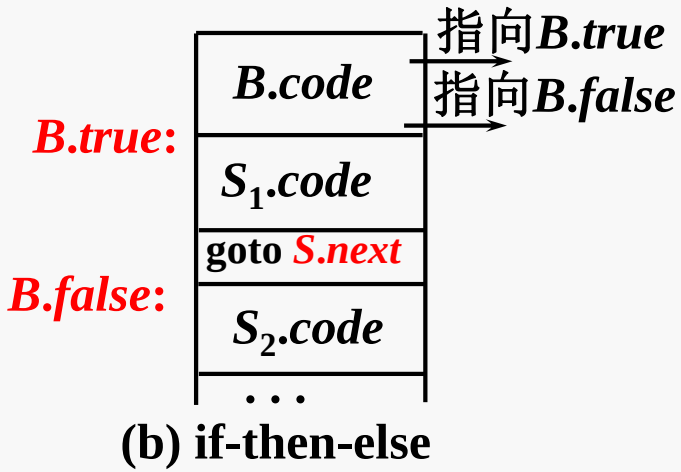
B.false = *newLabel*();

$S_1.next = S.next$;

$S_2.next = S.next$;

S.code = *B.code* || *gen*(*B.true*, ':') || S_1.*code* ||

gen('goto', *S.next*) || *gen*(*B.false*, ':') || S_2.*code*



(b) if-then-else

if 语句中间代码生成的SDD

- 考虑带有else的语句

$S \rightarrow$ if B then S_1 else S_2

语义规则：

$B.true$ = $newLabel()$;

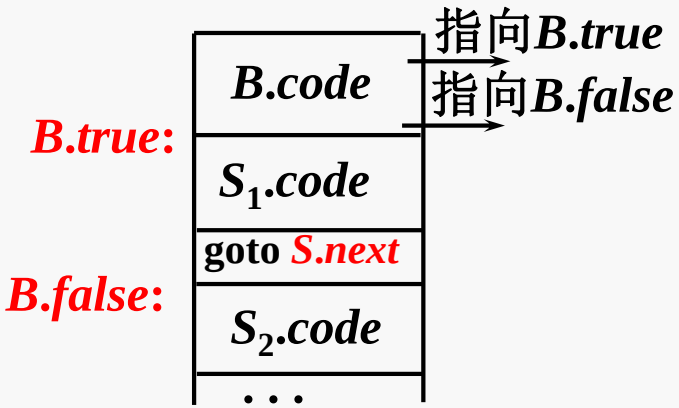
$B.false$ = $newLabel()$;

$S_1.next = S.next$;

$S_2.next = S.next$;

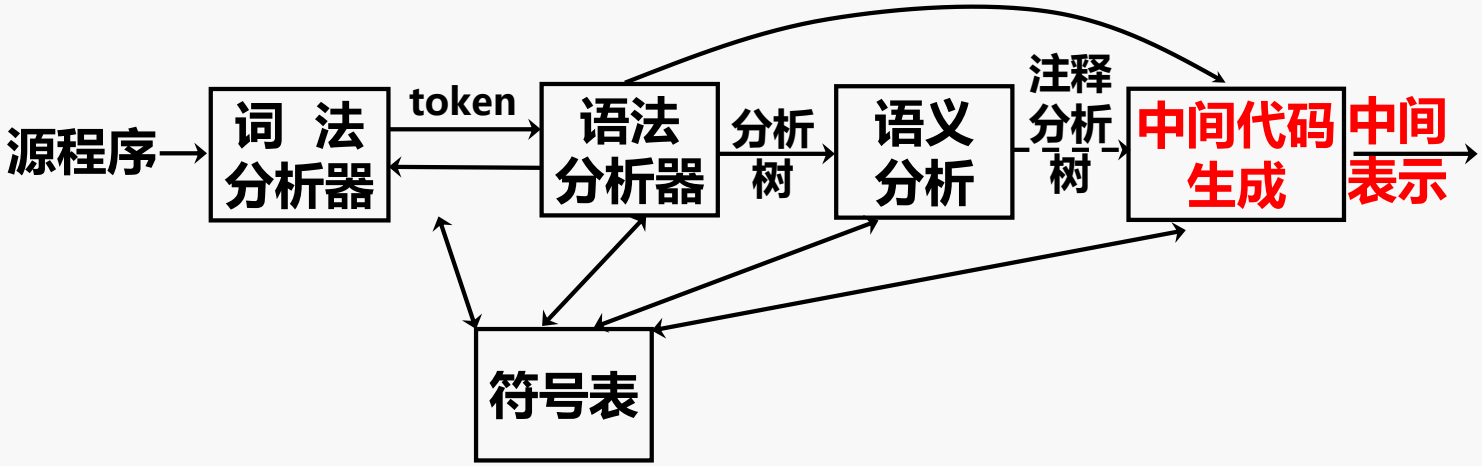
$S.code = B.code \,||\, gen(B.true, \text{':'}) \,||\, S_1.code \,||$

$gen(\text{'goto'}, S.next) \,||\, gen(B.false, \text{':'}) \,||\, S_2.code$



(b) if-then-else

本节提纲



- 控制流语句文法
- 简单控制流语句的翻译
  - if, if-then-else, while, 顺序语句
- 布尔表达式的翻译

while语句中间代码生成SDD

- 引入开始标号*S.begin*,作为循环的跳转目标

$S \rightarrow$ while B do S_1

语义规则：

S.begin = *newLabel*();

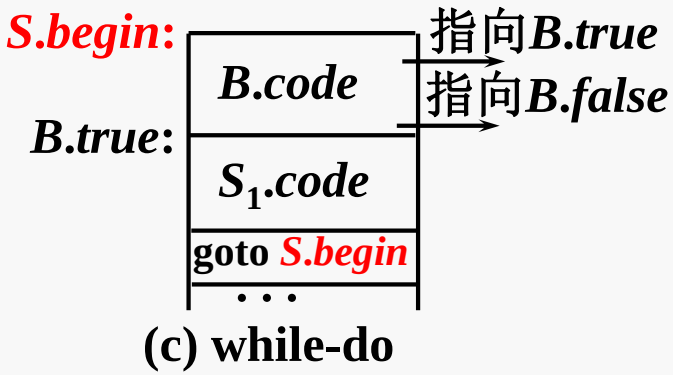
B.true = *newLabel*();

B.false = *S.next*;

S_1.*next* = *S.begin*;

S.code = *gen*(*S.begin*, ':') || *B.code* ||

gen(*B.true*, ':') || S_1.*code* || *gen*('goto', *S.begin*)



(c) while-do

while语句中间代码生成SDD

- 引入开始标号*S.begin*,作为循环的跳转目标

$S \rightarrow$ while B do S_1

语义规则:

S.begin = *newLabel*();

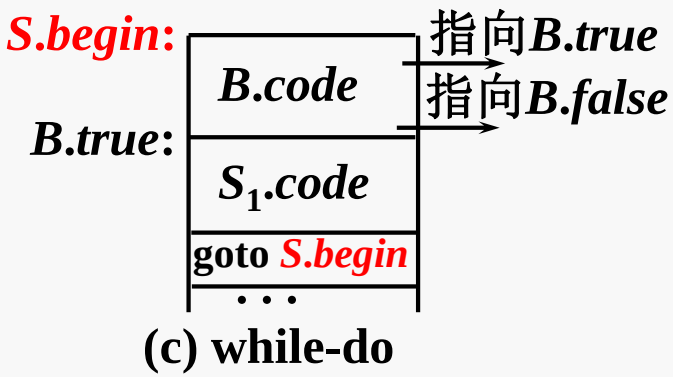
B.true = *newLabel*();

B.false = *S.next*;

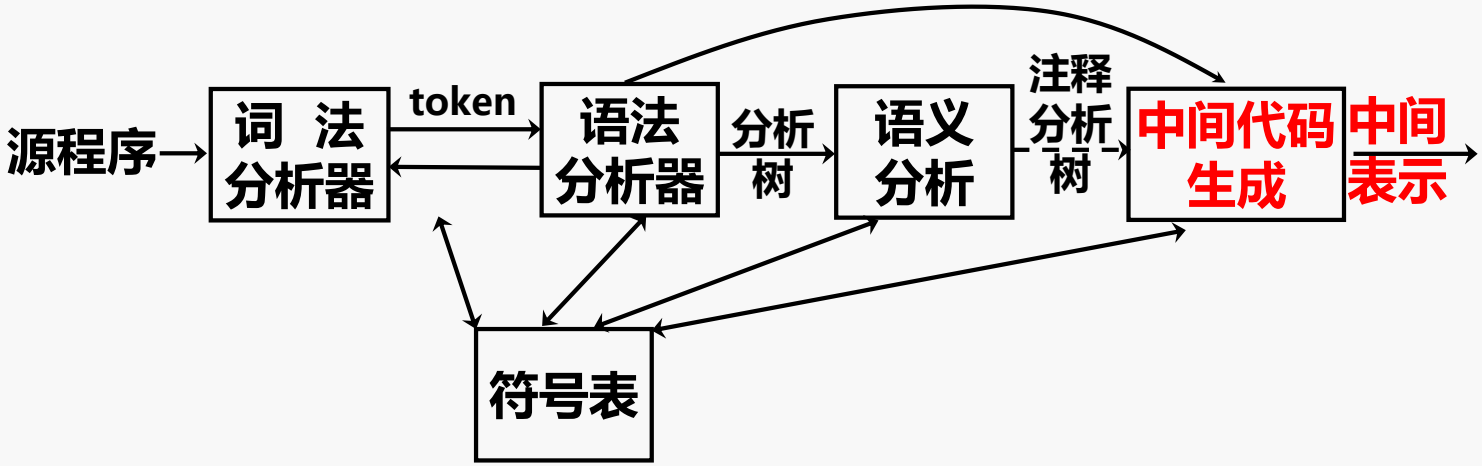
$S_1.next$ = *S.begin*;

S.code = *gen*(*S.begin*, ':') || *B.code* ||

gen(*B.true*, ':') || $S_1.code$ || *gen*('goto', *S.begin*)



本节提纲



- 控制流语句文法
- 简单控制流语句的翻译
  - if, if-then-else, while, 顺序语句
- 布尔表达式的翻译

顺序结构中间代码生成SDD

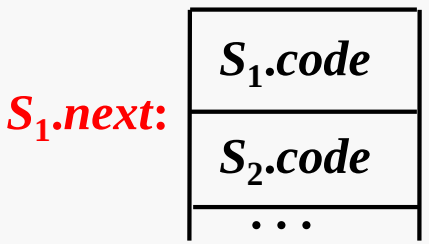
- 为每一语句S_1引入其后的下一条语句的标号$S_1.next$

$S \rightarrow S_1; S_2$

语义规则：

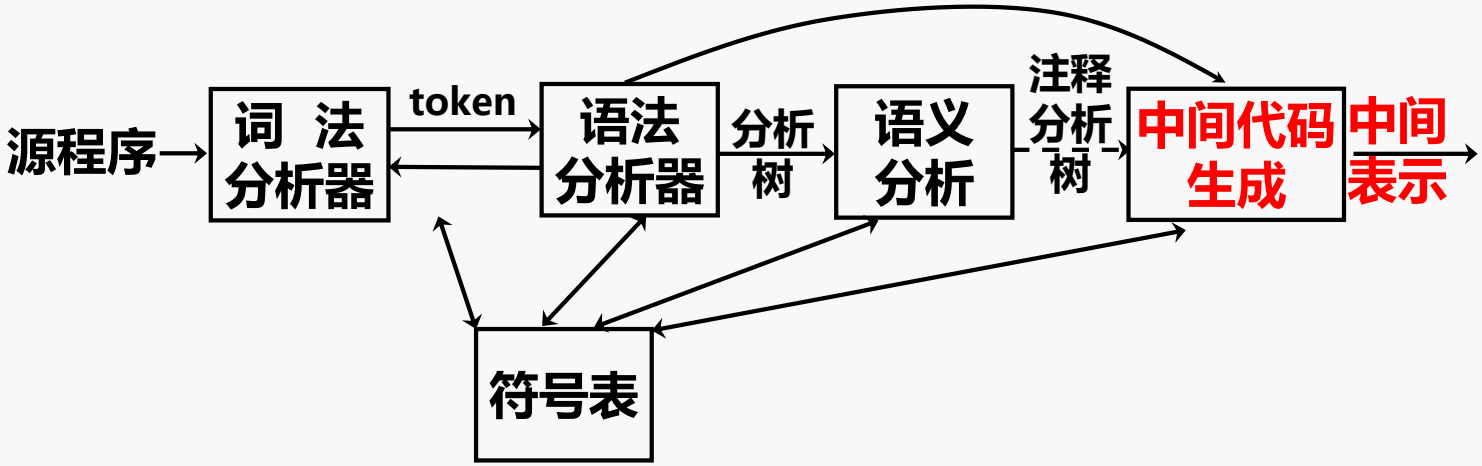
$S_1.next = newLabel(); S_2.next = S.next;$

$S.code = S_1.code \parallel gen(S_1.next, ':') \parallel S_2.code$



(d) $S_1; S_2$

本节提纲



- 控制流语句文法
- 简单控制流语句的翻译
  - if, if-then-else, while, 顺序语句
- 布尔表达式的翻译

布尔表达式的控制流翻译

- 在if-else以及while语句翻译中，并未对B.code进行展开，现在考虑B.code的三地址代码翻译
- 如果B是$a < b$的形式，

那么翻译生成的三地址码是：

if $a < b$ goto $B.true$

goto $B.false$

我们把这种翻译称作为“布尔表达式的控制流翻译”

布尔表达式

- 布尔表达式有两个基本目的
  - 计算逻辑值
    - 例如：作为赋值语句的右值
  - 在控制流语句中用作条件表达式
    - 例如：*if(B) then S*
- 本节所用的布尔表达式文法

$$B \rightarrow B \text{ or } B \mid B \text{ and } B \mid \text{not } B \mid (B) \mid E \text{ relop } E \mid \text{true} \mid \text{false}$$

布尔表达式

- **布尔表达式有两个基本目的**
  - 计算逻辑值
  - 在控制流语句中用作条件表达式
- **本节所用的布尔表达式文法**

$$B \rightarrow B \text{ or } B \mid B \text{ and } B \mid \text{not } B \mid (B) \mid E \text{ relop } E \mid \text{true} \mid \text{false}$$

  - 布尔运算符 or 、and 和 not（优先级、结合性）
  - 关系运算符 relop：＜、≤、＝、≠、＞和≥
  - 布尔常量：true和false

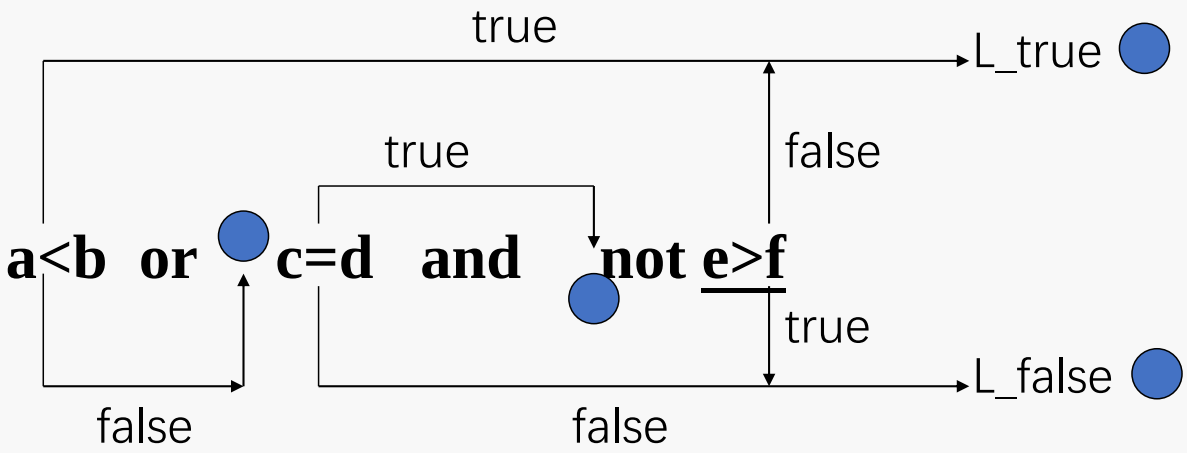
布尔表达式

- **布尔表达式的完全计算**
  - 值的表示数值化
  - 其计算类似于算术表达式的计算

  true and false or (2 > 1)的计算为

  ➔false or (2>1)➔false or true➔true
- **布尔表达式的“短路”计算**
  - B_1 or B_2　　B_1 为真即为真
  - B_1 and B_2　　B_1 为假即为假

布尔表达式的短路计算



L_true-真出口：整个布尔表达式为**真**时，控制流应转移到的目标语句（代码）；反之为**假**时则转到 L_false-假出口。

- 表示转移到的目标语句在有关布尔表达式翻译时尚未确定。

布尔表达式的控制流翻译

- **用控制流来实现计算**
  - 布尔运算符and, or, not不出现在翻译后的代码中
  - 用程序中的位置来表示值
- **例：if(x<3 or x>5 and x!=y) x =10;的翻译**

```
        if x<3 goto L2
        goto L3
L3:     if x>5 goto L4
        goto L1
L4:     if x!=y goto L2
        goto L1
L2:     x = 10
L1:
```

布尔表达式的控制流翻译

- 例 表达式

  $a < b$ or $c < d$ and $e < f$

的三地址码是：

if $a < b$ goto L_{true}

goto L_1

L_1: if $c < d$ goto L_2

goto L_{false}

L_2: if $e < f$ goto L_{true}

goto L_{false}

布尔表达式的控制流翻译

$B \rightarrow B_1$ or B_2

$\{B_1.true = B.true;$

$B_1.false = newLabel();$

$B_2.true = B.true;$

$B_2.false = B.false;$

$B.code = B_1.code \,||\, gen(B_1.false, ':') \,||\, B_2.code\ \}$

布尔表达式的控制流翻译

$B \rightarrow B_1$ or B_2

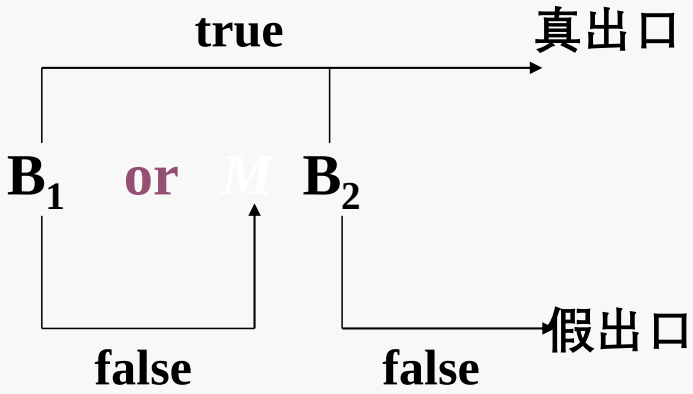
{ $B_1.true = B.true$;

$B_1.false = newLabel()$;

$B_2.true = B.true$;

$B_2.false = B.false$;

$B.code = B_1.code \,||\, gen(B_1.false, ':') \,||\, B_2.code$ }



布尔表达式控制流翻译SDD

$B \rightarrow B_1$ or B_2

语义规则：

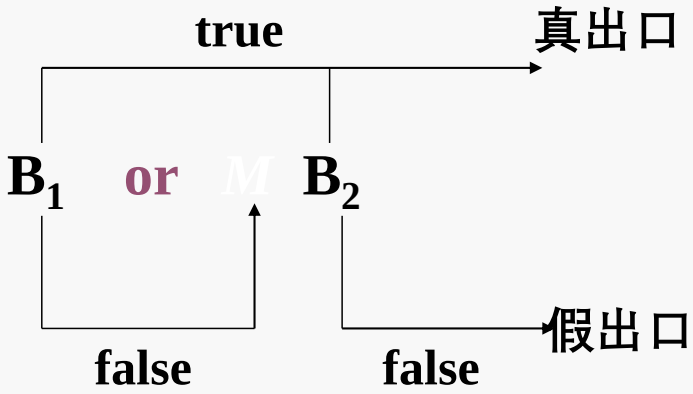
$B_1.true = B.true;$

$B_1.false = newLabel();$

$B_2.true = B.true;$

$B_2.false = B.false;$

$B.code = B_1.code \,||\, gen(B_1.false, ':') \,||\, B_2.code$



布尔表达式的控制流翻译

$B \rightarrow B_1$ and B_2

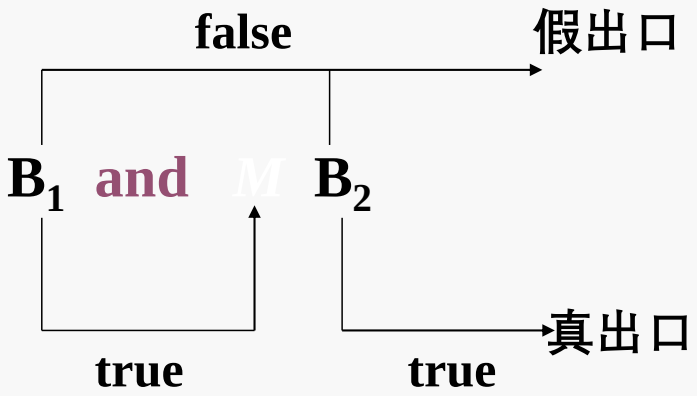
{ $B_1.true = newLabel();$

$B_1.false = B.false;$

$B_2.true = B.true;$

$B_2.false = B.false;$

$B.code = B_1.code \,||\, gen(B_1.true, ':') \,||\, B_2.code$ }



布尔表达式控制流翻译SDD

$B \rightarrow B_1$ and B_2

语义规则：

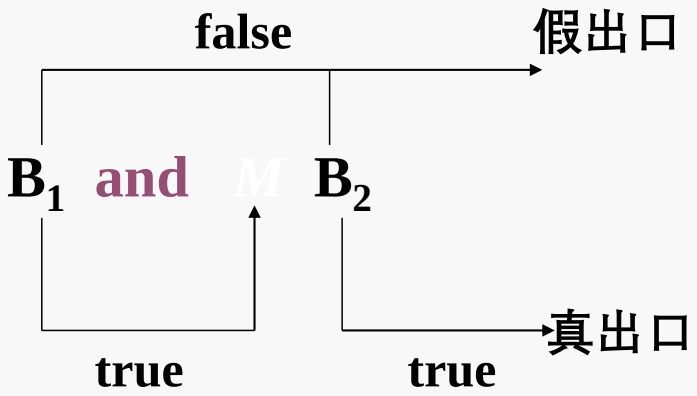
$B_1.true = newLabel();$

$B_1.false = B.false;$

$B_2.true = B.true;$

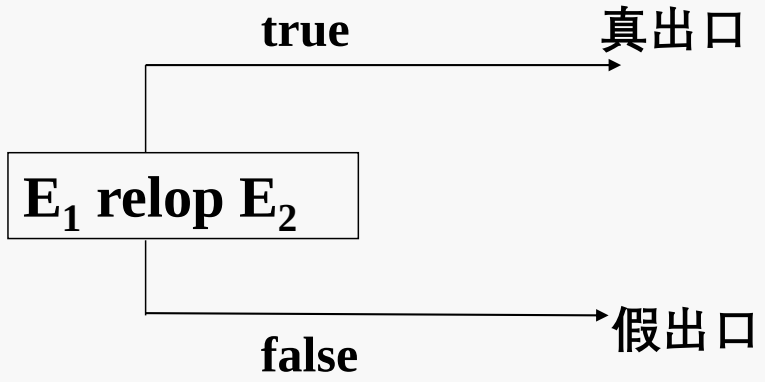
$B_2.false = B.false;$

$B.code = B_1.code \,||\, gen(B_1.true, ':') \,||\, B_2.code$



布尔表达式的控制流翻译

$B \rightarrow E_1$ relop E_2



{*B.code* = E_1.*code* || E_2.*code* ||

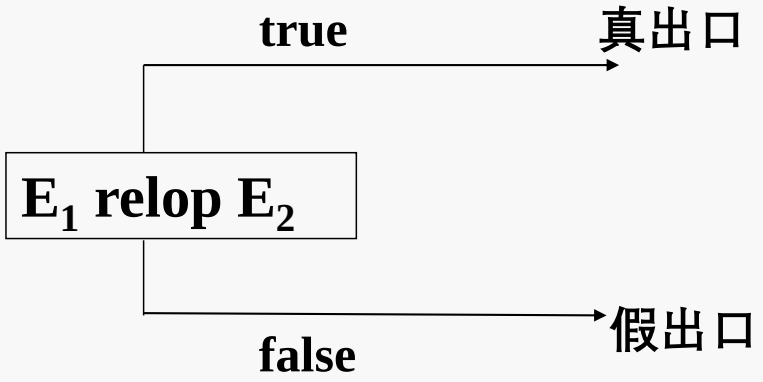
gen('if', E_1.*place*, relop.*op*, E_2.*place*,

'goto', *B.true*) ||

gen('goto', *B.false*) }

布尔表达式控制流翻译SDD

$B \rightarrow E_1$ relop E_2



语义规则：

$B.code = E_1.code \,||\, E_2.code \,||$

$gen(\text{‘if’}, E_1.place, \text{relop}.op, E_2.place,$

$\text{‘goto’}, B.true) \,||$

$gen(\text{‘goto’}, B.false)$

布尔表达式控制流翻译SDD

$B \rightarrow (B_1)$

语义规则：

$B_1.true = B.true;$

$B_1.false = B.false;$

$B.code = B_1.code$

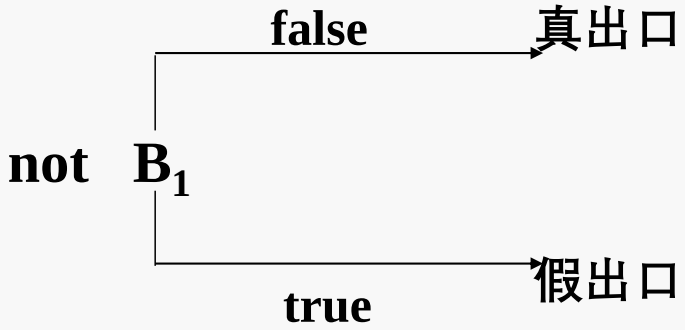
布尔表达式的控制流翻译

$B \rightarrow \text{not } B_1$

$\{B_1.true = B.false;$

$B_1.false = B.true;$

$B.code = B_1.code\ \}$



布尔表达式控制流翻译SDD

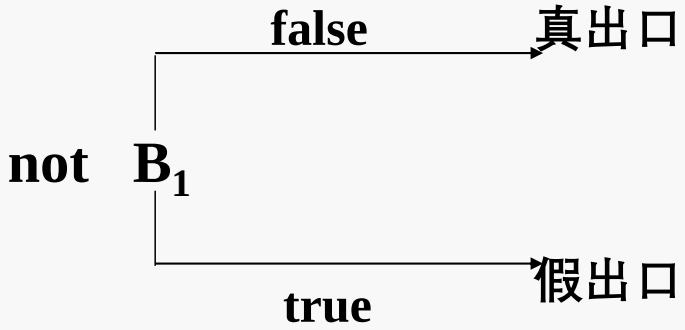
$B \rightarrow \text{not } B_1$

语义规则：

$B_1.true = B.false;$

$B_1.false = B.true;$

$B.code = B_1.code$



布尔表达式控制流翻译SDD

$B \rightarrow$ true

语义规则：

$B.code = gen(\text{‘goto’}, B.true)$

$B \rightarrow$ false

语义规则：

$B.code = gen(\text{‘goto’}, B.false)$

翻译难点

- **关键问题：将跳转指令与目标匹配起来**
- **B.true, B.false都是继承属性**
- **需要两趟分析来计算**
  - 1 pass：生成语法树
  - 2 pass：深度优先遍历树，计算属性值
    - 将标号和具体地址绑定起来
- **能否一趟完成?**
  - 标号回填技术

《编译原理和技术》

中间代码生成 II

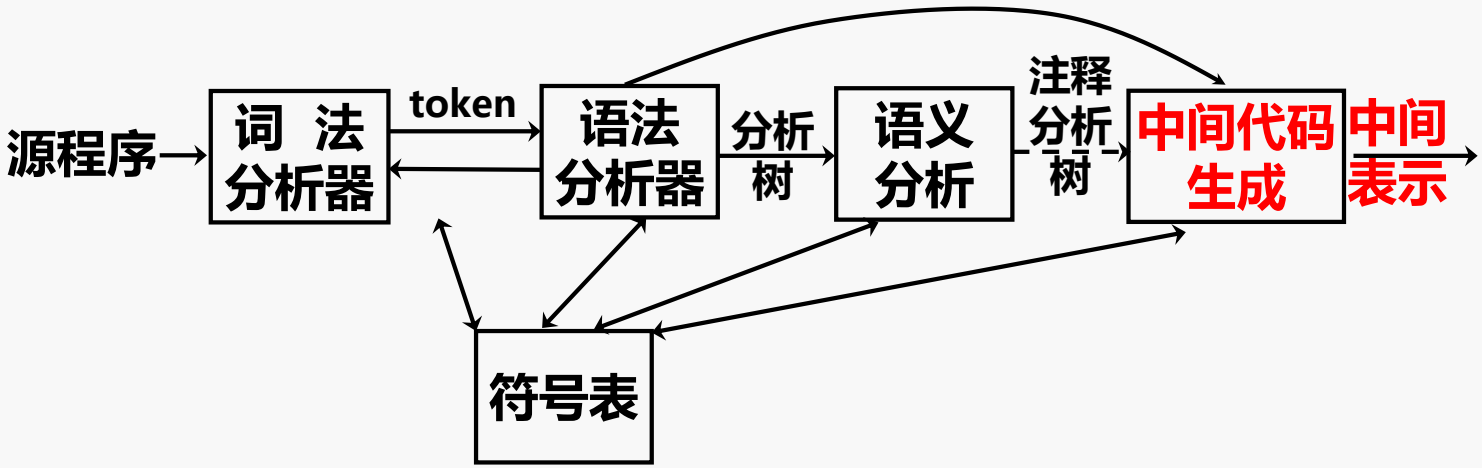
谢谢！

《编译原理和技术》

中间代码生成 III

中科大计算机学院
李 诚
2022-10-19

本节提纲



- 标号回填技术
- 基于标号回填的布尔表达式翻译
- 布尔表达式翻译举例

回填(backpatching)技术

- **问题：**
  - 布尔表达式短路计算翻译中，产生了转移目标不明确的条件或无条件代码；

回填(backpatching)技术

- **问题：**
  - 布尔表达式短路计算翻译中，产生了转移目标不明确的条件或无条件代码；
- **解决方案：**
  - 当生成跳转指令时，暂时不指定目标地址
  - 当有关目标地址确定后，再填回到翻译代码中

回填(backpatching)技术

- **问题：**
  - 布尔表达式短路计算翻译中，产生了转移目标不明确的条件或无条件代码；
- **解决方案：**
  - 当生成跳转指令时，暂时不指定目标地址
  - 当有关目标地址确定后，再填回到翻译代码中
- **具体实现：**
  - 将有相同转移目标的转移代码的**编号串起来形成链，**可以方便回填目标地址。
  - 该list变成了综合属性，可以与LR结合
  - 注：后面的翻译均是与LR结合的语法制导翻译方案

相关符号属性

- **对布尔表达式而言，有两个综合属性：**
  - B.truelist：代码中所有转向真出口的代码指令链；
  - B.falselist：所有转向假出口的代码指令链；
  - 在生成B的代码时，跳转指令goto是不完整的，目标标号尚未填写，用truelist和falselist来管理

实现：数据结构及语义函数

- 将生成的指令放入一个指令数组，指令的标号即为数组下标

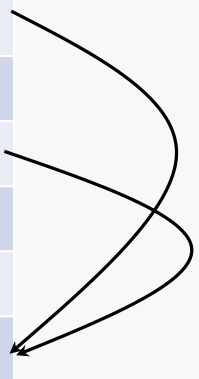
标号	指令数组
100	…
101	goto -
102	…
103	goto -
104	
105	
106	

- 假设100-103号指令都属于布尔表达式B

实现：数据结构及语义函数

- 将生成的指令放入一个指令数组，指令的标号即为数组下标

标号	指令数组
100	…
101	goto -
102	…
103	goto -
104	
105	
106	

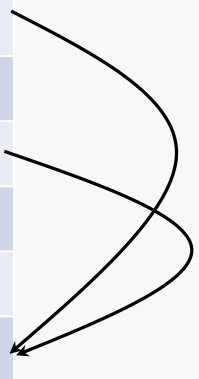


- 假设100-103号指令都属于布尔表达式B
- 101和103号指令都指向B真出口

实现：数据结构及语义函数

- 将生成的指令放入一个指令数组，指令的标号即为数组下标

标号	指令数组
100	…
101	goto -
102	…
103	goto -
104	
105	
106	

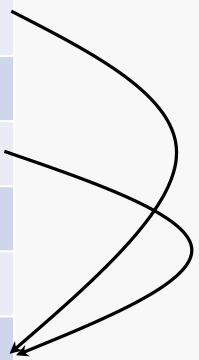


- 假设100-103号指令都属于布尔表达式B
- 101和103号指令都指向B真出口
- B真出口是106，但还未生成
- B.truelist = {101, 103}

实现：数据结构及语义函数

- 将生成的指令放入一个指令数组，指令的标号即为数组下标

标号	指令数组
100	…
101	goto 106
102	…
103	goto 106
104	…
105	…
106	…



- 假设100-103号指令都属于布尔表达式B
- 101和103号指令都指向B真出口
- B真出口是106，但还未生成
- B.truelist = {101, 103}
- 回填时，将101和103补齐

实现：数据结构及语义函数

makelist(i)

- 创建含标号为i的指令的链表
- i不是目标指令，而是源指令，也就是那一些不完整的goto指令

实现：数据结构及语义函数

backpatch(instruction-list, target-label)

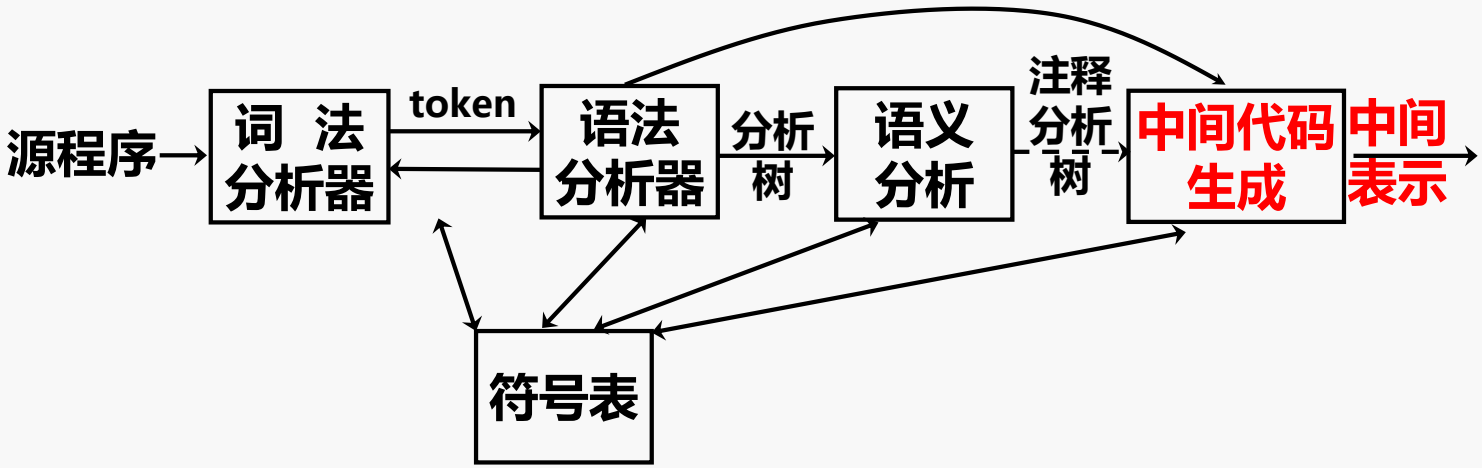
- 将目标地址target-label填回instruction-list中每条指令
- 也就是将goto – 指令中不明确的目标补齐

实现：数据结构及语义函数

merge(instruction-list$_1$, instruction-list$_2$)

- 合并链list$_1$和list$_2$
- 要求list$_1$和list$_2$中每条指令都会跳转到同一条指令

本节提纲



- 标号回填技术
- 基于标号回填的布尔表达式翻译
- 布尔表达式翻译举例

短路计算及回填的翻译方案

$B \rightarrow$ not B_1 { B.truelist = B_1.falselist;

B.falselist = B_1.truelist; }

$B \rightarrow (B_1)$ { B.truelist = B_1.truelist;

B.falselist = B_1.falselist; }

短路计算及回填的翻译方案

B→ true {

B.truelist = makelist(nextinstr);

/*为真时，执行无条件跳转指令，但是目标为空。当目标明确后，回填nextinstr所对应的跳转指令*/

gen(“goto” -); }

B→ false {

B.falselist = makelist(nextinstr);

/*为假时，执行无条件跳转指令，但是目标为空。当目标明确后，回填nextinstr所对应的跳转指令*/

gen(“goto” -);}

短路计算及回填的翻译方案

B→ true {

B.truelist = makelist(nextinstr);

/*为真时，执行无条件跳转指令，但是目标为空。当目标明确后，回填nextinstr所对应的跳转指令*/

gen(“goto” -); }

我们用变量nextinstr保存了紧跟着的下一条指令的序号

B→ false {

B.falselist = makelist(nextinstr);

/*为假时，执行无条件跳转指令，但是目标为空。当目标明确后，回填nextinstr所对应的跳转指令*/

gen(“goto” -);}

短路计算及回填的翻译方案

B→ true {

B.truelist = makelist(nextinstr);

/*为真时，执行无条件跳转指令，但是目标为空。当目标明确后，回填nextinstr所对应的跳转指令*/

gen(“goto” -); }

B→ false {

B.falselist = makelist(nextinstr);

/*为假时，执行无条件跳转指令，但是目标为空。当目标明确后，回填nextinstr所对应的跳转指令*/

gen(“goto” -);}

短路计算及回填的翻译方案

B→ true {

B.truelist = makelist(nextinstr);

/*为真时，执行无条件跳转指令，但是目标为空。当目标明确后，回填nextinstr所对应的跳转指令*/

gen(“goto” -); }

B→ false {

B.falselist = makelist(nextinstr);

/*为假时，执行无条件跳转指令，但是目标为空。当目标明确后，回填nextinstr所对应的跳转指令*/

gen(“goto” -);}

短路计算及回填的翻译方案

$B \rightarrow E_1$ relop E_2 {

B.truelist = makelist(nextinstr);

B.falselist = makelist(nextinstr+1);

/*为真时，执行条件跳转指令，但是目标为空。当目标明确后，回填nextinstr所对应的跳转指令*/

gen(“if” E_1.place relop.op E_2.place “goto” -);

/*为假时，执行无条件跳转指令，但是目标为空。当目标明确后，回填nextinstr+1所对应的跳转指令*/

gen(“goto” -); }

短路计算及回填的翻译方案

$B \rightarrow E_1$ relop E_2 {

B.truelist = makelist(nextinstr);

B.falselist = makelist(nextinstr+1);

/*为真时，执行条件跳转指令，但是目标为空。当目标明确后，回填nextinstr所对应的跳转指令*/

gen(“if” E_1.place relop.op E_2.place “goto” -);

/*为假时，执行无条件跳转指令，但是目标为空。当目标明确后，回填nextinstr+1所对应的跳转指令*/

gen(“goto” -); }

短路计算及回填的翻译方案

$B \rightarrow E_1$ relop E_2 {

B.truelist = makelist(nextinstr);

B.falselist = makelist(nextinstr+1);

/*为真时，执行条件跳转指令，但是目标为空。当目标明确后，回填nextinstr所对应的跳转指令*/

gen(“if” E_1.place relop.op E_2.place “goto” -);

/*为假时，执行无条件跳转指令，但是目标为空。当目标明确后，回填nextinstr+1所对应的跳转指令*/

gen(“goto” -); }

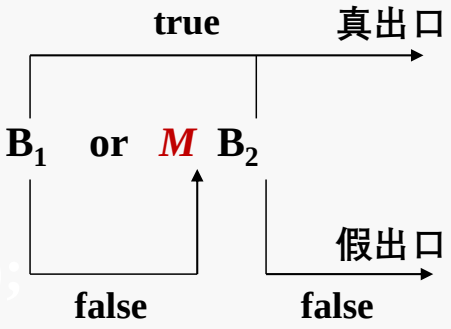
短路计算及回填的翻译方案

$B \rightarrow B_1$ or M B_2

{ backpatch(B_1.falselist, M.instr);

B.truelist = merge(B_1.truelist, B_2.truelist);

B.falselist = B_2.falselist; }



/*获取下一三地址代码（语句）的编号（作为转移目标来回填），在自底向上的语法分析中传递信息；
在分析B_2之前做，因此可以保存B_2开始的第一条指令的地址*/

$M \rightarrow \varepsilon$ { M.instr = nextinstr}

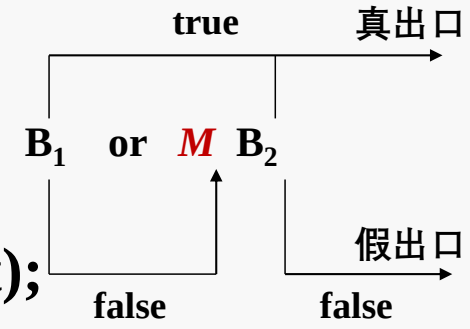
短路计算及回填的翻译方案

$B \rightarrow B_1$ or M B_2

{ backpatch(B_1.falselist, M.instr);

B.truelist = merge(B_1.truelist, B_2.truelist);

B.falselist = B_2.falselist; }



/*获取下一三地址代码（语句）的编号（作为转移目标来回填），
在自底向上的语法分析中传递信息；
在分析B_2之前做，因此可以保存B_2开始的第一条指令的地址*/

$M \rightarrow \varepsilon$ { M.instr = nextinstr}

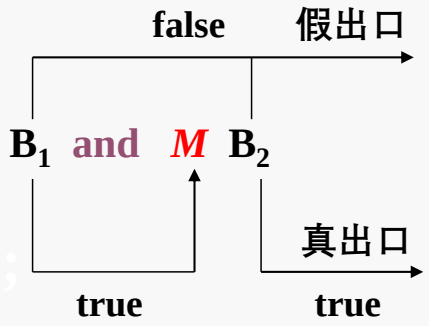
短路计算及回填的翻译方案

$B \rightarrow B_1$ and M B_2

{ backpatch(B_1.truelist, M.instr);

B.falselist = merge(B_1.falselist,B_2.falselist);

B.truelist = B_2.truelist; }



M→ε { M.instr = nextinstr}//在分析B_2之前做，因此可以保存B_2开始的第一条指令的地址

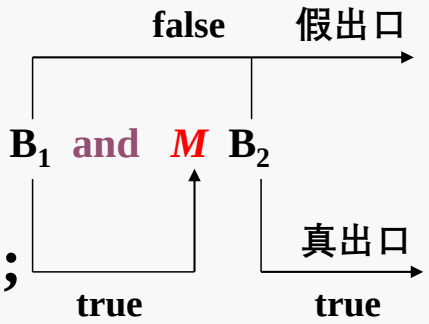
短路计算及回填的翻译方案

$B \rightarrow B_1$ and M B_2

{ backpatch(B_1.truelist, M.instr);

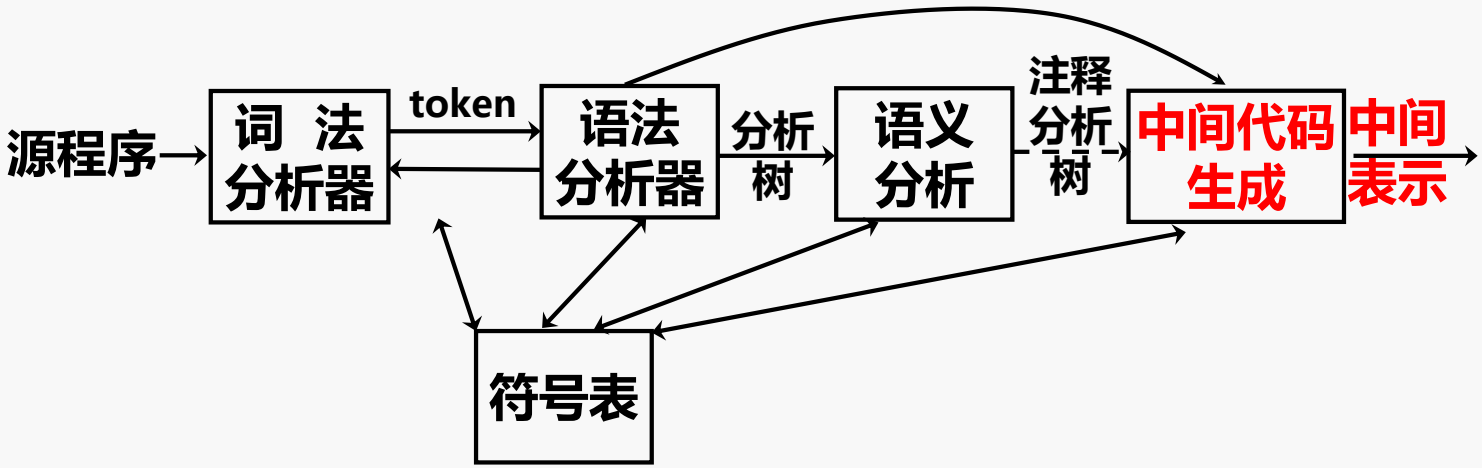
B.falselist = merge(B_1.falselist,B_2.falselist);

B.truelist = B_2.truelist; }



M→ε { M.instr = nextinstr}//在分析B_2之前做，因此可以保存B_2开始的第一条指令的地址

本节提纲

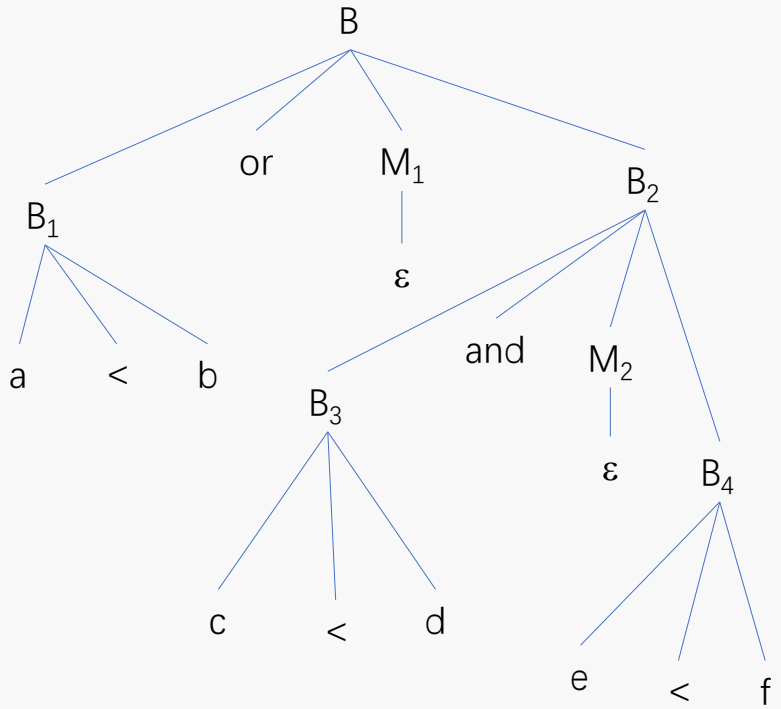


- 标号回填技术
- 基于标号回填的布尔表达式翻译
- **布尔表达式翻译举例**

布尔表达式的翻译

a<b or c<d and e<f

假设nextinstr = 100

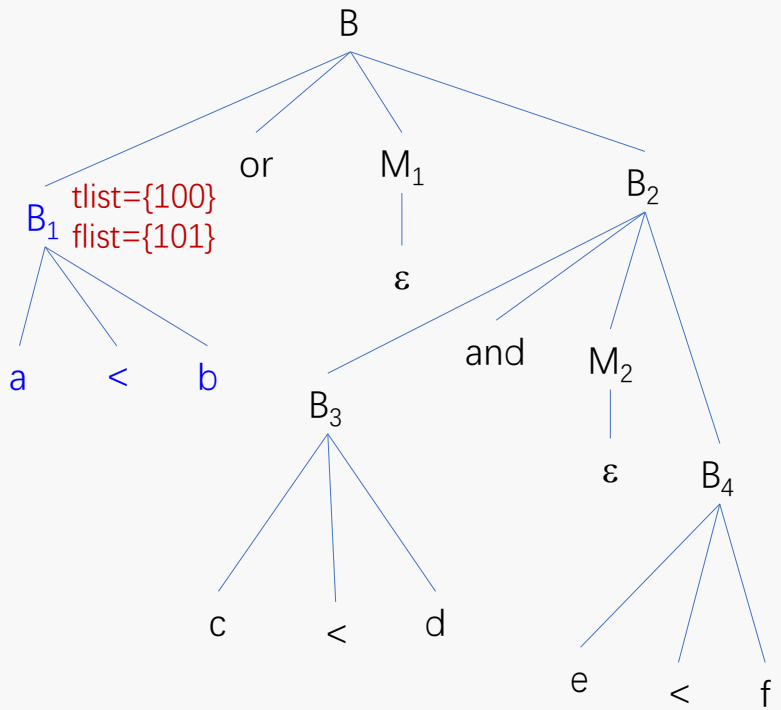


布尔表达式的翻译

a<b or c<d and e<f

假设nextinstr = 100

(100) if a<b goto -
(101) goto -

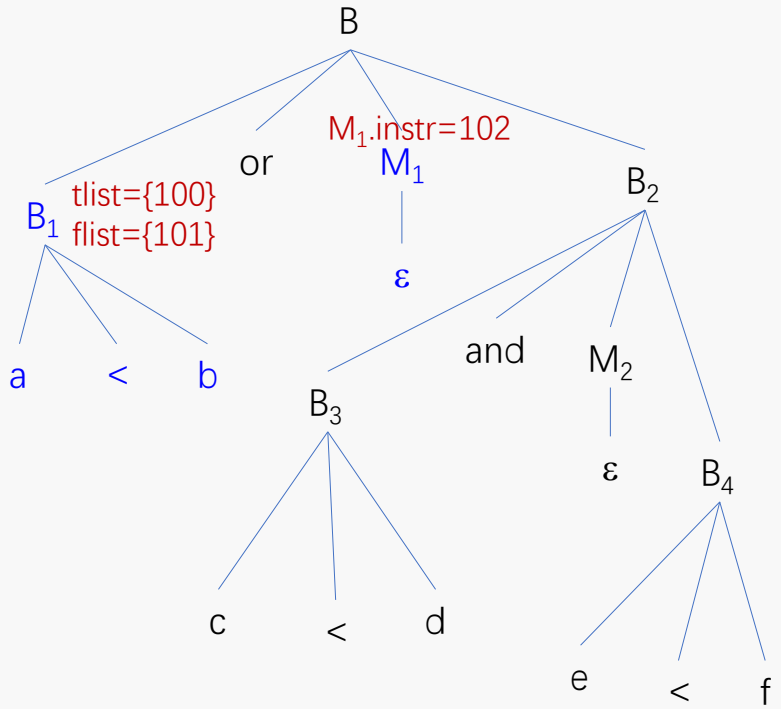


布尔表达式的翻译

a<b or c<d and e<f

假设nextinstr = 100

(100) if a<b goto -
(101) goto -

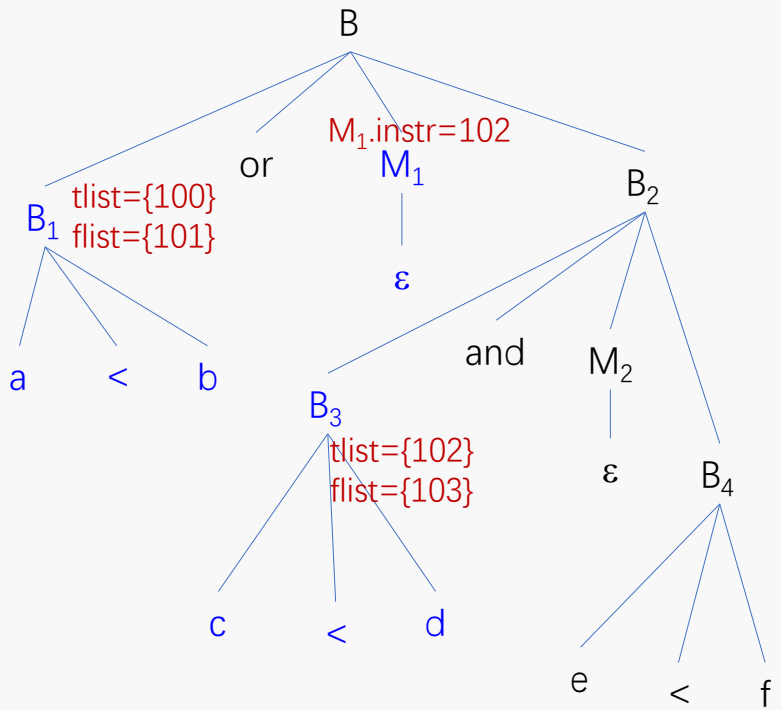


布尔表达式的翻译

a<b or c<d and e<f

假设nextinstr = 100

(100) if a<b goto -
(101) goto -
(102) if c<d goto -
(103) goto -

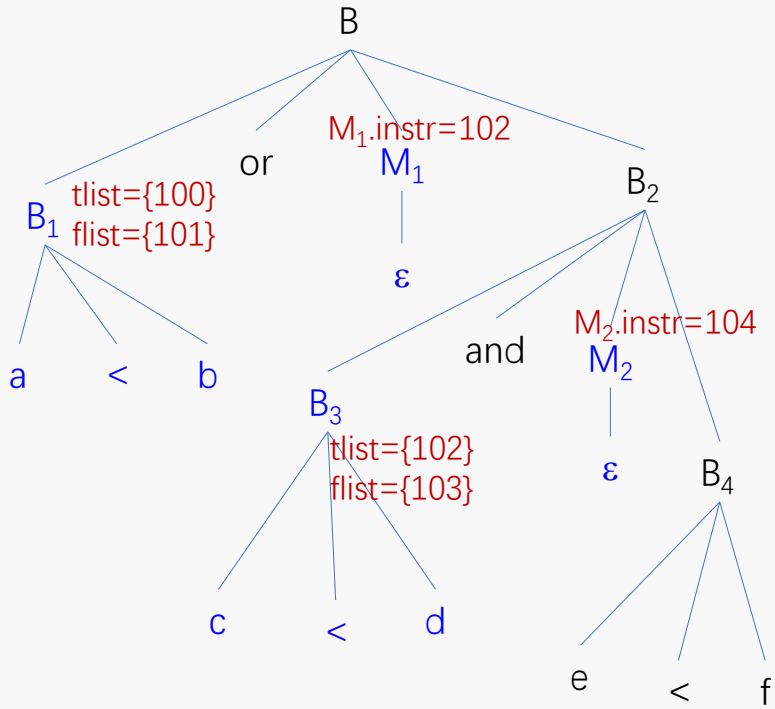


布尔表达式的翻译

a<b or c<d and e<f

假设nextinstr = 100

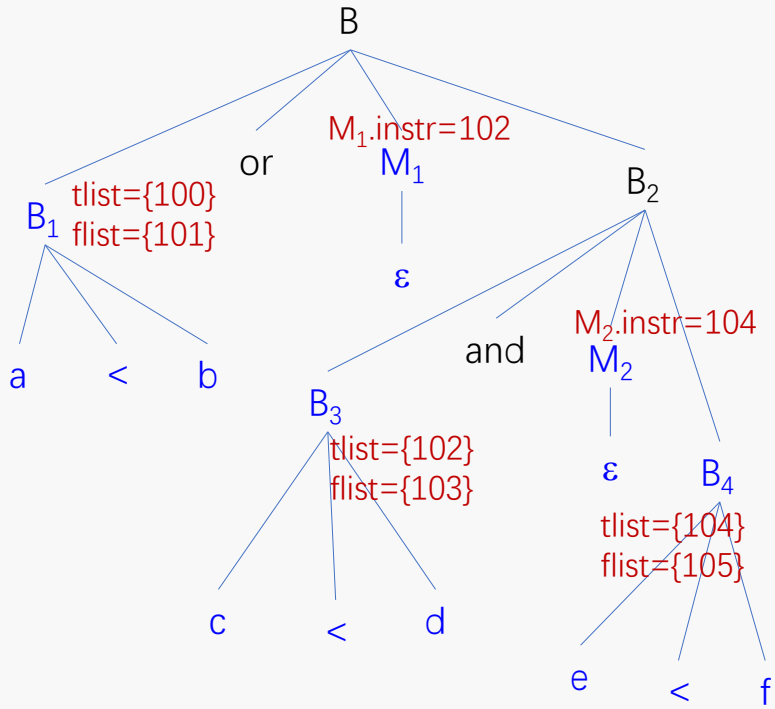
(100) if a<b goto -
(101) goto -
(102) if c<d goto -
(103) goto -



布尔表达式的翻译

a<b or c<d and e<f

假设nextinstr = 100

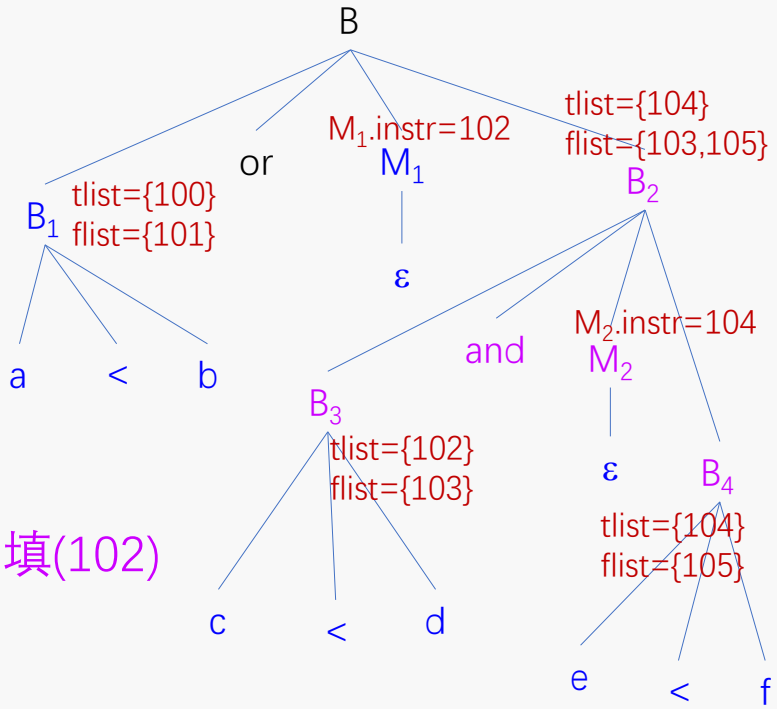


(100) if a<b goto -
(101) goto -
(102) if c<d goto -
(103) goto -
(104) if e<f goto -
(105) goto -

布尔表达式的翻译

a<b or c<d and e<f

假设nextinstr = 100

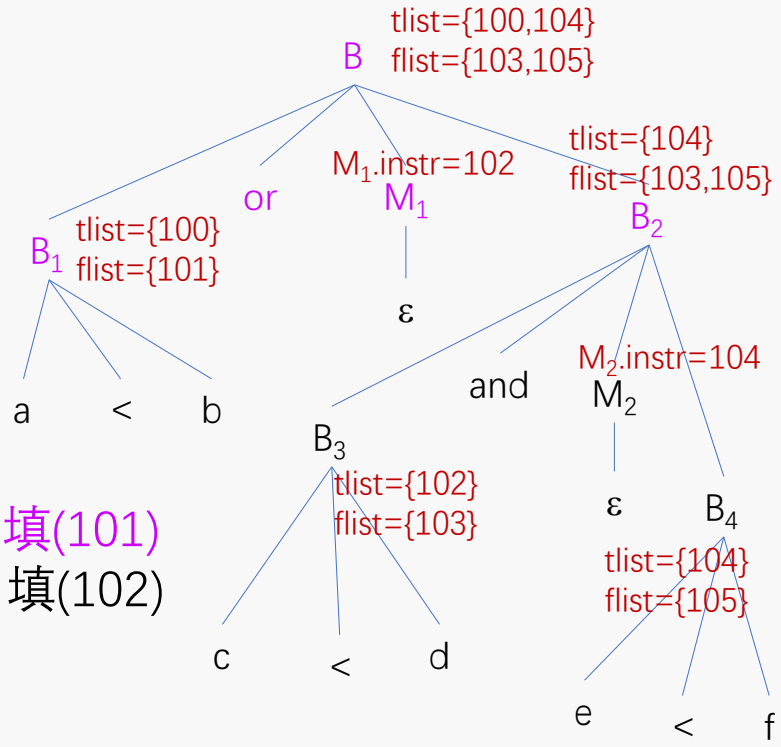


(100) if a<b goto -
(101) goto -
(102) if c<d goto 104//用104回填(102)
(103) goto -
(104) if e<f goto -
(105) goto -

布尔表达式的翻译

a<b or c<d and e<f

假设nextinstr = 100



(100) if a<b goto -
(101) goto 102 //用102回填(101)
(102) if c<d goto 104//用104回填(102)
(103) goto -
(104) if e<f goto -
(105) goto -

其他部分的回填要依赖与其他语句的翻译

作业

考虑布尔表达式

a>b or true and not c < f

- 参考本ppt中布尔表达式短路计算、标号回填等翻译技术，生成对应的三地址代码。
- 假设nextinstr = 200
- 除了三地址代码外，画出LR分析方法对应的注释分析树（如slide 35），标注出属性和属性值。
- 结合LR分析方法指出回填的具体细节
  - 在使用哪一个产生式归约时候进行的回填
  - 用哪一个标号，回填了哪一个不完整的goto指令

《编译原理和技术》

中间代码生成 III

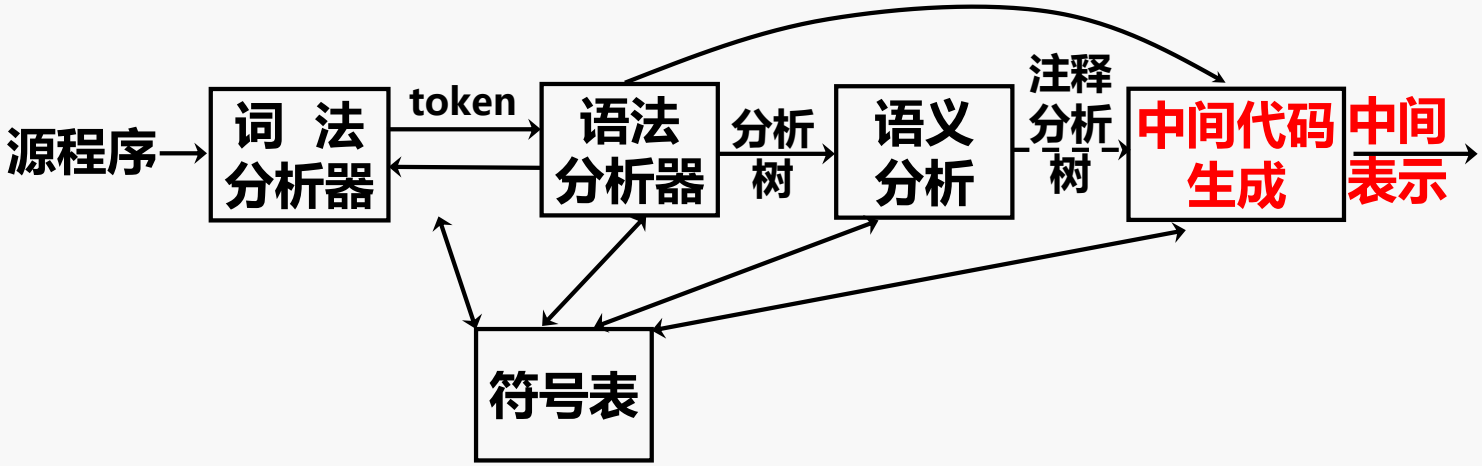
谢谢！

《编译原理和技术》

中间代码生成 IV

中科大计算机学院
李 诚
2022-10-19/24

本节提纲



- 基于标号回填的其他控制流语句翻译
- 控制流语句翻译举例

相关符号属性

- **对布尔表达式而言，有两个综合属性：**
  - B.truelist：代码中所有转向真出口的代码指令链；
  - B.falselist：所有转向假出口的代码指令链；
  - 在生成B的代码时，跳转指令goto是不完整的，目标标号尚未填写，用truelist和falselist来管理
- **对一般语句而言，有一个综合属性：**
  - S.nextlist：代码中所有跳转到紧跟S的代码之后的指令

例如：$S \rightarrow \{L\}$ //程序块

$S \rightarrow A$ //赋值语句

$S \rightarrow \text{if } B \text{ then } S_1 \text{ else } S_2$

控制流语句文法汇总

- **用S和L分别表示一条语句和语句列表**

  $S \rightarrow$ if B then S_1 // (B) 的括号此处省略

  $S \rightarrow$ if B then S_1 else S_2

  $S \rightarrow$ while B do S_1

  $S \rightarrow$ A //赋值语句

  $S \rightarrow$ {L} //大括号的作用是把内部的多个语句绑在一起，当成一个语句。

  $L \rightarrow L;S \mid S$ //语句列表，L和S之间用;分割

控制流语句文法-例

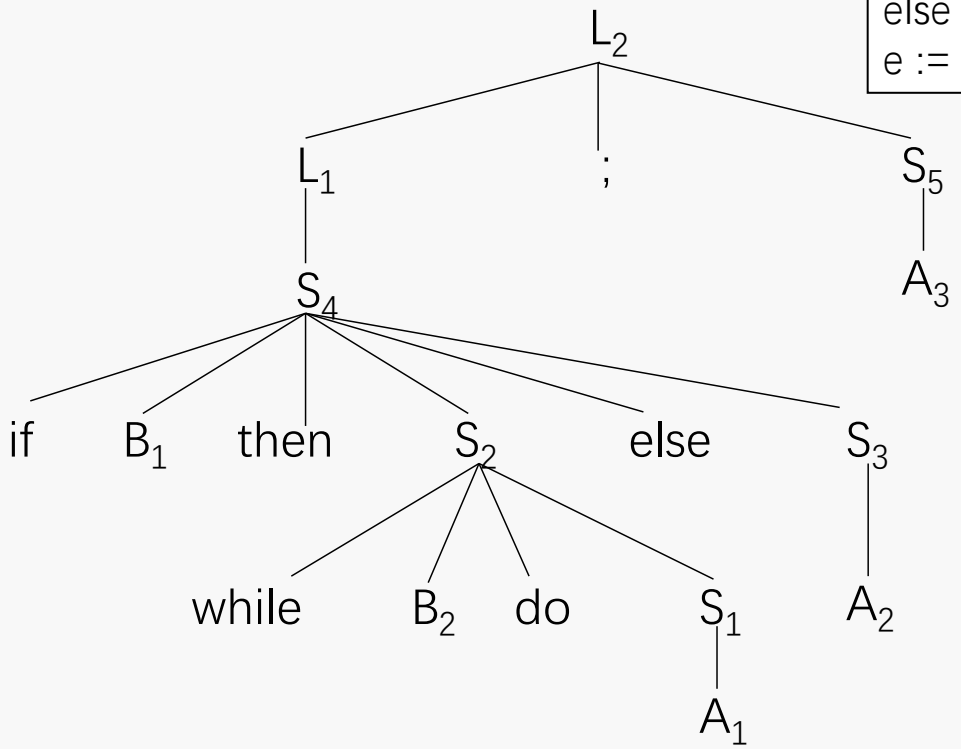
考虑以下语句序列：

```
if ( a<b or c<d and e<f )  then
    while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```

控制流语句文法-例

- 分析树

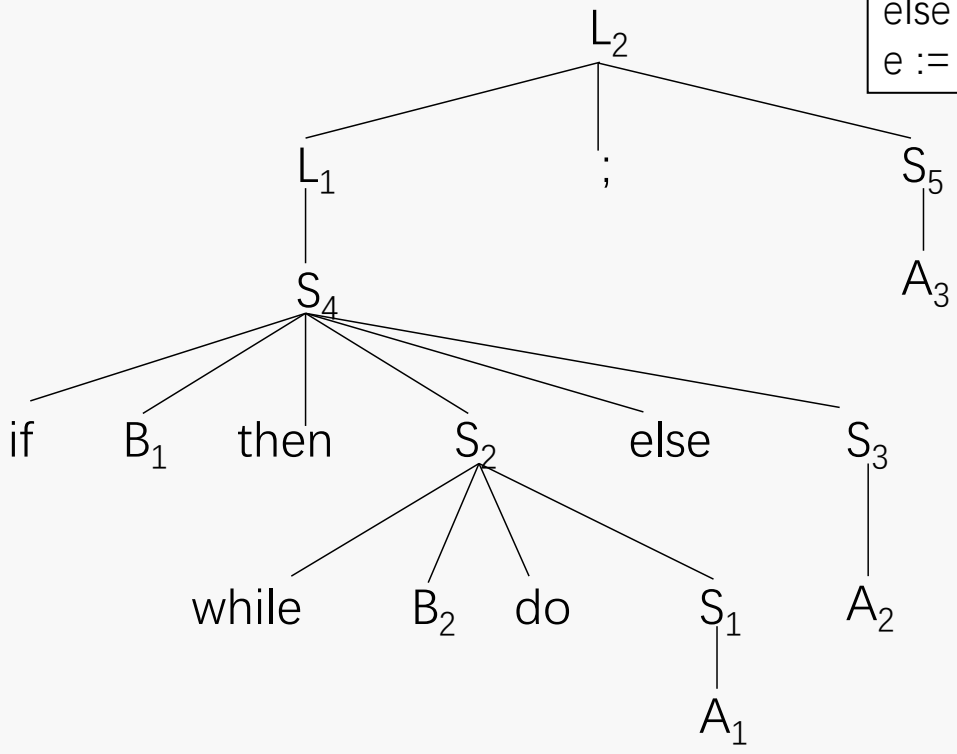
```
if ( a<b or c<d and e<f )  then
    while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



控制流语句文法-例

分析树

```
if ( a<b or c<d and e<f ) then
    while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



当前的任务：

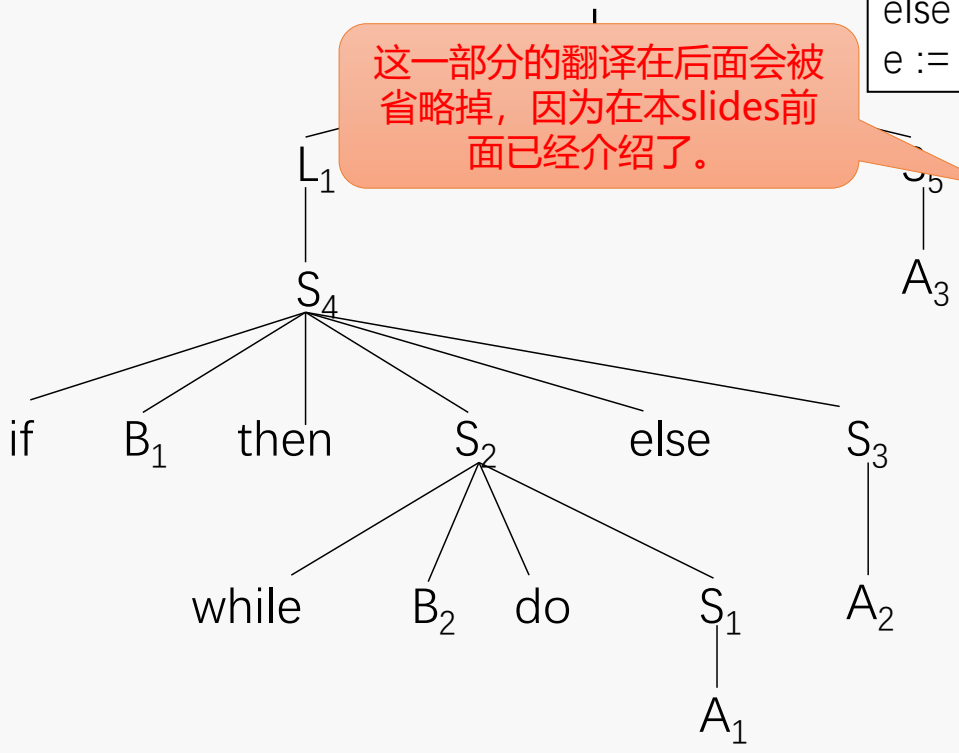
- 为每一条语句或表达式生成对应的三地址代码
- 为B_1和B_2创建分别指向真出口和假出口的truelist和 falselist
- 为五个S语句和两个L语句列表创建指向下一指令的 nextlist
- 在特定的归约环节，将跳转目标指令标号回填对应的列表中的待填指令
- 按需生成无条件转移指令

控制流语句文法-例

分析树

```
if ( a<b or c<d and e<f ) then
    while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```

这一部分的翻译在后面会被省略掉，因为在本slides前面已经介绍了。

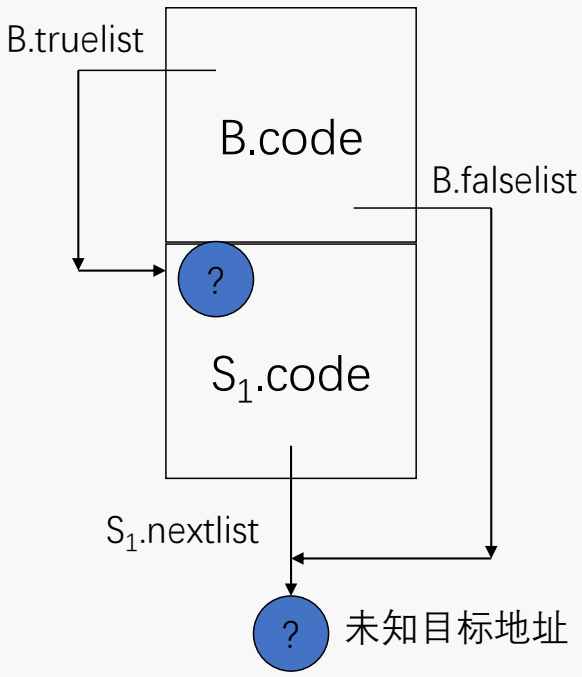


当前的任务：

- ~~为每一条语句或表达式生成对应的三地址代码~~
- 为B_1和B_2创建分别指向真出口和假出口的truelist和 falselist
- 为五个S语句和两个L语句列表创建指向下一指令的 nextlist
- 在特定的归约环节，将跳转目标指令标号回填对应的列表中的待填指令
- 按需生成无条件转移指令

条件语句的翻译方案 (1)

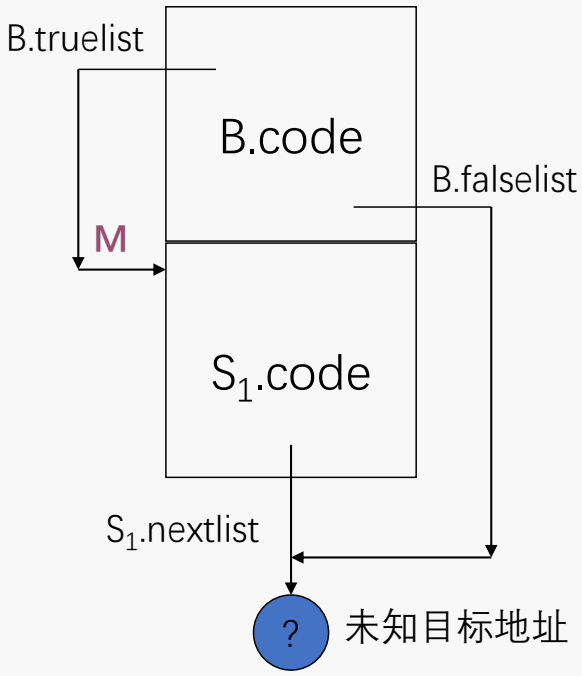
if B then S_1 的代码结构



- **四个跳转目标未知**
  - B的值为真跳转到S_1的开始
  - B的值为假/S_1结束/S结束应该跳转到同一个指令

条件语句的翻译方案 (1)

if B then S_1 的代码结构



- **四个跳转目标未知**
  - B的值为真跳转到S_1的开始
  - B的值为假/S_1结束/S结束应该跳转到同一个指令
- **解决方案**
  - 引入M标记，记录B.code之后的下一条新的指令标号，方便回填B.truelist
  - 将B.falselist、S_1.nextlist合并赋给S.nextlist

条件语句的翻译方案 (1)

$S \rightarrow$ if B then M S_1

{

backpatch(B.truelist, M.instr);

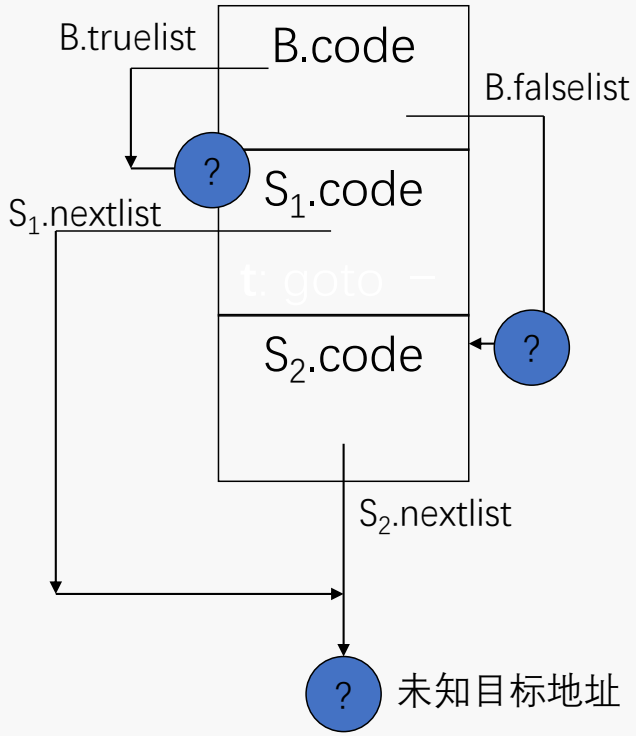
S.nextlist = merge(B.falselist, S_1.nextlist)

}

$M \rightarrow \varepsilon$ { M.instr = nextinstr}

条件语句的翻译方案 (2)

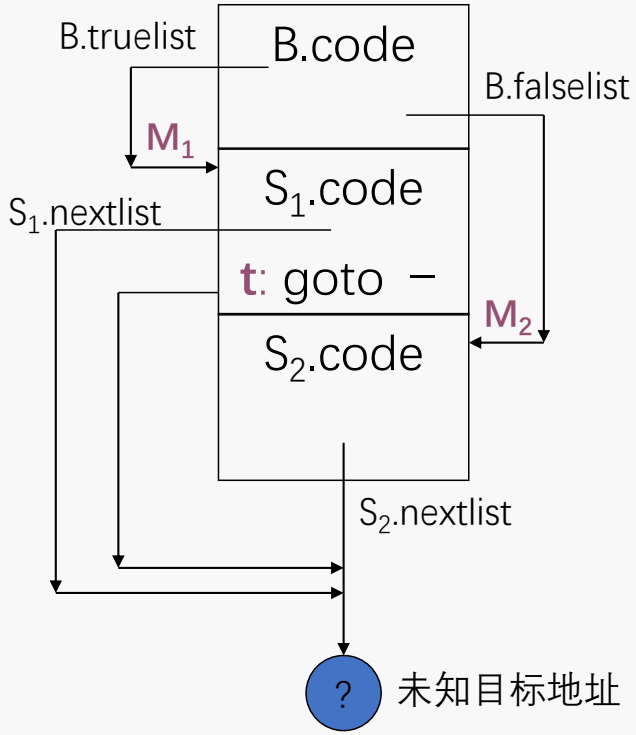
if B then S_1 else S2的结构



- **五个跳转目标未知**
  - B的值为真跳转到S_1的开始
  - B的值为假跳转到S_2的开始
  - S_1/ S_2 /S结束应该跳转到同一个指令
- **S_1结束要越过S_2**

条件语句的翻译方案 (2)

if B then S_1 else S2的结构



- **五个跳转目标未知**
  - B的值为真跳转到S_1的开始
  - B的值为假跳转到S_2的开始
  - S_1/ S_2 /S结束应该跳转到同一个指令
- **S_1结束要越过S_2**
- **解决方案**
  - 引入M_1和M_2标记
  - 引入N标记，在S_1后插入无条件跳转指令
  - 将S_1.nextlist、{t}和S_2.nextlist合并赋给S.nextlist

条件语句的翻译方案 (2)

$S \rightarrow$ if B then M_1 S_1 *N* else M_2 S_2

{ backpatch(B.truelist, M_1.instr);
backpatch(B.falselist, M_2.instr);
temp = merge(S_1.nextlist, N.nextlist);
S.nextlist = merge(temp, S_2.nextlist) ;
}

$N \rightarrow \varepsilon$ { N.nextlist = makelist(nextinstr); //标号t
gen(“goto” -);
}

条件语句的翻译方案 (2)

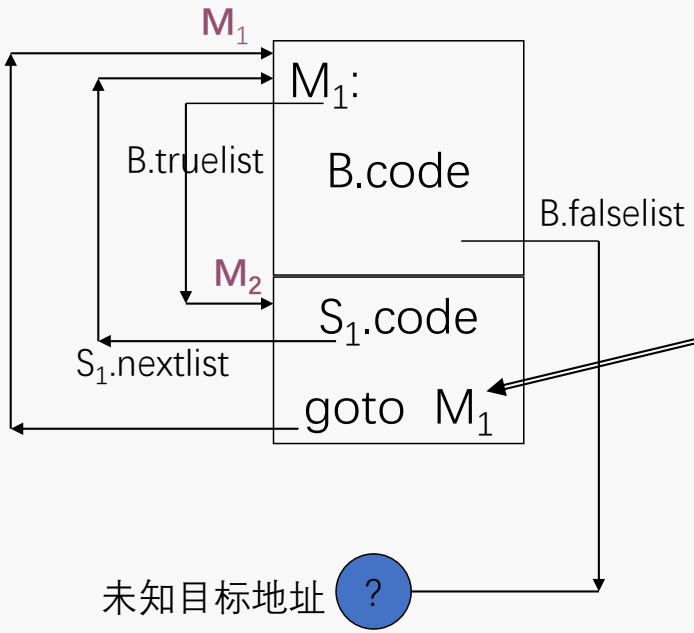
$S \rightarrow$ if B then M_1 S_1 *N* else M_2 S_2

{ backpatch(B.truelist, M_1.instr);
backpatch(B.falselist, M_2.instr);
temp = merge(S_1.nextlist, N.nextlist);
S.nextlist = merge(temp, S_2.nextlist) ;
}

$N \rightarrow \varepsilon$ { N.nextlist = makelist(nextinstr); //标号t
gen(“goto” -);
}

循环语句的翻译方案

while B do S_1 的代码结构

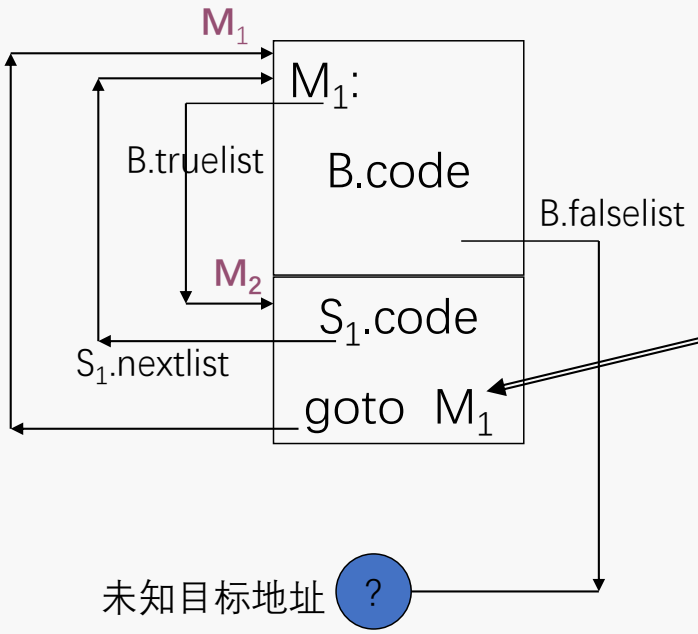


产生无条件跳转指令

goto M_1（跳转至循环条件测试代码开始处）

循环语句的翻译方案

while B do S_1 的代码结构



与 if B then M_1 S_1 N else M_2 S_2不同，此处不用引入N来生成一条无条件跳转指令，因为，S_1分析后已经看到句柄，因此，该指令的生成可以放到归约里。

产生无条件跳转指令

goto M_1（跳转至循环条件测试代码开始处）

循环语句的翻译方案

S→ while M_1 B do M_2 S_1

{ backpatch(B.truelist, M_2.instr);

backpatch(S_1.nextlist, M_1.instr);

S.nextlist = B.falselist;

gen(“goto” M_1.instr);//已知

}

M→ε { M.instr = nextinstr}

简单语句的翻译方案

$S \rightarrow A$ {S.nextlist = {};}

$S \rightarrow \{L\}$ {S.nextlist = L.nextlist}

$L \rightarrow S$ {L.nextlist = S.nextlist}

语句列表的翻译方案

$L \rightarrow L_1;S$

当分析完L_1时，由于不知道S是否会出现，因此需要引入M标记符，记录S的第一条指令；

当S分析完后，进行L的归约，此时，将M记录的指令标号回填L_1.nextlist, 并将S的nextlist赋给L的nextlist

语句列表的翻译方案

$L \rightarrow L_1;S$

当分析完L_1时，由于不知道S是否会出现，因此需要引入M标记符，记录S的第一条指令；

当S分析完后，进行L的归约，此时，将M记录的指令标号回填L_1.nextlist, 并将S的nextlist赋给L的nextlist

$L \rightarrow L_1;M\ S$ {

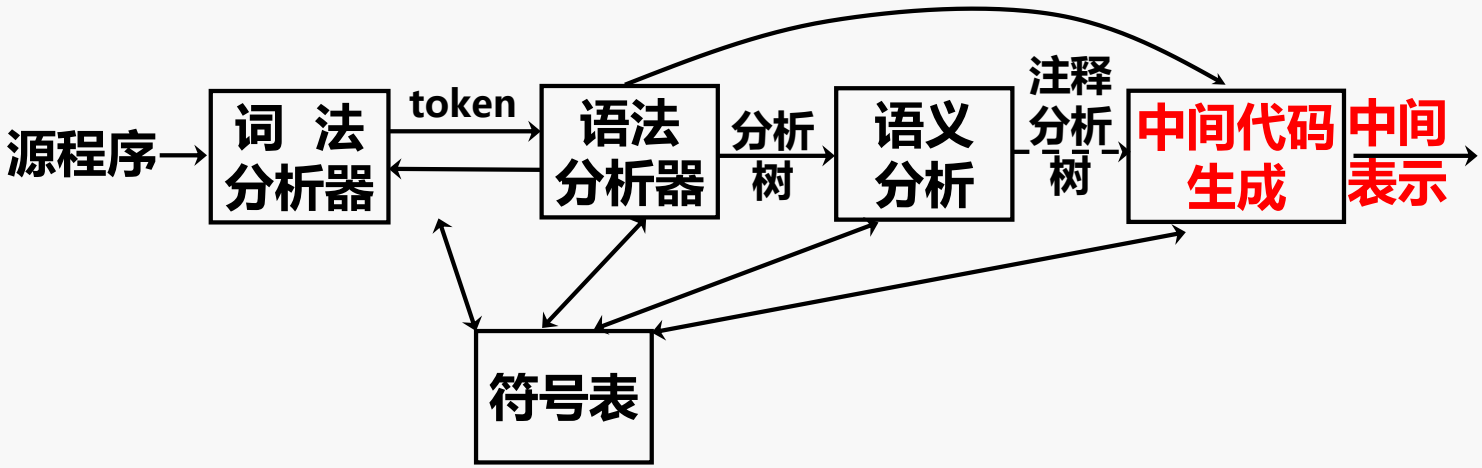
backpatch(L_1.nextlist, M.instr);

L.nextlist = S.nextlist;

}

$M \rightarrow \varepsilon$ { M.instr = nextinstr}

本节提纲



- 基于标号回填的其他控制流语句翻译
- 控制流语句翻译举例

控制流语句的翻译-例

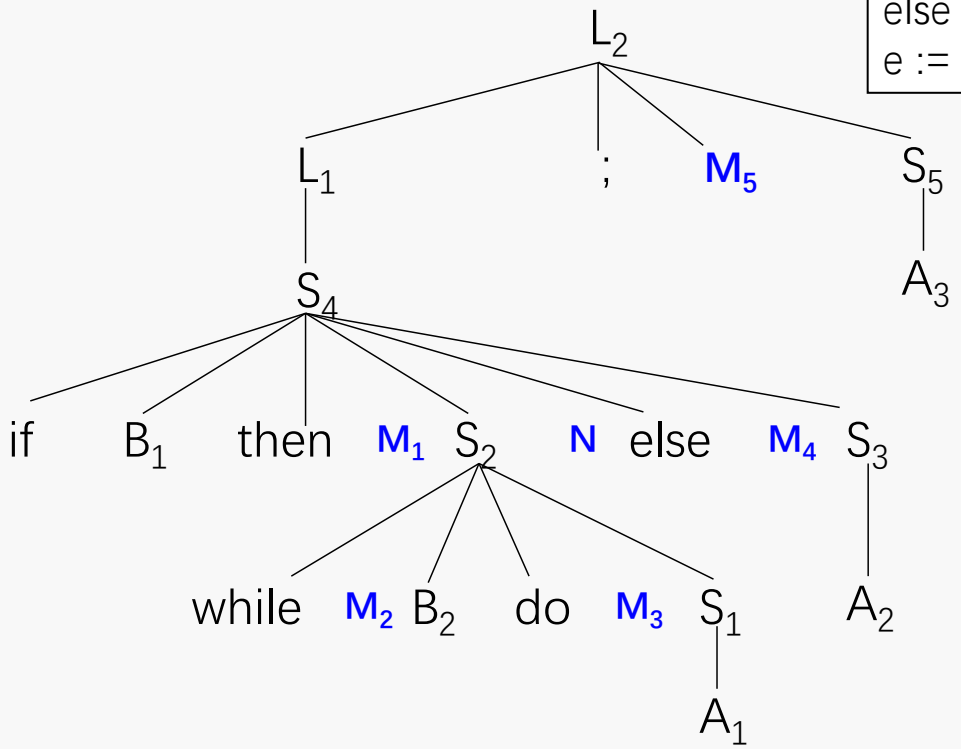
翻译以下语句序列：

```
if ( a<b or c<d and e<f )  then
    while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```

控制流语句的翻译-例

- 分析树

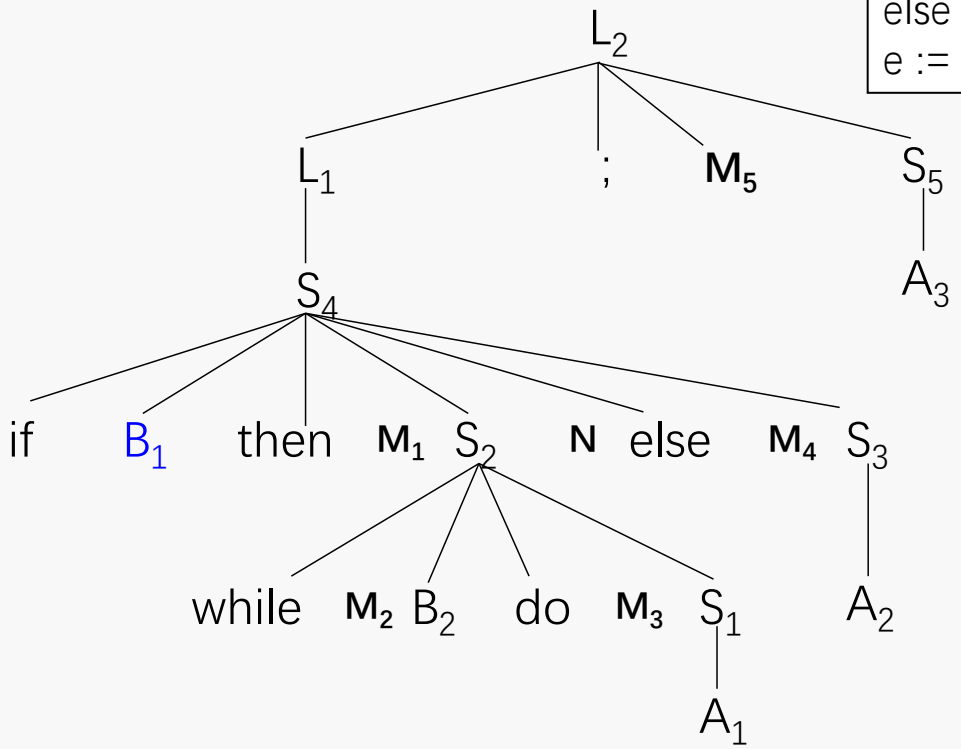
```
if ( a<b or c<d and e<f ) then
    while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



控制流语句的翻译-例

分析树

```
if ( a<b or c<d and e<f ) then
    while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填

控制流语句的翻译-例

一、翻译 B_1：(a<b or c<d and e<f)

(100) if a<b goto -

(101) goto 102

(102) if c<d goto 104

(103) goto -

(104) if e<f goto -

(105) goto -

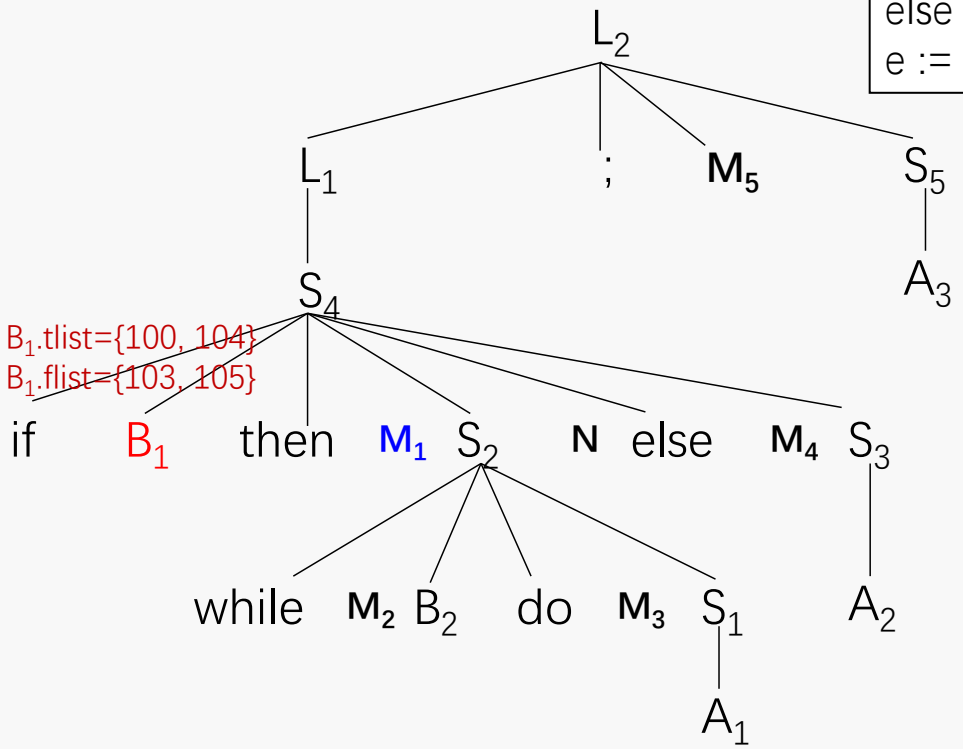
truelist: { 100, 104 } falselist: { 103, 105 }

在Lecture 15的ppt中已经进行了翻译，此处直接使用结果

控制流语句的翻译-例

分析树

```
if ( a<b or c<d and e<f ) then
    while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填

控制流语句的翻译-例

二、翻译 M_1： $M_1 \rightarrow \varepsilon$

记录下一个指令标号106，当if-then-else归约时，用106回填B_1的truelist { 100, 104 } 108) c := c + 1 // $S_1 \rightarrow A_1$ S_1.nextlist={}

(109) goto 106 // 转至循环入口(106)

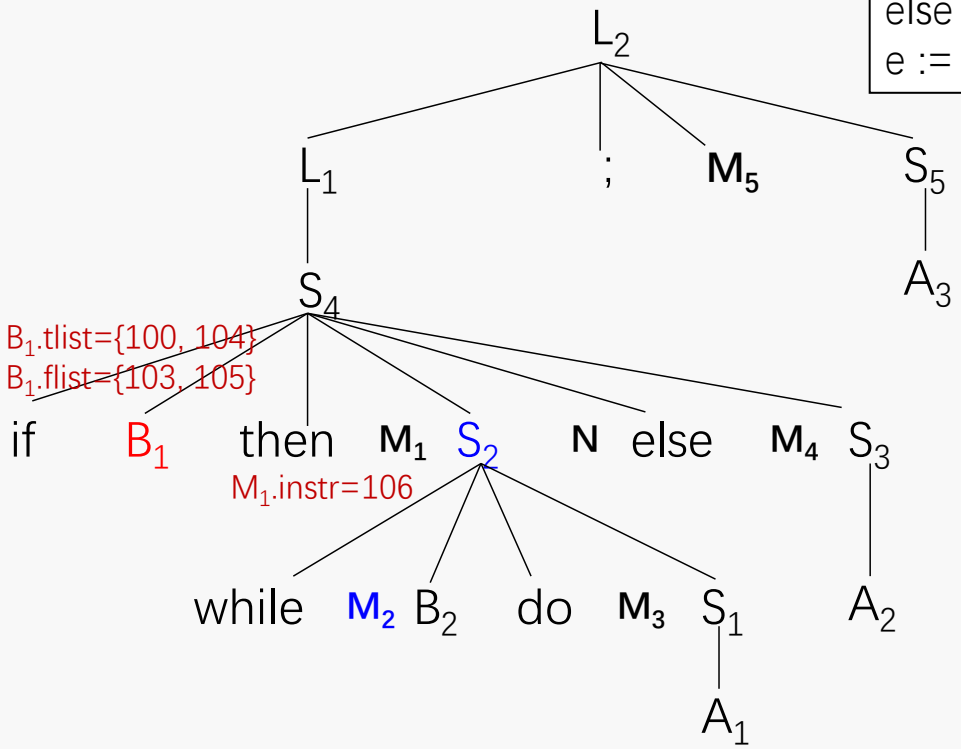
S_2.nextlist: { 107 } //转至循环外部

(110) goto 112 // 由$N \rightarrow \varepsilon$生成={}

控制流语句的翻译-例

分析树

```
if ( a<b or c<d and e<f ) then
    while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填

控制流语句的翻译-例

三、翻译 M_2： $M_2 \rightarrow \varepsilon$

记录下一个指令标号106，当while(B_2) S_1归约时，用106回填S_1的nextlist，并生成无条件跳转指令goto M_2.instr

(108) c := c + 1 //

$S_1 \rightarrow A_1$ S_1.nextlist={}

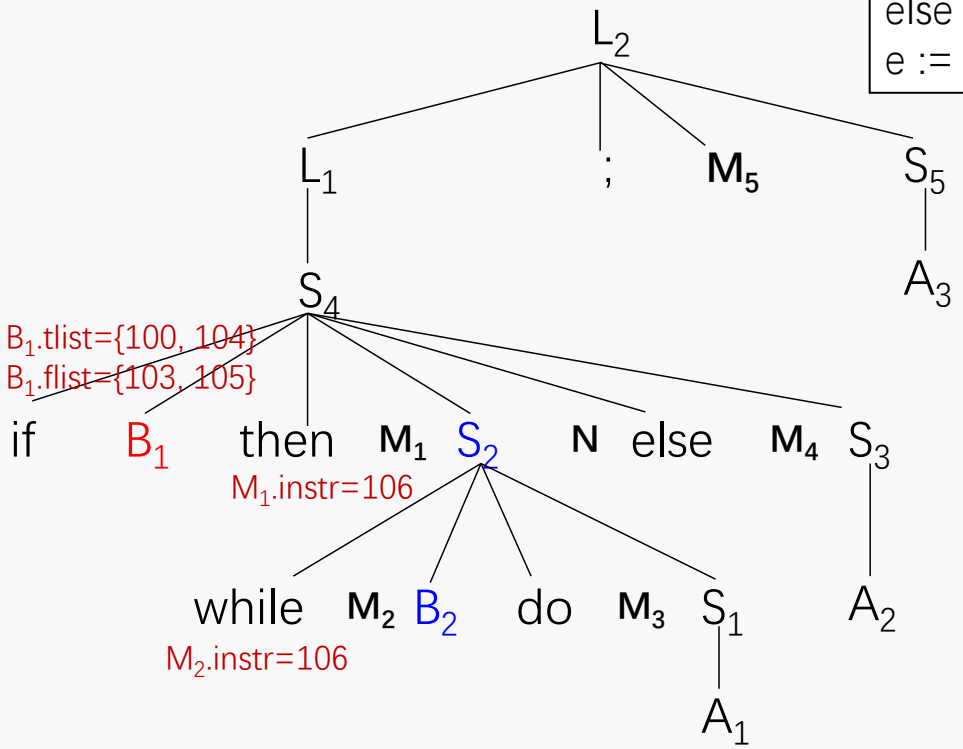
(109) goto 106 // 转至循环入口(106)

S_2.nextlist: { 107 } //转至循环外部={}

控制流语句的翻译-例

分析树

```
if ( a<b or c<d and e<f ) then
    while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填

控制流语句的翻译-例

四、翻译S_2中B_2：while B_2 do S_1

(106) if a>c goto -

(107) goto -

(truelist: { 106 } falselist: { 107 }//待回填

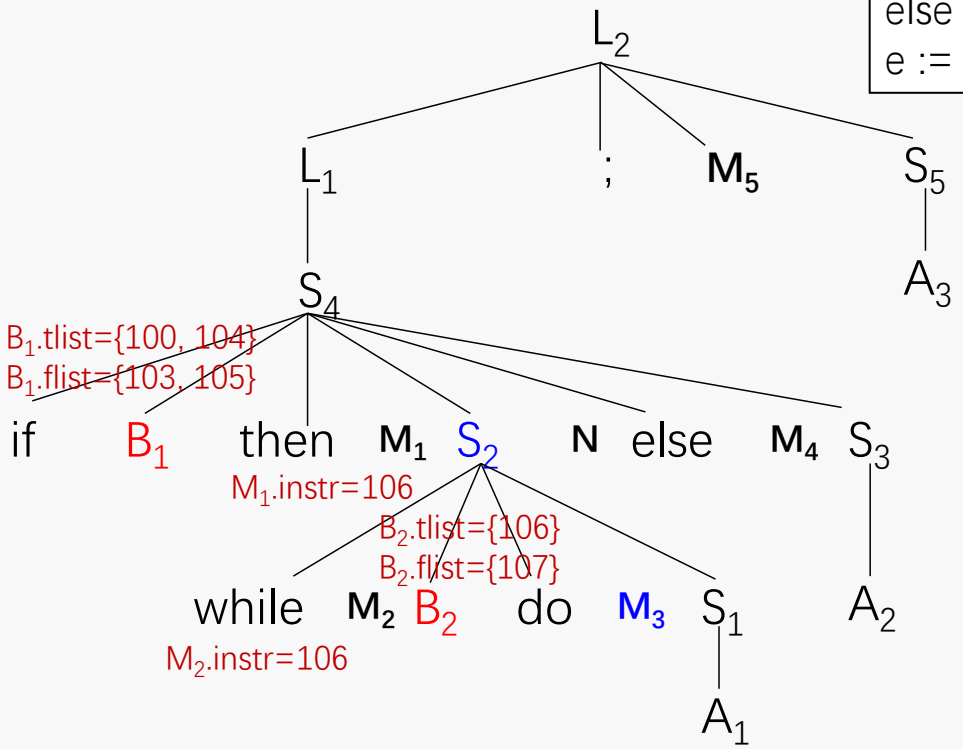
当while(B_2) S_1归约时，用106回填S_1的nextlist，并生成无条件跳转指令goto M_2.instr(110) goto 112　// 由N→ε生成

(111) d := d + 1　// S_3→A_2　S_3.nextlist={}

控制流语句的翻译-例

分析树

```
if ( a<b or c<d and e<f ) then
    while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填

控制流语句的翻译-例

五、翻译 M_3：　$M_3 \rightarrow \varepsilon$

记录下一个指令标号108，当while(B_2) S_1归约时，用108回填B_2的truelist

(108) c := c + 1　// $S_1 \rightarrow A_1$　S_1.nextlist={}

(109) goto 106　// 转至循环入口(106)

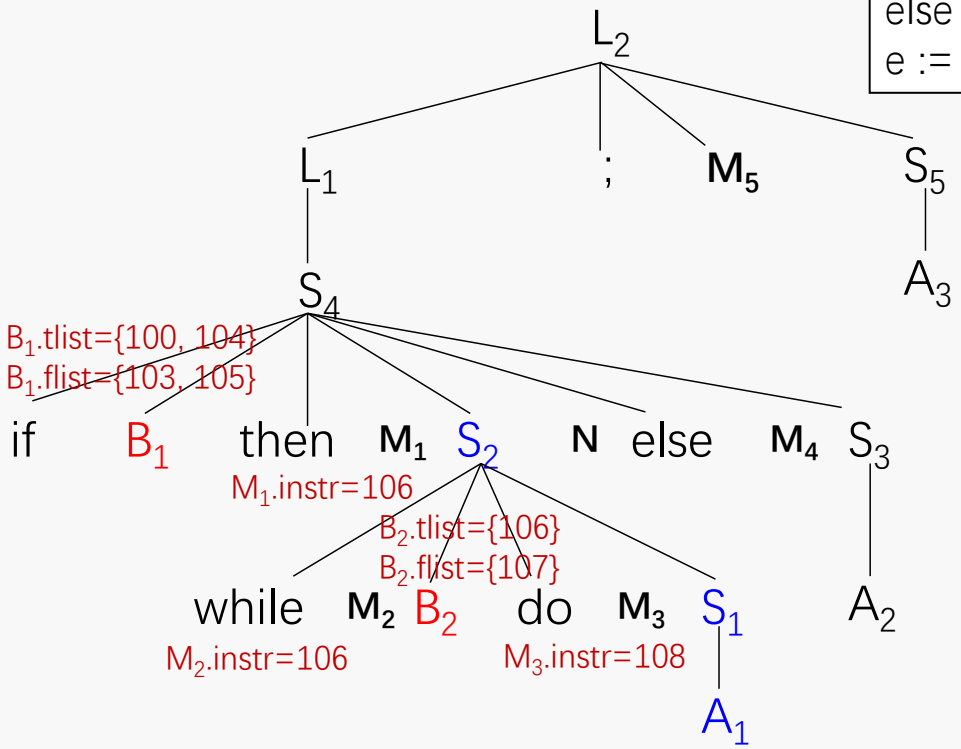
S_2.nextlist: { 107 } //转至循环外部

(110) goto 112　　// 由$N \rightarrow \varepsilon$生成={}

控制流语句的翻译-例

分析树

```
if ( a<b or c<d and e<f ) then
    while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填

控制流语句的翻译-例

六、翻译 S_1： while B_2 do S_1

(106) if a>c goto -

(107) goto -

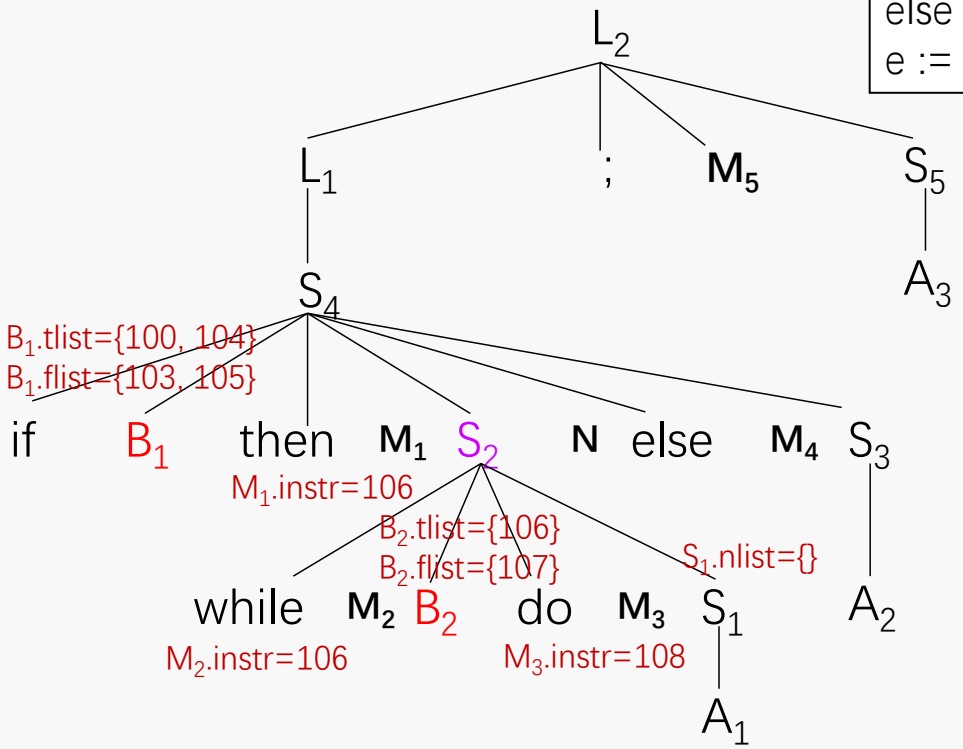
(108) c := c + 1 // $S_1 \rightarrow A_1$ S_1.nextlist={}

(109) goto 106 // 转至循环入口(106)

控制流语句的翻译-例

分析树

```
if ( a<b or c<d and e<f ) then
    while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填
紫色表示即将归约

控制流语句的翻译-例

七、归约 S_2： while B_2 do S_1

(106) if a>c goto -

(107) goto -

(108) c := c + 1 // $S_1 \rightarrow A_1$ S_1.nextlist={}

(109) goto 106 // 转至循环入口(106)

S_2.nextlist: { 107 } //转至循环外部

此处需要用106回填S_1.nextlist，但该list为空

控制流语句的翻译-例

七、归约 S_2： while B_2 do S_1

(106) if a>c goto 108 //用108回填106

(107) goto -

(108) c := c + 1 // $S_1 \rightarrow A_1$ S_1.nextlist={}

(109) goto 106 // 转至循环入口(106)

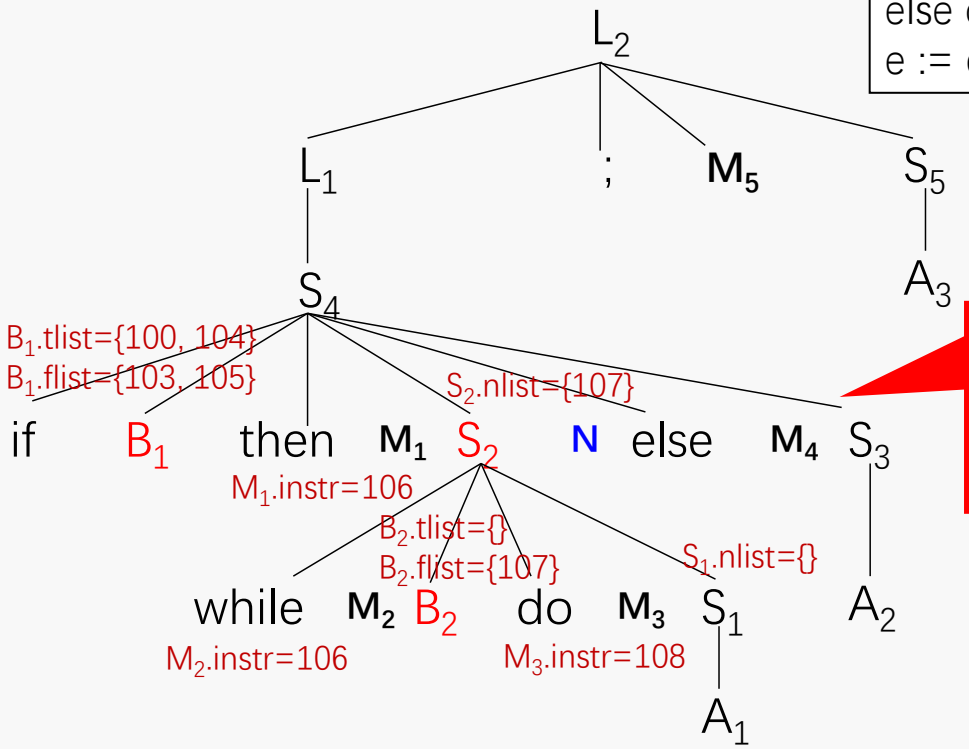
S_2.nextlist: { 107 } //转至循环外部

此处需要用106回填S_1.nextlist，但该list为空

控制流语句的翻译-例

分析树

```
if ( a<b or c<d and e<f ) then
    while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



虽然while语句归约到S_2，但是S_2的下一跳指令和B_2的假出口未定待填，它们指向相同。

蓝色表示即将翻译
红色表示需要回填

控制流语句的翻译-例

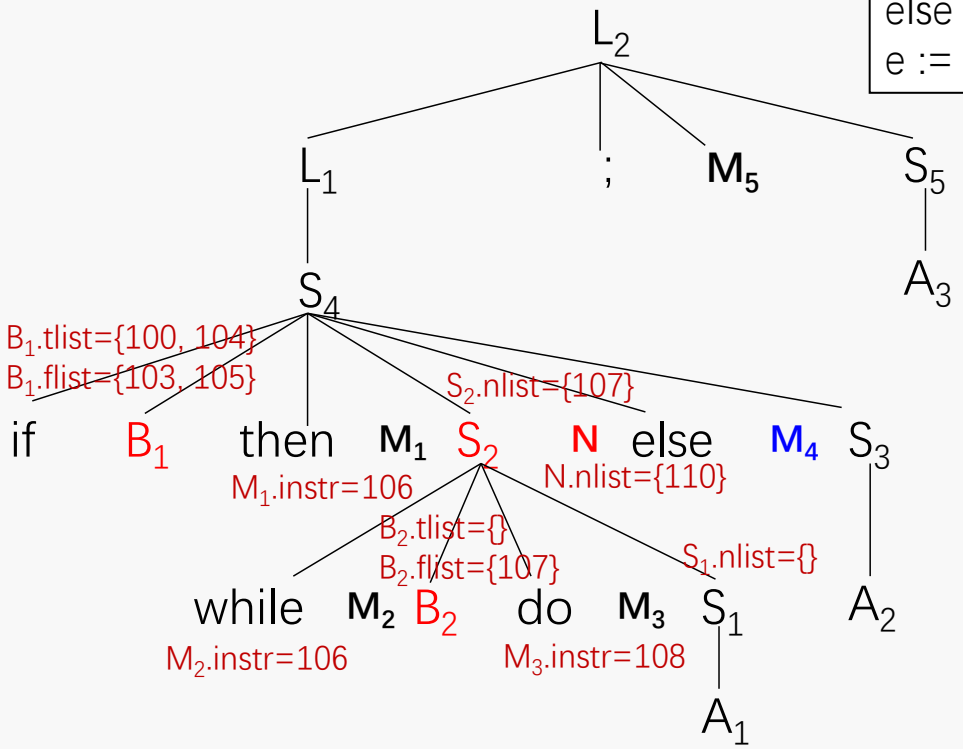
八、翻译 N： $N \rightarrow \varepsilon$

(110) goto - // N.nextlist = {110}

控制流语句的翻译-例

分析树

```
if ( a<b or c<d and e<f ) then
    while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填

控制流语句的翻译-例

九、翻译 M_4：　$M_4 \rightarrow \varepsilon$

记录下一个指令标号111，当if-then-else归约时，用111回填B_1的falselist{103, 105}

(108) c := c + 1　// $S_1 \rightarrow A_1$　S_1.nextlist={}

(109) goto 106　// 转至循环入口(106)

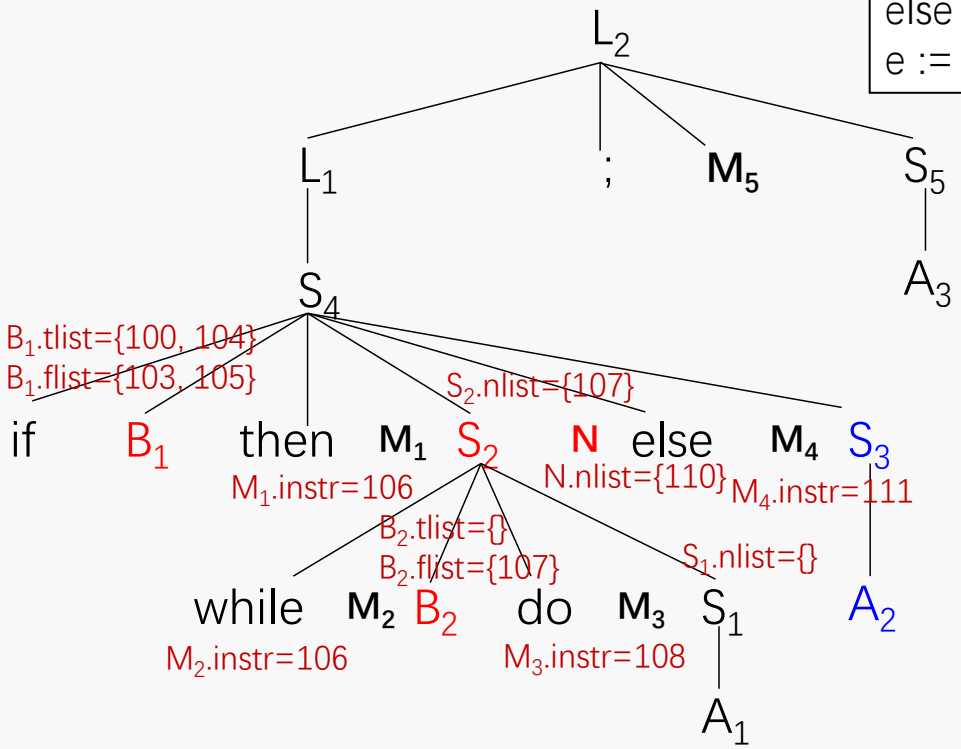
S_2.nextlist: { 107 } //转至循环外部

(110) goto 112　　// 由$N \rightarrow \varepsilon$生成={}

控制流语句的翻译-例

分析树

```
if ( a<b or c<d and e<f ) then
    while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填

控制流语句的翻译-例

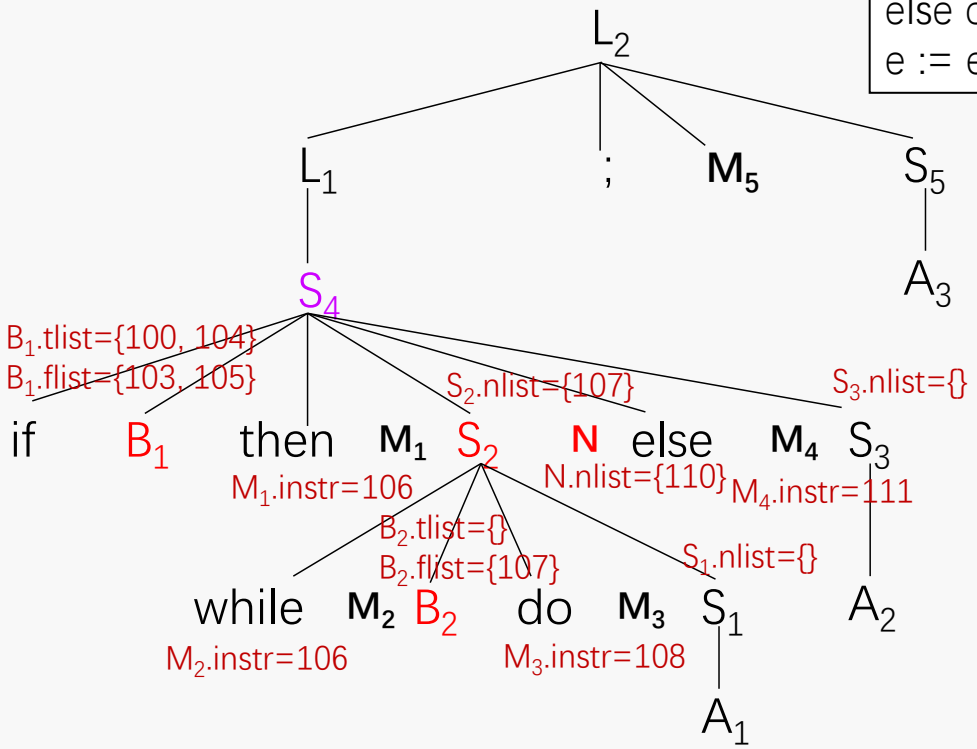
十、翻译 S_3： $S_3 \rightarrow A_2$

(111) d := d + 1 // S_3.nextlist={}

控制流语句的翻译-例

分析树

```
if ( a<b or c<d and e<f ) then
    while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填
紫色表示即将归约

控制流语句的翻译-例

十一、按if-then-else归约到S_4，进行如下操作

- 用M_1.instr即106回填布尔表达式B_1的truelist{100,104} 布尔表达式B_1的falselist{103,105}用111回填
- 整个if-then-else语句的nextlist，即S_4.nextlist合并为{ 107, 110 }

控制流语句的翻译-例

(100) if a<b goto 106

(101) goto 102

(102) if c<d goto 104

(103) goto -

(104) if e<f goto 106

(105) goto -

truelist: { 100, 104 } falselist: { 103, 105 }

控制流语句的翻译-例

十一、按if-then-else归约到S_4，进行如下操作

- 用M_1.instr即106回填布尔表达式B_1的truelist{100,104}
- 用M_4.instr即111回填布尔表达式B_1的falselist{103,105} 整个if-then-else语句的nextlist，即S_4.nextlist合并为{ 107, 110 }

控制流语句的翻译-例

(100) if a<b goto 106

(101) goto 102

(102) if c<d goto 104

(103) goto 111

(104) if e<f goto 106

(105) goto 111

truelist: { 100, 104 } falselist: { 103, 105 }

控制流语句的翻译-例

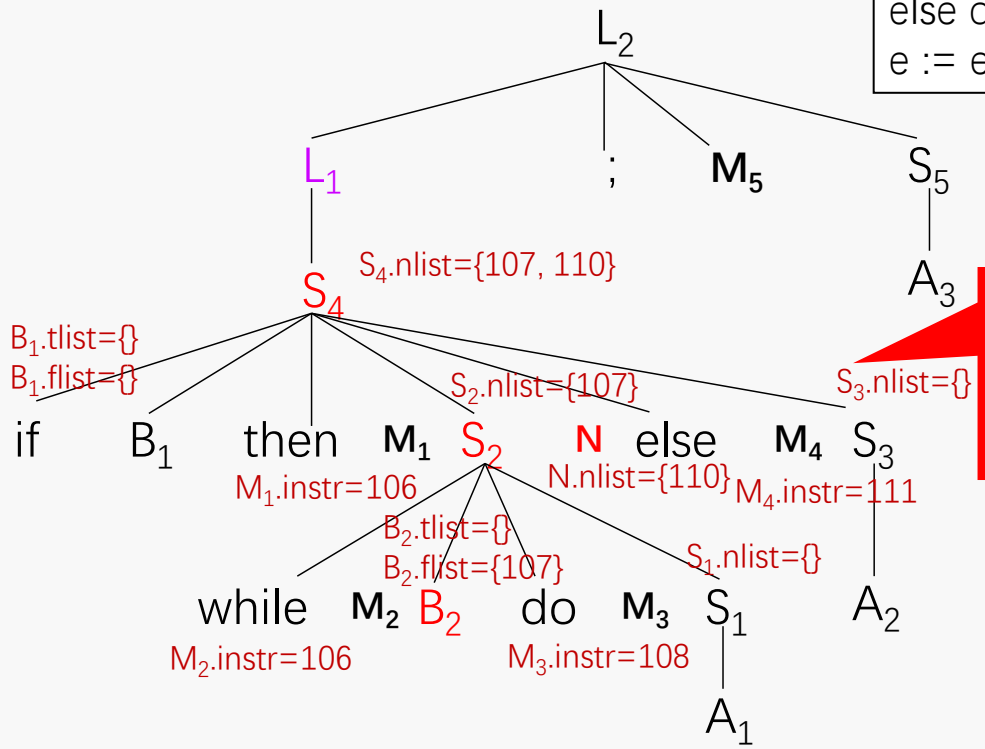
十一、按if-then-else归约到S_4，进行如下操作

- 用M_1.instr即106回填布尔表达式B_1的truelist{100,104}
- 用M_4.instr即111回填布尔表达式B_1的falselist{103,105}
- S_4.nextlist等于S_2、N和S_3的nextlist的并集，即为{107, 110}

控制流语句的翻译-例

分析树

```
if ( a<b or c<d and e<f ) then
    while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



S_4的下一跳指令、S_2的下一跳指令和B_2的假出口未定待填，它们指向相同。

蓝色表示即将翻译
红色表示需要回填
紫色表示即将归约

控制流语句的翻译-例

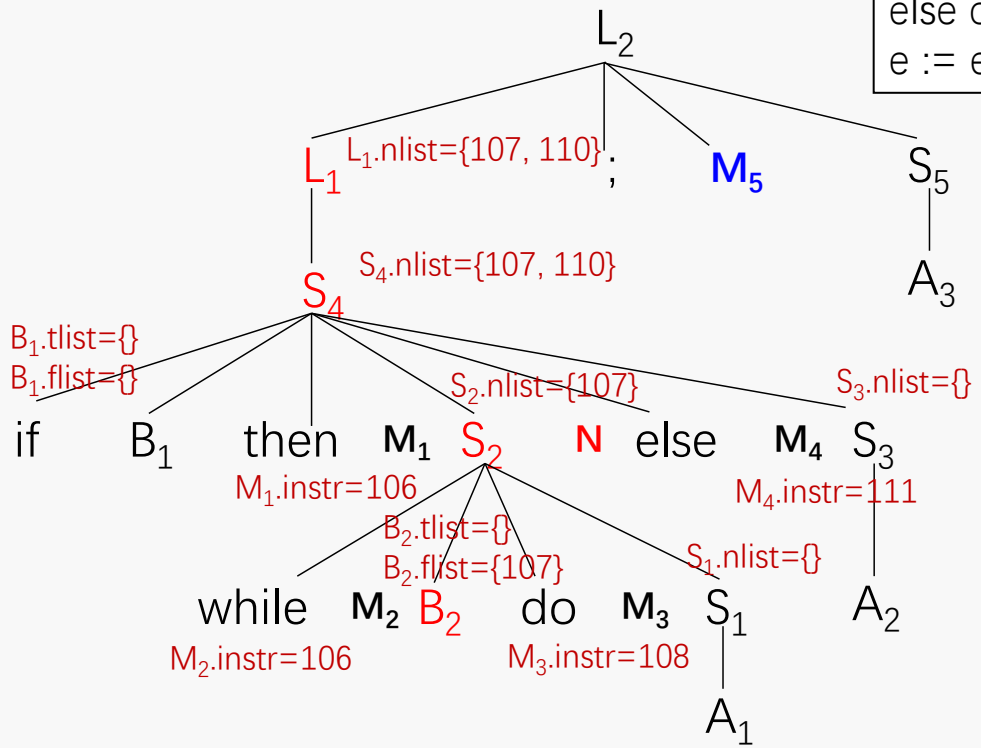
十二、归约L_1，将S_4.nextlist直接赋给L_1.nextlist

- L_1.nextlist: { 107, 110 }

控制流语句的翻译-例

分析树

```
if ( a<b or c<d and e<f ) then
    while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填

控制流语句的翻译-例

十三、翻译M_5： $M_5 \rightarrow \varepsilon$

记录下一个指令标号112，当L_1;S归约时，用112回填L_1的nextlist{107, 110}108

(112) e := e + d // $S_5 \rightarrow A_3$ S_5.nextlist={}

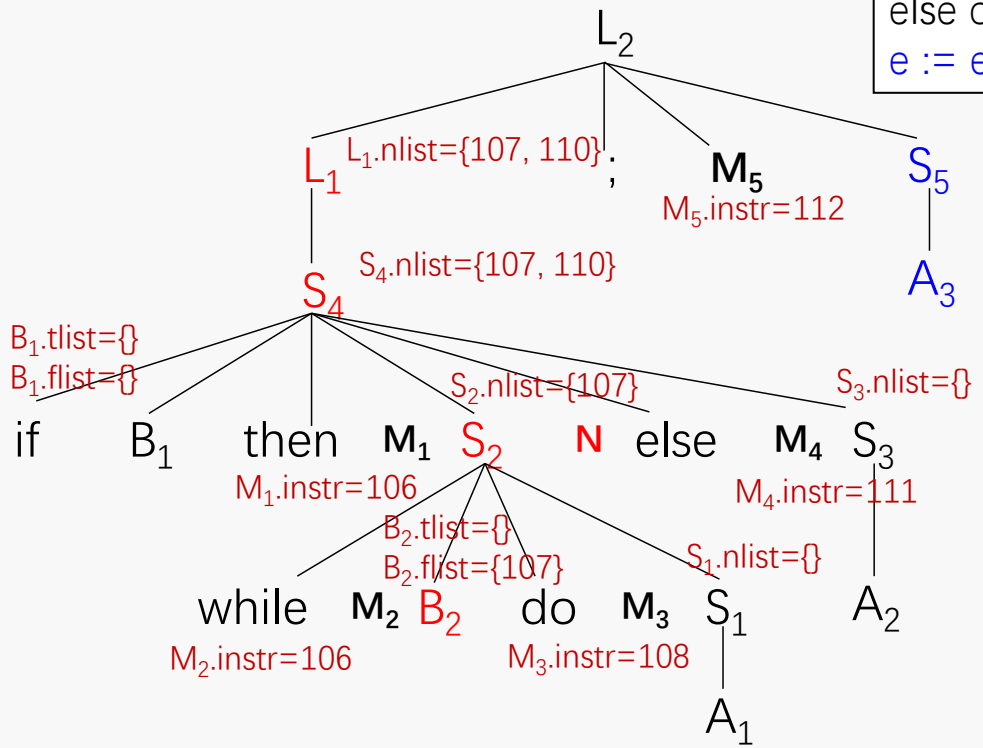
十四、翻译L_2

- 用112回填(107)和(110)
- L_2.nextlist: {}

控制流语句的翻译-例

分析树

```
if ( a<b or c<d and e<f ) then
    while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填

控制流语句的翻译-例

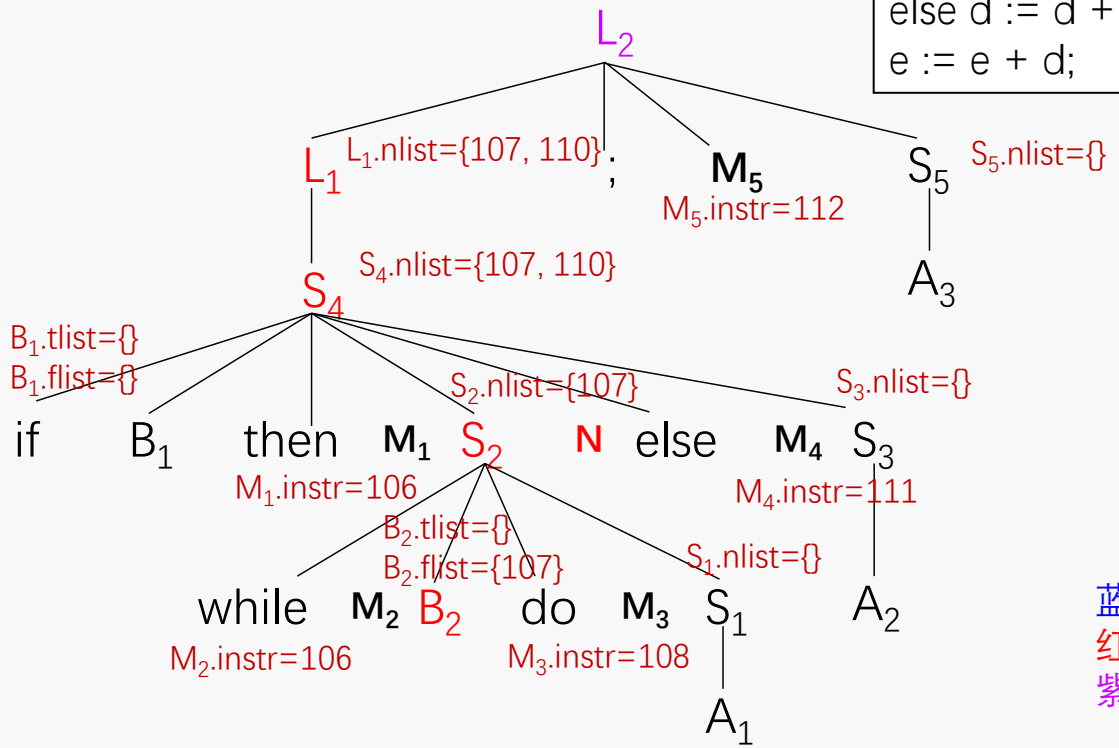
十四、翻译S_5

(112) e := e + d // $S_5 \rightarrow A_3$ S_5.nextlist={}

控制流语句的翻译-例

分析树

```
if ( a<b or c<d and e<f ) then
    while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填
紫色表示即将归约

控制流语句的翻译-例

十五、归约L_2

- 用M_5.instr即112回填L_1的nextlist{107，110}
- L_2.nextlist = S_5.nextlist，所以为空

控制流语句的翻译-例

(100) if a<b goto 106

(101) goto 102

(102) if c<d goto 104

(103) goto 111

(104) if e<f goto 106

(105) goto 111

(106) if a>c goto 108

(107) goto -

(108) c := c + 1

(109) goto 106

(110) goto -

(111) d := d + 1

(112) e := e + d

控制流语句的翻译-例-终

(100) if a<b goto 106

(101) goto 102

(102) if c<d goto 104

(103) goto 111

(104) if e<f goto 106

(105) goto 111

(106) if a>c goto 108

(107) goto 112

(108) c := c + 1

(109) goto 106

(110) goto 112

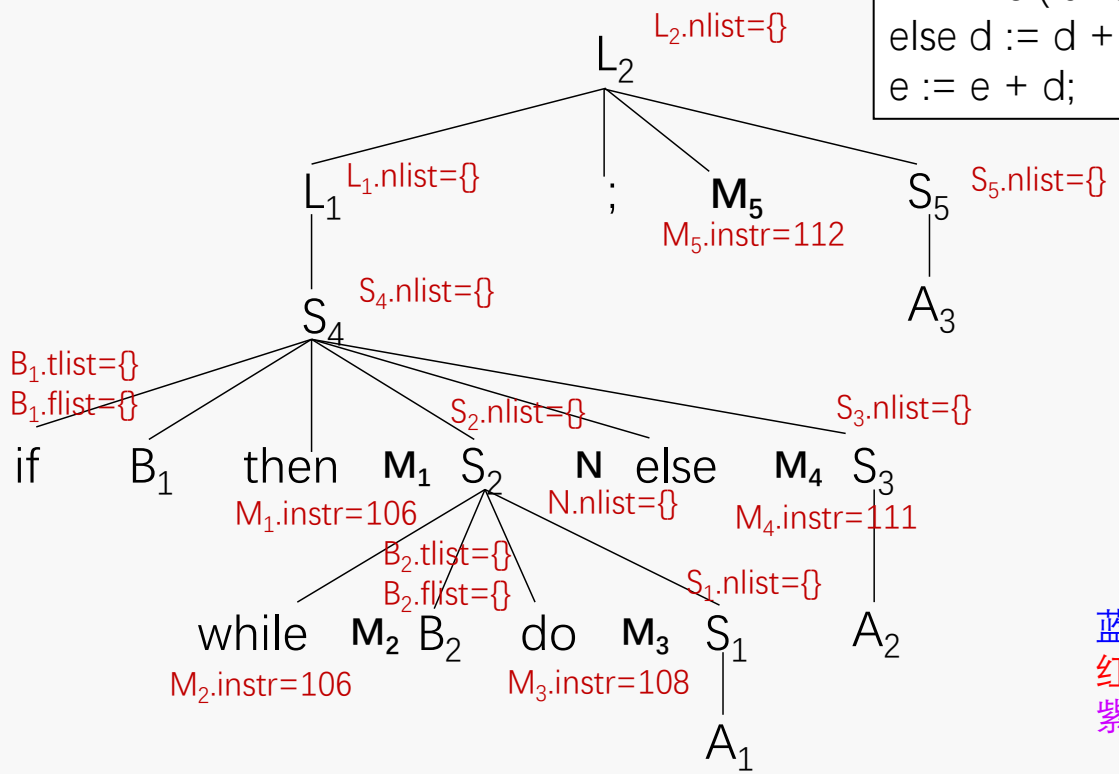
(111) d := d + 1

(112) e := e + d

控制流语句的翻译-例

分析树

```
if ( a<b or c<d and e<f ) then
    while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填
紫色表示即将归约

《编译原理和技术》

中间代码生成 IV

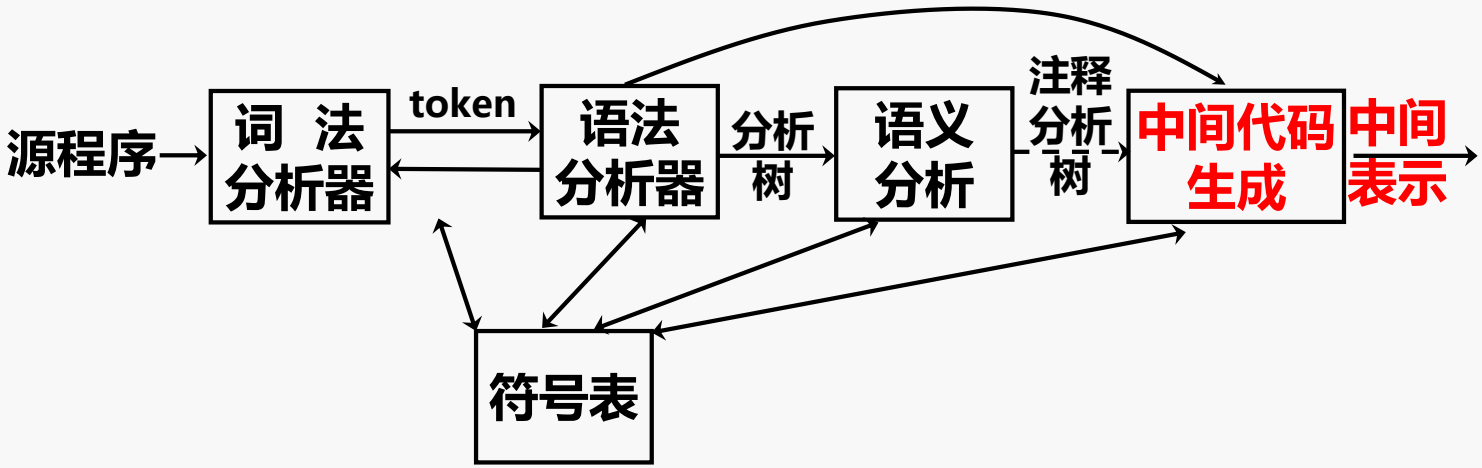
谢谢！

《编译原理和技术》

中间代码生成 V

中科大计算机学院
李 诚
2022-10-24

本节提纲



- 类型表达式
- 构造类型表达式的语法制导定义SDD
- 构造类型表达式的语法制导翻译SDT

类型表达式 (Type expression)

- 类型可以是语法的一部分，因此也是结构的

考虑以下文法，D代表声明语句，S代表一般语句

$P \rightarrow D\ ;\ S$

$D \rightarrow D\ ;\ D \mid \text{id}\ :\ T$

$T \rightarrow \text{boolean} \mid \text{integer} \mid \text{array [num] of } T \mid \uparrow T \mid T\ \text{‘}\rightarrow\text{’}\ T$

类型表达式 (Type expression)

- 类型可以是语法的一部分，因此也是结构的

考虑以下文法，D代表声明语句，S代表一般语句

$P \rightarrow D\ ;\ S$

$D \rightarrow D\ ;\ D \mid \mathrm{id}\ :\ T$

$T \rightarrow \mathrm{boolean} \mid \mathrm{integer} \mid \mathrm{array}\ [\mathrm{num}\ ]\ \mathrm{of}\ T \mid \uparrow T \mid T\ \text{‘}\rightarrow\text{’}\ T$

数组　指针　函数

基本类型　　复杂且可组合的类型

类型表达式 (Type expression)

- **基本类型是类型表达式**
  - *integer*
  - *real*
  - *char*
  - *boolean*
  - *type_error* //出错类型
  - *void* //无类型

在类型检查中传递错误

语句的类型

类型表达式 (Type expression)

- 基本类型是类型表达式
- 可为类型表达式命名，类名也是类型表达式

类型表达式 (Type expression)

- 基本类型是类型表达式
- 可为类型表达式命名，类名也是类型表达式
- 将类型构造算子(type constructor)作用于类型表达式可以构成新的类型表达式
  - 数组类型构造算子*array*
    - 如果T是类型表达式，N是一个整数，则*array(N, T)*是类型表达式

类型表达式 (Type expression)

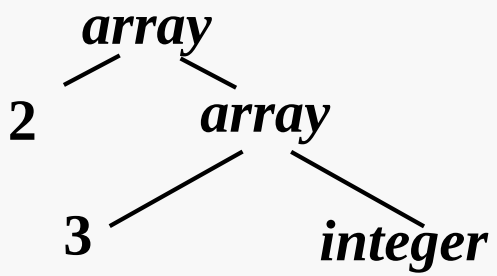
- 基本类型是类型表达式
- 可为类型表达式命名，类名也是类型表达式
- 将类型构造算子(type constructor)作用于类型表达式可以构成新的类型表达式
  - 数组类型构造算子*array*
    - 如果T是类型表达式，N是一个整数，则*array(N, T)*是类型表达式

类型	类型表达式
int[3]	*array* (3, *integer*)
int[2][3]	*array* (2, *array* (3, *integer*))

类型表达式 (Type expression)

- 基本类型是类型表达式
- 可为类型表达式命名，类名也是类型表达式
- 将类型构造算子(type constructor)作用于类型表达式可以构成新的类型表达式
  - 数组类型构造算子*array*
    - 如果T是类型表达式，N是一个整数，则*array(N, T)*是类型表达式

类型	类型表达式
int[3]	*array* (3, *integer*)
int[2][3]	*array* (2, *array* (3, *integer*))

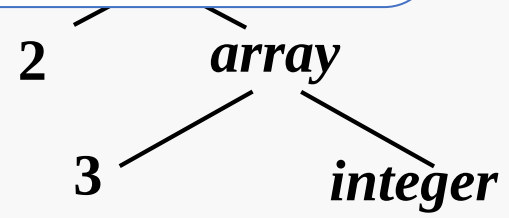


类型表达式 (Type expression)

- 基本类型是类型表达式
- 可为类型表达式命名，类名也是类型表达式
- 将类型构造算子(type constructor)作用于类型表达式可以构成新的类型表达式
  - 数组类型构造算子*array*
    - 如果T是类型表达式，N是一个整数，则*arra*

类型	类型表达式
int[3]	*array* (3, *integer*)
int[2][3]	*array* (2, *array* (3, *integer*))

也可以写为
array ({0,…,2}, integer)
其中{0,…,2}代表索引集合
如首元素索引从1开始，则
为*array ({1,…,3}, integer)*



类型表达式 (Type expression)

- 基本类型是类型表达式
- 可为类型表达式命名，类名也是类型表达式
- 将类型构造算子(type constructor)作用于类型表达式可以构成新的类型表达式
  - 数组类型构造算子*array*
  - 指针类型构造算子*pointer*
    - 如果T是类型表达式，则*pointer(T)*是类型表达式

类型表达式 (Type expression)

- 基本类型是类型表达式
- 可为类型表达式命名，类名也是类型表达式
- 将类型构造算子(type constructor)作用于类型表达式可以构成新的类型表达式
  - 数组类型构造算子*array*
  - 指针类型构造算子*pointer*
  - 笛卡尔乘积类型构造算子×
    - 如果T_1和T_2是类型表达式，则$T_1 \times T_2$也是类型表达式
    - 主要用于描述列表和元组，如：表示函数的参数

类型表达式 (Type expression)

- 基本类型是类型表达式
- 可为类型表达式命名，类名也是类型表达式
- 将类型构造算子(type constructor)作用于类型表达式可以构成新的类型表达式
  - 数组类型构造算子*array*
  - 指针类型构造算子*pointer*
  - 笛卡尔乘积类型构造算子×
  - 函数类型构造算子→
    - 若$T_1, T_2, \ldots, T_n$和R是类型表达式，则$T_1 \times T_2 \times \ldots \times T_n \rightarrow R$也是

函数参数　函数返回值

类型表达式 (Type expression)

- 基本类型是类型表达式
- 可为类型表达式命名，类名也是类型表达式
- 将类型构造算子(type constructor)作用于类型表达式可以构成新的类型表达式
  - 数组类型构造算子*array*
  - 指针类型构造算子*pointer*
  - 笛卡尔乘积类型构造算子×
  - 函数类型构造算子→
  - 记录类型构造算子*record*
    - 若有标识符$N_1, N_2, \ldots, N_n$以及对应的类型表达式$T_1, T_2, \ldots, T_n$，则*record*$((N_1 \times T_1) \times (N_2 \times T_2) \times \ldots \times (N_n \times T_n))$也是类型表达式

记录中的字段

字段对应的类型表达式

类型表达式-例1

- 考虑C语言中数组double a[10][20]，写出a、a[0]、a[0][0]的类型表达式

a[0][0]: double

a[0]: array(20,double);
pointer(double)

a: array(10,array(20,double));
pointer(array(20,double))

类型表达式-例1

- 考虑C语言中数组double a[10][20]，写出a、a[0]、a[0][0]的类型表达式

a[0][0]: double

a[0]: array(20,double);
pointer(double)

a: array(10,array(20,double));
pointer(array(20,double))

类型表达式-例1

- 考虑C语言中数组double a[10][20]，写出a、a[0]、a[0][0]的类型表达式

a[0][0]: double

a[0]: array(20,double);
pointer(double)

a: array(10,array(20,double));
pointer(array(20,double))

类型表达式-例1

- 考虑C语言中数组double a[10][20]，写出a、a[0]、a[0][0]的类型表达式

a[0][0]: double

a[0]: array(20,double);
pointer(double)

a: array(10,array(20,double));
pointer(array(20,double))

类型表达式-例2

- **为row、table和p分别写出类型表达式：**

```
typedef struct{
    int address;
    char lexeme[15];
} row;
row table[101];
row *p;
```

row的类型表达式：

record((address×integer) × (lexeme × (array(15, char))))

table的类型表达式：

array(101, row) //此处row是类型名，因此也是类型表达式

p的类型表达式：

pointer(row)

类型表达式-例2

- **为row、table和p分别写出类型表达式：**

```
typedef struct{
    int address;
    char lexeme[15];
} row;
row table[101];
row *p;
```

row的类型表达式：

record((address×integer) × (lexeme × (array(15, char))))

table的类型表达式：

array(101, row) //此处row是类型名，因此也是类型表达式

p的类型表达式：

pointer(row)

类型表达式-例3

- **考虑下面的函数f，写出其类型表达式。**

  int *f(char a, char b);

  f的类型表达式：

  (char×char) → pointer(integer)

类型表达式-例3

- **考虑下面的函数f，写出其类型表达式。**

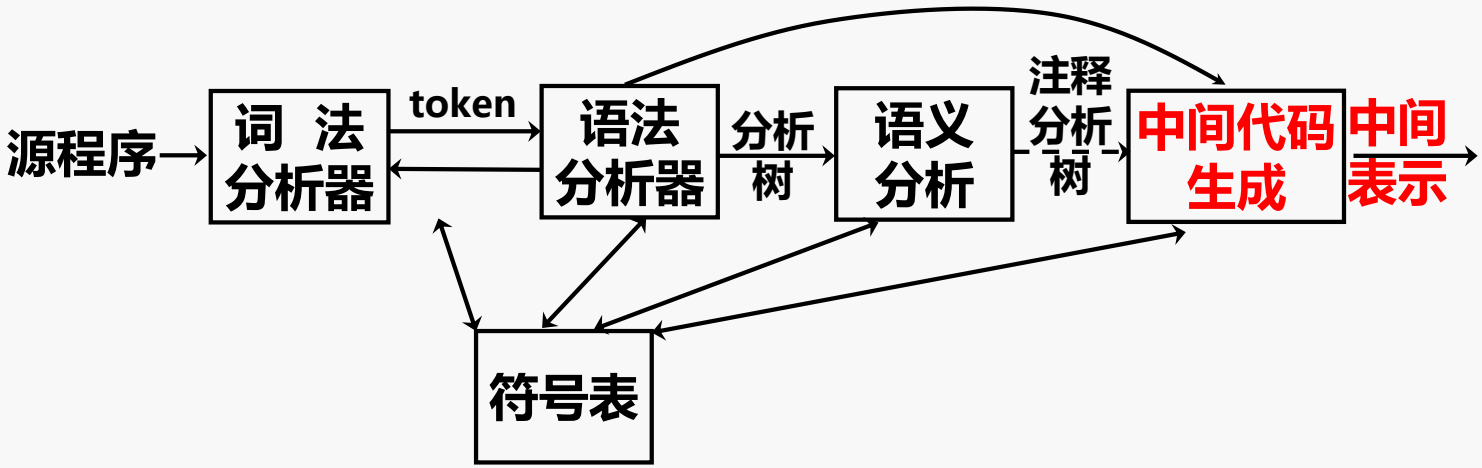
  int *f(char a, char b);

  f的类型表达式：

  (char×char) → pointer(integer)

问题：可否自动化地实现类型表达式的生成?

本节提纲



- 类型表达式
- 构造类型表达式的语法制导定义SDD
- 构造类型表达式的语法制导翻译SDT

构造类型表达式的SDD

- 为以下文法制定构造类型表达式的语义规则

产　生　式	语 义 规 则
$T \rightarrow B\ C$	
$B \rightarrow$ int	
$B \rightarrow$ float	
$C \rightarrow$ [num] C_1	
$C \rightarrow \varepsilon$	

构造类型表达式的SDD

- 为以下文法制定构造类型表达式的语义规则

产　生　式	语　义　规　则
$T \rightarrow B\ C$	
$B \rightarrow$ int	
$B \rightarrow$ float	
$C \rightarrow$ [num] C_1	
$C \rightarrow \varepsilon$	

- 为每个文法符号设置**综合属性**t和**继承属性**b
  - t：该符号对应的类型表达式
  - b：将类型信息从左到右传递

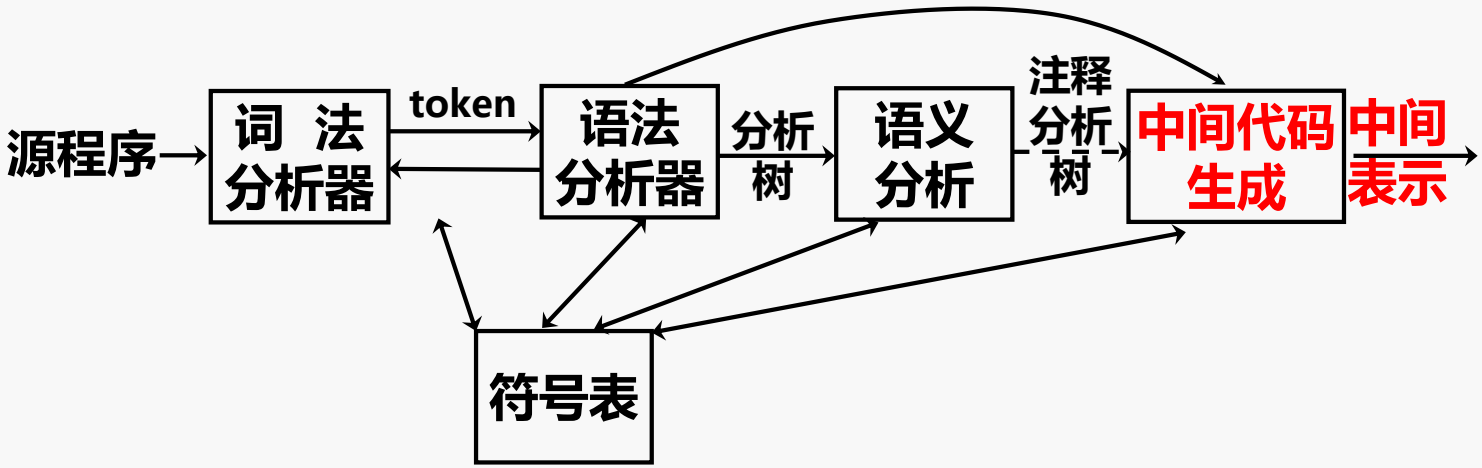
构造类型表达式的SDD

- 为以下文法制定构造类型表达式的语义规则

产　生　式	语　义　规　则
$T \rightarrow B\ C$	$T.t = C.t;\ C.b = B.t$
$B \rightarrow$ int	$B.t = integer$
$B \rightarrow$ float	$B.t = float$
$C \rightarrow$ [num] C_1	$C.t = array(num.val, C_1.t);\ C_1.b = C.b$
$C \rightarrow \varepsilon$	$C.t = C.b$

- 为每个文法符号设置**综合属性**t和**继承属性**b
  - t：该符号对应的类型表达式
  - b：将类型信息从左到右传递

本节提纲



- 类型表达式
- 构造类型表达式的语法制导定义SDD
- **构造类型表达式的语法制导翻译SDT**

构造类型表达式的SDT

- 将SDD改造为SDT

$T \rightarrow B$ {*C.b = B.t*} C {*T.t = C.t;* }

$B \rightarrow$ int {*B.t = integer*}

$B \rightarrow$ float {*B.t = float*}

$C \rightarrow$ [num] {C_1*.b = C.b*} C_1 {*C.t = array(num.val,* C_1*.t);* }

$C \rightarrow \varepsilon$ {*C.t = C.b*}

- 但是继承属性的计算与LR分析方法不适配
- 因此，如果要使用LR，就需要改造文法

构造类型表达式的SDT

- 通过改造文法，与LR适配
  - 引入标记M，C归约时可在栈顶以下位置找到B.t
  - 引入标记N，把继承属性C.b当做综合属性记录

$T \rightarrow B\ M\ C\ \{T.t = C.t;\ \}$

$M \rightarrow \varepsilon\ \{M.t = B.t\}$

$B \rightarrow \text{int}\ \{B.t = integer\}$

$B \rightarrow \text{float}\ \{B.t = float\}$

$C \rightarrow [\text{num}]\ N\ C_1\ \{C.t = array(num.val, C_1.t);\ \}$

$N \rightarrow \varepsilon\ \{N.t = C.b\}$

$C \rightarrow \varepsilon\ \{C.t = C.b\}$

构造类型表达式的SDT

- 分析int[2][3]的LR栈操作

$T \rightarrow B\ M\ C$ {$T.t = C.t$; }

$M \rightarrow \varepsilon$ {$M.t = B.t$}

$B \rightarrow$ int {$B.t = integer$}

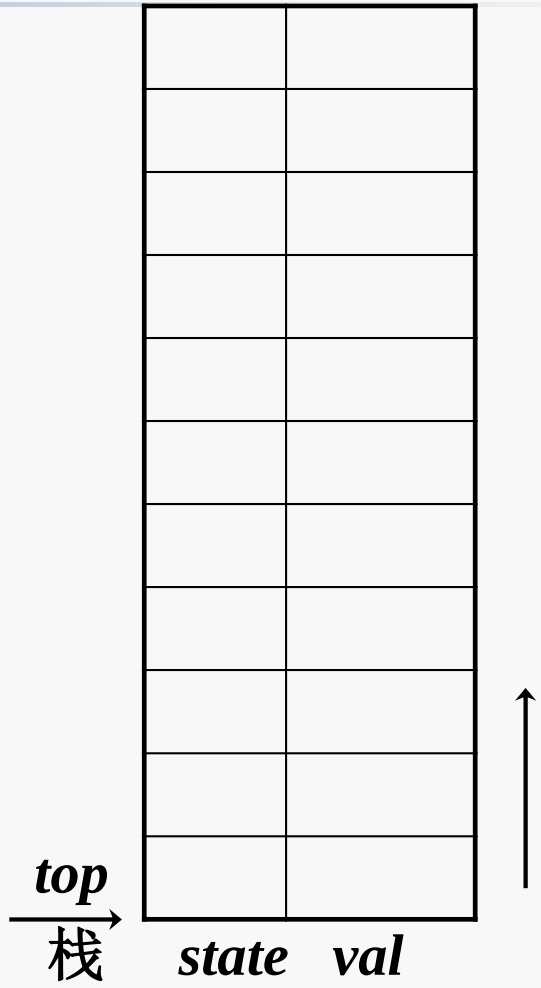
$B \rightarrow$ float {$B.t = float$}

$C \rightarrow$ [num] $N\ C_1$

{$C.t = array(num.val, C_1.t)$; }

$N \rightarrow \varepsilon$ {$N.t = C.b$}

$C \rightarrow \varepsilon$ {$C.t = C.b$}



构造类型表达式的SDT

- 分析int[2][3]的LR栈操作

$T \rightarrow B\ M\ C$ {$T.t = C.t$; }

$M \rightarrow \varepsilon$ {$M.t = B.t$}

$B \rightarrow$ int {$B.t = integer$}

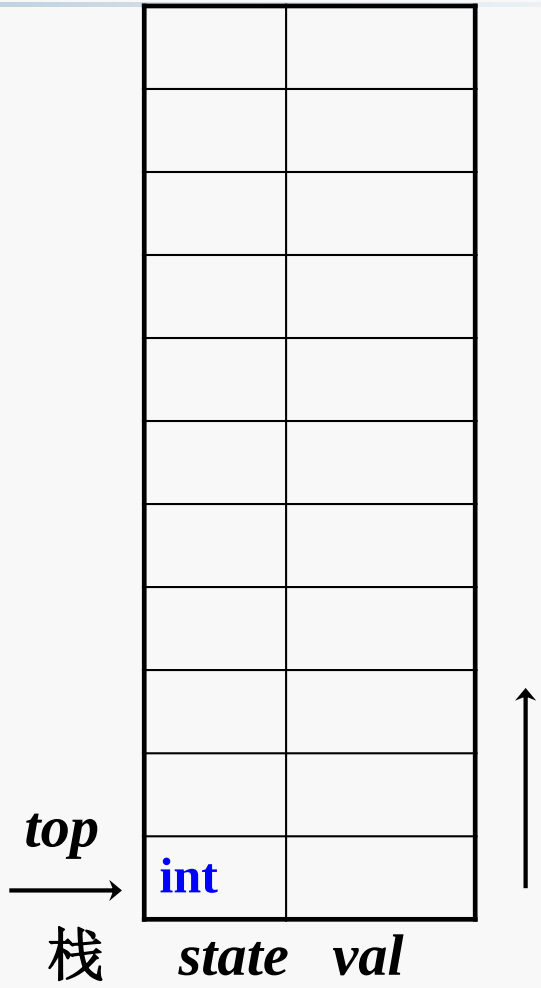
$B \rightarrow$ float {$B.t = float$}

$C \rightarrow$ [num] $N\ C_1$

{$C.t = array(num.val, C_1.t)$; }

$N \rightarrow \varepsilon$ {$N.t = C.b$}

$C \rightarrow \varepsilon$ {$C.t = C.b$}



构造类型表达式的SDT

- 分析int[2][3]的LR栈操作

$T \rightarrow B\ M\ C$ {$T.t = C.t$; }

$M \rightarrow \varepsilon$ {$M.t = B.t$}

$B \rightarrow$ int {$B.t = integer$}

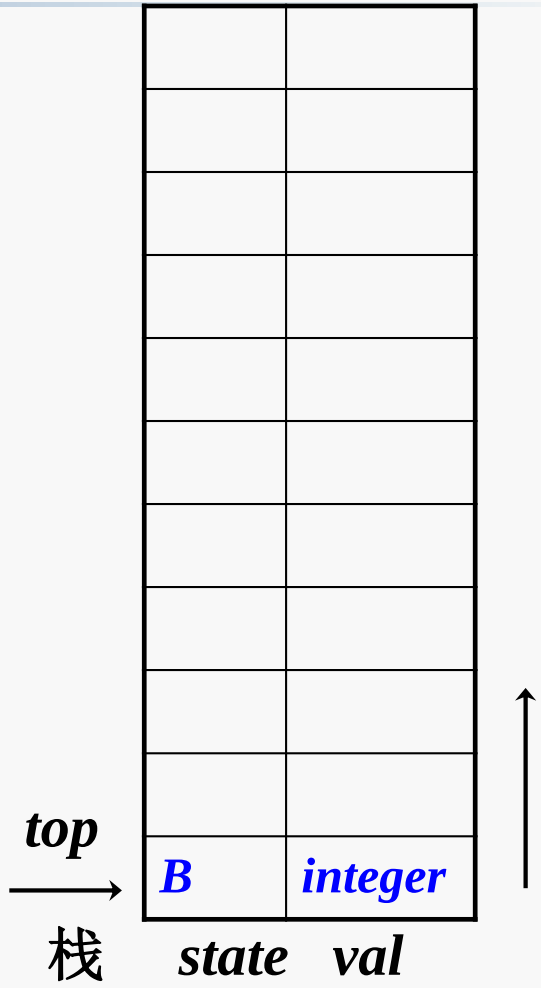
$B \rightarrow$ float {$B.t = float$}

$C \rightarrow$ [num] $N\ C_1$

{$C.t = array(num.val, C_1.t)$; }

$N \rightarrow \varepsilon$ {$N.t = C.b$}

$C \rightarrow \varepsilon$ {$C.t = C.b$}



构造类型表达式的SDT

- 分析int[2][3]的LR栈操作

$T \rightarrow B\ M\ C$ {$T.t = C.t$; }

$M \rightarrow \varepsilon$ {$M.t = B.t$}

$B \rightarrow$ int {$B.t = integer$}

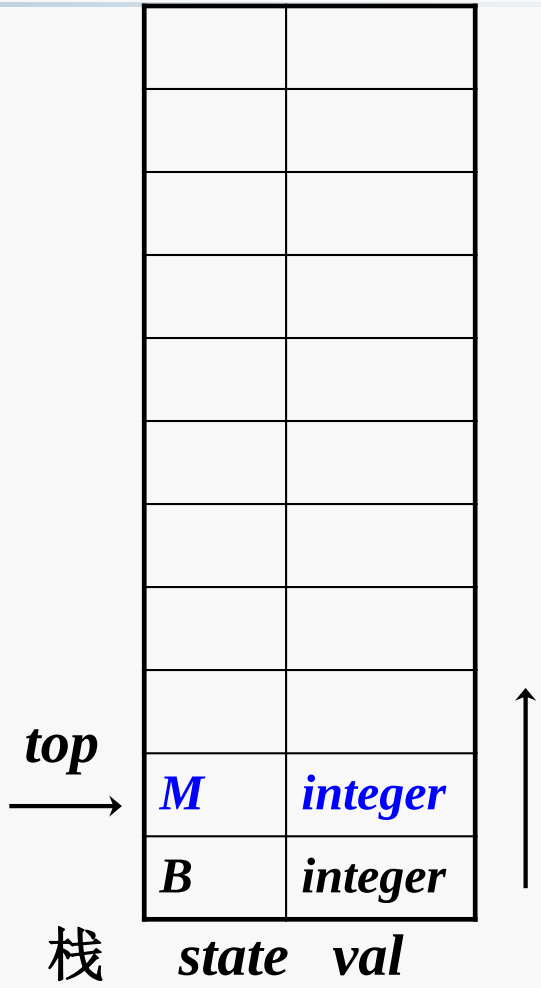
$B \rightarrow$ float {$B.t = float$}

$C \rightarrow$ [num] $N\ C_1$

{$C.t = array(num.val, C_1.t)$; }

$N \rightarrow \varepsilon$ {$N.t = C.b$}

$C \rightarrow \varepsilon$ {$C.t = C.b$}



构造类型表达式的SDT

- 分析int[2][3]的LR栈操作

$T \rightarrow B\ M\ C$ {$T.t = C.t$; }

$M \rightarrow \varepsilon$ {$M.t = B.t$}

$B \rightarrow$ int {$B.t = integer$}

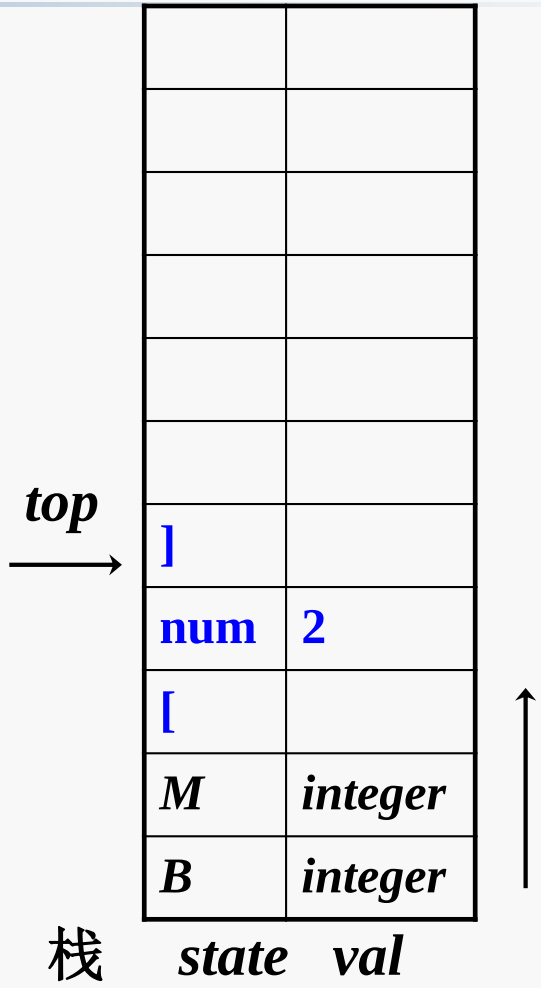
$B \rightarrow$ float {$B.t = float$}

$C \rightarrow$ [num] $N\ C_1$

{$C.t = array(num.val, C_1.t)$; }

$N \rightarrow \varepsilon$ {$N.t = C.b$}

$C \rightarrow \varepsilon$ {$C.t = C.b$}



构造类型表达式的SDT

- 分析int[2][3]的LR栈操作

$T \rightarrow B\ M\ C$ {$T.t = C.t$; }

$M \rightarrow \varepsilon$ {$M.t = B.t$}

$B \rightarrow$ int {$B.t = integer$}

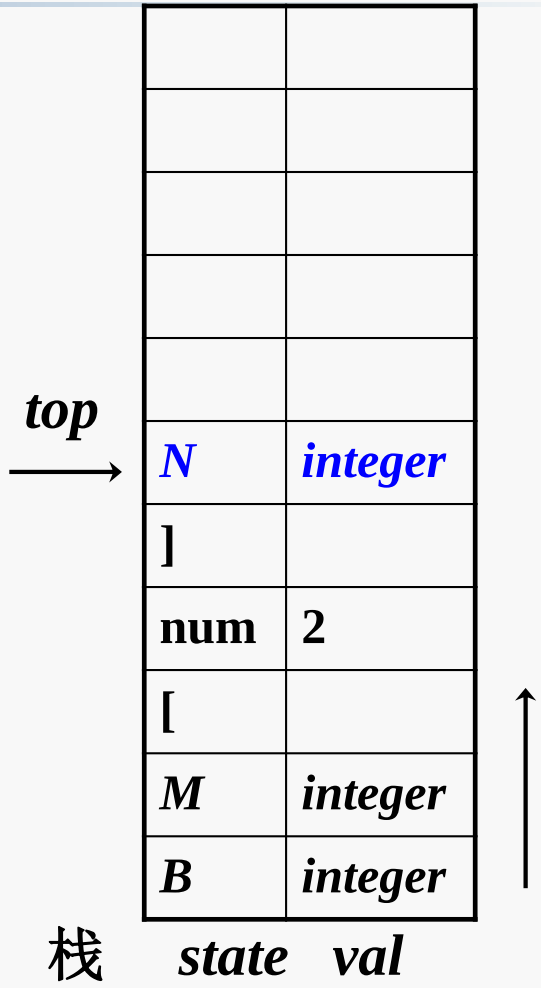
$B \rightarrow$ float {$B.t = float$}

$C \rightarrow$ [num] $N\ C_1$

{$C.t = array(num.val, C_1.t)$; }

$N \rightarrow \varepsilon$ {$N.t = C.b$}

$C \rightarrow \varepsilon$ {$C.t = C.b$}



构造类型表达式的SDT

- 分析int[2][3]的LR栈操作

$T \rightarrow B\ M\ C$ {$T.t = C.t$; }

$M \rightarrow \varepsilon$ {$M.t = B.t$}

$B \rightarrow$ int {$B.t = integer$}

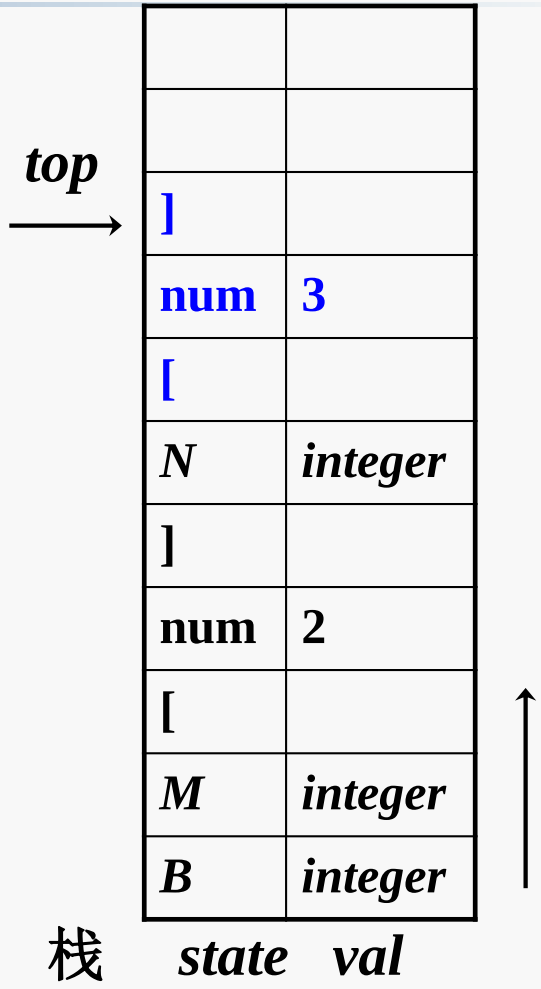
$B \rightarrow$ float {$B.t = float$}

$C \rightarrow$ [num] $N\ C_1$

{$C.t = array(num.val, C_1.t)$; }

$N \rightarrow \varepsilon$ {$N.t = C.b$}

$C \rightarrow \varepsilon$ {$C.t = C.b$}



构造类型表达式的SDT

- 分析int[2][3]的LR栈操作

$T \rightarrow B\ M\ C$ {$T.t = C.t$; }

$M \rightarrow \varepsilon$ {$M.t = B.t$}

$B \rightarrow$ int {$B.t = integer$}

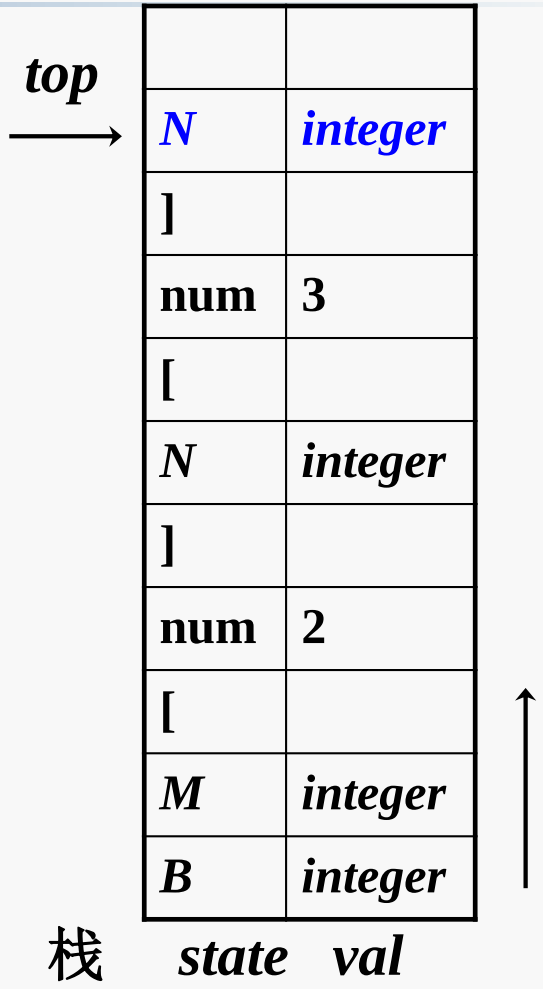
$B \rightarrow$ float {$B.t = float$}

$C \rightarrow$ [num] $N\ C_1$

{$C.t = array(num.val, C_1.t)$; }

$N \rightarrow \varepsilon$ {$N.t = C.b$}

$C \rightarrow \varepsilon$ {$C.t = C.b$}



构造类型表达式的SDT

- 分析int[2][3]的LR栈操作

$T \rightarrow B\ M\ C$ {$T.t = C.t$; }

$M \rightarrow \varepsilon$ {$M.t = B.t$}

$B \rightarrow$ int {$B.t = integer$}

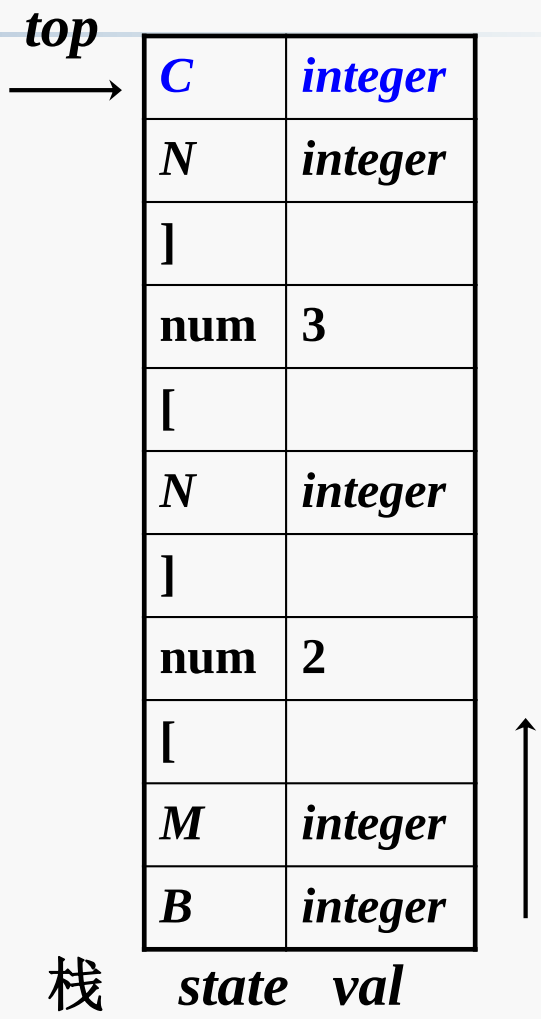
$B \rightarrow$ float {$B.t = float$}

$C \rightarrow$ [num] $N\ C_1$

{$C.t = array(num.val, C_1.t)$; }

$N \rightarrow \varepsilon$ {$N.t = C.b$}

$C \rightarrow \varepsilon$ {$C.t = C.b$}



构造类型表达式的SDT

- 分析int[2][3]的LR栈操作

$T \rightarrow B\ M\ C$ {$T.t = C.t$; }

$M \rightarrow \varepsilon$ {$M.t = B.t$}

$B \rightarrow$ int {$B.t = integer$}

$B \rightarrow$ float {$B.t = float$}

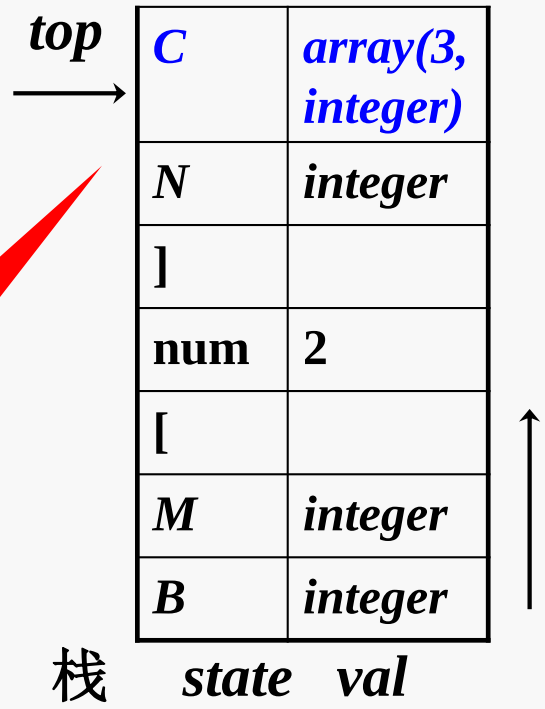
$C \rightarrow$ [num] $N\ C_1$

{$C.t = array(num.val, C_1.t)$; }

$N \rightarrow \varepsilon$ {$N.t = C.b$}

$C \rightarrow \varepsilon$ {$C.t = C.b$}

完成第一次归约
$C \rightarrow$ [num] $N\ C_1$



构造类型表达式的SDT

- 分析int[2][3]的LR栈操作

$T \rightarrow B\ M\ C$ {$T.t = C.t$; }

$M \rightarrow \varepsilon$ {$M.t = B.t$}

$B \rightarrow$ int {$B.t = integer$}

$B \rightarrow$ float {$B.t = float$}

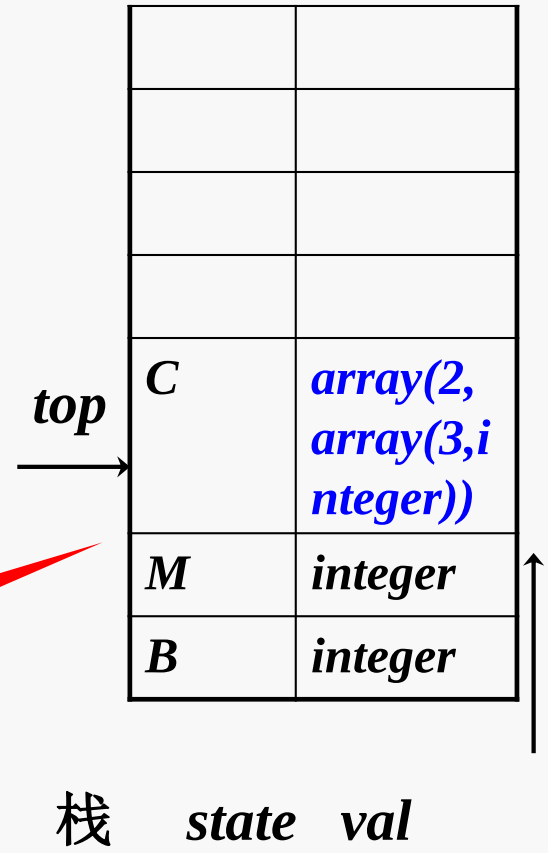
$C \rightarrow$ [num] $N\ C_1$

{$C.t = array(num.val, C_1.t)$; }

$N \rightarrow \varepsilon$ {$N.t = C.b$}

$C \rightarrow \varepsilon$ {$C.t = C.b$}

完成第二次归约
$C \rightarrow$ [num] $N\ C_1$



构造类型表达式的SDT

- 分析int[2][3]的LR栈操作

$T \rightarrow B\ M\ C$ {$T.t = C.t$; }

$M \rightarrow \varepsilon$ {$M.t = B.t$}

$B \rightarrow$ int {$B.t = integer$}

$B \rightarrow$ float {$B.t = float$}

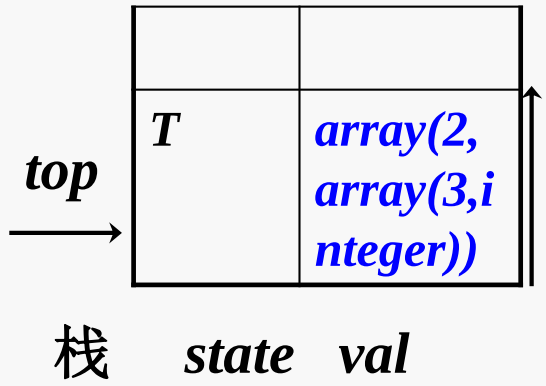
$C \rightarrow$ [num] $N\ C_1$

{$C.t = array(num.val, C_1.t)$; }

$N \rightarrow \varepsilon$ {$N.t = C.b$}

$C \rightarrow \varepsilon$ {$C.t = C.b$}

完成第三次归约
$T \rightarrow B\ M\ C$



《编译原理和技术》

中间代码生成 V

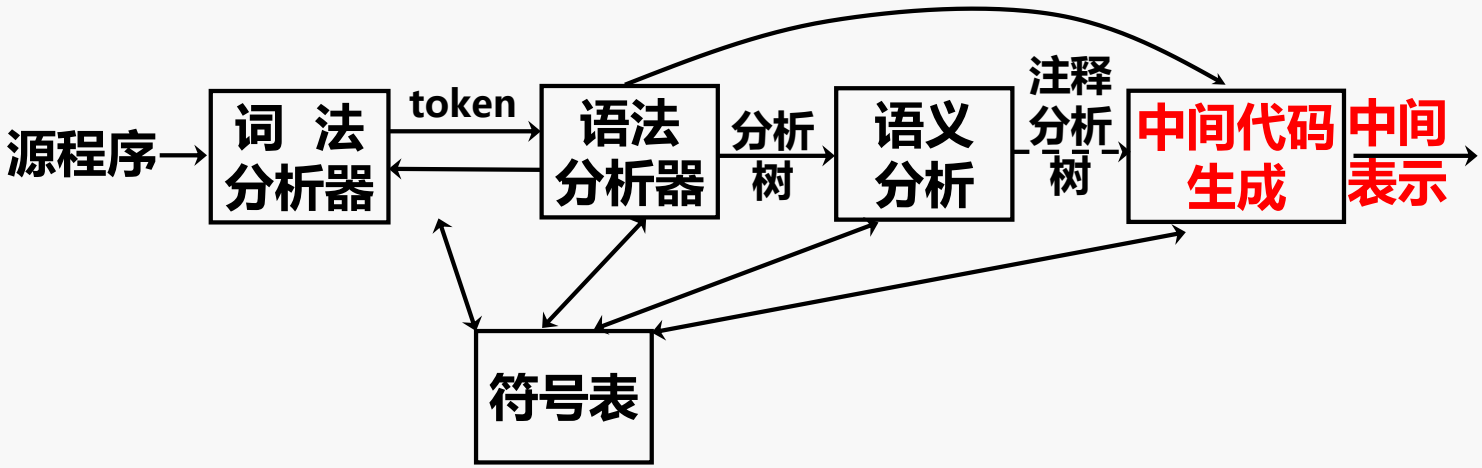
谢谢！

《编译原理和技术》

中间代码生成 VI

中科大计算机学院
李 诚
2022-10-26

本节提纲



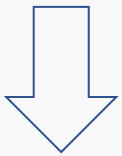
- 符号表的组织
- 声明语句的翻译
  - 存储空间计算、作用域、记录

符号表 (Symbol table)

- 符号表的使用和修改伴随编译的全过程
- 存储entity的各种信息
  - 如variable names, function names, objects, classes, interfaces 等
  - 如类型信息、所占用内存空间、作用域
- 用于编译过程中的分析与合成
  - 语义分析：如使用前声明检查、类型检查、确定作用域等
  - 合成：如类型表达式构造、内存空间分配等

符号表 (Symbol table)

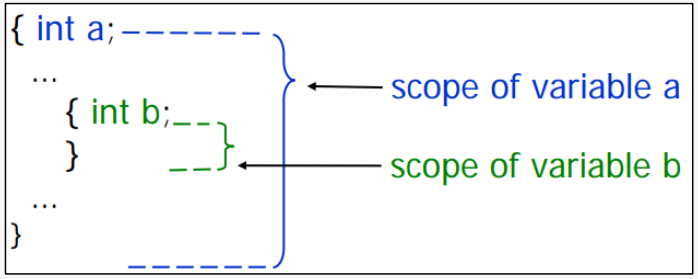
代码片段:
extern bool foo(auto int m, const int n);
const bool tmp;



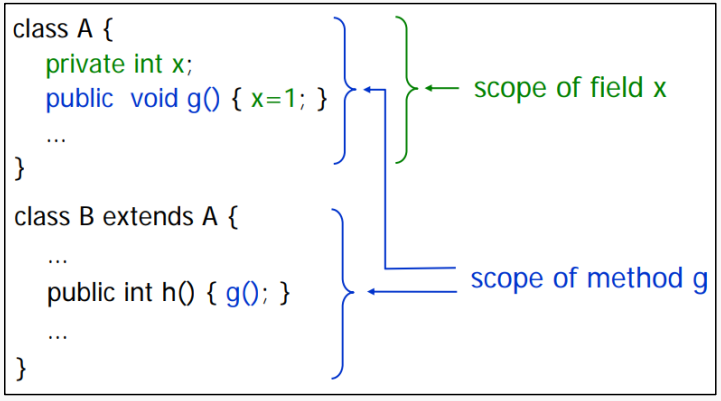
NAME	KIND	TYPE	OTHER
foo	fun	int x int → bool	extern
m	par	int	auto
n	par	int	const
tmp	var	bool	const

符号表

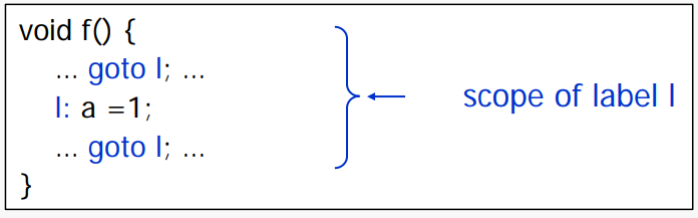
符号表——作用域



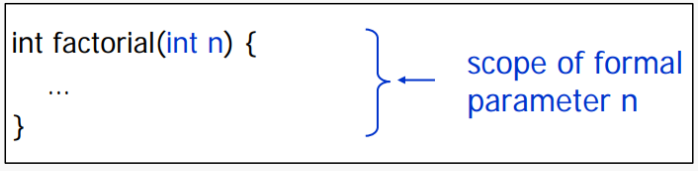
程序块中



对象中的field和methods



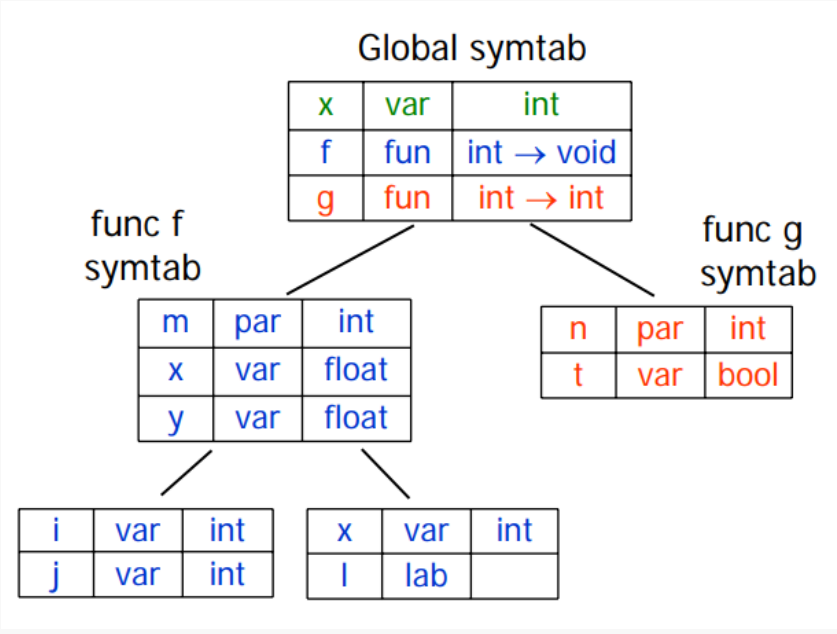
语句标号



过程或函数定义中的参数

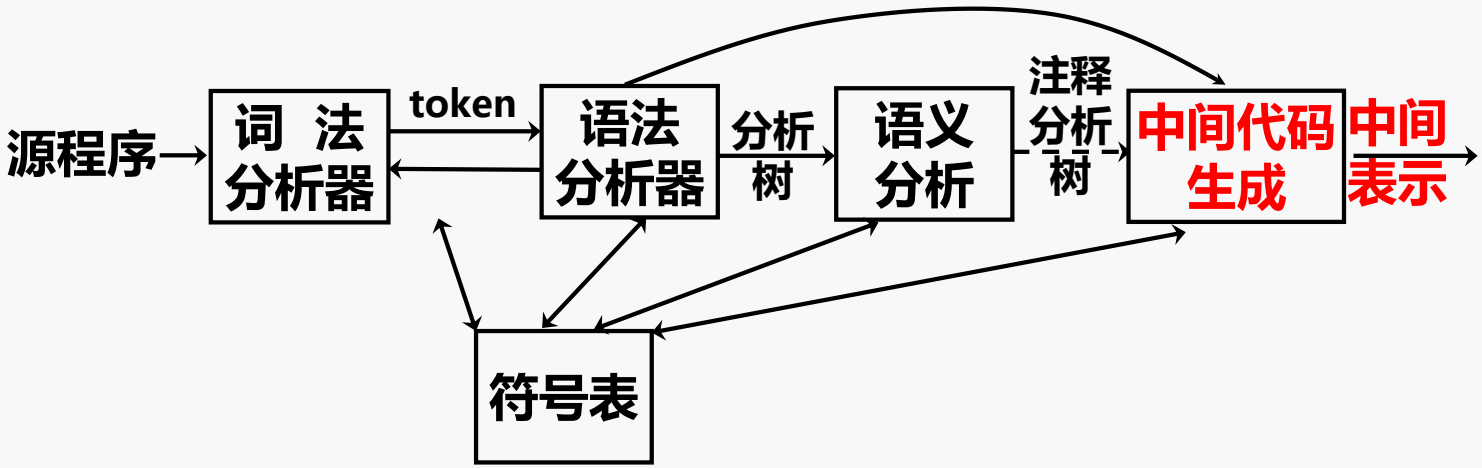
符号表 (Symbol table)

```
int x;

void f(int m) {
     float x, y;
     ...
    { int i, j; …; }
    { int x; l: …; }
}

int g(int n) {
    bool t;
    …;
}
```



注：l代表label

本节提纲



- 符号表的组织
- **声明语句的翻译**
  - 存储空间计算、作用域、记录

声明语句翻译的要点

- **分配存储单元**
  - 名字、类型、字宽、偏移
- **作用域的管理**
  - 过程调用
- **记录类型的管理**
- **不产生中间代码指令，但是要更新符号表**

声明语句的翻译

- 例：文法G_1如下：

  $P \rightarrow D ; S$

  $D \rightarrow D ; D$

  $D \rightarrow id : T$

  $T \rightarrow integer \mid real \mid array [num] of T_1 \mid \uparrow T_1$

声明语句的翻译

- 有关符号的属性

  T.type - 变量所具有的类型，如

  整型 INT

  实型 REAL

  数组类型 array（元素个数，元素类型）

  指针类型 pointer（所指对象类型）

  T.width - 该类型数据所占的字节数

  offset - 变量的存储偏移地址

声明语句的翻译

T.type		T.width
整型	INT	4
实型	REAL	8
数组	array（num, T_1）	num.val * T_1.width
指针	pointer（T_1）	4
enter(name, type, offset)—将类型type和偏移offset填入符号表中name所在的表项。		

声明语句的翻译

计算被声明名字的类型和相对地址

P→ {*offset = 0*} D ; S

相对地址初始化为0

D→ D ; D

D→ id : T {*enter(id.lexeme, T.type, offset) ;*

***offset = offset + T.width* }**

更新符号表信息

T→integer {*T.type = integer; T.width = 4*}

T→real {*T.type = real; T.width = 8*}

类型=>字宽

T→array [number] of T_1

{*T.type = array(num.val, T_1.type);*

***T.width = num.val * T_1.width*}**

T→ ↑ T_1 {*T.type = pointer(T_1.type); T.width = 4*}

声明语句翻译的要点

- 分配存储单元
  - 名字、类型、字宽、偏移
- 作用域的管理
  - 过程调用
- 记录类型的管理
- 不产生中间代码指令，但是要更新符号表

允许自定义过程时的翻译

- 所讨论语言的文法

  $P \rightarrow D; S$

  $D \rightarrow D\ ;\ D \mid \text{id} : T \mid$

  proc id ; D ; S

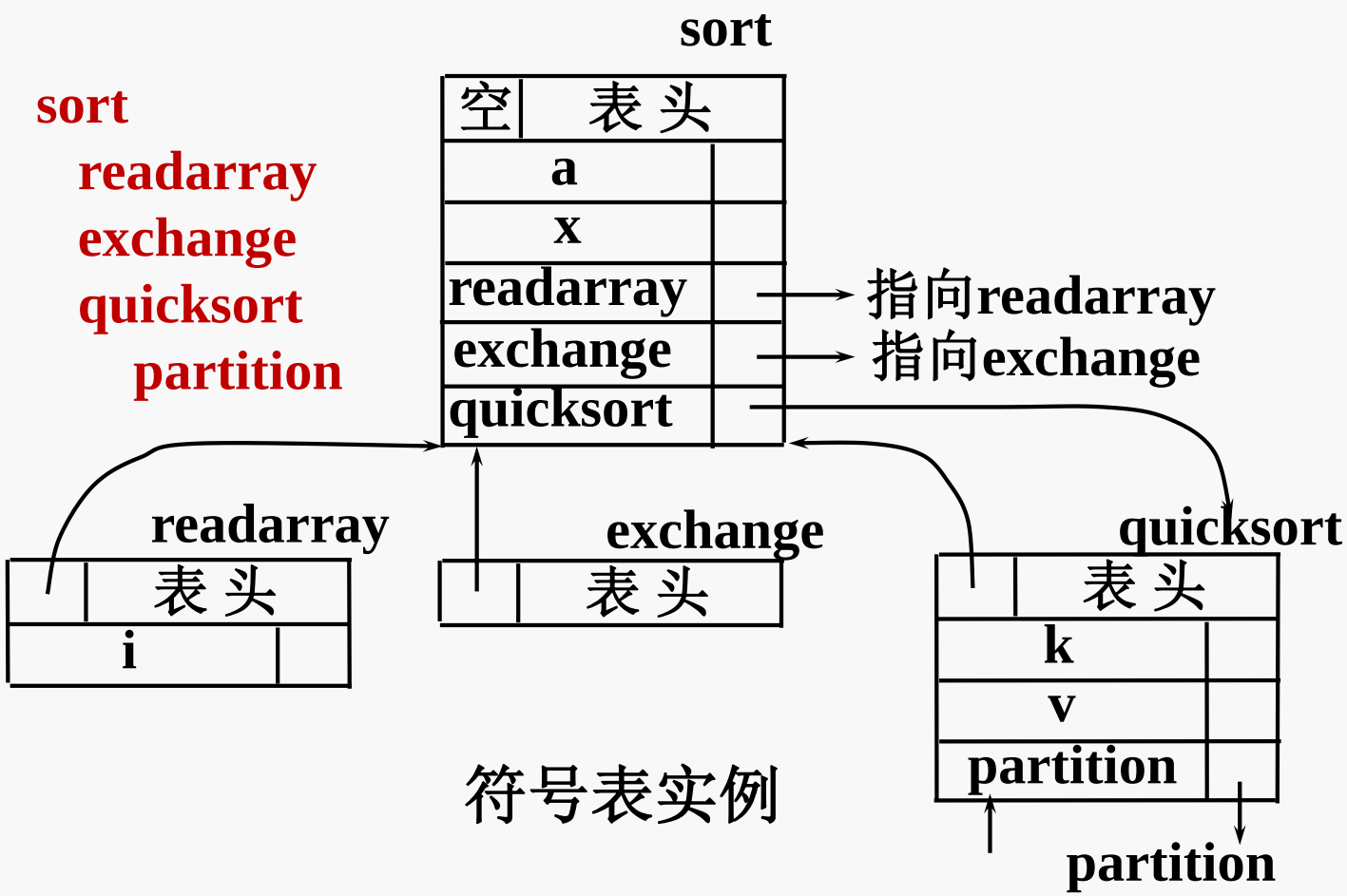
- 管理作用域(过程嵌套声明)
  - 每个过程内声明的符号要置于该过程的符号表中
  - 方便地找到子过程和父过 程对应的符号

```
sort
    var a:…; x:…;
    readarray
        var i:…;
    exchange
    quicksort
        var k, v:…;
        partition
            var i, j:…;
```

教科书186页图6.14

过程参数被略去

各过程的符号表


sort
sort
readarray
exchange
quicksort
partition
空
表 头
a
x
readarray
exchange
quicksort
指向readarray
指向exchange
readarray
表 头
i
exchange
表 头
quicksort
表 头
k
v
partition
partition
符号表实例

符号表的组织与管理

符号表的特点及数据结构

- 各过程有各自的符号表：哈希表
- 符号表之间有双向链
  - 父→子：过程中包含哪些子过程定义
  - 子→父：分析完子过程后继续分析父过程
- 维护符号表栈(*tblptr*)和地址偏移量栈(*offset*)
  - 保存尚未完成的过程的符号表指针和相对地址

符号表的组织与管理

语义动作用到的函数

/* 建立新的符号表，其表头指针指向父过程符号表*/

1. *mkTable*(*parent-table*)

/* 将所声明变量的类型、偏移填入当前符号表*/

2. *enter*(*current-table*, *name*, *type*, *current-offset*)

/* 在父过程符号表中建立子过程名的条目*/

3. *enterProc*(*parent-table*, *sub-proc-name*, *sub-table*)

/*在符号表首部添加变量累加宽度，可利用符号表栈tblptr和偏移栈offset（栈顶值分别表示当前分析的过程的符号表及可用变量偏移位置）*/

4. addWidth(table, width)

声明语句的处理

$P \rightarrow M\ D;\ S$ {*addWidth (top (tblptr), top (offset));*
pop(tblptr); pop (offset) }

$M \rightarrow \varepsilon$ {*t = mkTable (nil);*
push(t, tblprt); push (0, offset) }

$D \rightarrow D_1\ ;\ D_2$

$D \rightarrow$ proc id ; $N\ D_1;\ S$ {*t = top(tblptr);*
addWidth(t, top(offset)); pop(tblptr); pop(offset);
enterProc(top(tblptr), id.lexeme, t) }

$D \rightarrow$ id :T {*enter(top(tblptr), id.lexeme, T.type, top(offset));*
top(offset) = top(offset) + T.width }

$N \rightarrow \varepsilon$ {*t = mkTable(top(tblptr));*
push(t, tblptr); push(0, offset) }

声明语句的处理

tblptr：符号表栈
offset：偏移量栈

$P \rightarrow M\ D;\ S$ {*addWidth* (*top* (*tblptr*), *top* (*offset*));
pop(*tblptr*); *pop* (*offset*) }

$M \rightarrow \varepsilon$ {*t* = *mkTable* (*nil*);
push(*t*, *tblptr*); *push* (0, *offset*) }

$D \rightarrow D_1\ ;\ D_2$

$D \rightarrow$ proc id ; $N\ D_1;\ S$ {*t* = *top*(*tblptr*);
addWidth(*t*, *top*(*offset*)); *pop*(*tblptr*); *pop*(*offset*);
enterProc(*top*(*tblptr*), id.*lexeme*, *t*) }

$D \rightarrow$ id :T {*enter*(*top*(*tblptr*), id.*lexeme*, *T*.*type*, *top*(*offset*));
top(*offset*) = *top*(*offset*) + *T*.*width* }

$N \rightarrow \varepsilon$ {*t* = *mkTable*(*top*(*tblptr*));
push(*t*, *tblptr*); *push*(0, *offset*) }

建立主程序（最外围）的符号表偏移从0开始

声明语句的处理

$P \rightarrow M\ D;\ S$ {*addWidth* (*top* (*tblptr*), *top* (*offset*));
pop(*tblptr*); *pop* (*offset*) }

$M \rightarrow \varepsilon$ {*t* = *mkTable* (*nil*);
push(*t*, *tblptr*); *push* (0, *offset*) }

$D \rightarrow D_1\ ;\ D_2$

$D \rightarrow$ proc id ; $N\ D_1;\ S$ {*t* = *top*(*tblptr*);
addWidth(*t*, *top*(*offset*)); *pop*(*tblptr*); *pop*(*offset*);
enterProc(*top*(*tblptr*), id.*lexeme*, *t*) }

$D \rightarrow$ id :*T* {*enter*(*top*(*tblptr*), id.*lexeme*, *T.type*, *top*(*offset*));
top(*offset*) = *top*(*offset*) + *T.width* }

$N \rightarrow \varepsilon$ {*t* = *mkTable*(*top*(*tblptr*));
push(*t*, *tblptr*); *push*(0, *offset*) }

将变量name的有关属性填入当前符号表

声明语句的处理

$P \rightarrow M\ D;\ S$ {*addWidth* (*top* (*tblptr*), *top* (*offset*));
pop(*tblptr*); *pop* (*offset*) }

$M \rightarrow \varepsilon$ {*t* = *mkTable* (*nil*);
push(*t*, *tblptr*); *push* (0, *offset*) }

$D \rightarrow D_1\ ;\ D_2$

$D \rightarrow$ proc id ; $N\ D_1;\ S$ {*t* = *top*(*tblptr*);
addWidth(*t*, *top*(*offset*)); *pop*(*tblptr*); *pop*(*offset*);
enterProc(*top*(*tblptr*), id.*lexeme*, *t*) }

$D \rightarrow$ id :*T* {*enter*(*top*(*tblptr*), id.*lexeme*, *T.type*, *top*(*offset*));
top(*offset*) = *top*(*offset*) + *T.width* }

$N \rightarrow \varepsilon$ {*t* = *mkTable*(*top*(*tblptr*));
push(*t*, *tblptr*); *push*(0, *offset*) }

建立子过程的符号表和偏移从0开始

声明语句的处理

$P \rightarrow M\ D;\ S$ {*addWidth* (*top* (*tblptr*), *top* (*offset*));
pop(*tblptr*); *pop* (*offset*) }

$M \rightarrow \varepsilon$ {*t* = *mkTable* (*nil*);
push(*t*, *tblptr*); *push* (0, *offset*) }

$D \rightarrow D_1\ ;\ D_2$

$D \rightarrow$ proc id ; $N\ D_1$; S {*t* = *top*(*tblptr*);
addWidth(*t*, *top*(*offset*)); *pop*(*tblptr*); *pop*(*offset*);
enterProc(*top*(*tblptr*), id.*lexeme*, *t*) }

$D \rightarrow$ id :T {*enter*(*top*(*tblptr*), id.*lexeme*, *T.type*, *top*(*offset*));
top(*offset*) = *top*(*offset*) + *T.width* }

$N \rightarrow \varepsilon$ {*t* = *mkTable*(*top*(*tblptr*));
push(*t*, *tblptr*); *push*(0, *offset*) }

保留当前过程声明的总空间；弹出符号表和偏移栈顶（露出父过程的符号表和偏移；在父过程符号表中填写子过程名有关条目

声明语句的处理

$P \rightarrow M\ D;\ S$ {*addWidth* (*top* (*tblptr*), *top* (*offset*));
pop(*tblptr*); *pop* (*offset*) }

$M \rightarrow \varepsilon$ {*t* = *mkTable* (*nil*);
push(*t*, *tblptr*); *push* (0, *offset*) }

$D \rightarrow D_1\ ;\ D_2$

$D \rightarrow$ proc id ; $N\ D_1$; S {*t* = *top*(*tblptr*);
addWidth(*t*, *top*(*offset*)); *pop*(*tblptr*); *pop*(*offset*);
enterProc(*top*(*tblptr*), id.*lexeme*, *t*) }

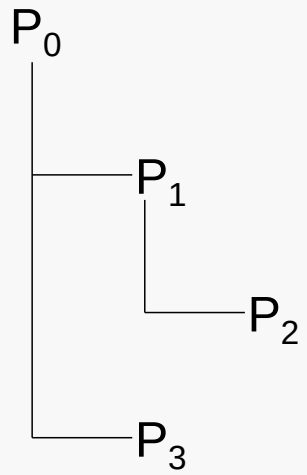
$D \rightarrow$ id :T {*enter*(*top*(*tblptr*), id.*lexeme*, *T.type*, *top*(*offset*));
top(*offset*) = *top*(*offset*) + *T.width* }

$N \rightarrow \varepsilon$ {*t* = *mkTable*(*top*(*tblptr*));
push(*t*, *tblptr*); *push*(0, *offset*) }

修改变量分配空间大小并清空符号表和偏移栈

举例：过程嵌套声明

```
i : int;  j : int ;
PROC P1 ;
    k : int;  f : real ;
    PROC P2;
          l : int ;
         a1 ;
    a2;
PROC P3;
    temp : int ;  max : int ;
    a3;
```



过程声明层次图

举例：过程嵌套声明

- 初始：M→ε

null	总偏移：

P_0

top → | P_0 | 0 |

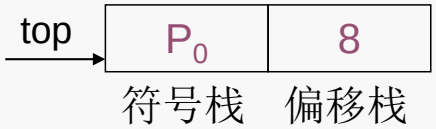
符号栈　偏移栈

举例：过程嵌套声明

- i : int ; j : int ;

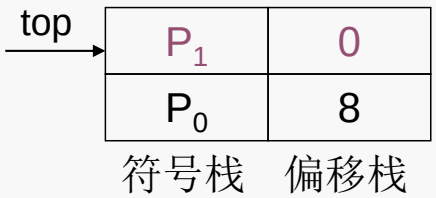
null	总偏移：	
i	INT	0
j	INT	4

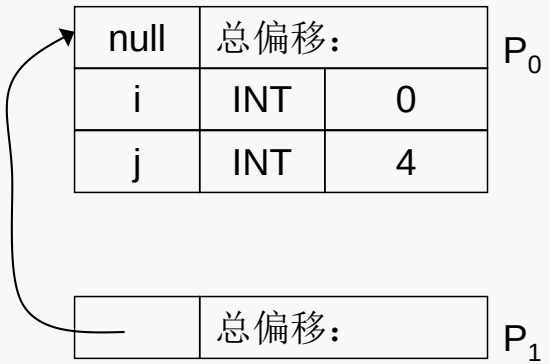
P_0



举例：过程嵌套声明

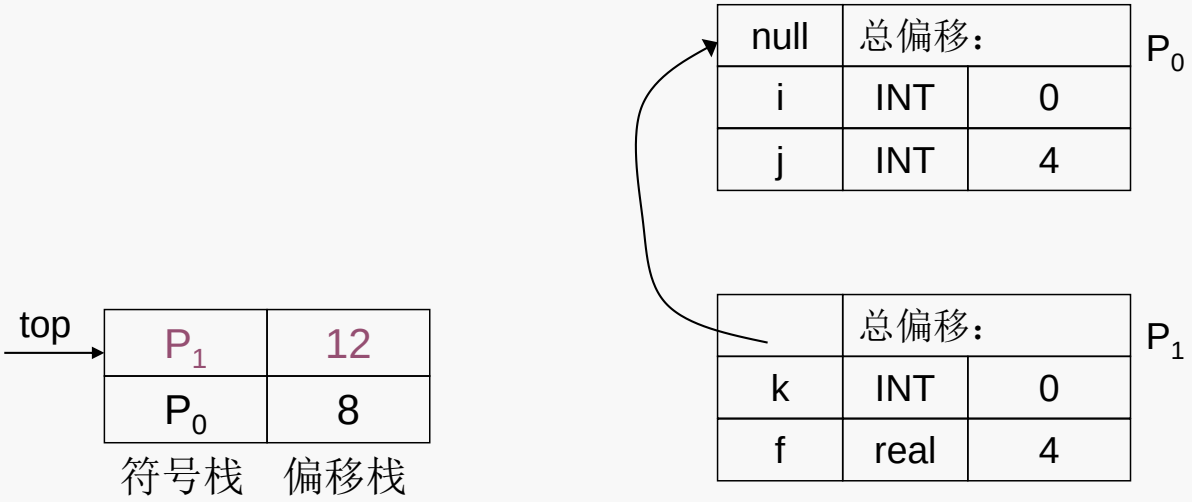
- PROC P_1；（$N \rightarrow \varepsilon$）





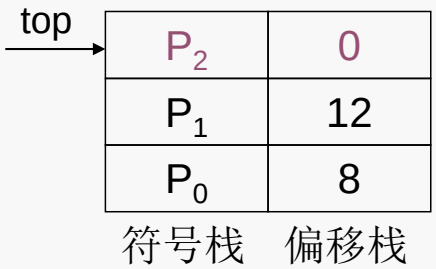
举例：过程嵌套声明

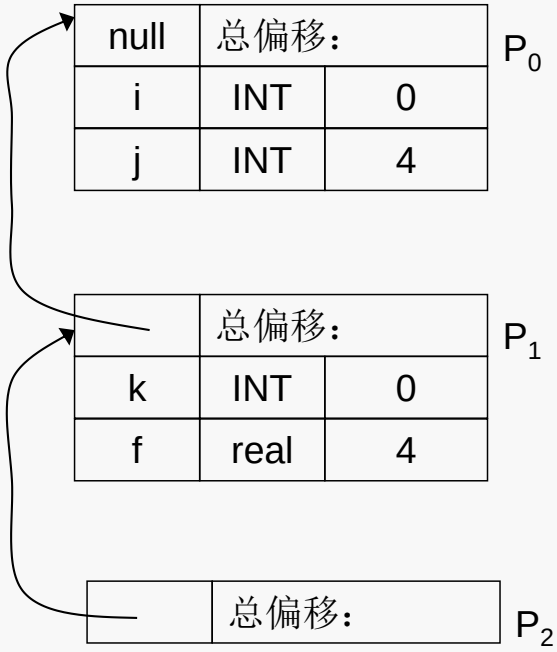
- k : int ; f : real ;



举例：过程嵌套声明

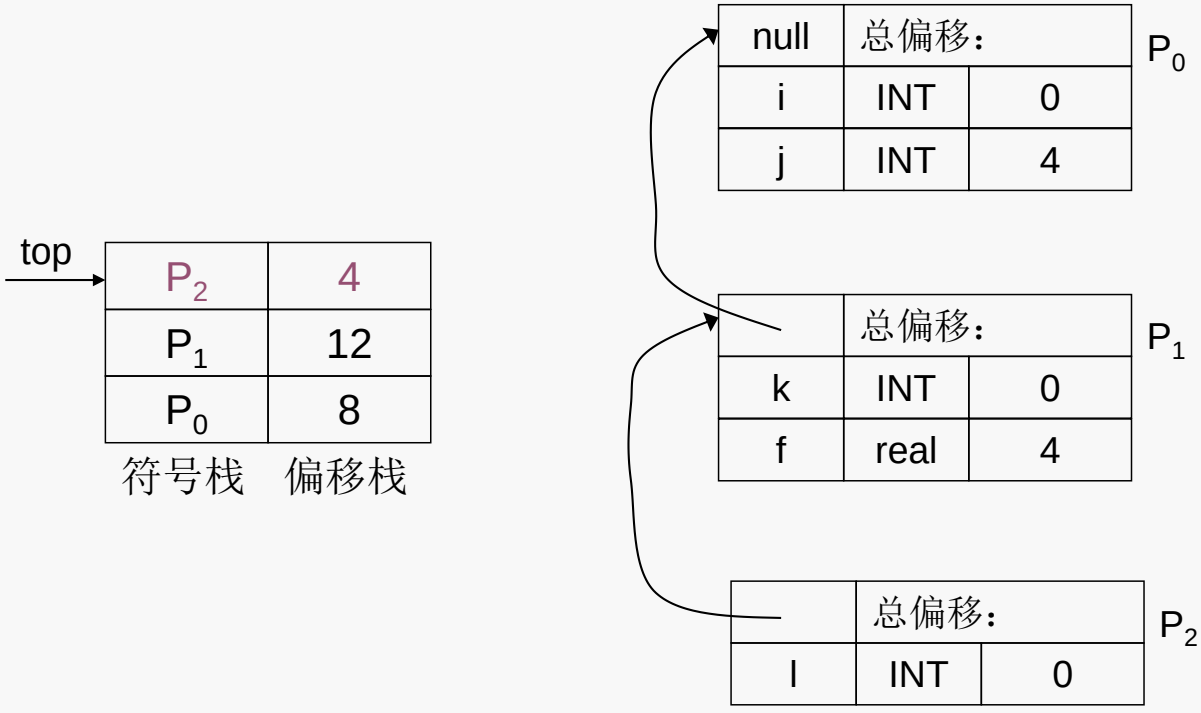
- PROC P_2 ; （N→ε）





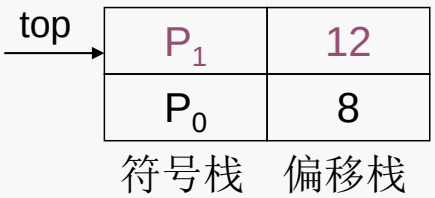
举例：过程嵌套声明

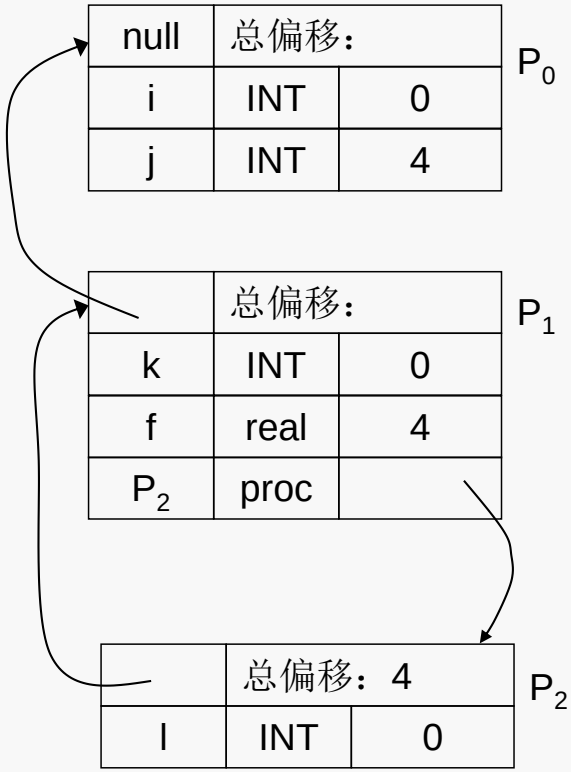
- l : int ;



举例：过程嵌套声明

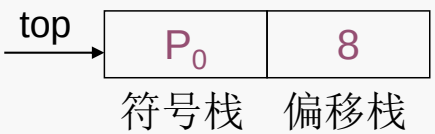
- a_1；

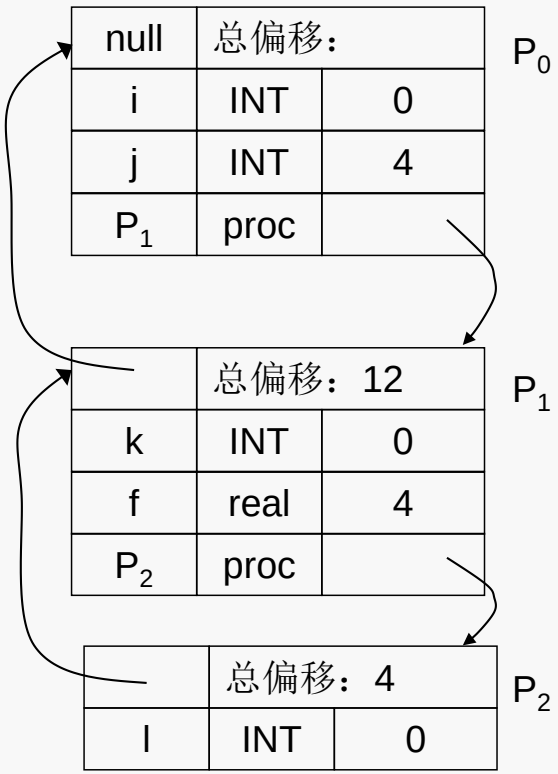

top
P1
12
P0
8
符号栈
偏移栈


null
总偏移：
i
INT
0
j
INT
4
P0
总偏移：
k
INT
0
f
real
4
P2
proc
P1
总偏移：4
l
INT
0
P2

举例：过程嵌套声明

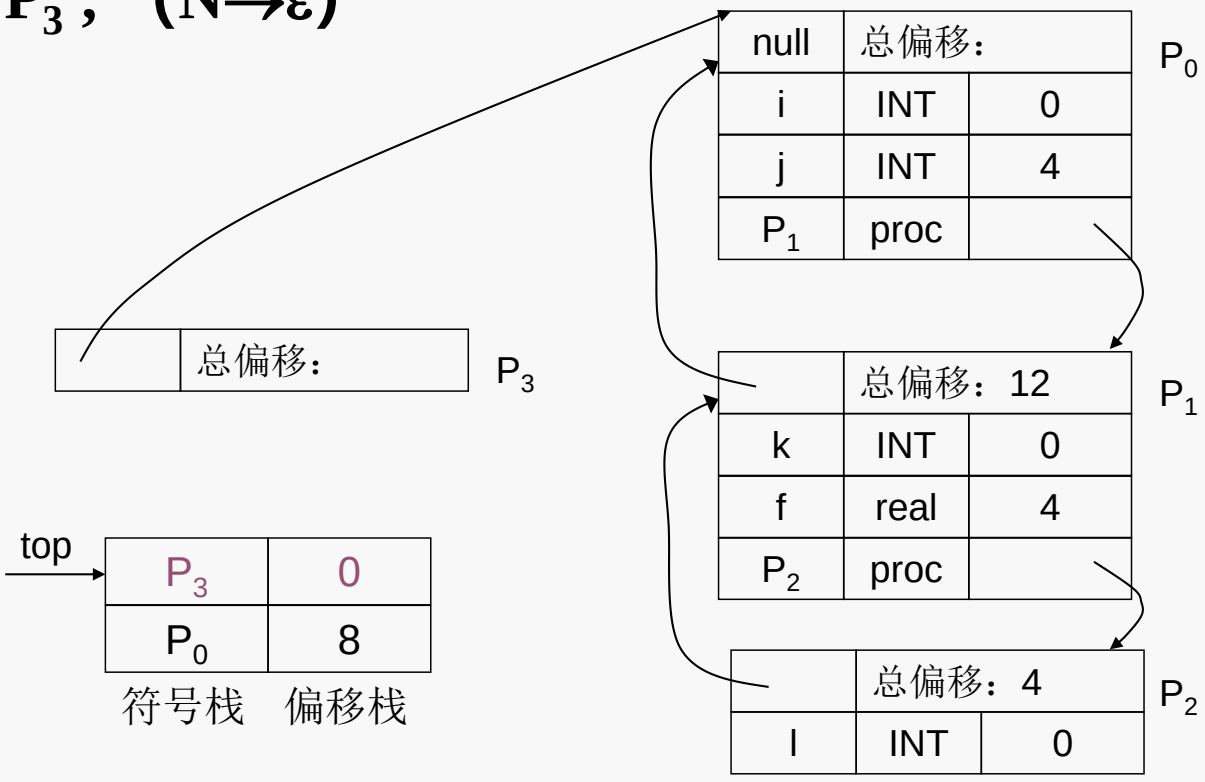
- a_2 ;


top
P0
8
符号栈
偏移栈


null
总偏移：
P0
i
INT
0
j
INT
4
P1
proc
总偏移：12
P1
k
INT
0
f
real
4
P2
proc
总偏移：4
P2
l
INT
0

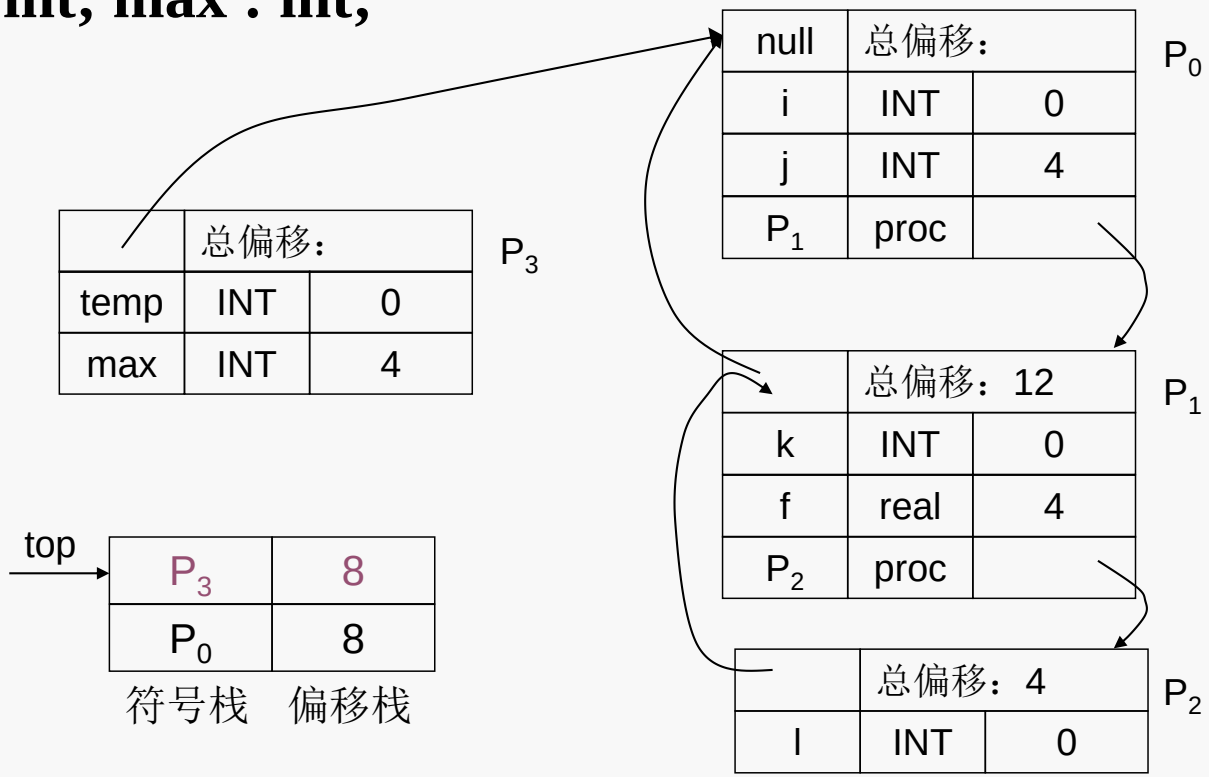
举例：过程嵌套声明

- PROC P_3；（N→ε）



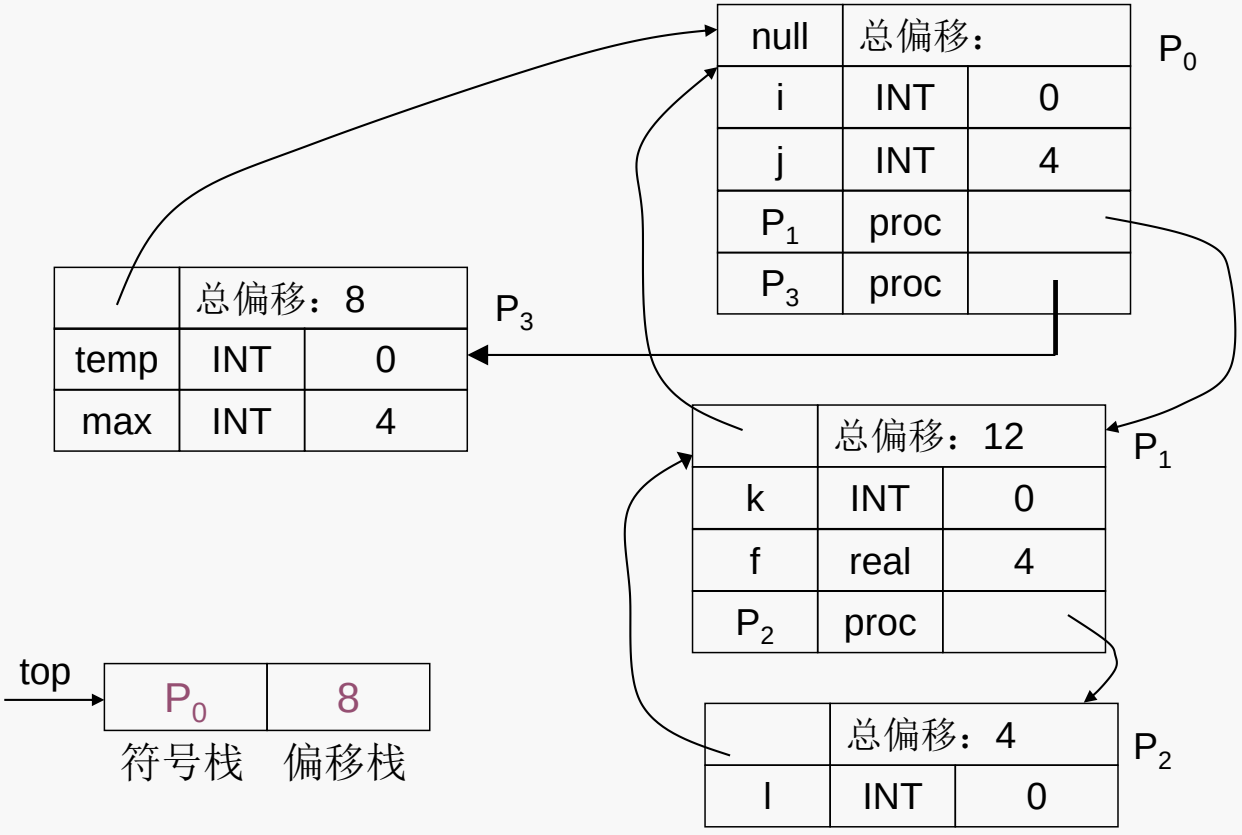
举例：过程嵌套声明

- temp : int; max : int;



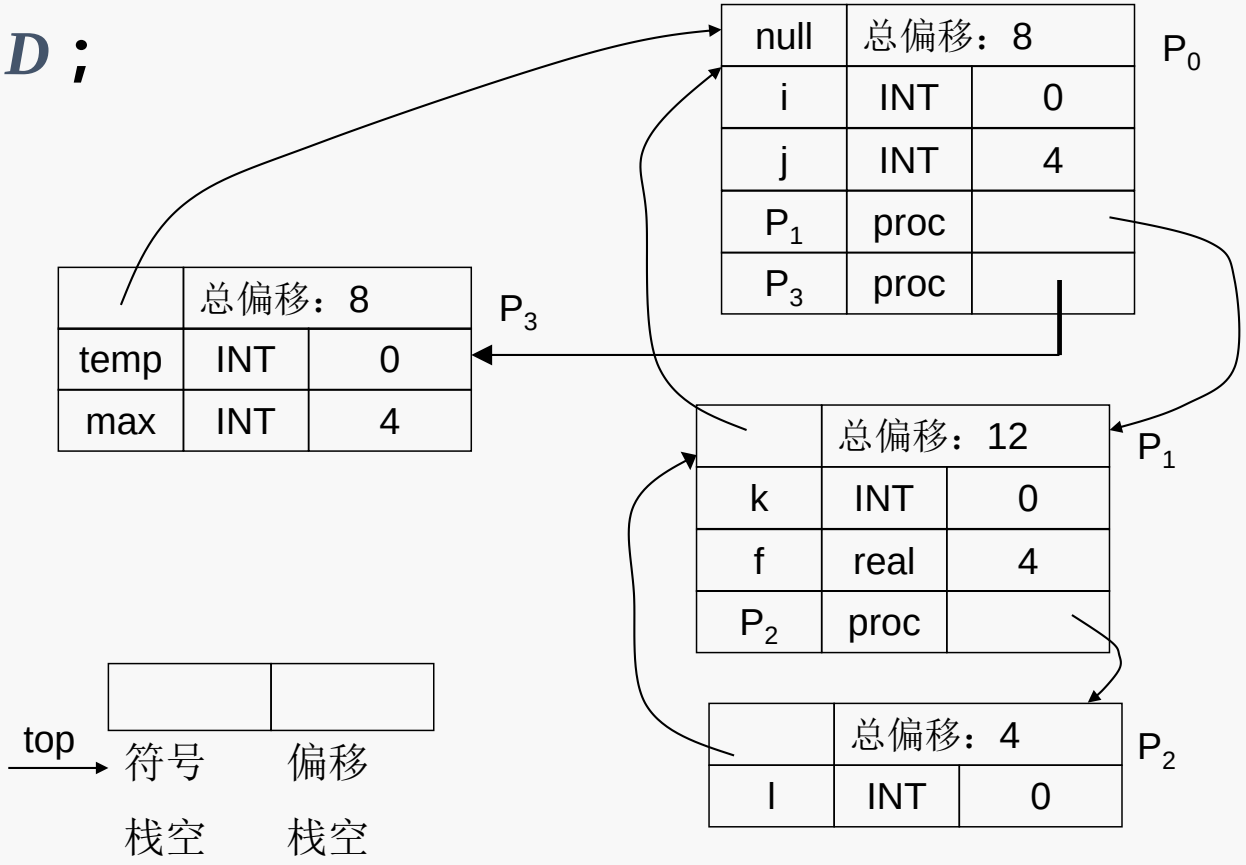
举例：过程嵌套声明

- a_3;


null
总偏移：
P0
i
INT
0
j
INT
4
P1
proc
P3
proc
总偏移：8
P3
temp
INT
0
max
INT
4
总偏移：12
P1
k
INT
0
f
real
4
P2
proc
top
P0
8
符号栈
偏移栈
总偏移：4
P2
l
INT
0

举例：过程嵌套声明

- P→ M *D* ;



声明语句翻译的要点

- 分配存储单元
  - 名字、类型、字宽、偏移
- 作用域的管理
  - 过程调用
- 记录类型的管理
- 不产生中间代码指令，但是要更新符号表

记录的域名管理

- 描述记录的文法

$T \rightarrow$ record D end

记录类型单独建符号表，域的相对地址从0开始

$T \rightarrow$ record L D end

{*T.type* = *record* (*top*(*tblptr*));
T.width = *top*(*offset*);
pop(*tblptr*); *pop*(*offset*) }

$L \rightarrow \varepsilon$ {*t* = *mkTable*(*nil*);
push(*t*, *tblprt*); *push*(0, *offset*) }

```
record
   a :…;
   r : record
          i :…;
          . . .
       end;
    k : …;
end
```

记录的域名管理

描述记录的文法

$T \rightarrow$ record D end

记录类型单独建符号表，域的相对地址从0开始

$T \rightarrow$ record L D end

{*T.type = record (top(tblptr));*
T.width = top(offset);
pop(tblptr); pop(offset) }

$L \rightarrow \varepsilon$ { *t = mkTable(nil);*
push(t, tblptr); push(0, offset) }

```
record
   a :…;
   r : record
          i :…;
          . . .
       end;
    k : …;
end
```

建立符号表，进入作用域

记录的域名管理

描述记录的文法

$T \rightarrow$ record D end

记录类型单独建符号表，域的相对地址从0开始

$T \rightarrow$ record L D end

{*T.type = record (top(tblptr));*

T.width = top(offset);

pop(tblptr); pop(offset) }

$L \rightarrow \varepsilon$ { *t = mkTable(nil);*

push(t, tblptr); push(0, offset) }

```
record
   a :…;
   r : record
          i :…;
          . . .
       end;
    k : …;
end
```

设置记录的类型表达式和宽度，退出作用域

记录的域名管理

描述记录的文法

$T \rightarrow$ record D end

记录类型单独建符号表，域的相对地址从0开始

$T \rightarrow$ record L D end

{*T.type* = *record* (*top*(*tblptr*));
T.width = *top*(*offset*);
pop(*tblptr*); *pop*(*offset*) }

$L \rightarrow \varepsilon$ { *t* = *mkTable*(*nil*);
push(*t*, *tblptr*); *push*(0, *offset*) }

D的翻译同前

```
record
    a :…;
    r : record
            i :…;
            . . .
        end;
     k : …;
end
```

举例：记录域的偏移

- 有2个C语言的结构定义如下：

```
struct A {
    char c1;
    char c2;
    long l;
    double d;
} S1;
```

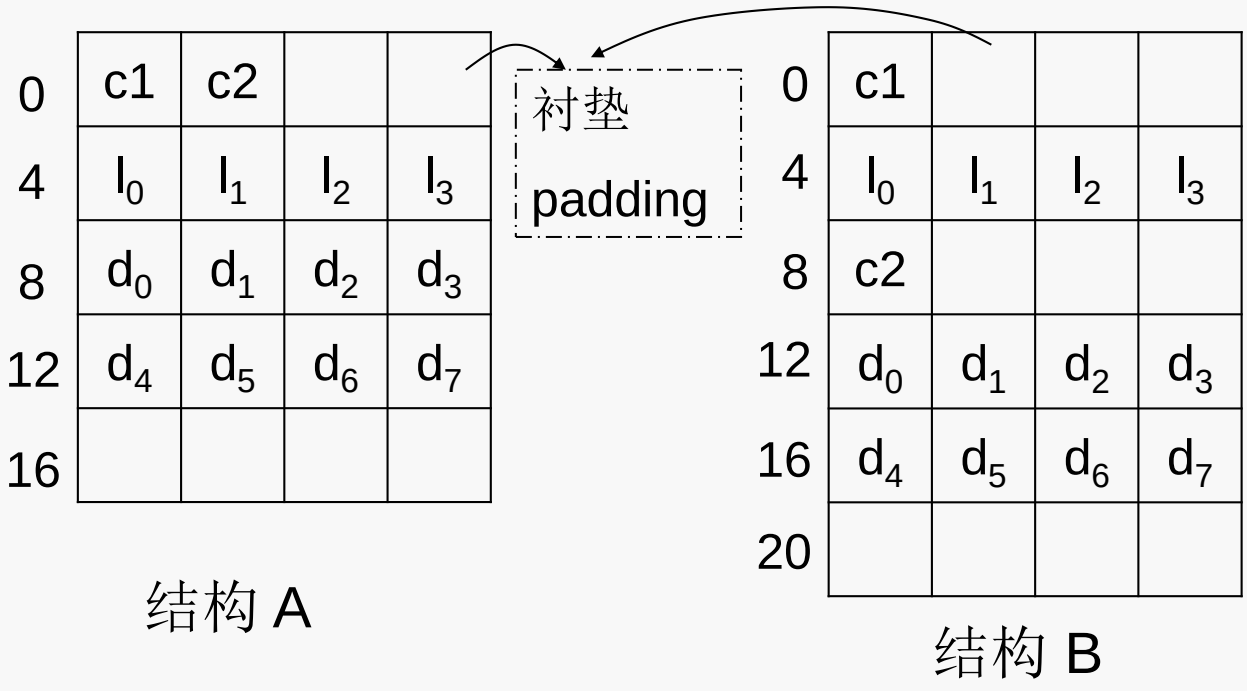
```
struct B {
    char c1;
    long l;
    char c2;
    double d;
} S2;
```

举例：记录域的偏移

- **数据（类型）的对齐 - alignment**
- **在 X86-Linux下：**
  - char：对齐1，起始地址可分配在任意地址
  - int，long，double：对齐4，即从被4整除的地址开始分配
- **注*：其它类型机器，double可能对齐到8**
  - 如sun－SPARC

举例：记录域的偏移

- 结构A 和 B的大小分别为16和20字节(Linux)



举例：记录域的偏移

- 2个结构中域变量的偏移如下：

```
struct A {
    char c1;   0
    char c2;   1
    long l;    4
    double d;  8
} S1;
```

```
struct B {
    char c1;    0
    long l;     4
    char c2;    8
    double d;  12
} S2;
```

《编译原理和技术》

中间代码生成 VI

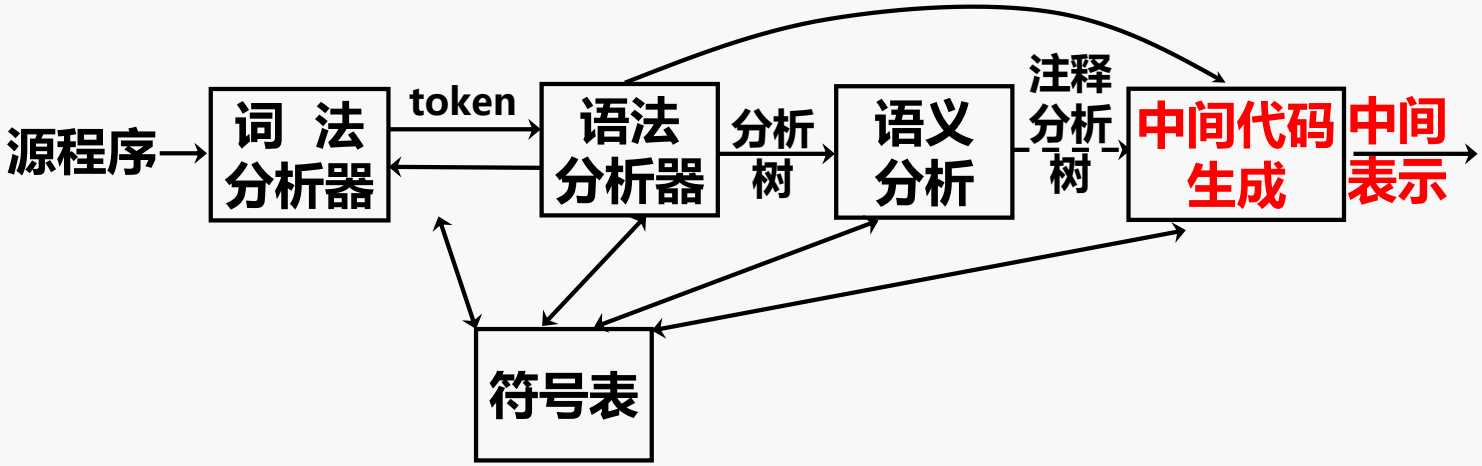
谢谢！

《编译原理和技术》

中间代码生成VII

中科大计算机学院
李 诚
2022-10-26

本节提纲



- **数组寻址的翻译**
  - 数组元素地址的计算
  - 数组元素地址计算翻译方案
  - 举例说明

数组元素的翻译

- **数组类型的声明**

e.g. Pascal的数组声明，

A : array[low_1..$high_1$,…,low_n..$high_n$] of integer ;

数组元素：A[i , j, k,…] 或 A[i][j][k]…

（下界）$low_1 \le i \le high_1$（上界） , …

e.g. C的数组声明，

int A [100][100][100];

数组元素：A[i][30][40] 0 ≤ i ≤ (100-1)

数组元素的翻译

- **翻译的主要任务**
  - 输出(gen/emit)地址计算的指令
  - “基址[偏移]”相关的中间指令：t =b[o]，b[o]=t

数组元素的地址计算

- **一维数组A的第*i*个元素的地址计算**

  $base + (i - low) \times w$

  base: 整个数组的基地址，也是分配给该数组的内存块的相对地址

  ***low*: 下标的下界**

  w: 每个数组元素的宽度

可以变换成

$i \times w + (base - low \times w)$

$low \times w$是常量，编译时计算，减少了运行时计算

数组元素的地址计算

- **一维数组A的第*i*个元素的地址计算**

$$base + (i - low) \times w$$

base: 整个数组的基地址，也是分配给该数组的内存块的相对地址

***low*: 下标的下界**

w: 每个数组元素的宽度

可以变换成

$$i \times w + (base - low \times w)$$

***low* × *w*是常量，编译时计算，减少了运行时计算**

数组元素的地址计算

- **二维数组**

  A: array[1..2, 1..3] of T

- **列为主**

  A[1, 1], A[2, 1], A[1, 2], A[2, 2], A[1, 3], A[2, 3]

- **行为主**

  A[1, 1], A[1, 2], A[1, 3], A[2, 1], A[2, 2], A[2, 3]

数组元素的地址计算

- **二维数组**

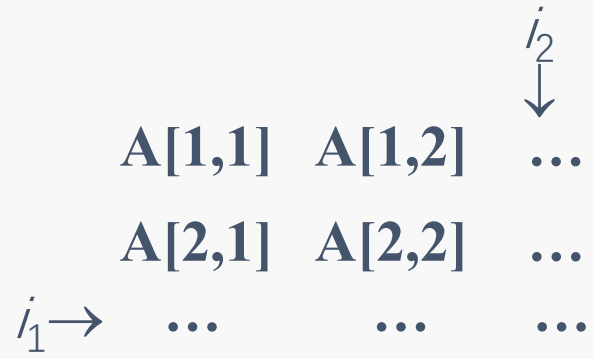
  A: array[1..2, 1..3] of T

  - **列为主**

    A[1, 1], A[2, 1], A[1, 2], A[2, 2], A[1, 3], A[2, 3]

  - **行为主**

    A[1, 1], A[1, 2], A[1, 3], A[2, 1], A[2, 2], A[2, 3]



数组元素的地址计算

- **二维数组**

  A: array[1..2, 1..3] of T

  - **列为主**

    A[1, 1], A[2, 1], A[1, 2], A[2, 2], A[1, 3], A[2, 3]

  - **行为主**

    A[1, 1], A[1, 2], A[1, 3], A[2, 1], A[2, 2], A[2, 3]

$$base + ((i_1 - low_1) \times n_2 + (i_2 - low_2)) \times w$$

（A[i_1, i_2]的地址，其中$n_2 = high_2 - low_2 + 1$）

变换成 $((i_1 \times n_2) + i_2) \times w +$

$$(base - ((low_1 \times n_2) + low_2) \times w)$$

i_1 \ i_2		
	A[1,1]	A[1,2]
	A[2,1]	A[2,2]
	…	…

数组元素的地址计算

- 多维数组下标变量A[i_1, i_2, ..., i_k]的地址表达式
  - 以行为主

$$((\ldots((i_1 \times n_2 + i_2) \times n_3 + i_3) \ldots) \times n_k + i_k) \times w$$

$$+ base - ((\ldots ((low_1 \times n_2 + low_2) \times n_3 + low_3) \ldots) \times n_k + low_k) \times w$$

数组元素的地址计算

- 多维数组下标变量A[i_1, i_2, ..., i_k]的地址表达式
  - 以行为主

$$((\ldots((i_1 \times n_2 + i_2) \times n_3 + i_3) \ldots) \times n_k + i_k) \times w$$

$$+ base - ((\ldots ((low_1 \times n_2 + low_2) \times n_3 + low_3) \ldots) \times n_k + low_k) \times w$$

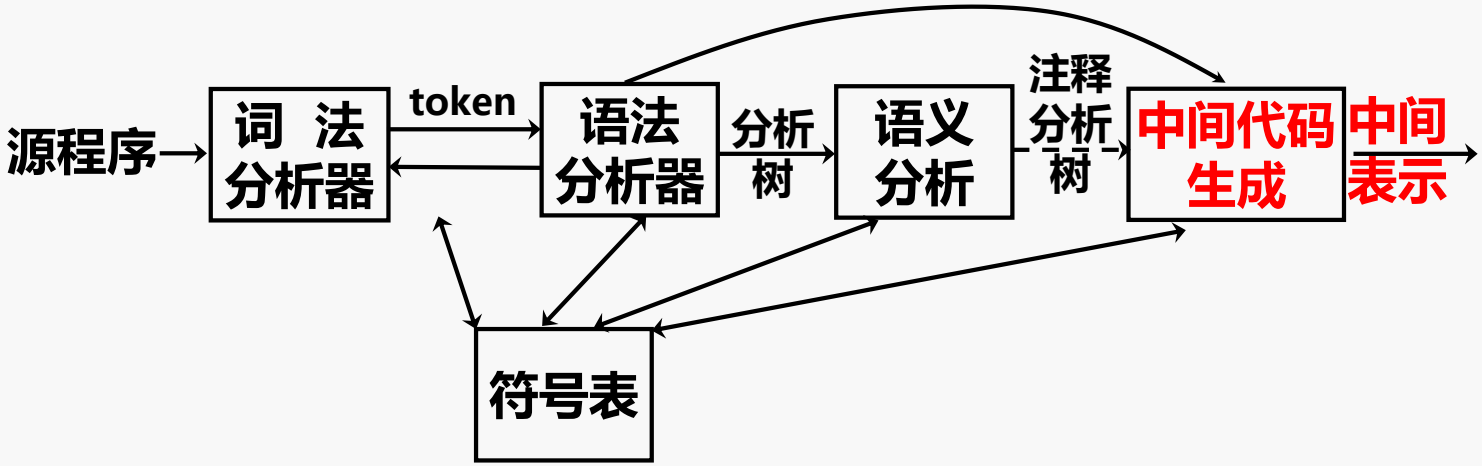
红色部分是数组访问翻译中的最重要的内容

递推公式：

$e_1 = i_1$

$e_m = e_{m-1} \times n_m + i_m$

本节提纲



- 数组寻址的翻译
  - 数组元素地址的计算
  - 数组元素地址计算翻译方案
  - 举例说明

数组元素地址计算翻译方案

- 下标变量访问的产生式

$S \rightarrow L := E$　　$L \rightarrow \text{id}\ [\ Elist\ ] \mid \text{id}$

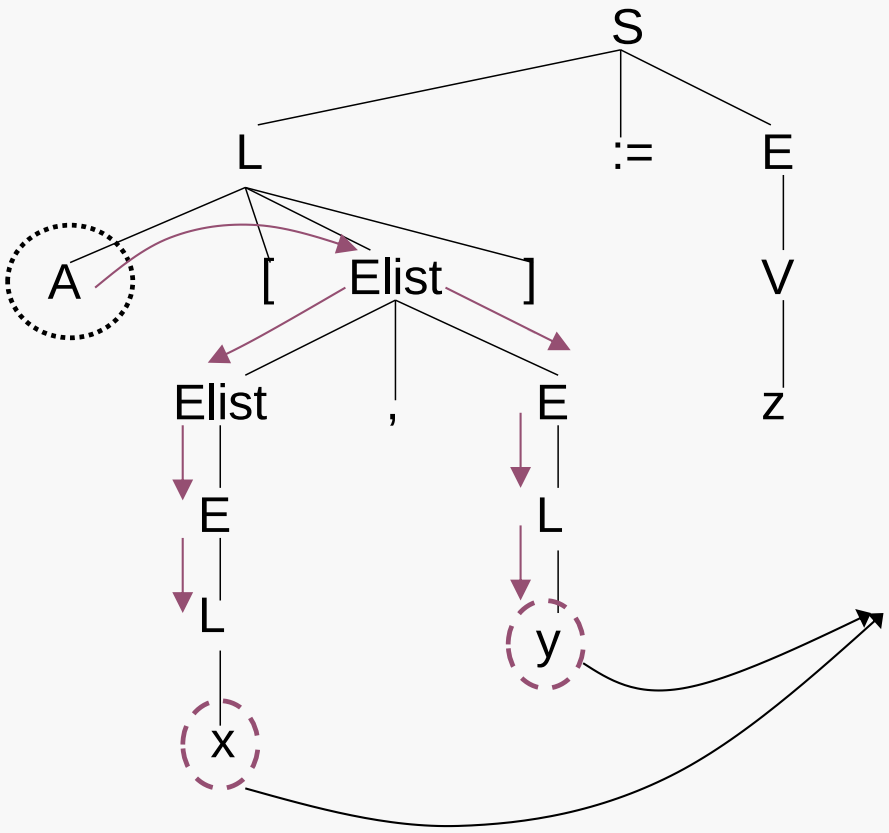
$Elist \rightarrow Elist, E \mid E$　　$E \rightarrow \text{L} \mid \ldots$

- 采用语法制导的翻译方案时存在的问题

$Elist \rightarrow Elist, E \mid E$

由Elist的结构只能得到各维的下标值，但无法获得数组的信息（如各维的长度）

A[x,y] := z的分析树



当分析到下标（表达式）x和y时，要计算地址中的“可变部分”。这时需要知晓数组A的有关的属性，如n_m，类型宽度w等，而这些信息存于在结点A处。若想使用必须定义有关继承属性来传递之。

但在移进－归约分析不适合继承属性的计算！

修改文法

- 所有产生式

$S \rightarrow L := E$

$E \rightarrow E + E$

$E \rightarrow (E)$

$E \rightarrow L$

$L \rightarrow Elist\ ]$

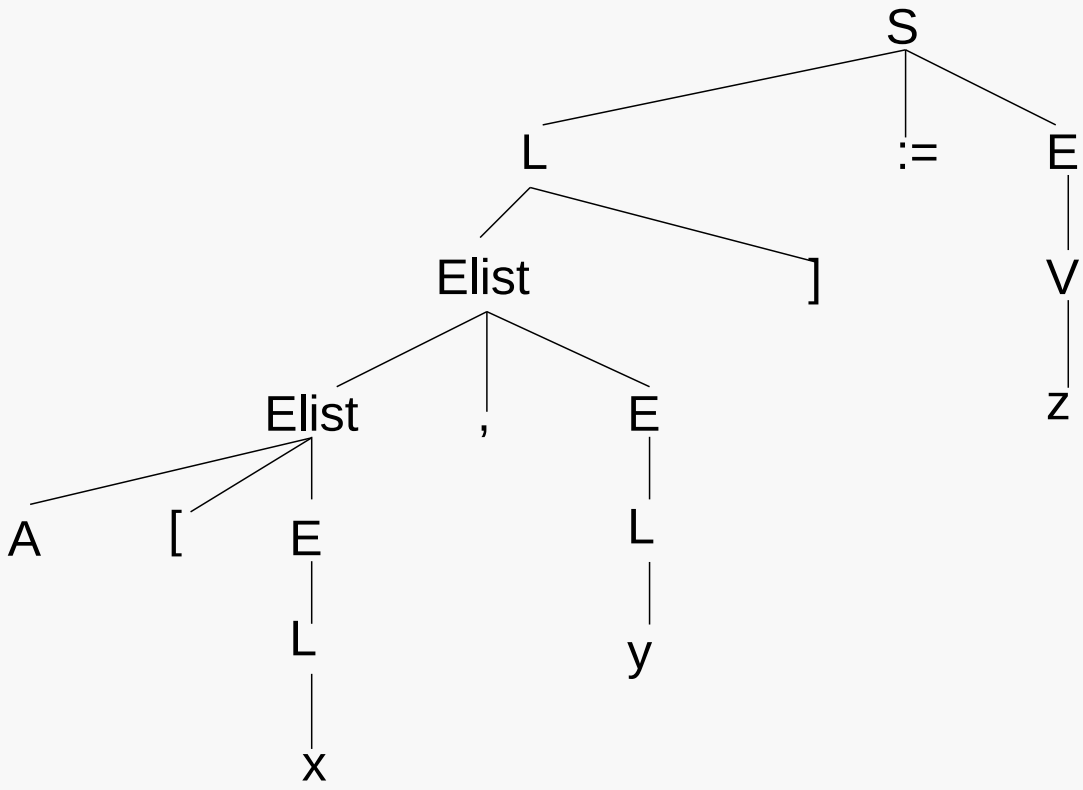
$L \rightarrow \mathrm{id}$

$Elist \rightarrow Elist, E$

$Elist \rightarrow \mathrm{id}\ [\ E$

修改文法，使数组名id成为Elist的子结点（类似于前面的类型声明），从而避免继承属性的出现

A[x,y] := z的分析树



相关符号属性定义：

L.place，L.offset ：

- 若L是简单变量，L.place为其“值”的存放场所，而L.offset为空（null）；
- 当L表示数组元素时，L.place是其地址的“常量值”部分；而此时L.offset为数组元素地址中可变部分的“值”存放场所，数组元素的表示为：**L.place [L.offset]**

相关符号属性定义：

Elist.place：存放“可变部分”值(下标计算值)的地址

Elist.array：数组名条目的指针，比如可以查询base

Elist.ndim　　：当前处理的维数

limit(array, j)：第j维的大小

width(array)：数组元素的宽度

invariant(array)：静态可计算的值，即紫书7.4公式

数组元素的翻译

- 翻译时重点关注三个表达式：
  - $Elist \rightarrow \text{id}\ [\ E$ ： 计算第1维
  - $Elist \rightarrow Elist_1, E$ ： 传递信息
  - $L \rightarrow Elist\ ]$ ： 计算最终结果

数组元素的翻译

$S \rightarrow L := E$ {if *L.offset* == null then /* *L*是简单变量 */

gen (*L.place*, '= ', *E.place*)

else

/*取数组元素的左值*/

gen (*L.place* , '[', *L.offset*, ']', '=', *E.place*) }

数组元素的翻译

Elist → id [*E* {*Elist.place* = *E.place*;

/*第一维下标*/

Elist.ndim = 1;

Elist.array = id.*place* }

数组元素的翻译

$Elist \rightarrow Elist_1, E$ {/*维度增加1*/

$m = Elist_1.ndim + 1;$

/* 第m维的大小*/

$n_m = limit(Elist_1.array, m);$

$t = newTemp();$

/*计算公式7.6 $e_{m-1} * n_m$ */

$gen(t, '=', Elist_1.place, '*', n_m);$

/*计算公式7.6 $e_m = e_{m-1} * n_m + i_m$*/

$gen\ (t, '=', t, '+', E.place);$

$Elist.array = Elist_1.array;$

$Elist.place = t;$

$Elist.ndim = m$}

数组元素的翻译

$L \rightarrow Elist$] { $L.place = newTemp()$;

/*获取数组元素地址的常量值*/

$gen\ (L.place,\ '=',\ invariant\ (Elist.array)\ )$;

$L.offset = newTemp()$;

/*获取数组元素地址的可变部分*/

$gen\ (L.offset,\ '=',\ Elist.place,\ '*',\ width(Elist.array))$}

数组元素的翻译

$L \rightarrow$ id {*L.place* = id.*place*; *L.offset* = null }

$E \rightarrow L$ { if *L.offset* == null then /* *L*是简单变量 */

E.place = *L.place*

else begin *E.place* = *newTemp*();

gen (*E.place*, '=', *L.place*, '[', *L.offset*, ']') end }

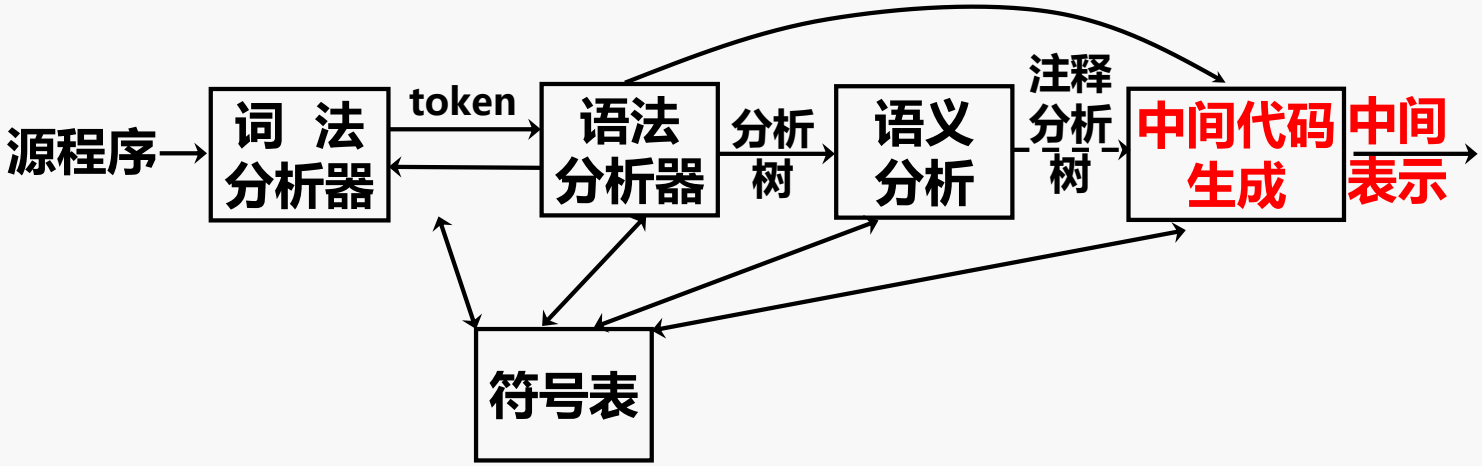
$E \rightarrow E_1 + E_2$ {*E.place* = *newTemp*();

gen (*E.place*, '=', E_1.*place* , '+', E_2.*place*) }

$E \rightarrow (E_1)$ {*E.place* = E_1.*place* }

其他翻译同前

本节提纲



- **数组寻址的翻译**
  - 数组元素地址的计算
  - 数组元素地址计算翻译方案
  - 举例说明

数组元素的翻译-举例

- 数组A的定义为：A[1…10, 1…20] of integer
- 数组的下界为1，即low为1
- 为赋值语句 x := A[y, z]生成中间代码

举例： x := A[y, z]

L.place = x
L.offset = null
|
x

A[1…10, 1…20] of integer

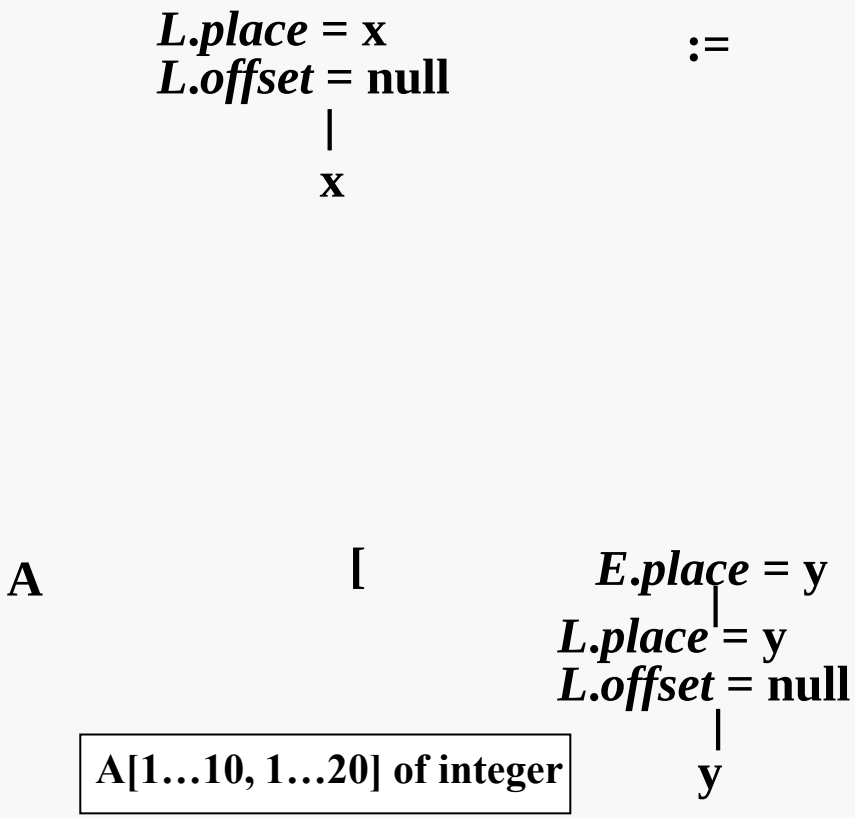
举例： x := A[y, z]

L.place = x
L.offset = null
|
x

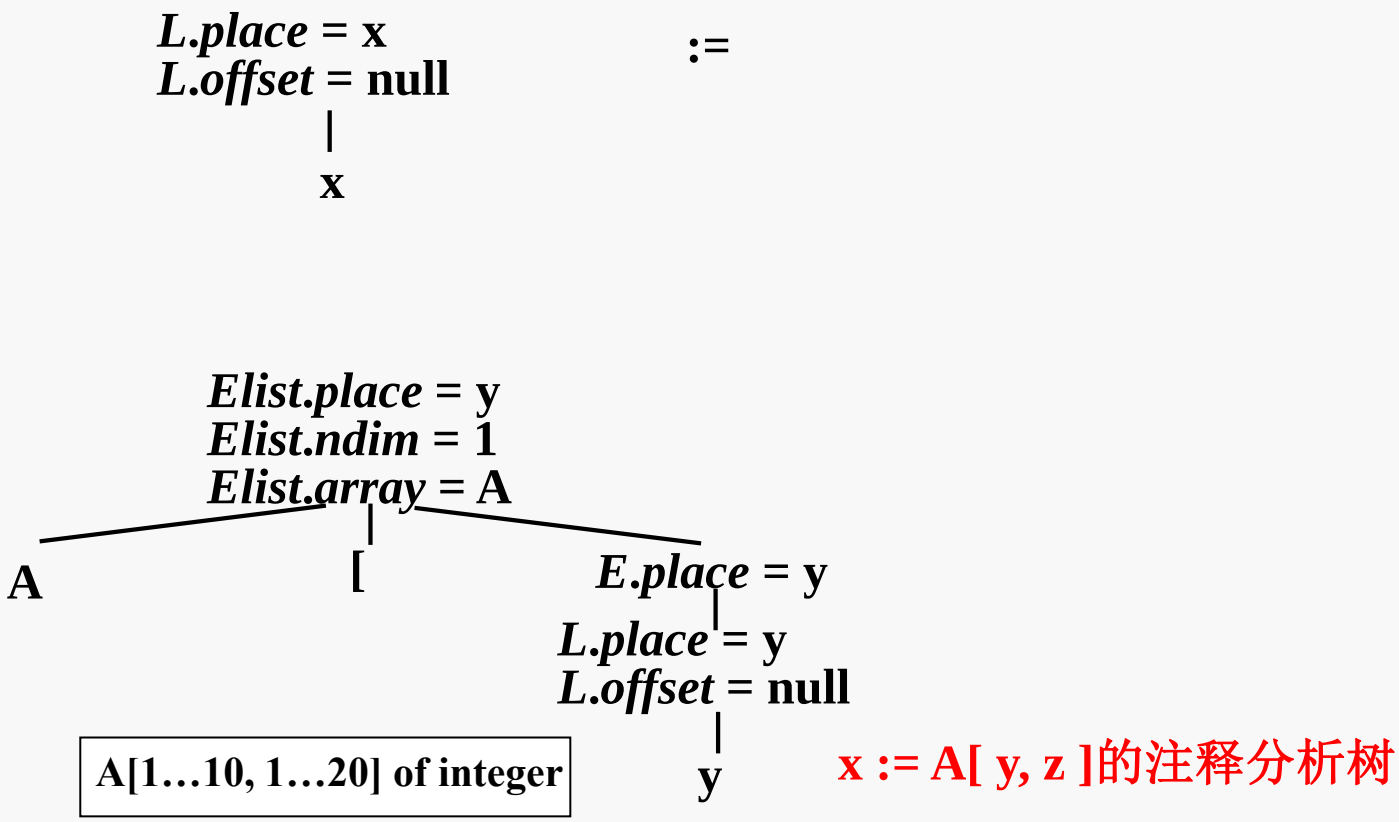
:=

A[1…10, 1…20] of integer

举例： x := A[y, z]

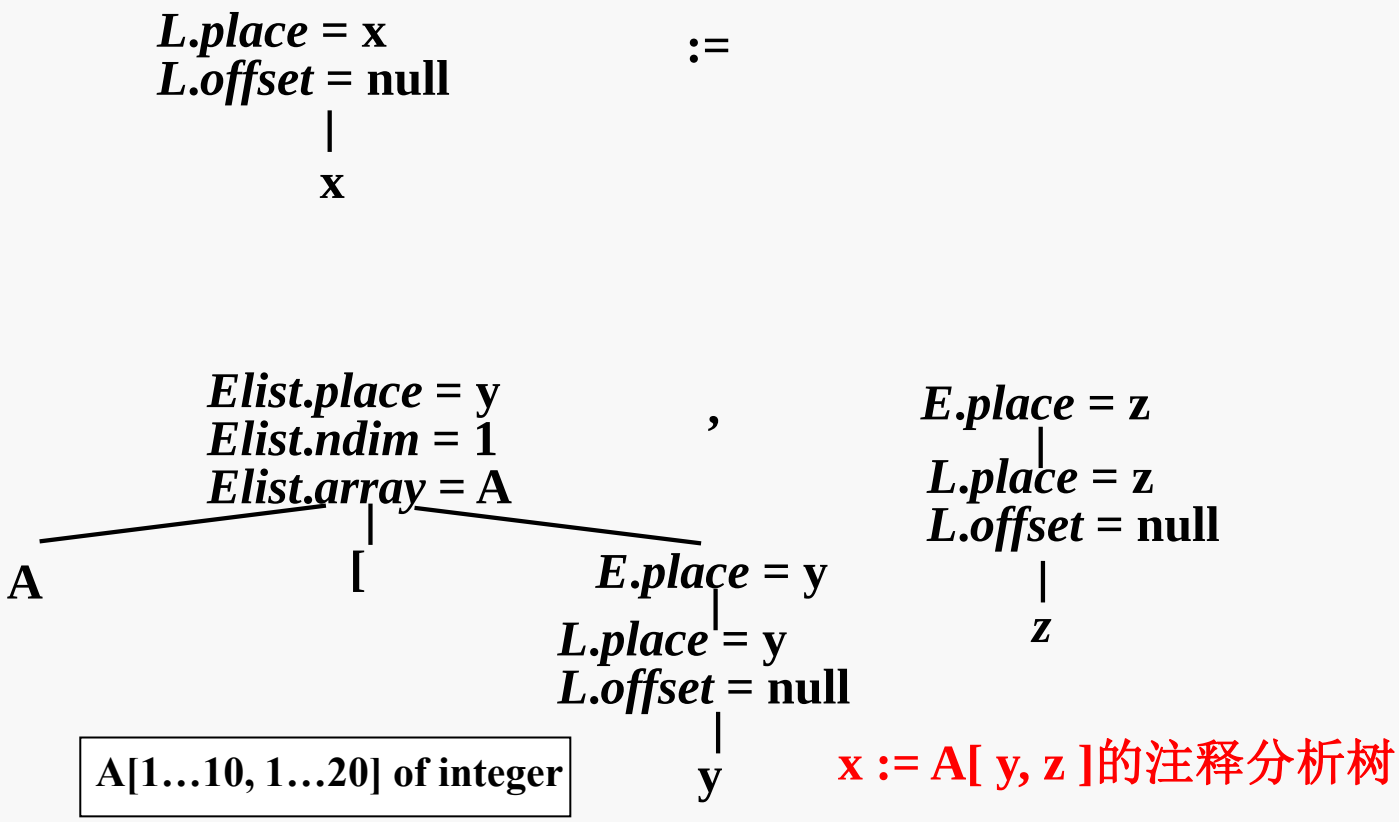

L.place = x
L.offset = null
x
:=
A
[
E.place = y
L.place = y
L.offset = null
y
A[1…10, 1…20] of integer

举例： x := A[y, z]



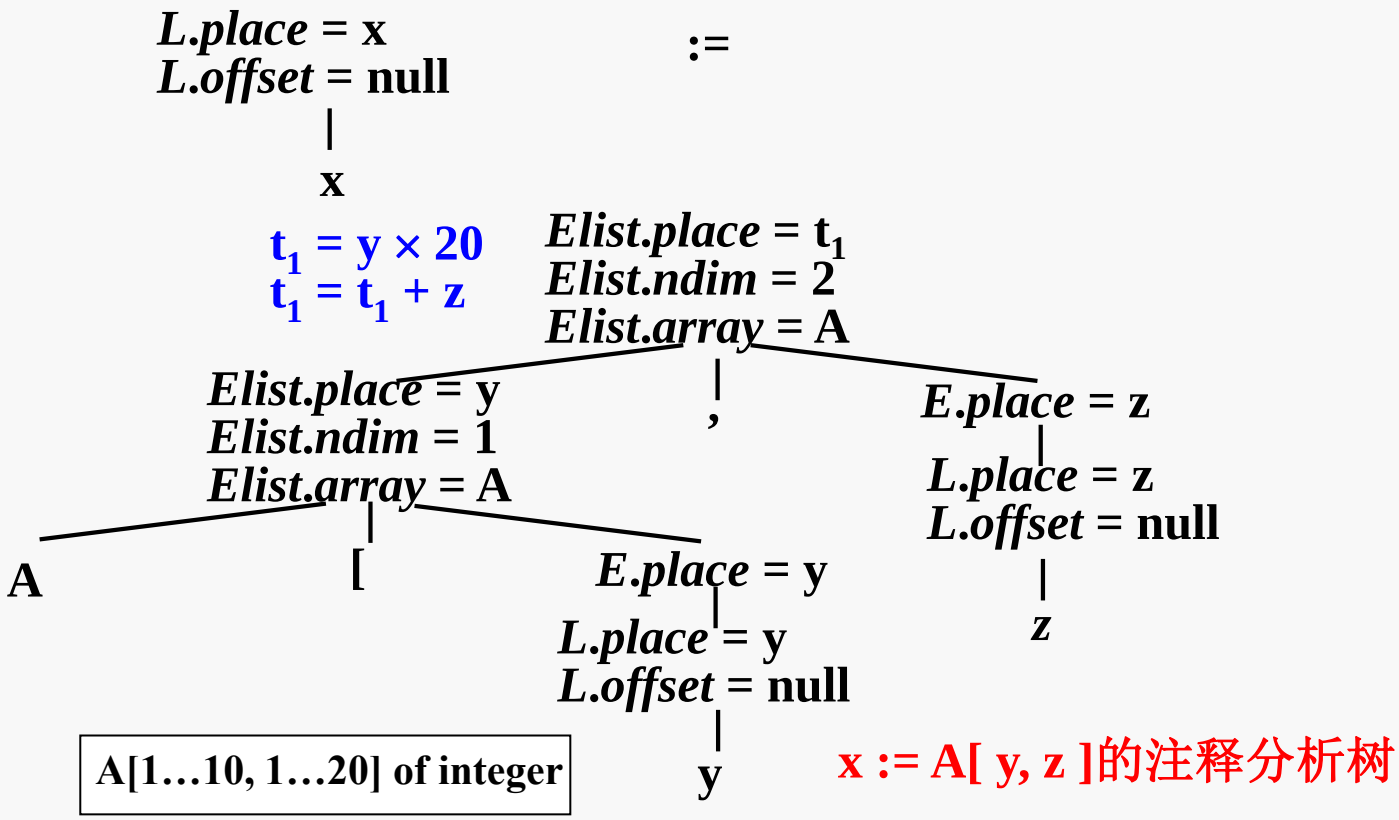
x := A[y, z]的注释分析树

举例： x := A[y, z]



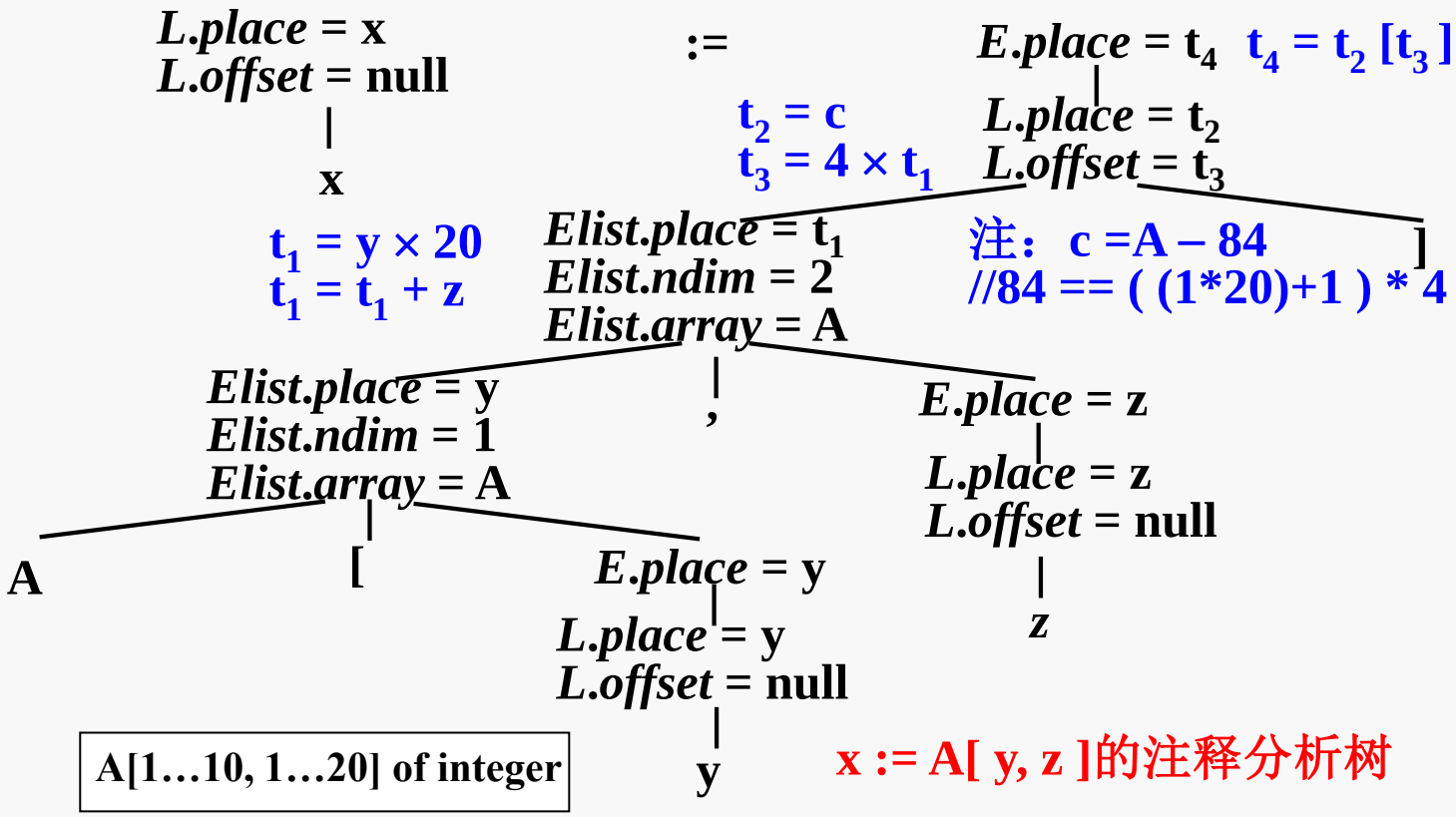
x := A[y, z]的注释分析树

举例： x := A[y, z]



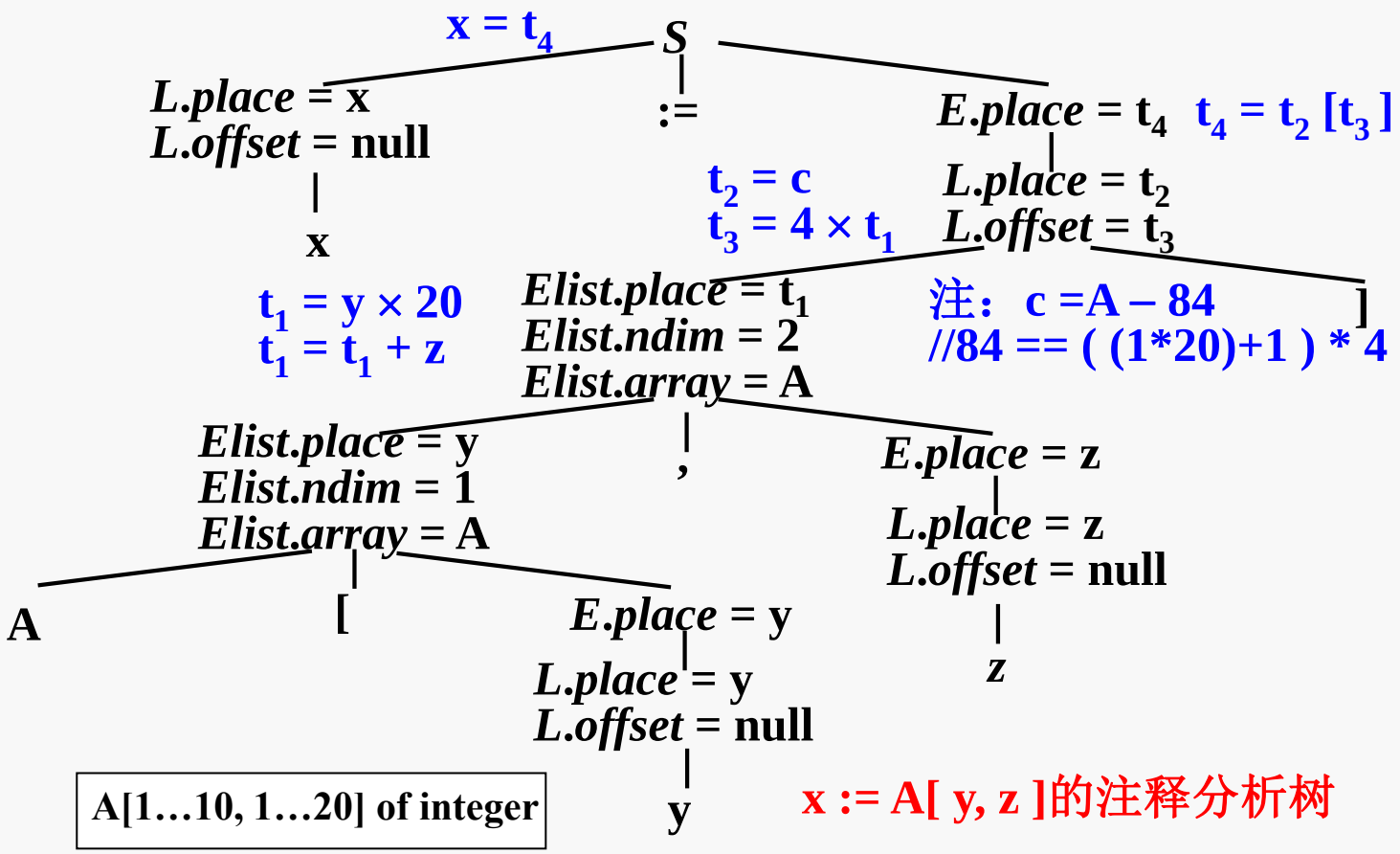
x := A[y, z]的注释分析树

举例： x := A[y, z]



x := A[y, z]的注释分析树

举例： x := A[y, z]



x := A[y, z]的注释分析树

举例：A[i, j] := B[i, j] * k

- 数组A：A[1..10, 1..20] of integer;

  数组B：B[1..10, 1..20] of integer;

  w ：4 （integer）

- TAC如下：

  (1) t_1 := i * 20

  (2) t_1 := t_1 + j

  (3) t_2 := A - 84 // 84 == ((1*20)+1) * 4

  (4) t_3 := t_1 * 4 // 以上A[i, j]的（左值）翻译

举例：A[i, j] := B[i, j] * k

TAC如下（续）：

(5) t_4 := i * 20

(6) t_4 := t_4 + j

(7) t_5 := B - 84

(8) t_6 := t_4 * 4

(9) t_7 := t_5[t_6]

//以上计算B[i,j]的右值

TAC如下（续）：

(10) t_8 := t_7 * k

//以上整个右值表达

//式计算完毕

(11) t_2[t_3] := t_8

// 完成数组元素的赋值

《编译原理和技术》

中间代码生成 VII

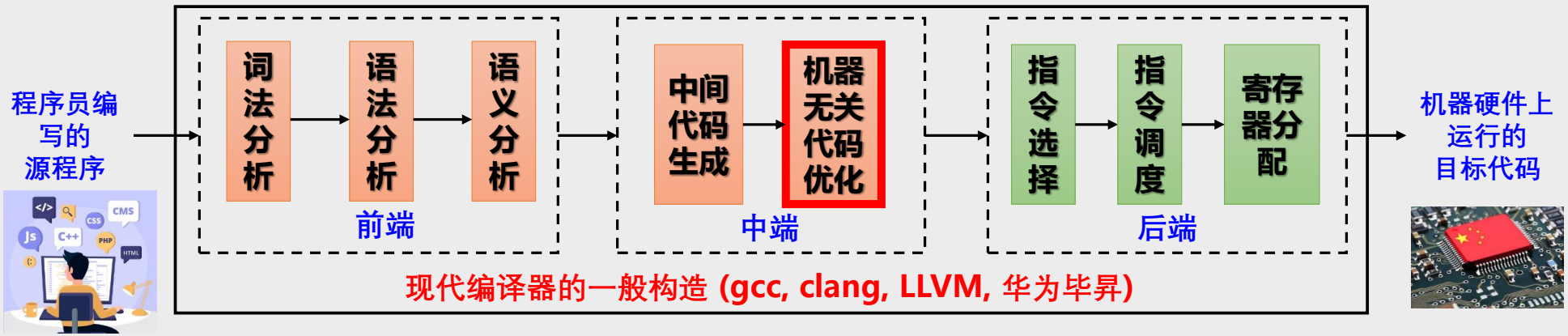
谢谢！

《编译原理和技术》

机器无关的代码优化 I

中科大计算机学院
李 诚
2022-10-31

本节提纲



- 代码优化的定义及背景
- 常见的优化方式
  - 公共子表达式删除优化
  - 死代码删除、复制传播、常量合并
  - 循环系列优化
    - 强度削弱、删除归纳变量、代码移动

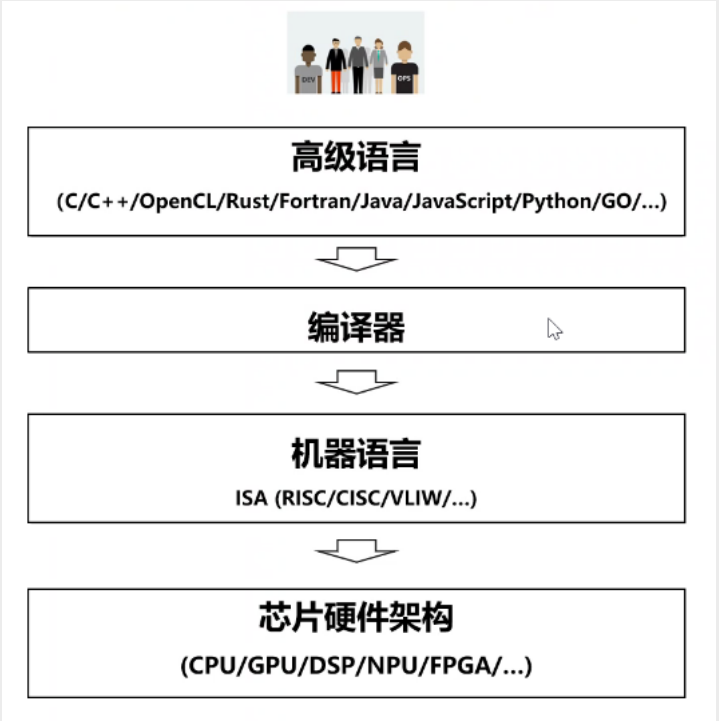
什么是代码优化?

- 在不改变程序运行效果的前提下，对程序代码进行**等价变换**，使之能**生成更加高效目标代码**。
- 优化目标：
  - 运行时间更短
  - 占用空间更小
- 代码优化是编写程序过程中不可或缺的关键环节！

为什么需要代码优化?

源头1：程序员编写的代码存在低效计算

源头2：代码翻译过程中，产生了冗余代码



- **高级语言**
  - 直接面向开发者
  - 与数学公式类似
  - 编程效率高
- **机器语言**
  - 驱动硬件完成具体任务
  - 编程效率低
- **编译器**
  - 实现人机交流，将人类易懂的高级语言翻译成硬件可执行的目标机器语言
  - 但目标代码可能运行效率低下

优化的源头和主要种类

- **程序中存在许多程序员无法避免的冗余运算**

  如A[i][j]和X.f1这样访问数组元素和结构体的域的操作

  - 编译后，这些访问操作展开成多步低级算术运算
  - 对同一个数据结构多次访问导致许多公共低级运算

举例——快速排序代码

- 排序是最基本、应用最广泛的可通过计算机高效求解的问题之一
- 快速排序是最经典、效率较高的排序算法之一

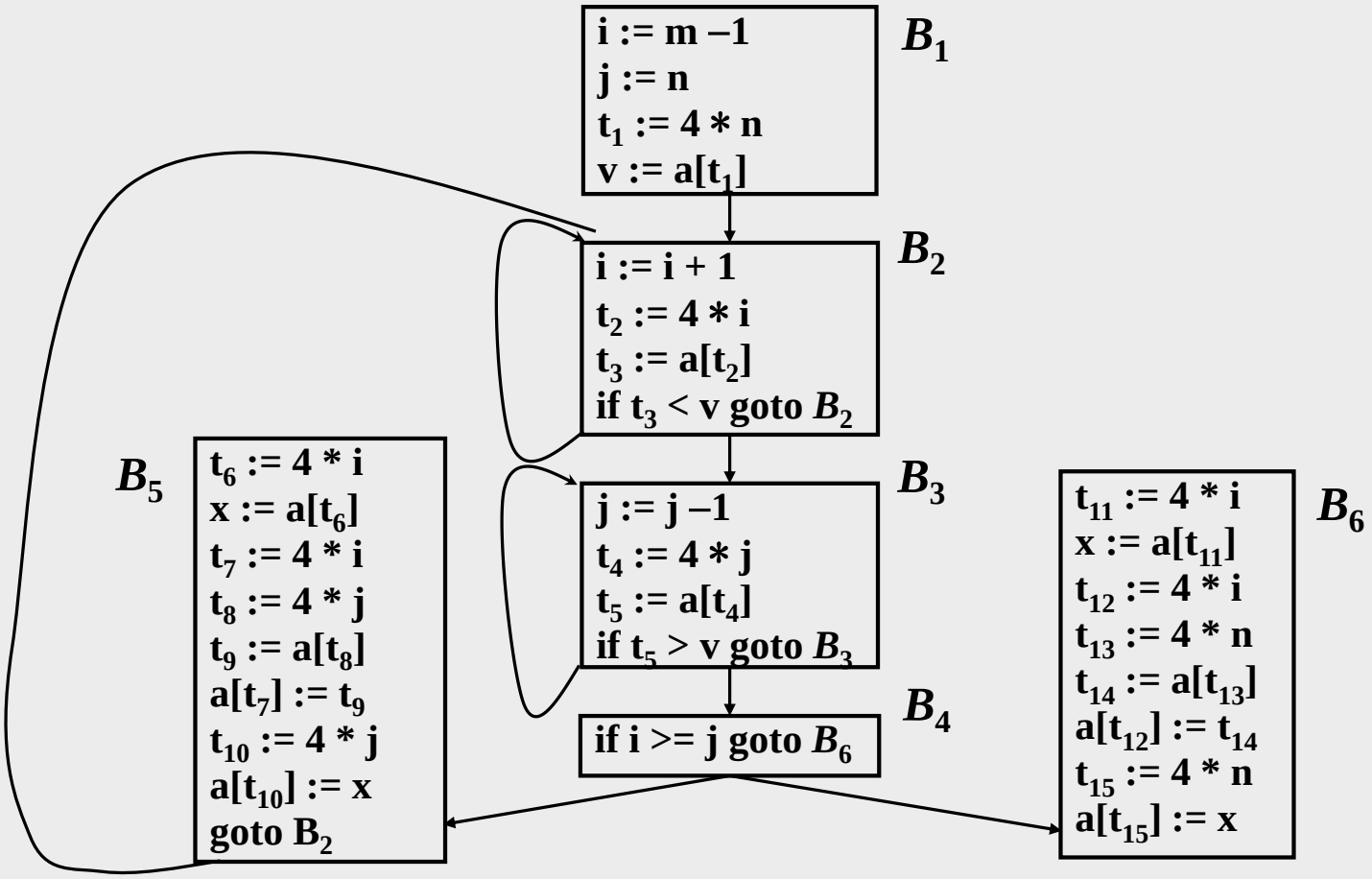
```
i = m – 1; j = n; v = a[n];
while (1) {
   do i = i +1; while(a[i] < v);
   do j = j - 1; while (a[j] > v);
   if (i  >=  j) break;
   x = a[i]; a[i] = a[j]; a[j] = x;
}
x = a[i]; a[i] = a[n]; a[n] = x;
```

高级语言代码

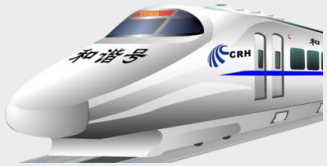
(1) i := m –1	(11) t5 := a[t4]	(21) a[t10] := x
(2) j := n	(12) if t5 > v goto (9)	(22) goto (5)
(3) t1 := 4 * n	(13) if i >= j goto (23)	(23) t11 := 4 * i
(4) v := a[t1]	(14) t6 := 4 * i	(24) x := a[t11]
(5) i := i + 1	(15) x := a[t6]	(25) t12 := 4 * i
(6) t2 := 4 * i	(16) t7 := 4 * i	(26) t13 := 4 * n
(7) t3 := a[t2]	(17) t8 := 4 * j	(27) t14 := a[t13]
(8) if t3 < v goto (5)	(18) t9 := a[t8]	(28) a[t12] := t14
(9) j := j –1	(19) a[t7] := t9	(29) t15 := 4 * n
(10) t4 := 4 * j	(20) t10 := 4 * j	(30) a[t15] := x

生成的目标机器代码

举例——快速排序代码


i := m –1
j := n
t1 := 4 * n
v := a[t1]
B1
i := i + 1
t2 := 4 * i
t3 := a[t2]
if t3 < v goto B2
B2
j := j –1
t4 := 4 * j
t5 := a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
t6 := 4 * i
x := a[t6]
t7 := 4 * i
t8 := 4 * j
t9 := a[t8]
a[t7] := t9
t10 := 4 * j
a[t10] := x
goto B2
B5
t11 := 4 * i
x := a[t11]
t12 := 4 * i
t13 := 4 * n
t14 := a[t13]
a[t12] := t14
t15 := 4 * n
a[t15] := x
B6

编译器的优化选项



优化等级	简要说明
-Ofast	在-O3级别的基础上，开启更多**激进优化项**，该优化等级不会严格遵循语言标准
-O3	在-O2级别的基础上，开启了更多的**高级优化项**，以编译时间、代码大小、内存为代价获取更高的性能。
-Os	在-O2级别的基础上，开启**降低生成代码体量**的优化
-O2	开启了大多数**中级优化**，会改善编译时间开销和最终生成代码性能
-O/-O1	优化效果介于-O1和-O2之间
-O0	默认优化等级，即**不开启编译优化**，只尝试减少编译时间

延伸阅读：https://clang.llvm.org/docs/CommandGuide/clang.html#code-generation-options

案例演示——优化对代码性能的影响

- 1000000000次循环迭代累加

```
#include <stdlib.h>
#include <time.h>
void main() {
    int loop = 1000000000;
    long sum = 0;
    int start_time = clock();
    int index = 0;
    for (index = 0; index < loop; index ++)
    {
        sum += index;
    }
    int end_time = clock();
    printf("Sum : %ld, Time Cost : %lf \n", sum, (end_time - start_time) * 1.0 / CLOCKS_PER_SEC);
}
```

循环次数定义

开始计时

循环体

结束计时

代码运行时间输出

案例演示——优化对代码性能的影响

- gcc –O0 无优化执行

```
gloit@gloit-x1c ~/2022_compiler_demo master gcc -O0 add.c
gloit@gloit-x1c ~/2022_compiler_demo master ./a.out
Sum: 499999999500000000, Time Cost: 3.415244
```

- gcc –O1 中级优化执行

```
gloit@gloit-x1c ~/2022_compiler_demo master gcc -O1 add.c
gloit@gloit-x1c ~/2022_compiler_demo master ./a.out
Sum: 499999999500000000, Time Cost: 0.554717
```

- gcc –O2 高级优化执行

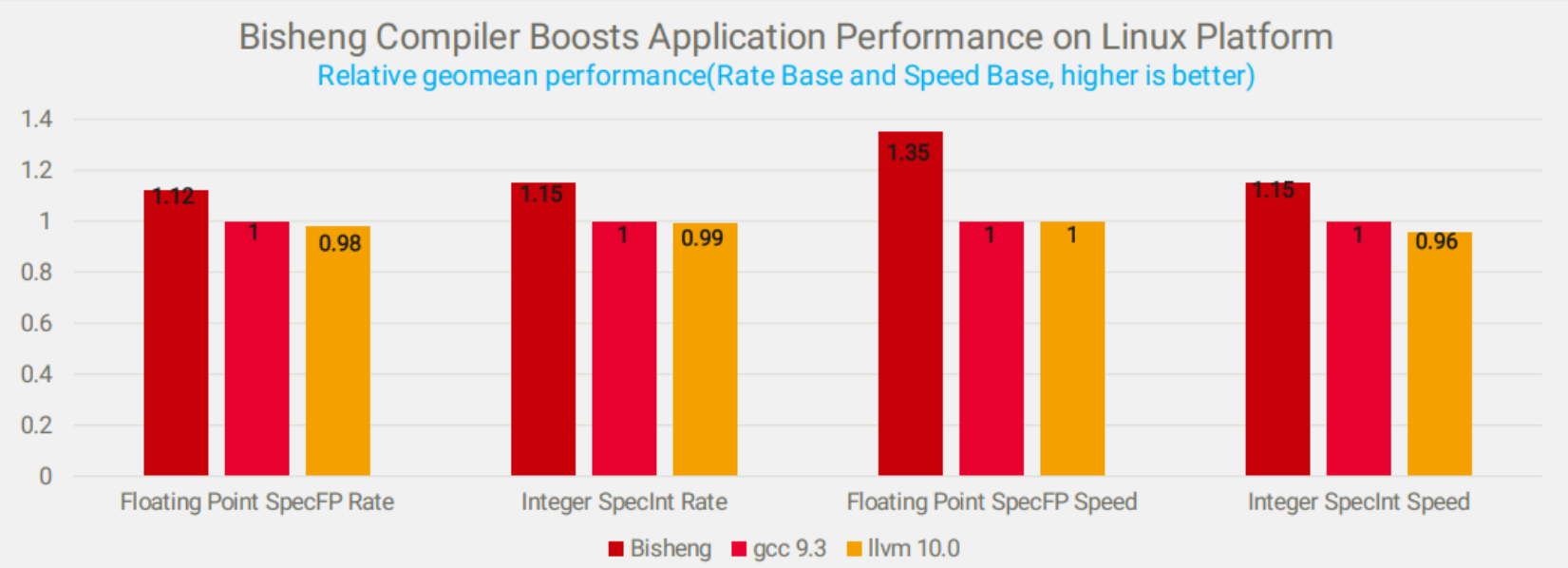
```
gloit@gloit-x1c ~/2022_compiler_demo master gcc -O2 add.c
gloit@gloit-x1c ~/2022_compiler_demo master ./a.out
Sum: 499999999500000000, Time Cost: 0.000002
```

性能提升5倍

性能提升数十万倍

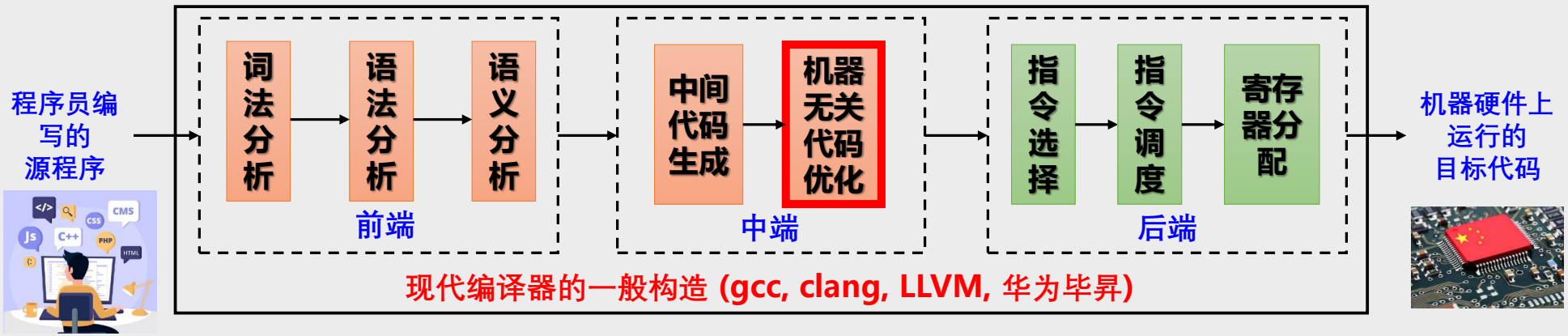
国产开源编译器——毕昇编译器

- 毕昇编译器通过**编译优化**提升鲲鹏硬件平台上业务的性能体验，SPEC2017性能较业界编译器平均高15%以上。



SPEC作为业界芯片性能评分标准，SPEC的分数可以直观的体现出硬件的性能，越高越好

本节提纲

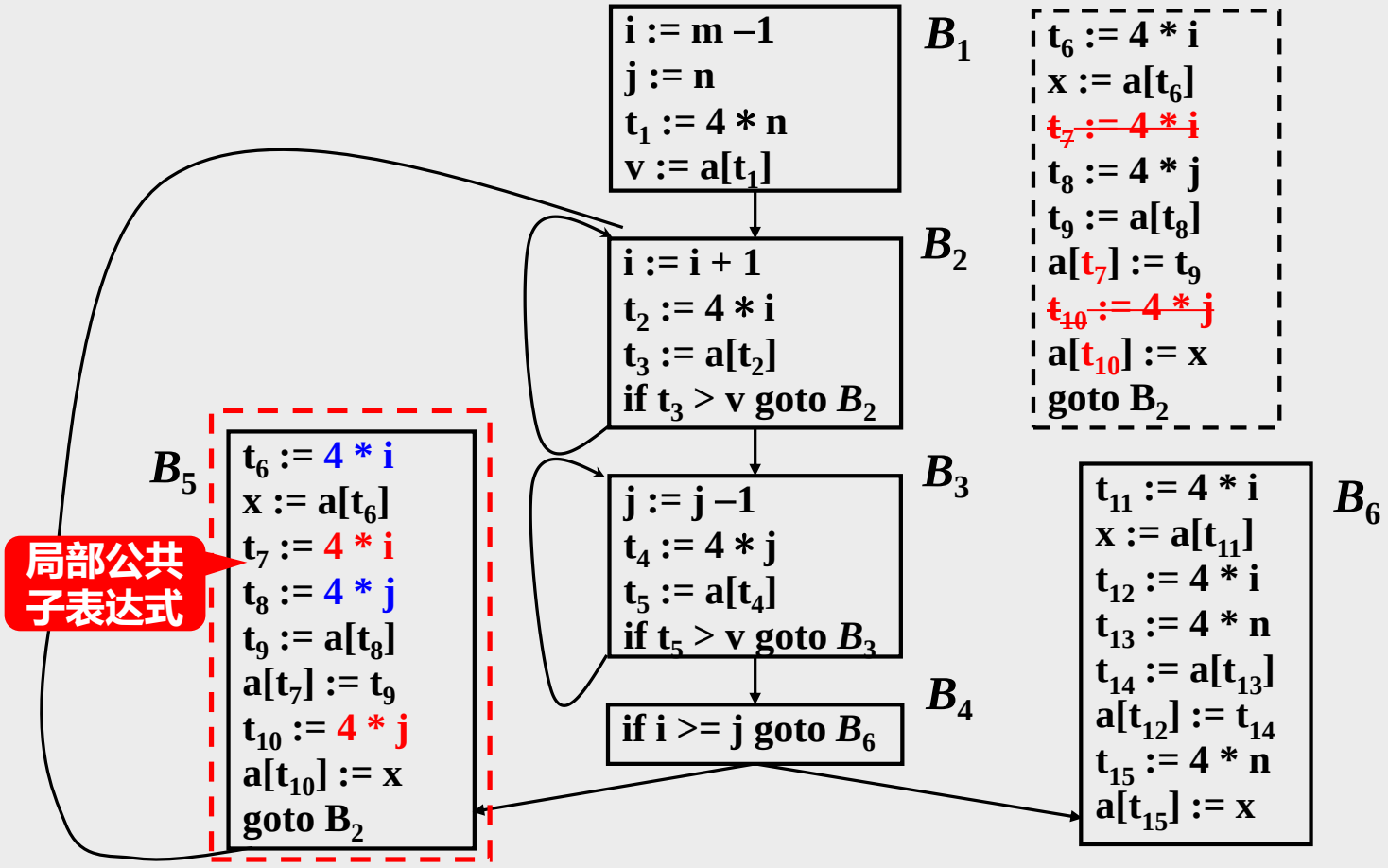


- 代码优化的定义及背景
- 常见的优化方式
  - 公共子表达式删除优化
  - 死代码删除、复制传播、常量合并
  - 循环系列优化
    - 强度削弱、删除归纳变量、代码移动

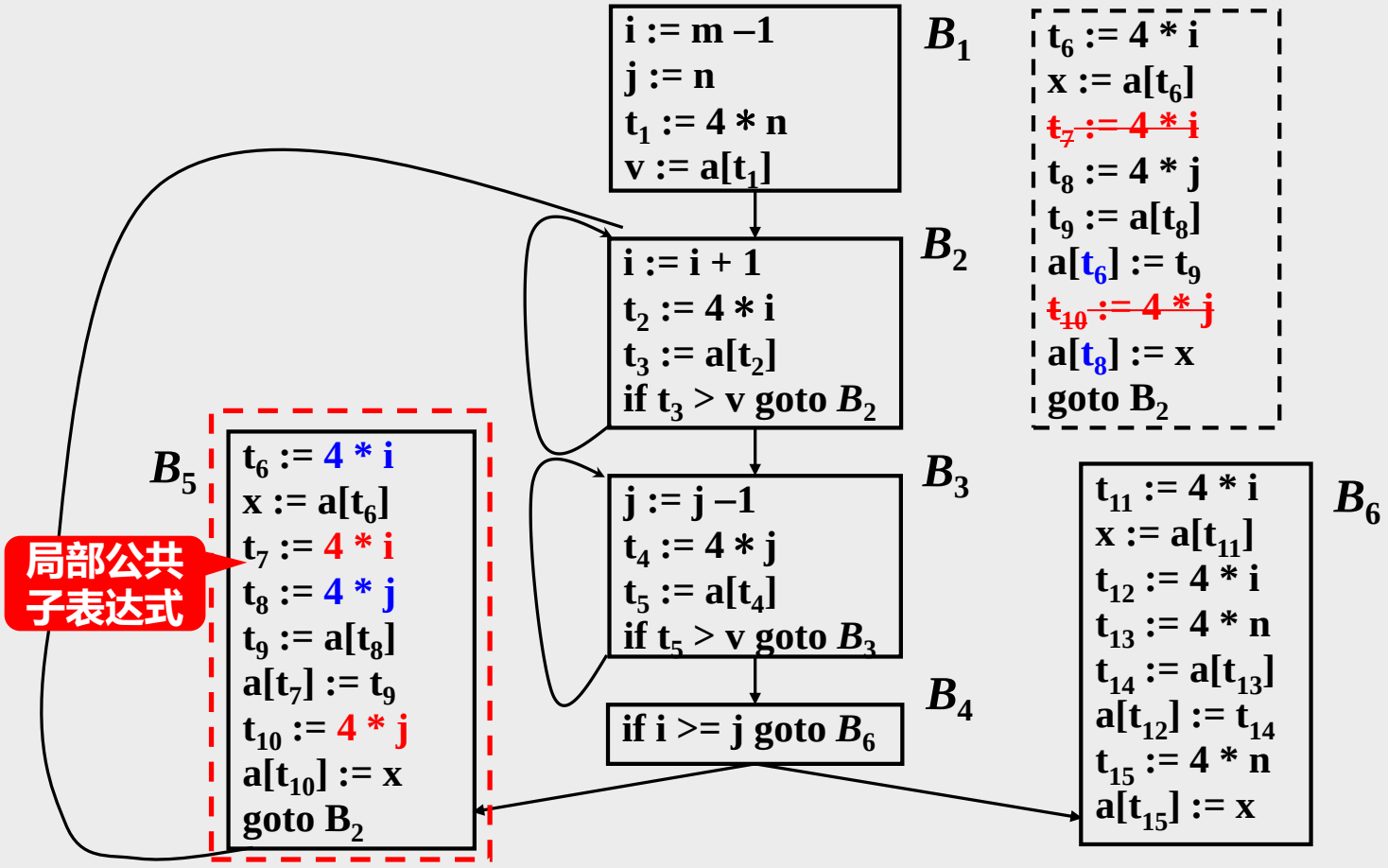
公共子表达式删除

- **公共子表达式**
  - 如x op y已被计算过，且到现在为止，x和y的值未变，那么该表达式的本次出现是公共子表达式
- **公共子表达式的删除**
  - 本次计算可以被删除
  - 其值用上一次计算值替代

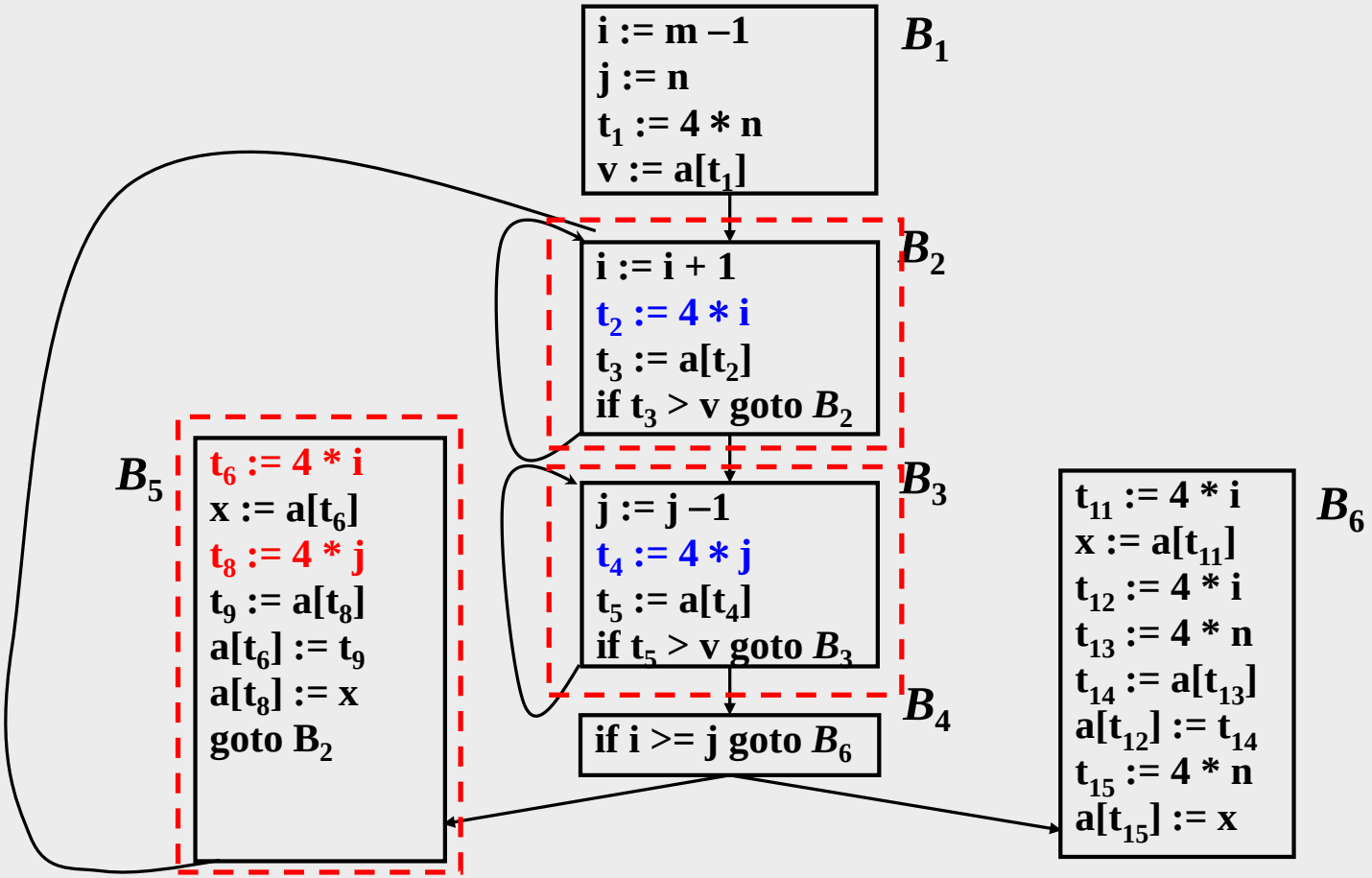
快排中的公共子表达式删除



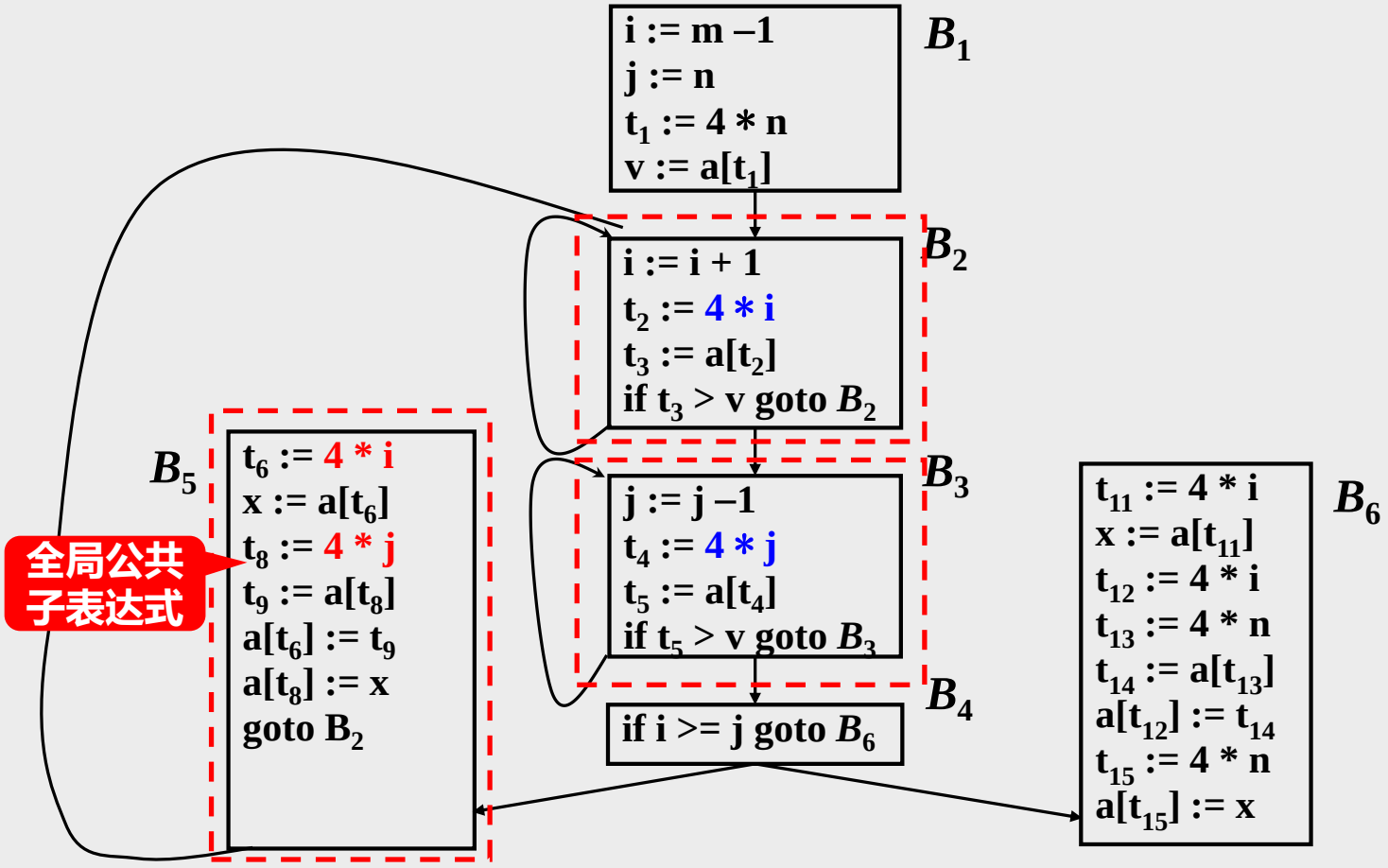
快排中的公共子表达式删除



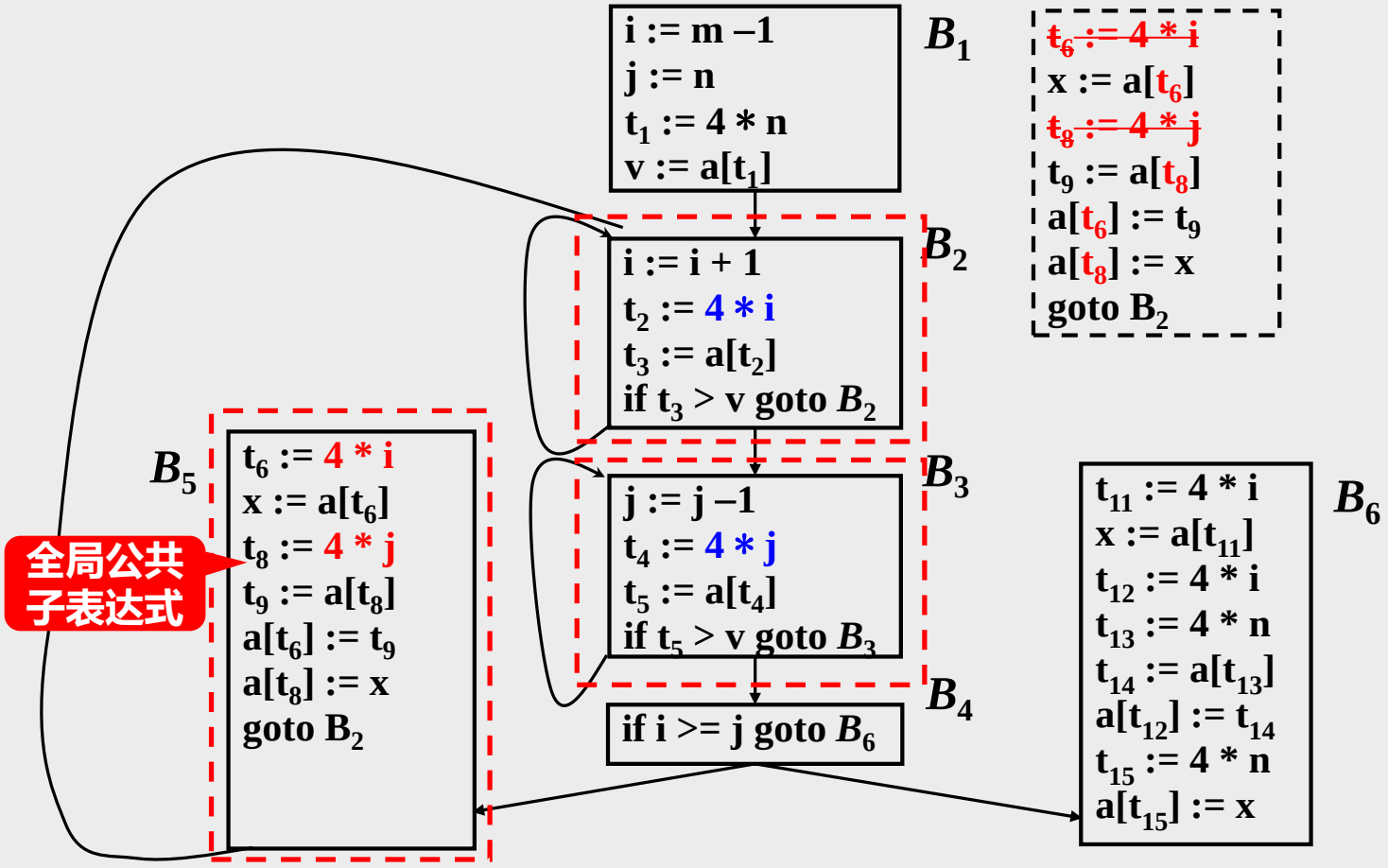
快排中的公共子表达式删除



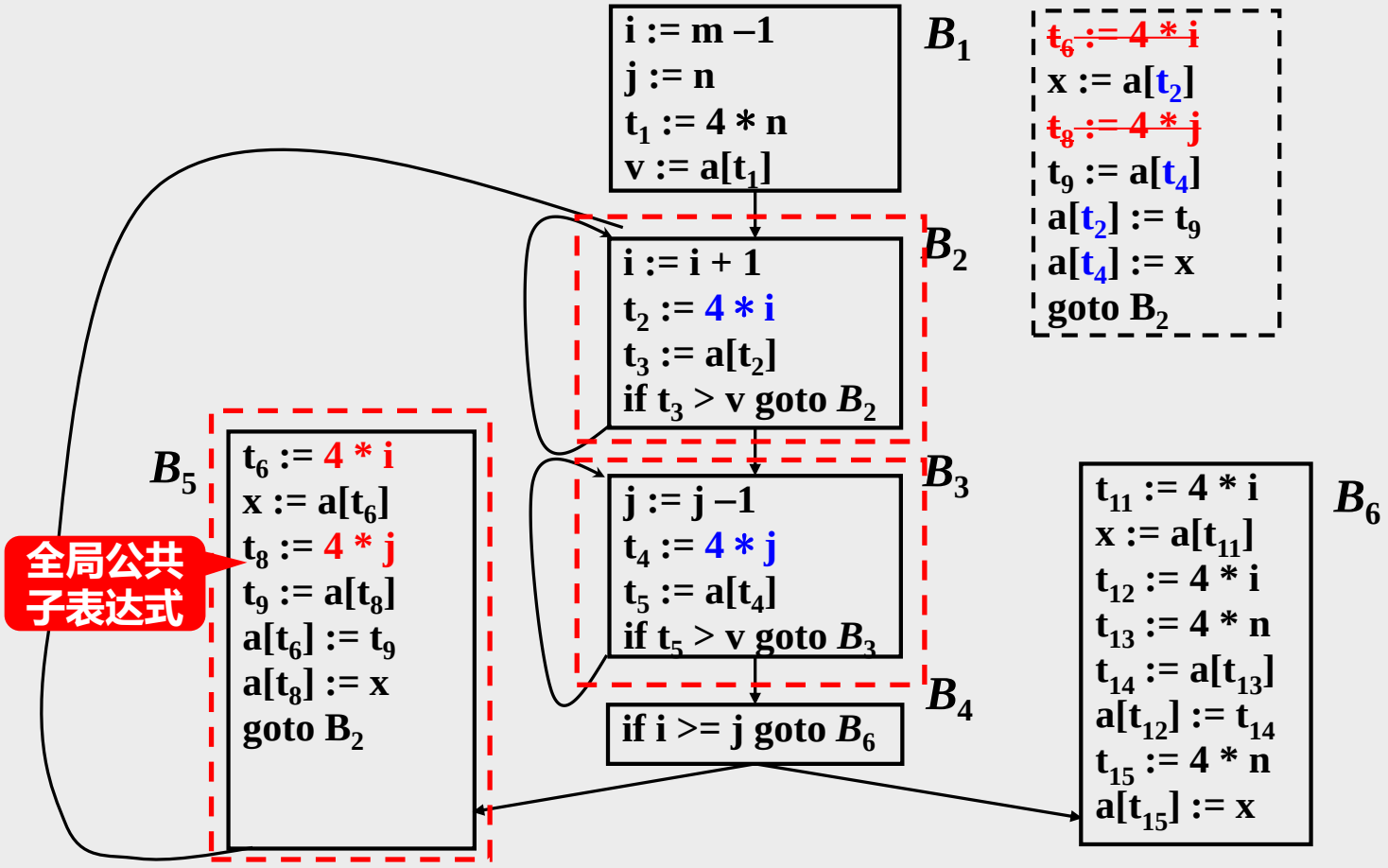
快排中的公共子表达式删除



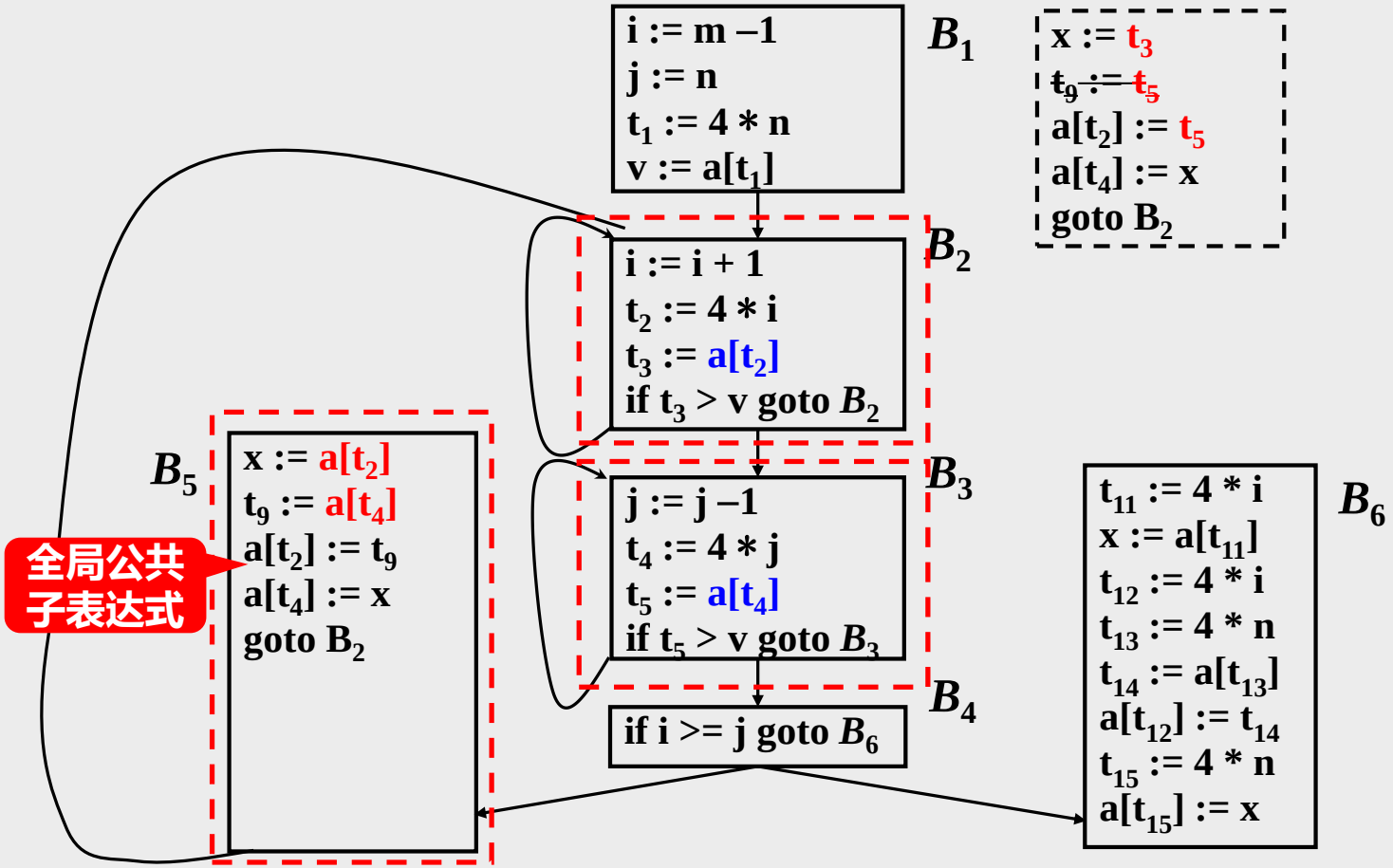
快排中的公共子表达式删除



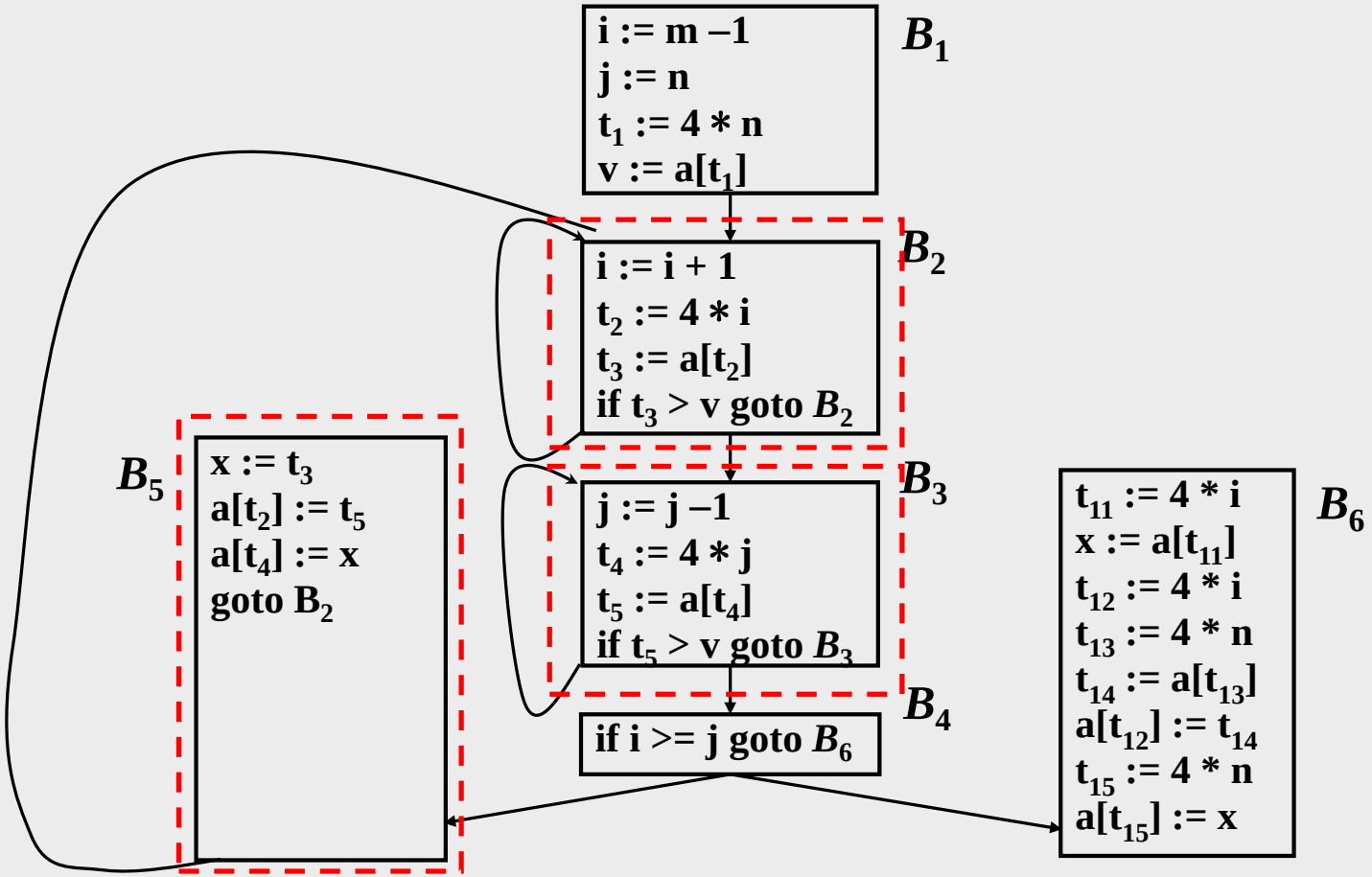
快排中的公共子表达式删除



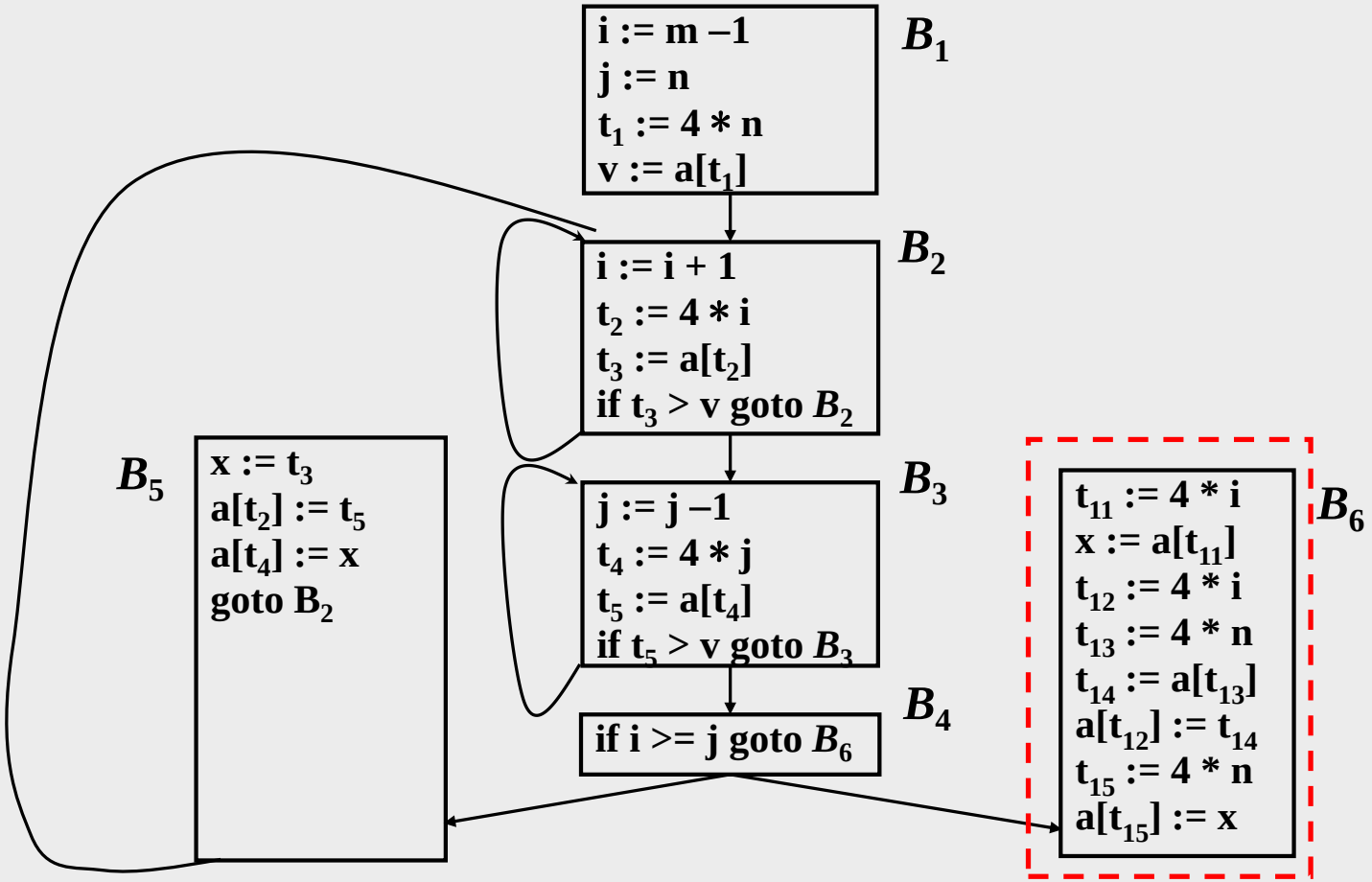
快排中的公共子表达式删除



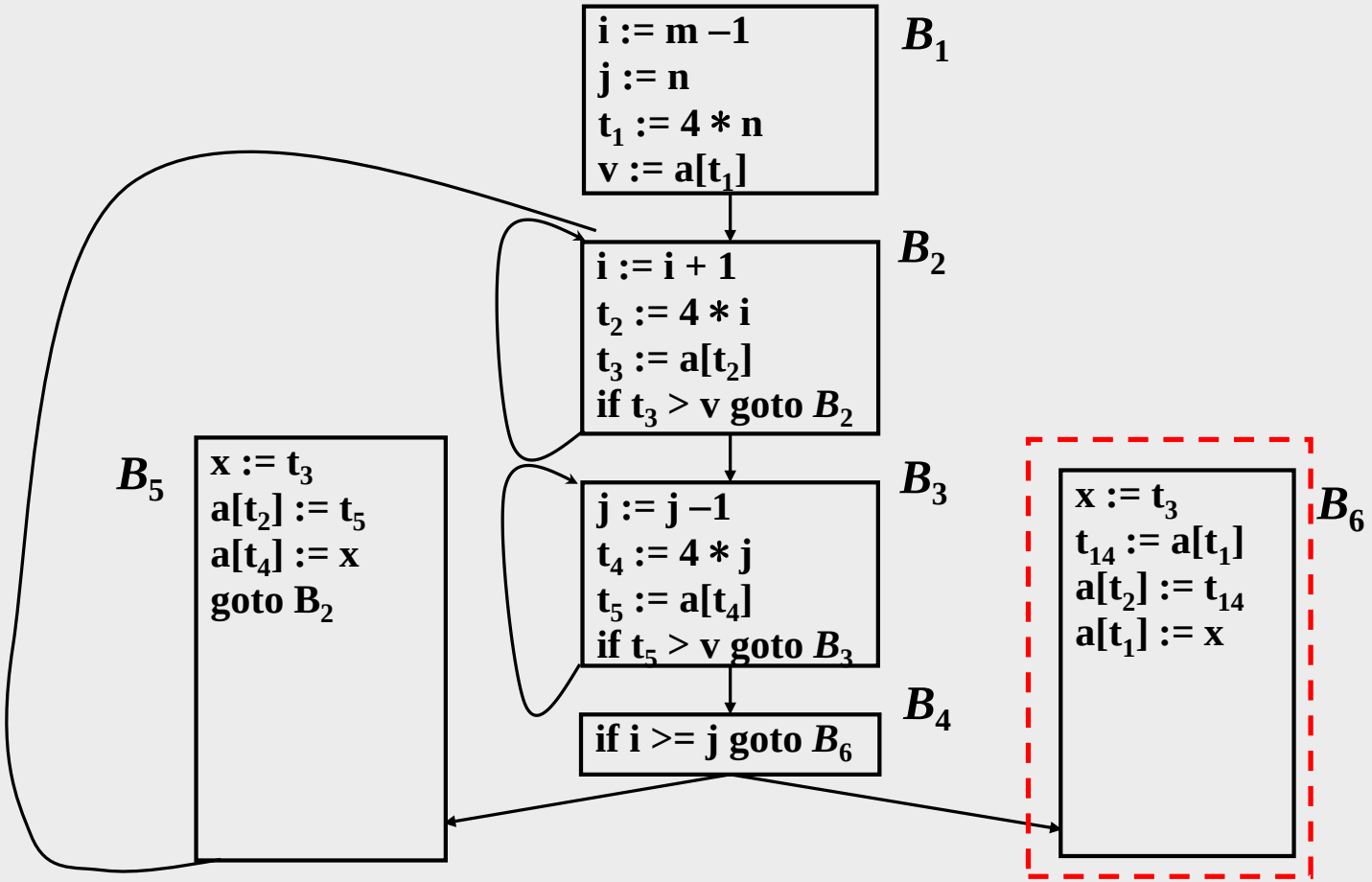
快排中的公共子表达式删除



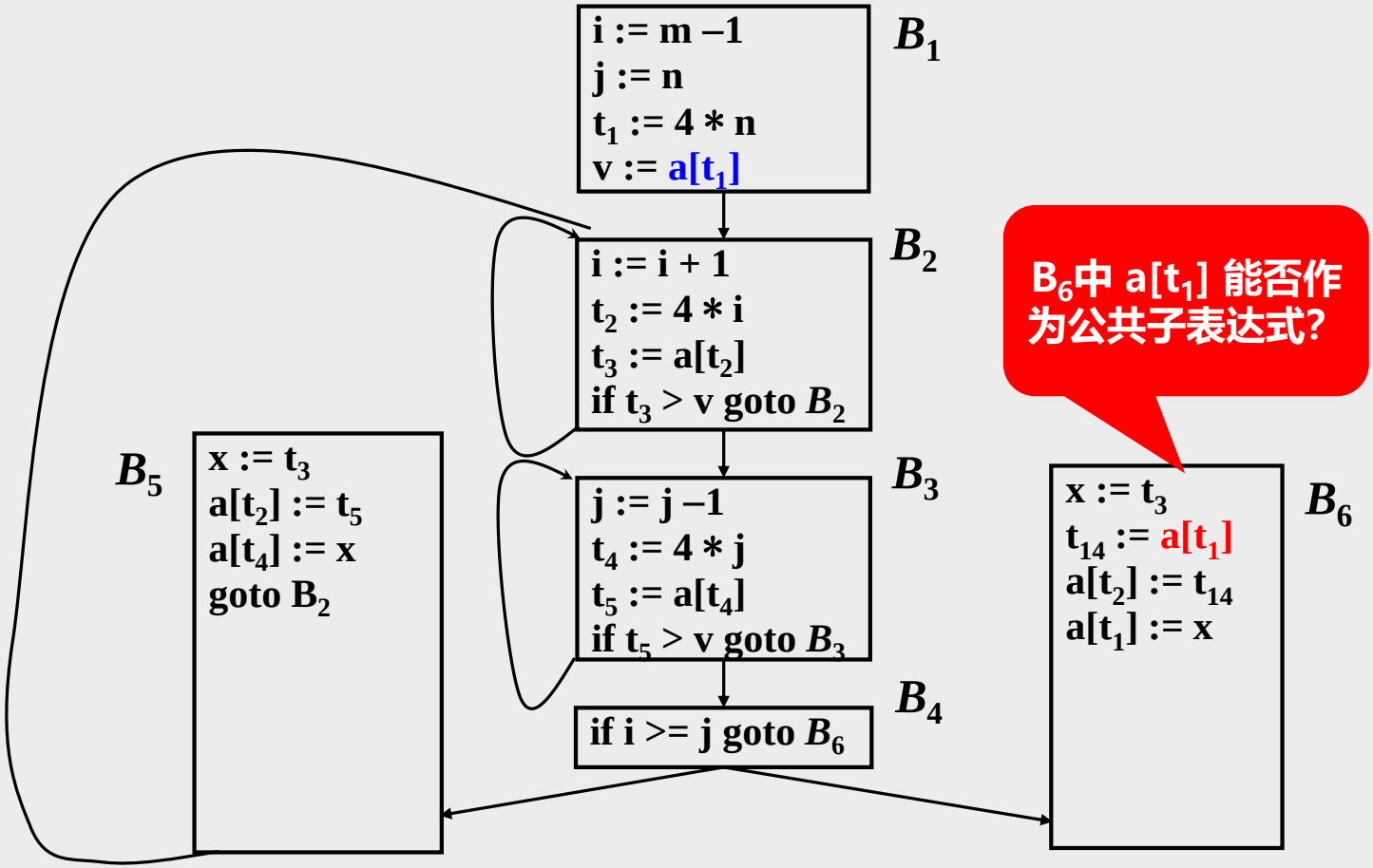
快排中的公共子表达式删除



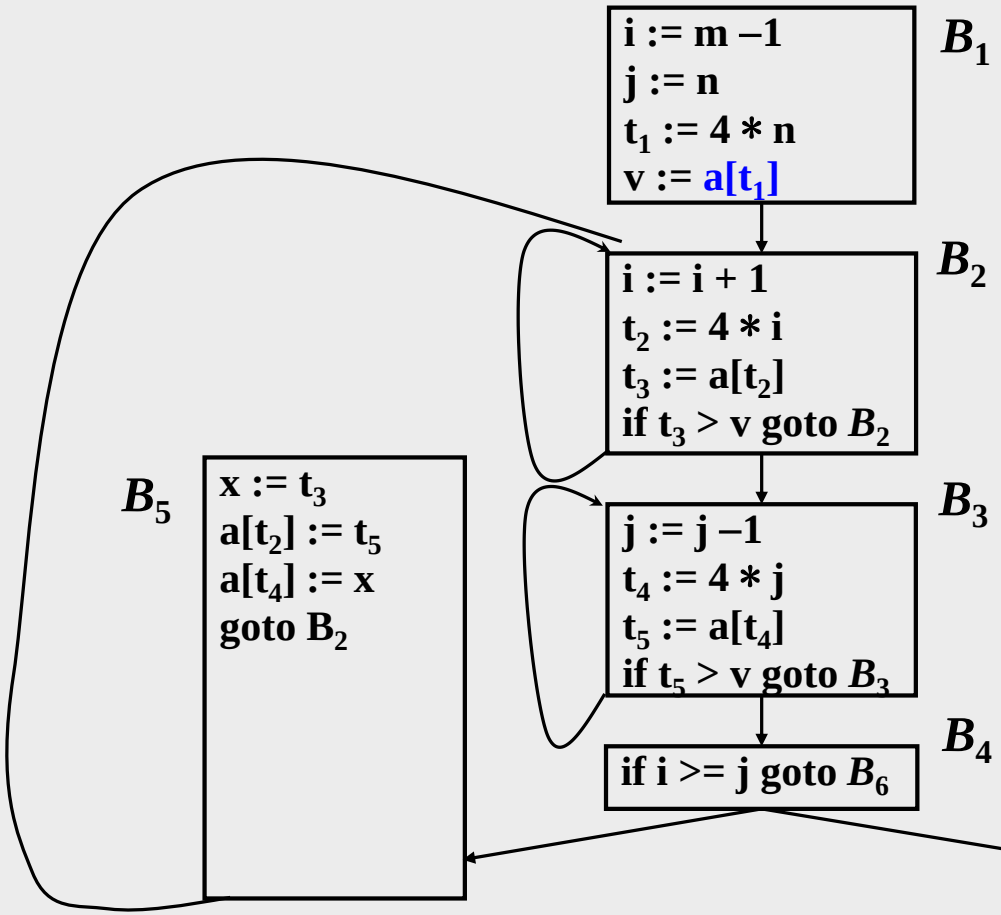
快排中的公共子表达式删除



快排中的公共子表达式删除



快排中的公共子表达式删除



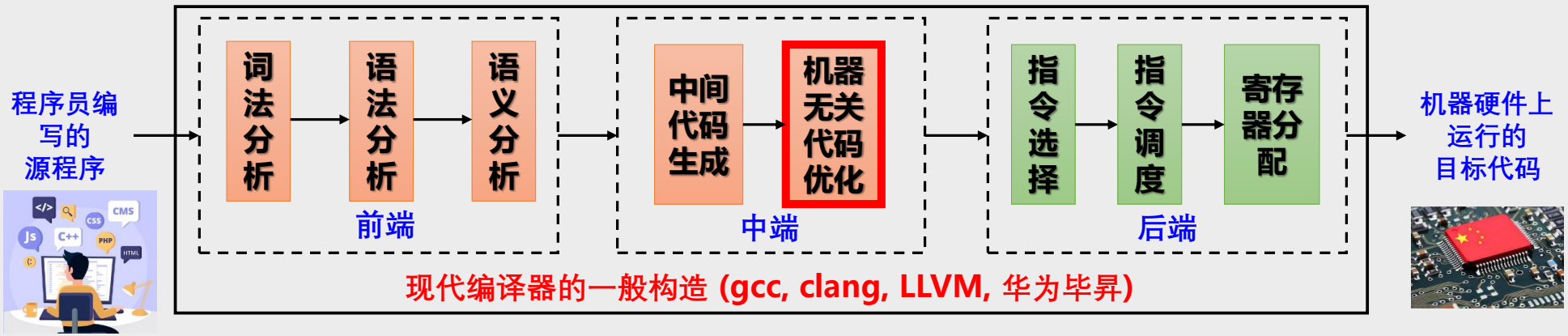
把$a[t_1]$作为公共子表达式是不稳妥的

因为B_5有对下标变量$a[t_2]$和$a[t_4]$的赋值

理论联系实际——课外延伸阅读

- **工程实现公共子表达式删除的关键理论与技术点**
  - 可用表达式数据流分析 (教材第9章第9.2节)
  - 公共子表达式删除的工业界实现代码
    - **文档**链接： https://llvm.org/doxygen/EarlyCSE_8cpp.html
    - **源码**链接： https://llvm.org/doxygen/EarlyCSE_8cpp_source.html
  - 基于Global Value Numbering算法的工业界实现代码
    - **文档**链接： https://llvm.org/docs/Passes.html#gvn-global-value-numbering
    - **源码**链接： https://llvm.org/doxygen/GVN_8cpp_source.html

本节提纲



- 代码优化的定义及背景
- 常见的优化方式
  - 公共子表达式删除优化
  - 死代码删除、复制传播、常量合并
  - 循环系列优化
    - 强度削弱、删除归纳变量、代码移动

死代码删除

- 死代码是指计算的结果永远不被引用的语句

例： 为便于调试，可能在程序中加打印语句，测试后改成右边的形式

debug = true;	debug = false;
. . .	. . .
if (debug) print …	if (debug) print …

死代码删除

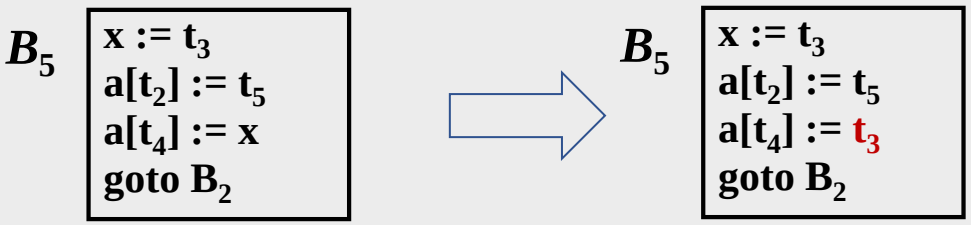
- 死代码是指计算的结果永远不被引用的语句
- 一些优化变换可能会引起死代码
  - 如：复制传播、常量合并

例： 为便于调试，可能在程序中加打印语句，测试后改成右边的形式

debug = true;	debug = false;
. . .	. . .
if (debug) print …	if (debug) print …

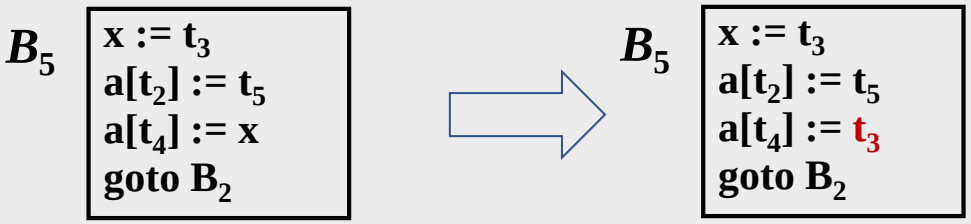
复制传播

- 定义：在复制语句x = y之后尽可能用y代替x



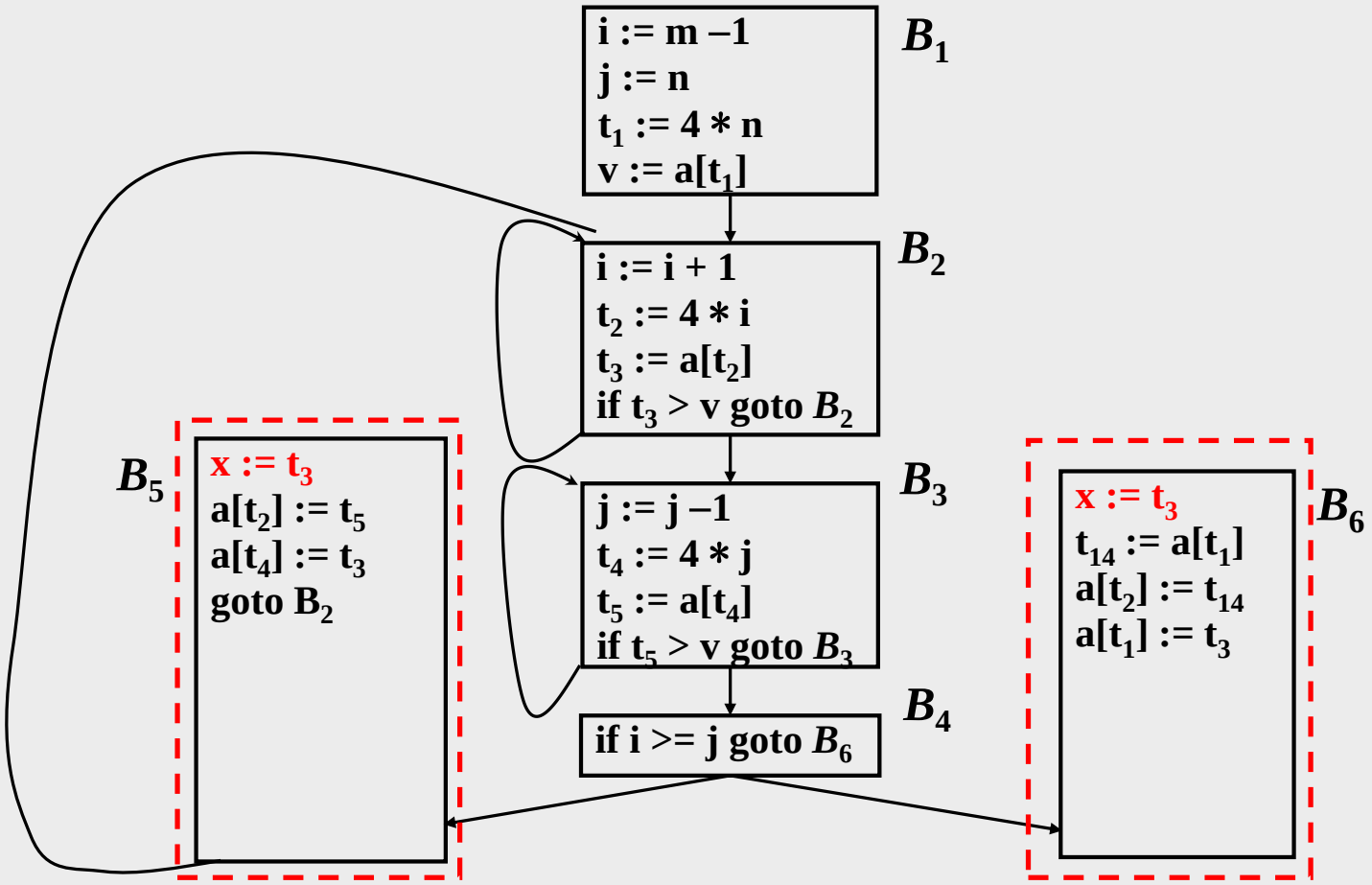
复制传播

- 定义：在复制语句x = y之后尽可能用y代替x

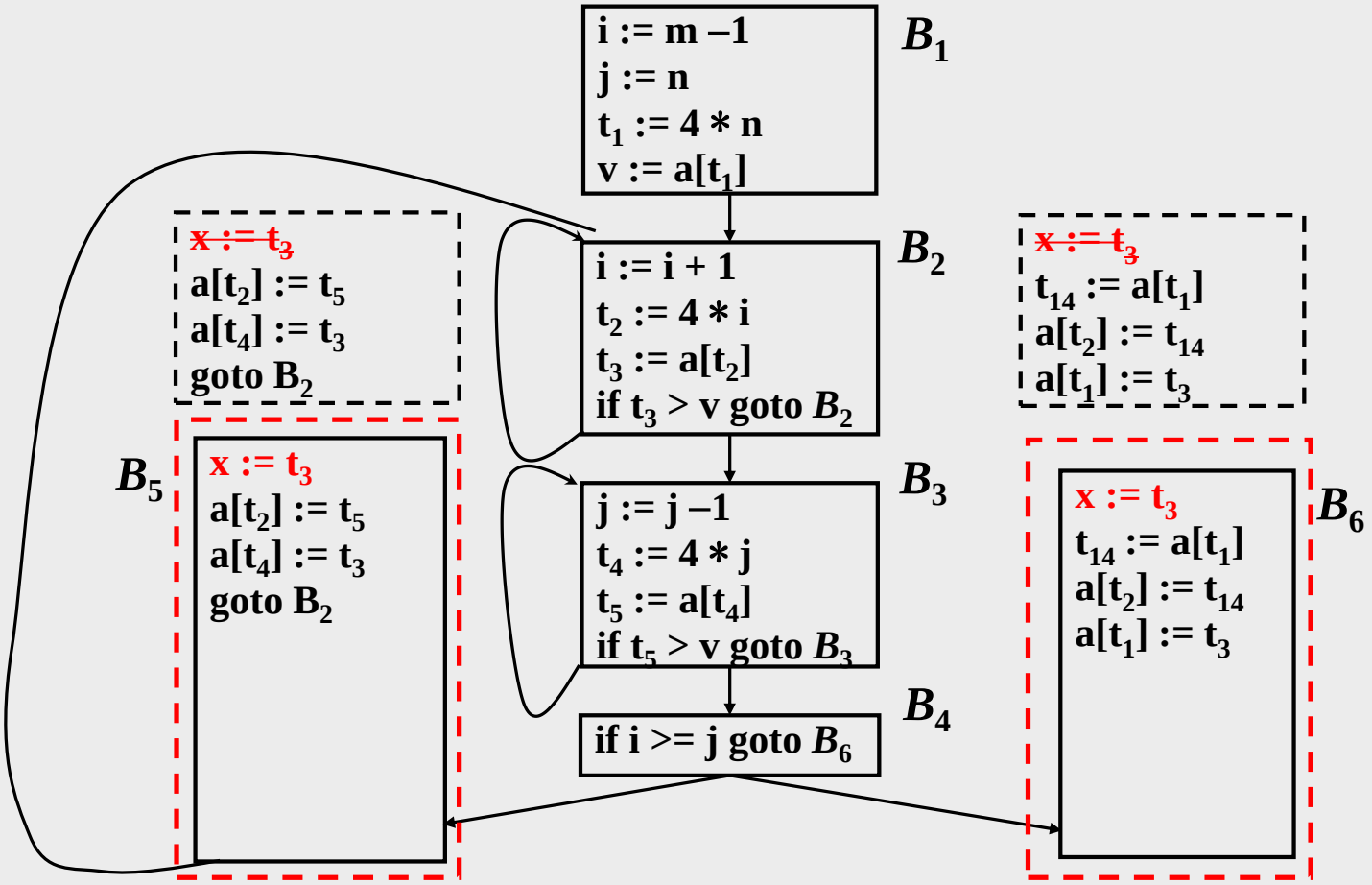


- 常用的公共子表达式删除和其他一些优化会引入一些复制语句
- 复制传播本身没有优化的意义，但可以给死代码删除创造机会

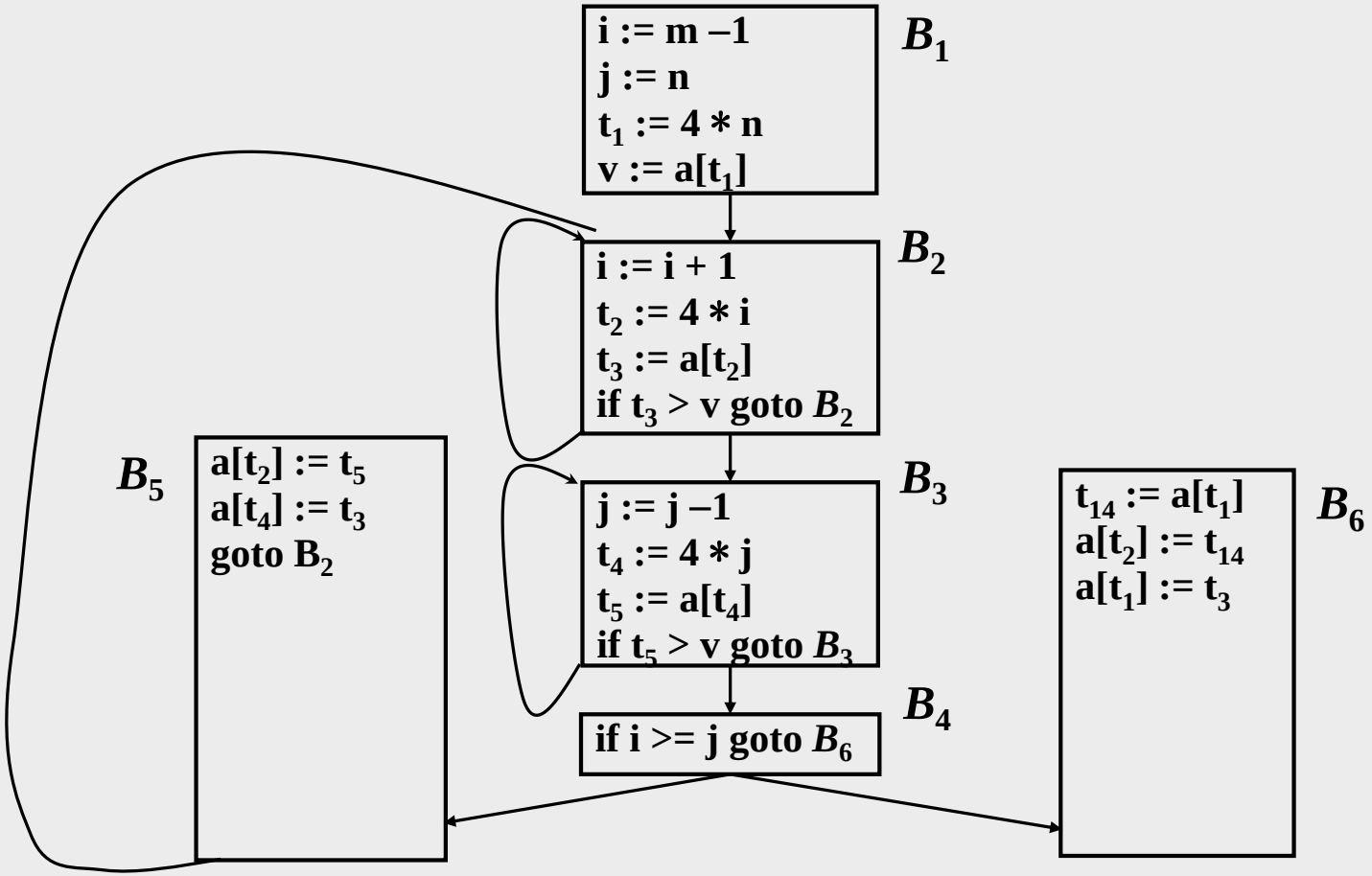
快排中的死代码删除


i := m –1
j := n
t1 := 4 * n
v := a[t1]
B1
i := i + 1
t2 := 4 * i
t3 := a[t2]
if t3 > v goto B2
B2
j := j –1
t4 := 4 * j
t5 := a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
x := t3
a[t2] := t5
a[t4] := t3
goto B2
B5
x := t3
t14 := a[t1]
a[t2] := t14
a[t1] := t3
B6

快排中的死代码删除

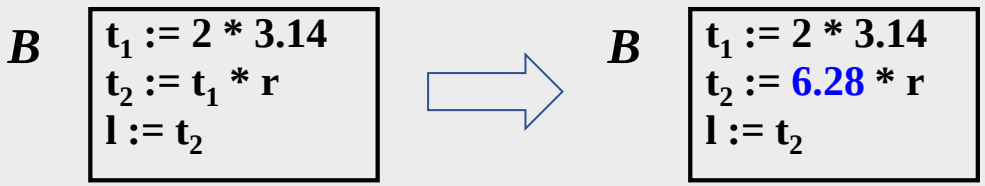

i := m –1
j := n
t1 := 4 * n
v := a[t1]
B1
i := i + 1
t2 := 4 * i
t3 := a[t2]
if t3 > v goto B2
B2
j := j –1
t4 := 4 * j
t5 := a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
x := t3
a[t2] := t5
a[t4] := t3
goto B2
B5
x := t3
t14 := a[t1]
a[t2] := t14
a[t1] := t3
B6

快排中的死代码删除


i := m –1
j := n
t1 := 4 * n
v := a[t1]
B1
i := i + 1
t2 := 4 * i
t3 := a[t2]
if t3 > v goto B2
B2
a[t2] := t5
a[t4] := t3
goto B2
B5
j := j –1
t4 := 4 * j
t5 := a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
t14 := a[t1]
a[t2] := t14
a[t1] := t3
B6

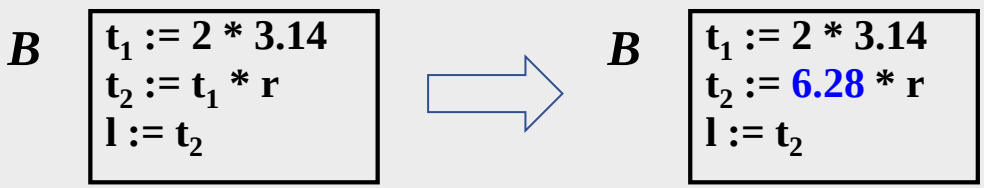
常量合并

- 如果在**编译时刻**推导出**一个表达式的值是常量**，就可以使用该常量来代替这个表达式。
  - 例：计算圆周长的表达式 $l = 2*3.14*r$



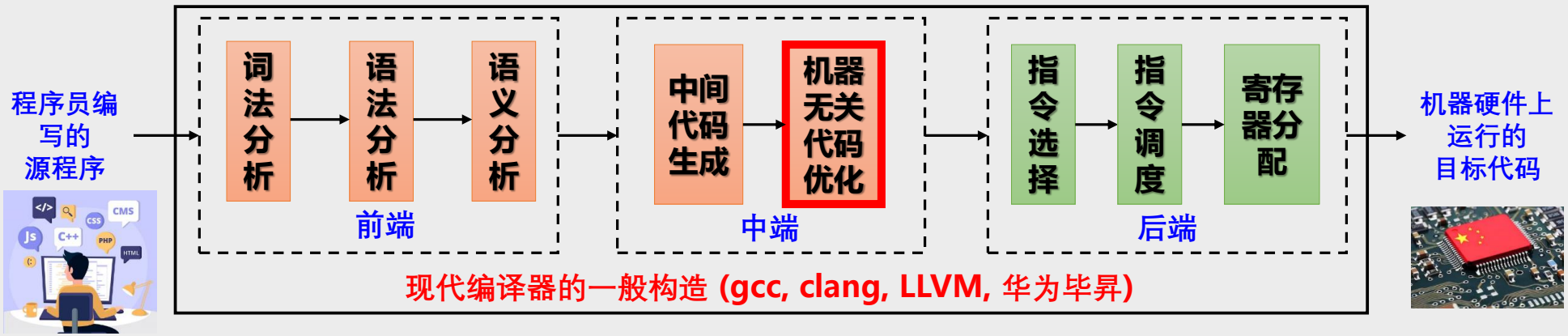
常量合并

- 如果在**编译时刻**推导出**一个表达式的值是常量**，就可以使用该常量来代替这个表达式。
  - 例：计算圆周长的表达式 $l = 2*3.14*r$



- 常量合并本身没有优化的意义，**但可以给死代码删除创造机会**

本节提纲



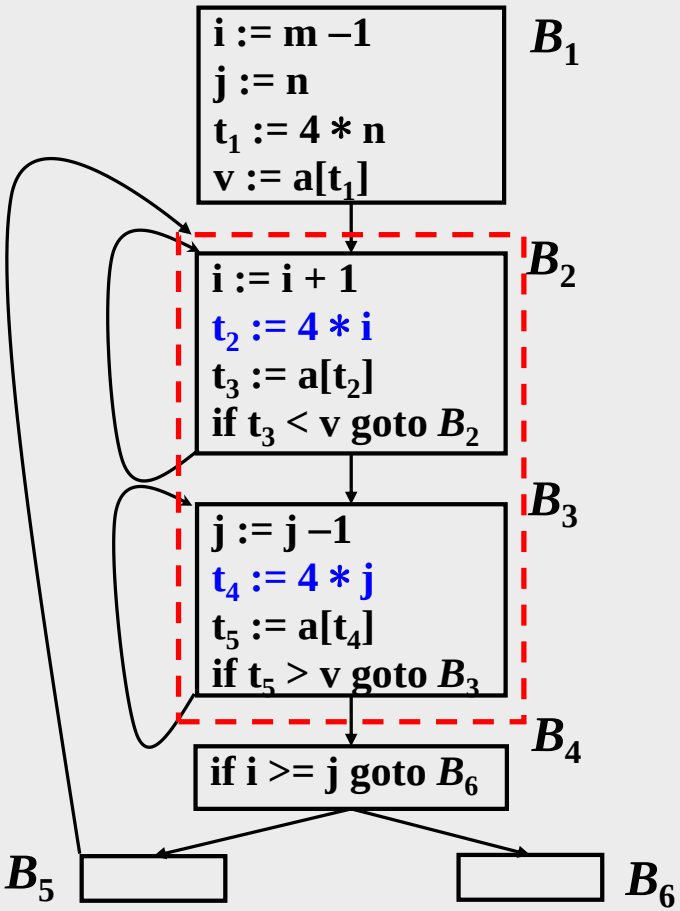
- 代码优化的定义及背景
- 常见的优化方式
  - 公共子表达式删除优化
  - 死代码删除、复制传播、常量合并
  - 循环系列优化
    - 强度削弱、删除归纳变量、代码移动

循环中的强度削弱

- 在循环中的代码会被执行多次
  - 迭代次数越多，执行时间越长
  - 降低每次迭代计算的复杂度是循环优化的重要方法
- 潜在的优化可能——强度削弱(Strength Reduction)
  - 将程序中执行时间较长的运算替换为执行时间较短的运算

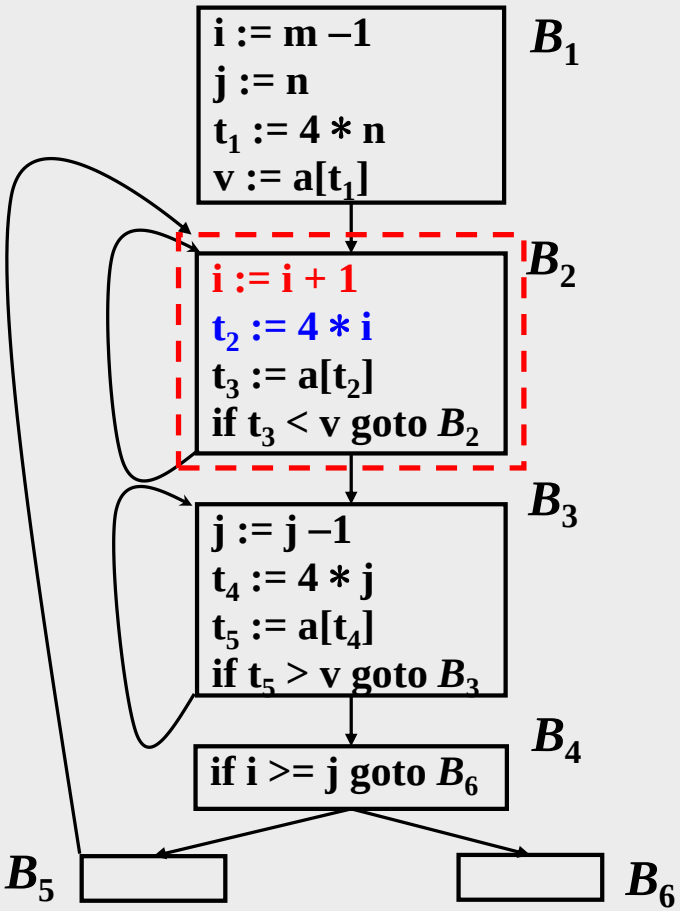
开销较高的运算	开销较低的等价运算
2 * x 或者 2.0 * x	x + x
x/2	x * 0.5
x^2	x * x
$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x^1 + a_0$	$((\ldots (a_n x + a_{n-1})x + a_{n-2})\ldots)x + a_1) + a_0$

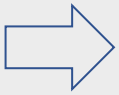
循环中的强度削弱

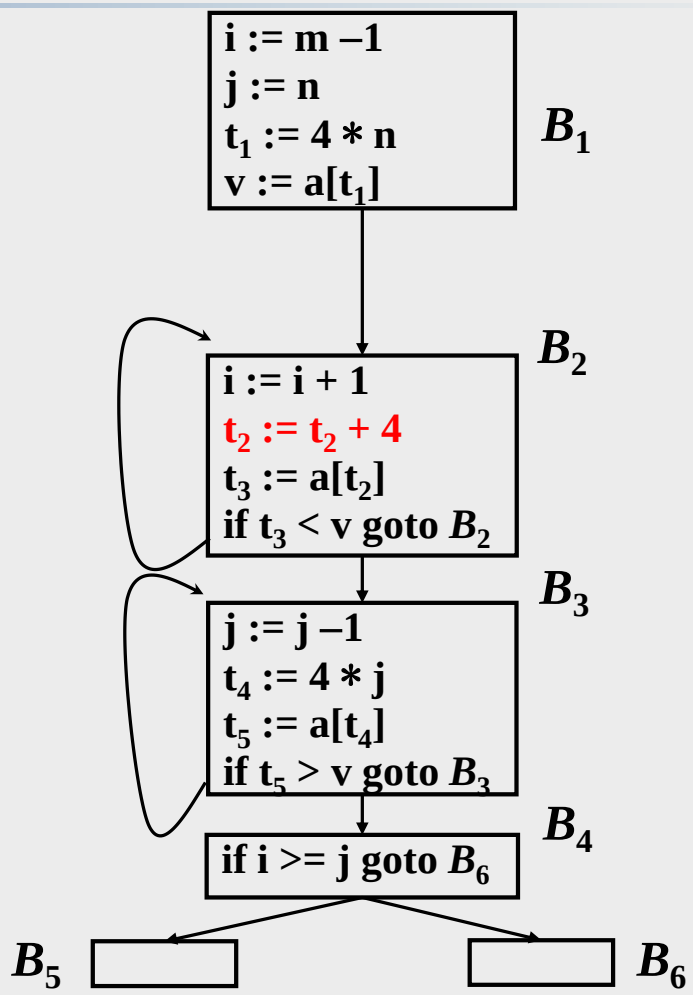


- 观察B_2和B_3中的变量t_2和t_4
  - t2和t4总是按照一定的步幅递增或递减
- 归纳变量
  - 如果存在一个常量c，使x的每一次赋值总是增加c，则称x为归纳变量。
  - 基本形式：x = x + c
  - 高级形式：x = c*i+d （c, d常量，i归纳变量）
  - 强度削弱：用增量运算(加或减)替代

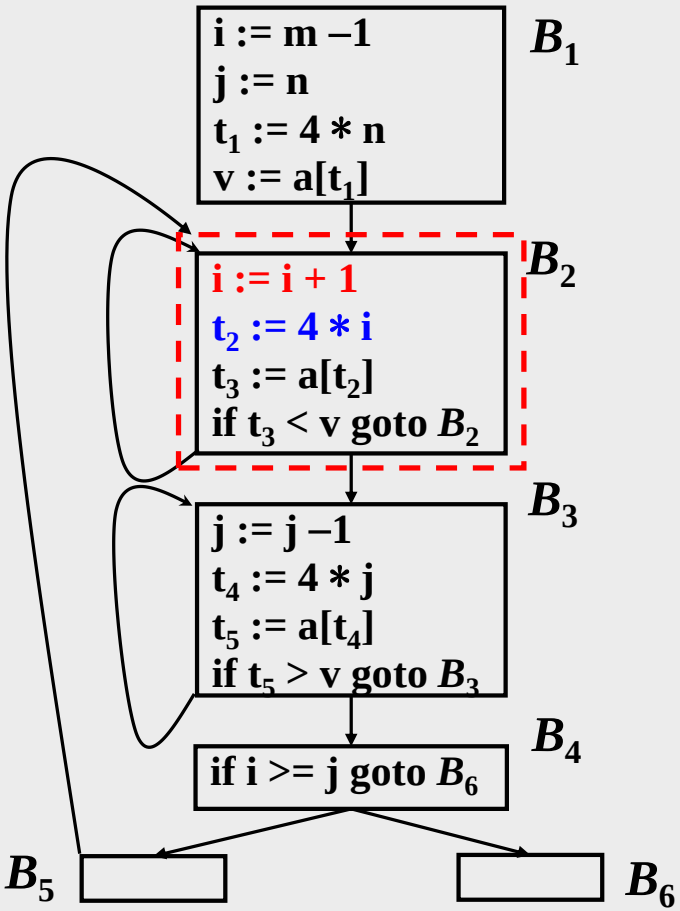
循环中的强度削弱


i := m –1
j := n
t1 := 4 * n
v := a[t1]
B1
i := i + 1
t2 := 4 * i
t3 := a[t2]
if t3 < v goto B2
B2
j := j –1
t4 := 4 * j
t5 := a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
B5
B6

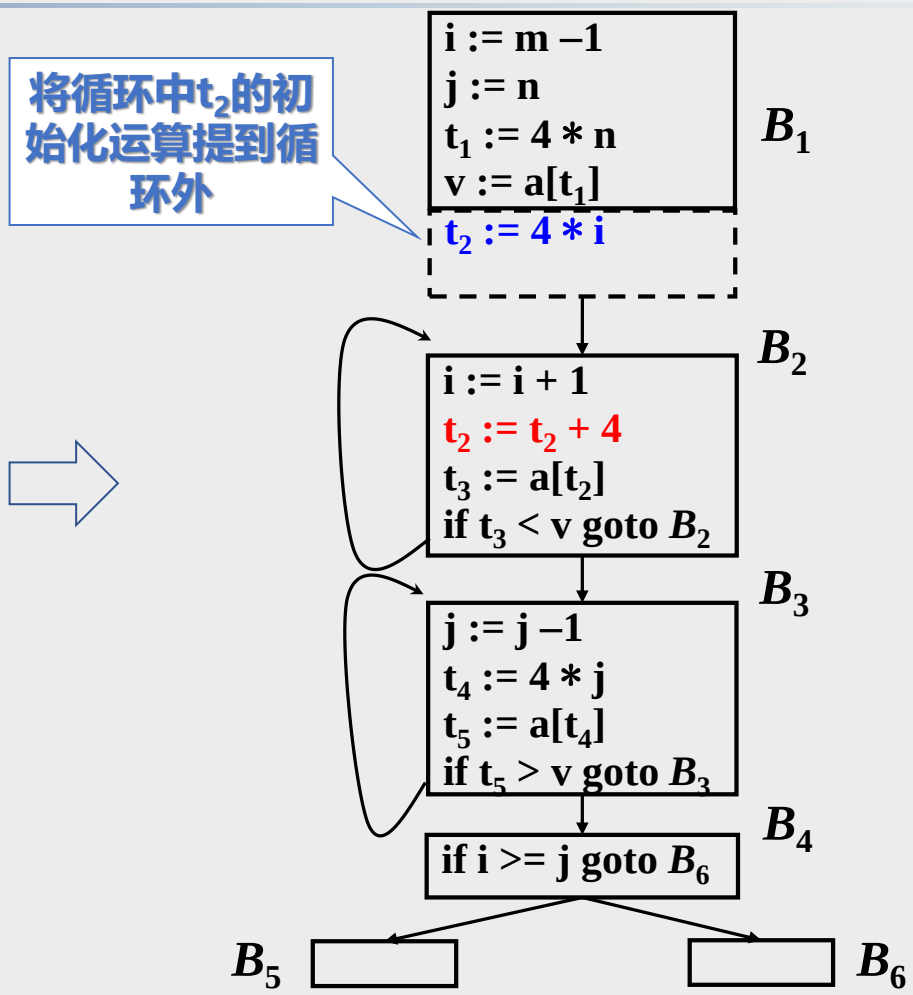



i := m –1
j := n
t1 := 4 * n
v := a[t1]
B1
i := i + 1
t2 := t2 + 4
t3 := a[t2]
if t3 < v goto B2
B2
j := j –1
t4 := 4 * j
t5 := a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
B5
B6

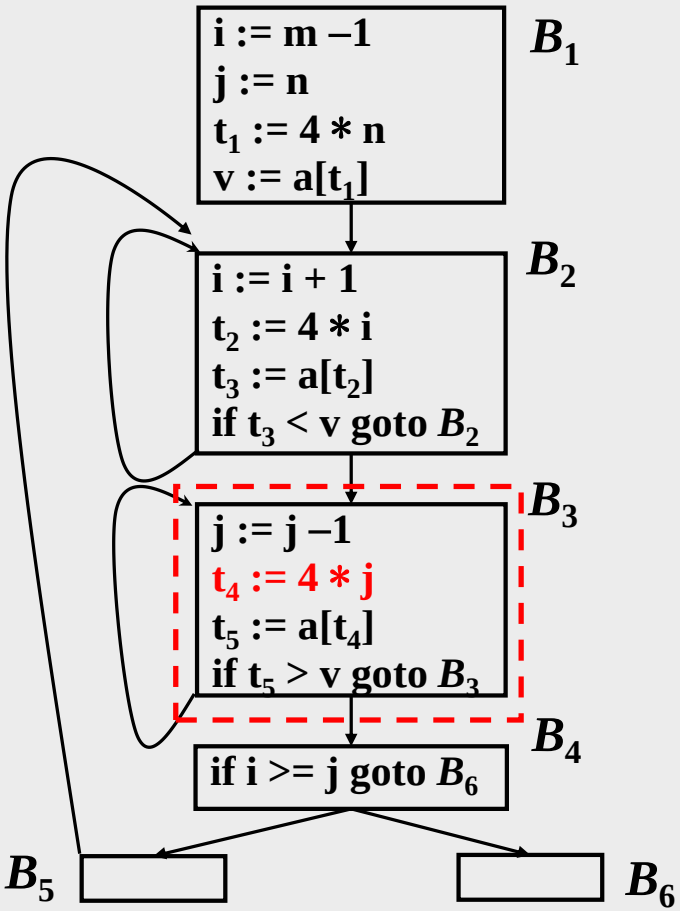
循环中的强度削弱

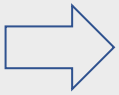


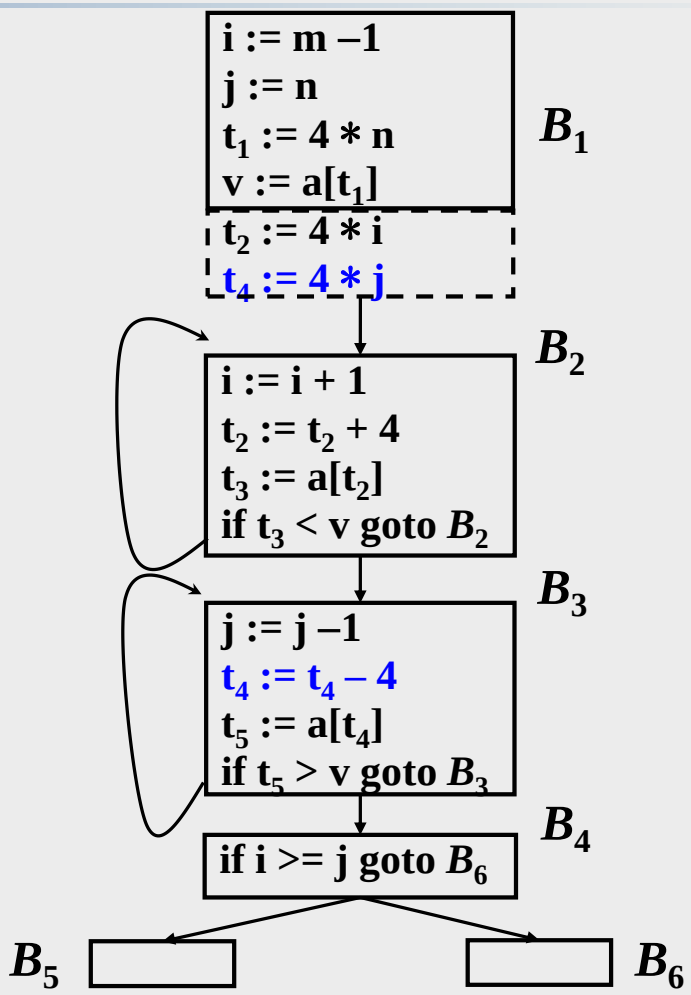
将循环中t_2的初始化运算提到循环外



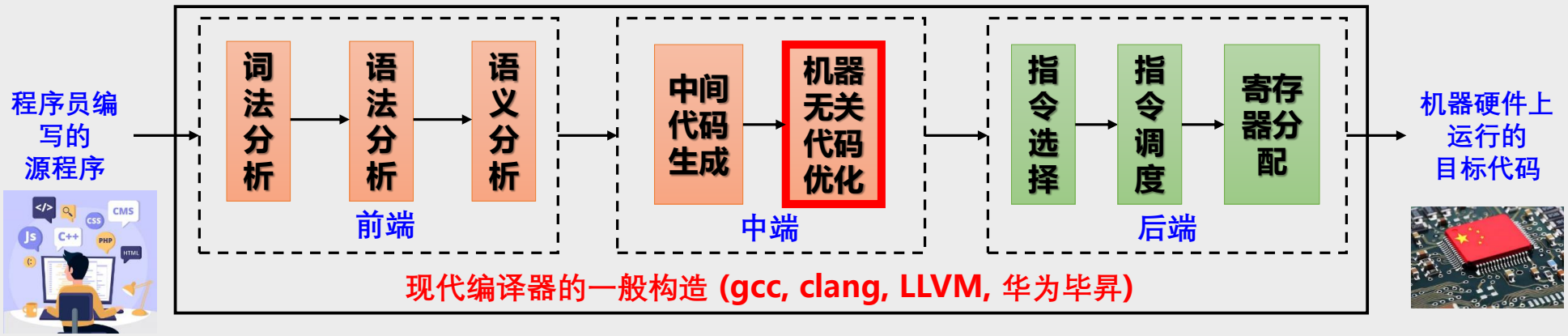
循环中的强度削弱


i := m –1
j := n
t1 := 4 * n
v := a[t1]
B1
i := i + 1
t2 := 4 * i
t3 := a[t2]
if t3 < v goto B2
B2
j := j –1
t4 := 4 * j
t5 := a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
B5
B6



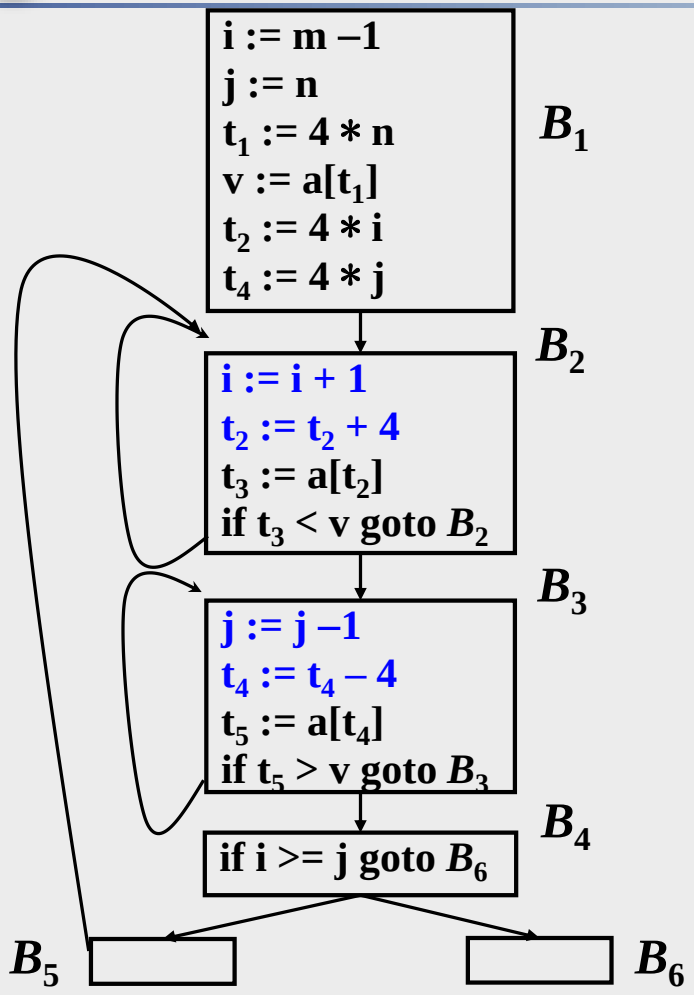

i := m –1
j := n
t1 := 4 * n
v := a[t1]
t2 := 4 * i
t4 := 4 * j
B1
i := i + 1
t2 := t2 + 4
t3 := a[t2]
if t3 < v goto B2
B2
j := j –1
t4 := t4 – 4
t5 := a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
B5
B6

本节提纲



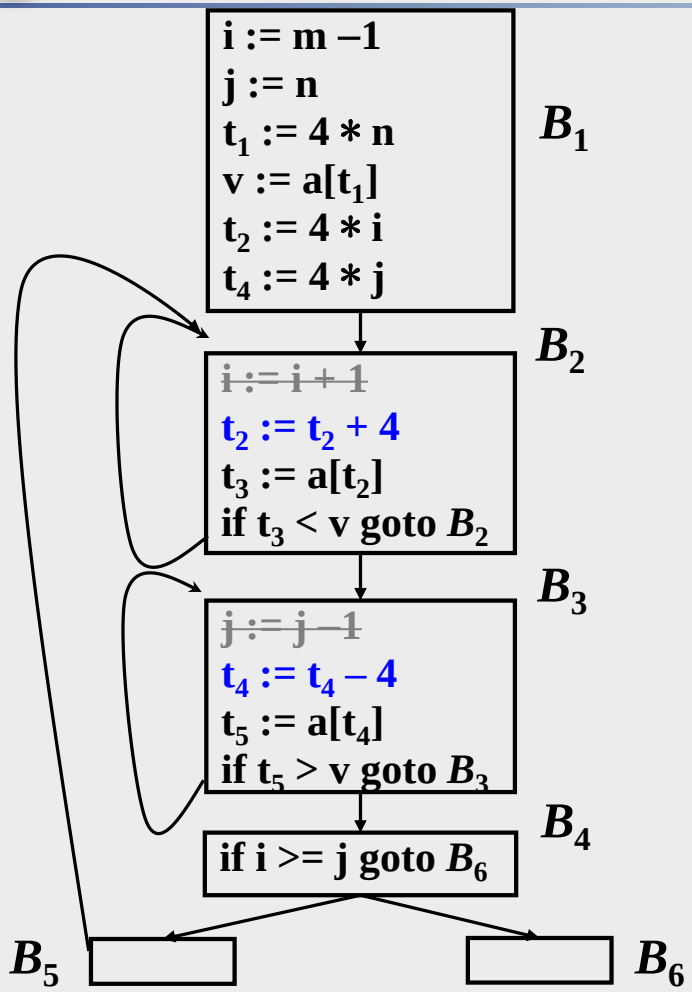
- 代码优化的定义及背景
- 常见的优化方式
  - 公共子表达式删除优化
  - 死代码删除、复制传播、常量合并
  - 循环系列优化
    - 强度削弱、删除归纳变量、代码移动

循环中的归纳变量删除



- i, j, t_2与t_4均为归纳变量
- 按照变化步调对归纳变量进行分组
  - i与t_2，j与t_4
- 删除冗余的归纳变量
  - 一个循环中，如一组归纳变量的值的变化保持步调一致，可只保留一个。

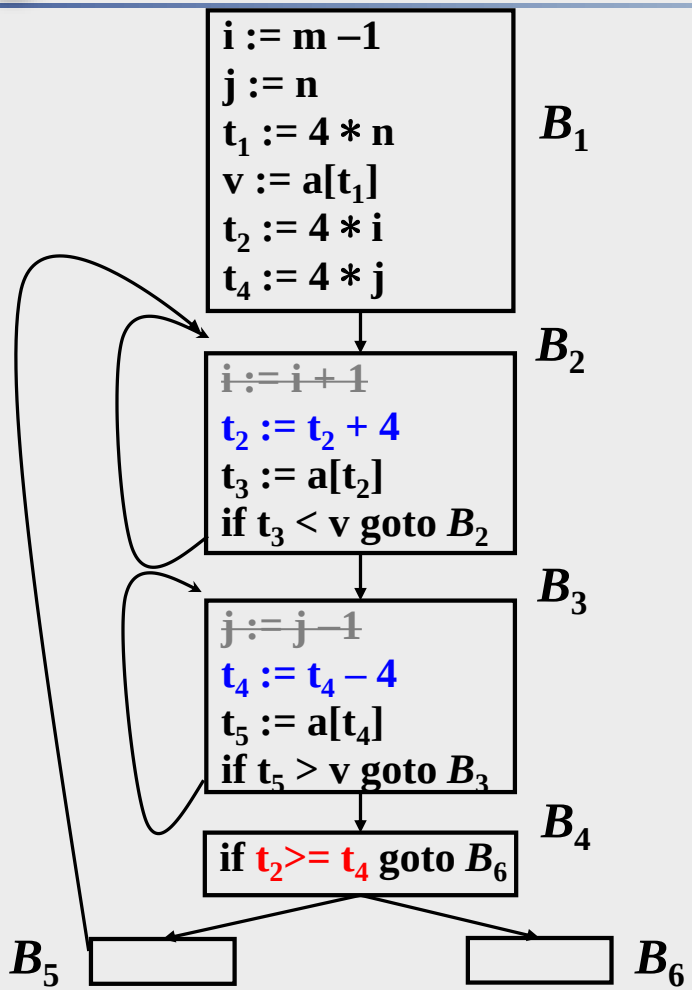
循环中的归纳变量删除



- i, j, t_2与t_4均为归纳变量
- 按照变化步调对归纳变量进行分组
  - i 与t_2， j与t_4
- 删除冗余的归纳变量
  - 一个循环中，如一组归纳变量的值的变化保持步调一致，可只保留一个。

第一步：删除循环中对 i 和 j 的计算

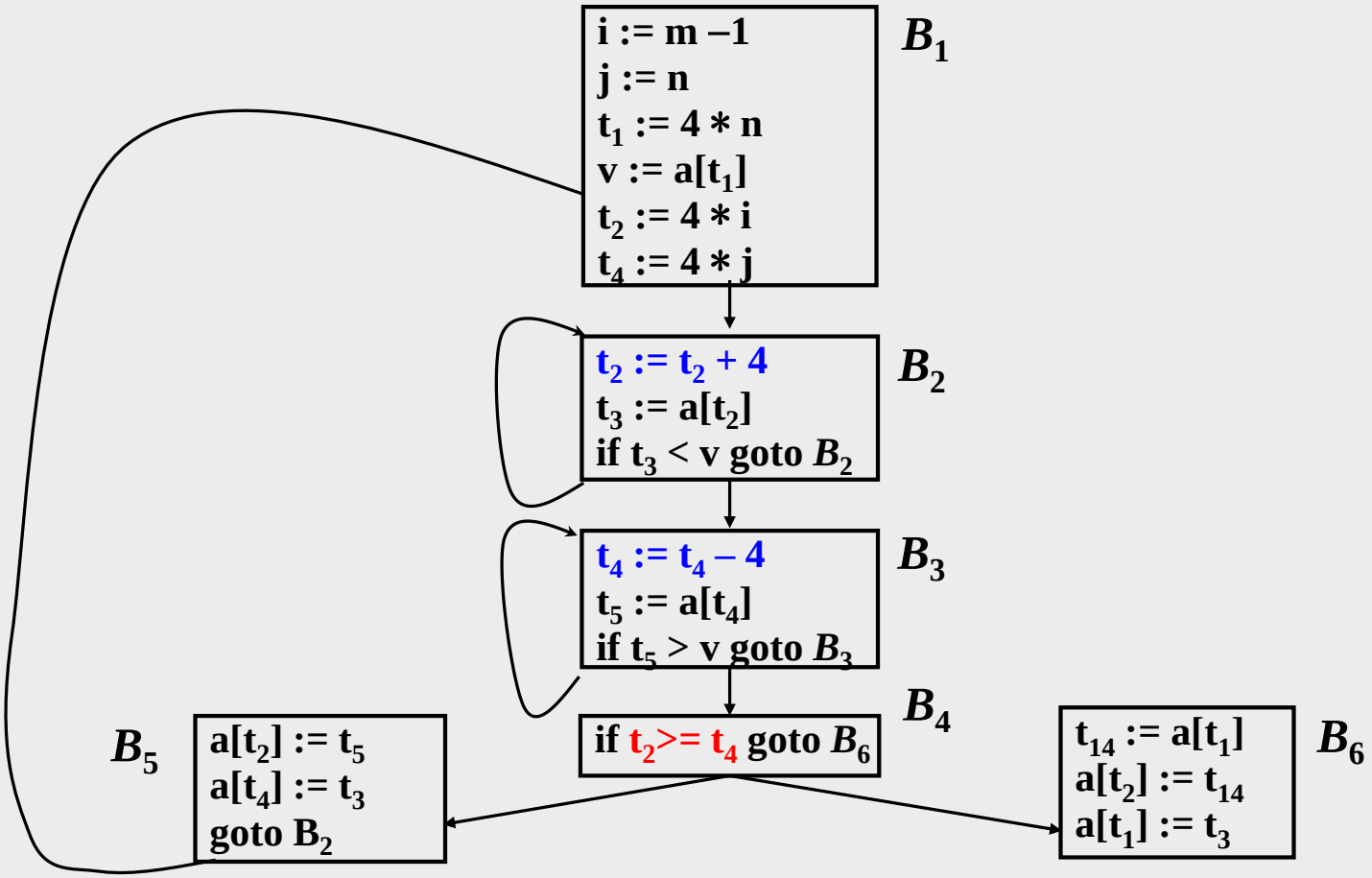
循环中的归纳变量删除



- i, j, t_2与t_4均为归纳变量
- 按照变化步调对归纳变量进行分组
  - i 与t_2， j与t_4
- 删除冗余的归纳变量
  - 一个循环中，如一组归纳变量的值的变化保持步调一致，可只保留一个。

第二步：将对 i 和 j 的引用分别替换为 t2 与 t4

优化后的快速排序代码

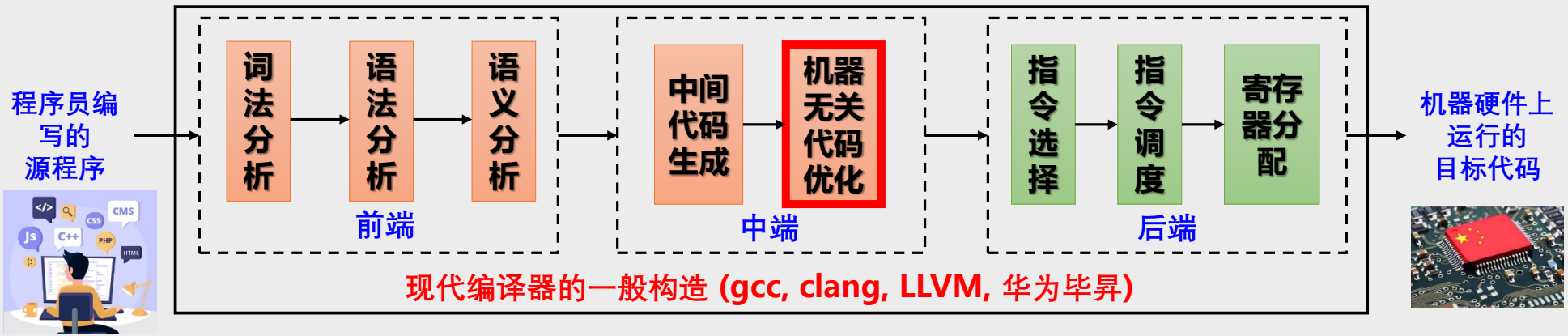

i := m –1
j := n
t1 := 4 * n
v := a[t1]
t2 := 4 * i
t4 := 4 * j
B1
t2 := t2 + 4
t3 := a[t2]
if t3 < v goto B2
B2
t4 := t4 – 4
t5 := a[t4]
if t5 > v goto B3
B3
if t2>= t4 goto B6
B4
B5
a[t2] := t5
a[t4] := t3
goto B2
t14 := a[t1]
a[t2] := t14
a[t1] := t3
B6

理论联系实际——课外延伸阅读

工程实现循环优化的关键理论与技术点

- 如何识别归纳变量？
  - **文档**链接：https://llvm.org/docs/Passes.html#iv-users-induction-variable-users
  - **源码**链接：https://llvm.org/doxygen/IVUsers_8cpp_source.html
- 强度削弱的工业界实现代码
  - **文档**链接：https://llvm.org/docs/Passes.html#loop-reduce-loop-strength-reduction
  - **源码**链接：https://llvm.org/doxygen/LoopStrengthReduce_8cpp.html
- 归纳变量删除的工业界实现代码
  - **文档**链接：https://llvm.org/docs/Passes.html#indvars-canonicalize-induction-variables
  - **源码**链接：https://llvm.org/doxygen/IndVarSimplify_8cpp.html

本节提纲



- 代码优化的定义及背景
- 常见的优化方式
  - 公共子表达式删除优化
  - 死代码删除、复制传播、常量合并
  - 循环系列优化
    - 强度削弱、删除归纳变量、代码移动

代码移动

- 循环不变计算（loop-invariant computation）是指不管循环执行多少次都得到相同结果的表达式
- 代码移动是循环优化的一种，在进入循环前就对循环不变计算进行求值

代码移动

- 循环不变计算（loop-invariant computation）是指不管循环执行多少次都得到相同结果的表达式
- 代码移动是循环优化的一种，在进入循环前就对循环不变计算进行求值。

例：while (i <= limit – 2) …

代码移动后变换成

```
t = limit – 2;
while (i <= t ) …
```

代码移动

- 循环不变计算（loop-invariant computation）是指不管循环执行多少次都得到相同结果的表达式
- 代码移动是循环优化的一种，在进入循环前就对循环不变计算进行求值。
- 对于多重嵌套循环， loop-invariant computation是相对于某一个循环的，可能对于更加外层的循环，它就不成立了。
- 因此，处理循环时，按照由里到外的方式

结束语

- **代码优化是编译技术的重要组成部分，是发挥硬件能力、提升编程水平的重要技术手段。**
- **常见的代码优化方法有循环优化和公共子表达式删除等。**
  - 循环的强度削弱和归纳变量删除依赖于归纳变量识别
  - 公共子表达式删除依赖于公共子表达式的识别
  - 可以看出，大部分的高级优化均需要利用代码分析

图灵奖获得者——法兰 艾伦

- 由于在编译优化方面的杰出贡献被授予2006年计算机图灵奖
- 是世界上第一位获得图灵奖的女科学家
- 重要的理论与实践工作有：
  - Program Optimization， 1966
  - Control Flow Analysis， 1970
  - A Basis for Program Optimization, 1970
  - A Catalog of Optimizing Transformations， 1971
  - ……

Frances Allen
(1932-2020)
ACM/IEEE Fellow
美国科学院院士

《编译原理和技术》

机器无关的代码优化 I

谢谢！

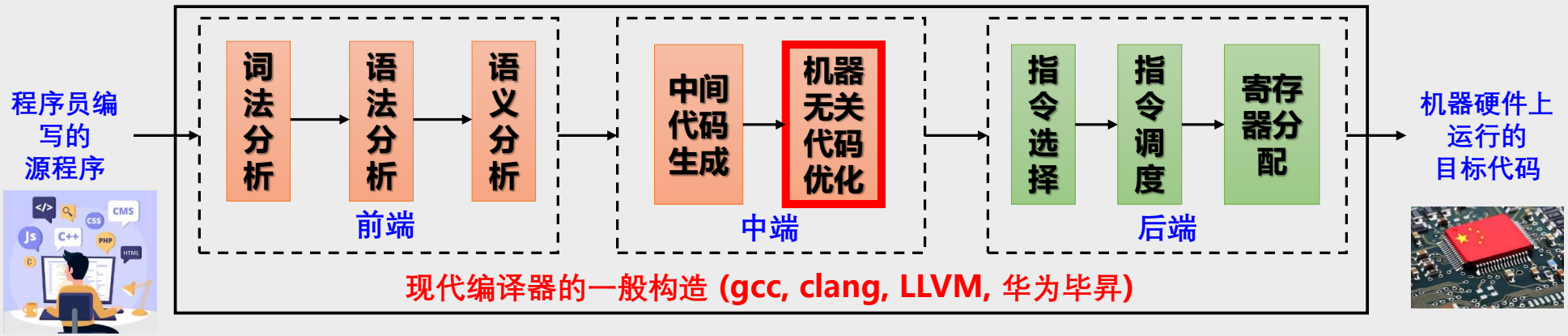
《编译原理和技术》

机器无关的代码优化 II

——数据流分析

中科大计算机学院
李 诚
2022-11-02

本节提纲

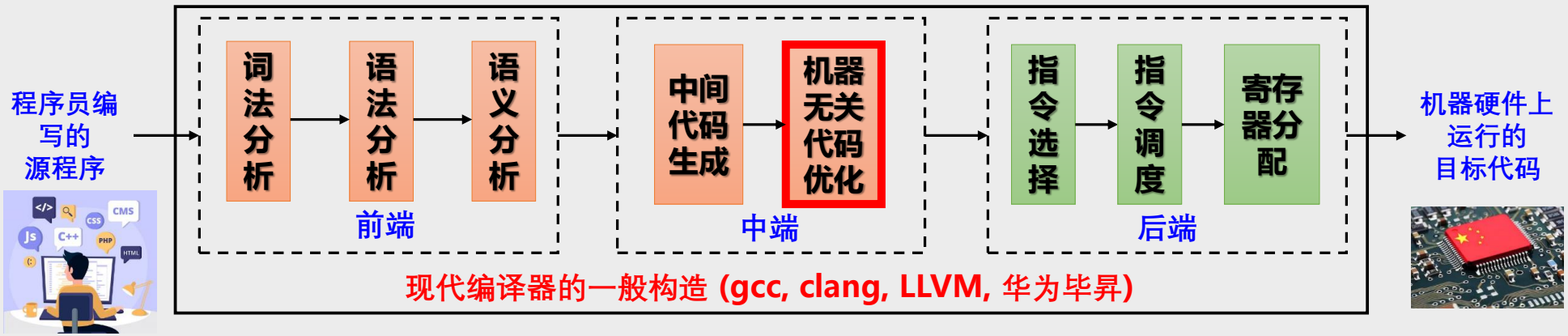


- 代码优化的主要实现方式
- 数据流分析概述
- 数据流分析理论框架
- 到达-定值分析算法

优化的实现方式

- **全局视角-跨基本块的优化**
  - 数据流分析
- **局部视角-基本块的优化**
  - DAG表示

本节提纲



- 代码优化的主要实现方式
- 数据流分析概述
- 数据流分析理论框架
- 到达-定值分析算法

数据流分析

- **Data-flow analysis**
  - 一组用来获取程序执行路径上的数据流信息的技术
- **数据流分析应用**
  - 到达-定值分析(Reaching-Definition Analysis)
  - 活跃变量分析(Live-Variable Analysis)
  - 可用表达式分析(Available-Expression Analysis)
- **在每一种数据流分析应用中，都会把每个程序点和一个数据流值关联起来**

数据流抽象

- 流图上的点(程序点)
  - 基本块中，两个相邻的语句之间为程序的一个点
  - 基本块的开始点和结束点

- 流图上的路径
  - 点序列$p_1, p_2, \ldots, p_n$，对1和$n-1$间的每个i，满足

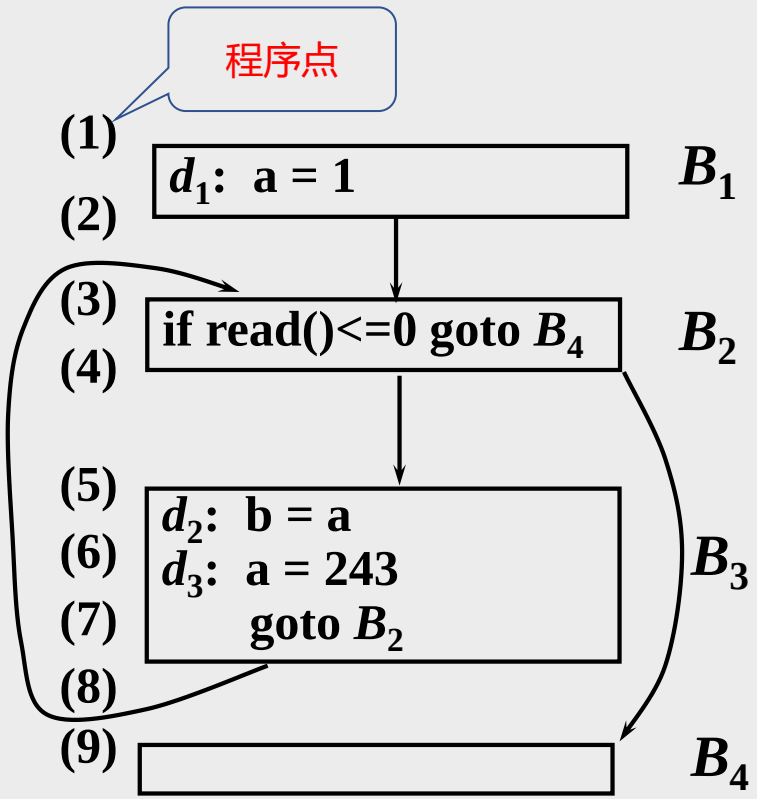
(1) p_i是先于一个语句的点，p_{i+1}是同一块中位于该语句后的点，或者

(2) p_i是某块的结束点，p_{i+1}是后继块的开始点

数据流抽象

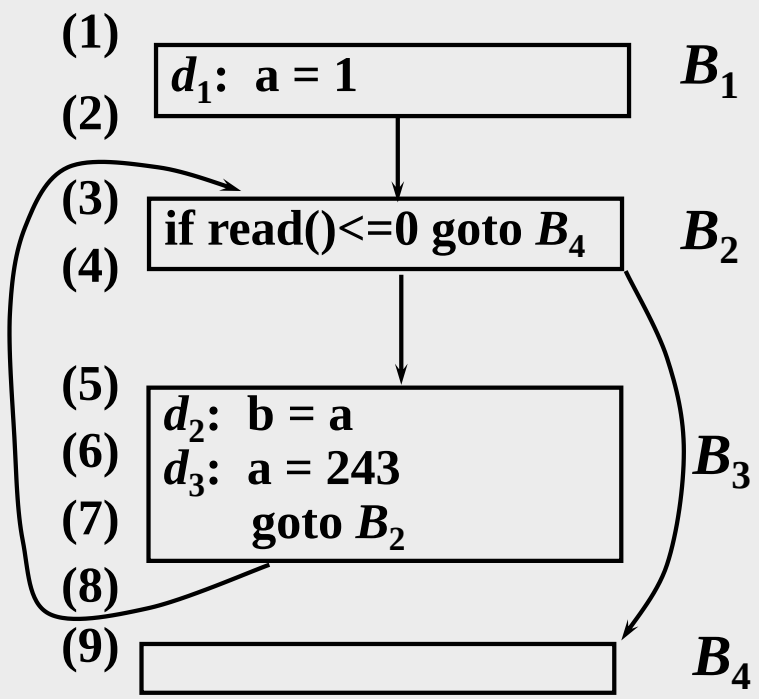
流图上路径实例

- (1, 2, 3, 4, 9)
- (1, 2, 3, 4, 5, 6, 7, 8, 3, 4, 9)
- (1, 2, 3, 4, 5, 6, 7, 8,
  3, 4, 5, 6, 7, 8, 3, 4, 9)
- (1, 2, 3, 4, 5, 6, 7, 8,
  3, 4, 5, 6, 7, 8,
  3, 4, 5, 6, 7, 8, …)
- … …



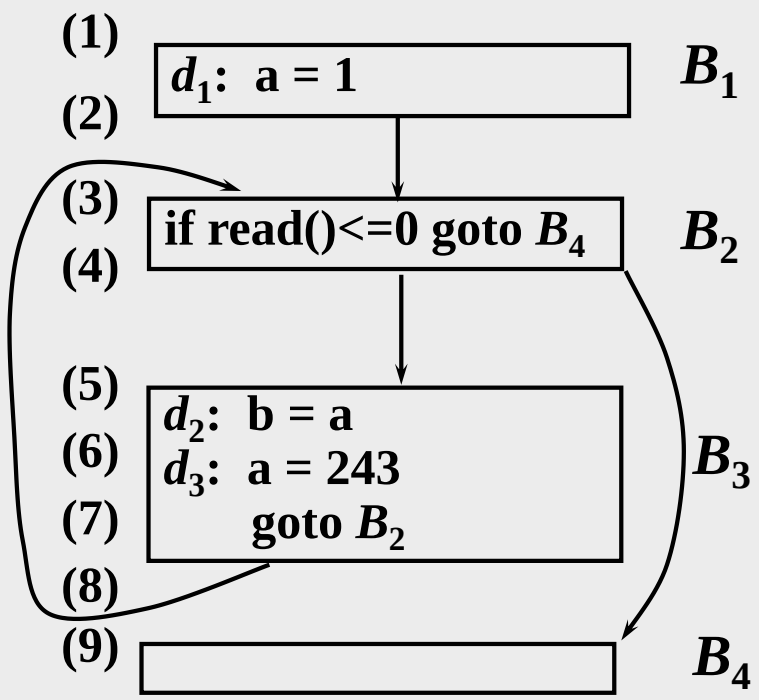
数据流分析介绍

- **分析程序的行为时，必须在其流图上考虑所有的执行路径（在调用或返回语句被执行时，还需要考虑执行路径在多个流图之间的跳转）**
  - 通常，从流图得到的程序执行路径数无限，且执行路径长度没有有限的上界

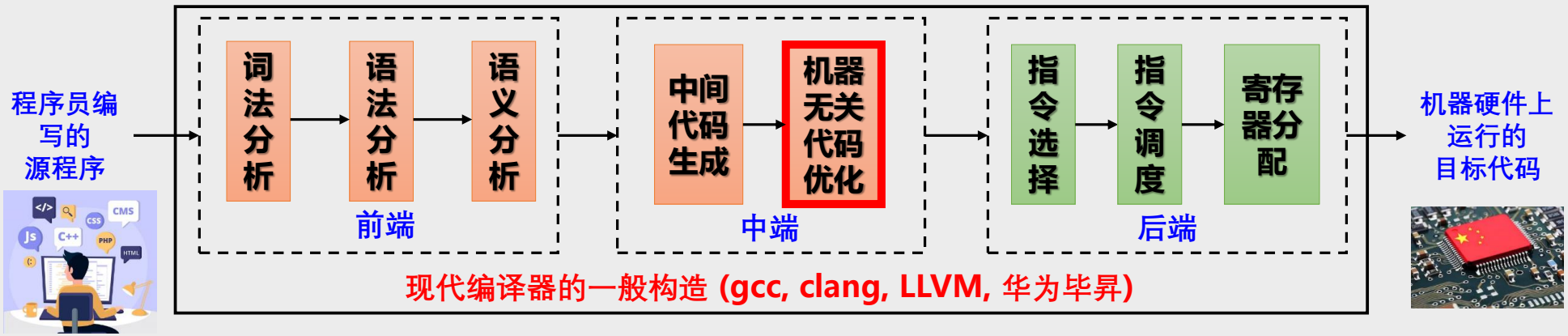


数据流分析介绍

- **分析程序的行为时，必须在其流图上考虑所有的执行路径（在调用或返回语句被执行时，还需要考虑执行路径在多个流图之间的跳转）**
  - 通常，从流图得到的程序执行路径数无限，且执行路径长度没有有限的上界
  - 每个程序点的不同状态数也可能无限
    (程序状态：存储单元到值的映射)



本节提纲



- 代码优化的主要实现方式
- 数据流分析概述
- **数据流分析理论框架**
- **到达-定值分析算法**

数据流分析模式

- 数据流值代表在任一程序点能观测到的所有可能程序状态集合的一个**抽象**
- 对于一个语句*s*
  - s之前的程序点对应的数据流值用IN[*s*]表示
  - s之后的程序点对应的数据流值用OUT[*s*]表示
- 对于一个基本块呢?

数据流分析模式

- **传递函数(transfer function) f**
  - 语句前后两点的数据流值受该语句的语义约束
  - 若沿执行路径正向传播，则$\mathrm{OUT}[s] = f_s(\mathrm{IN}[s])$
  - 若沿执行路径逆向传播，则$\mathrm{IN}[s] = f_s(\mathrm{OUT}[s])$

若基本块B由语句$s_1, s_2, \ldots, s_n$依次组成，则

- $\mathrm{IN}[s_{i+1}] = \mathrm{OUT}[s_i],\ i = 1, 2, \ldots, n-1$

考虑的是在语句执行后输入输出之间的变化关系

基本块上的数据流模式

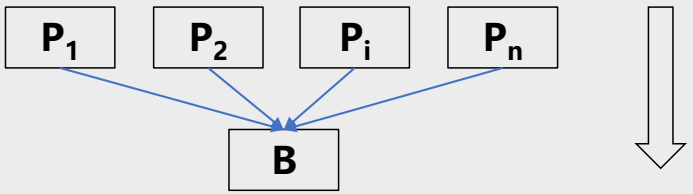
- IN[B]：紧靠基本块B之前的数据流值
  - IN[B] = IN[s_1]
- OUT[B]：紧靠基本块B之后的数据流值
  - OUT[B] = OUT[s_n]
- f_B :基本块B的传递函数
  - 前向数据流： OUT[B] = f_B (IN[B])
    - $f_B = f_n \circ \ldots \circ f_2 \circ f_1$
  - 逆向数据流： IN[B] = f_B (OUT[B])
    - $f_B = f_1 \circ \ldots \circ f_{n-1} \circ f_n$

基本块间的数据流分析模式

- 控制流约束
  - 正向传播

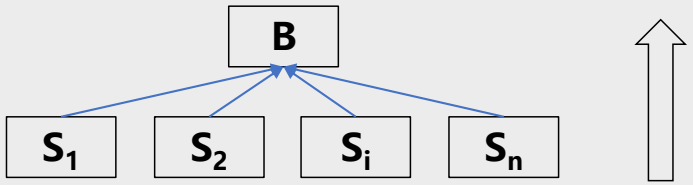
$$IN[B] = \cup_{P\text{是}B\text{的前驱}} OUT[P]$$

U 是汇合的意思，并不一定代表并集，也可能是交集等运算



  - 逆向传播

$$OUT[B] = \cup_{S\text{是}B\text{的后继}} IN[S]$$



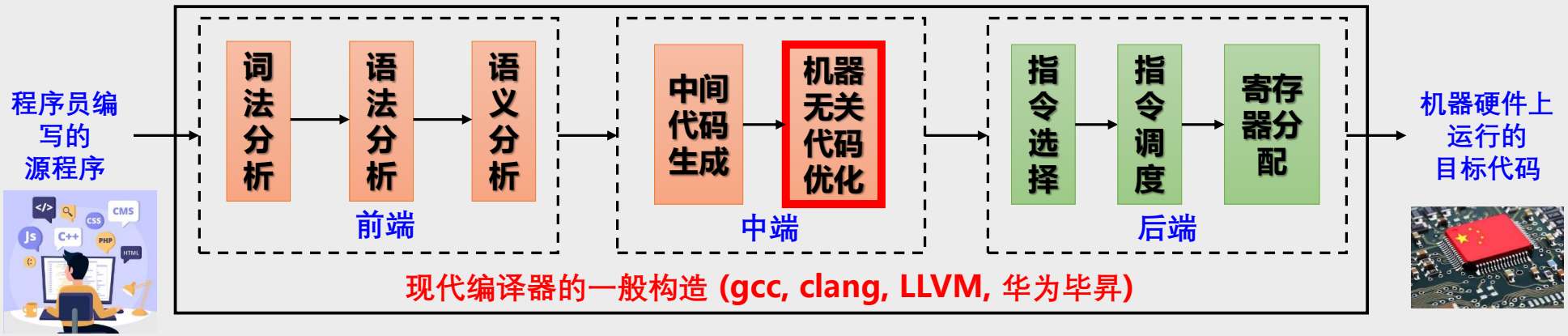
- 约束方程组的解通常不是唯一的
  - 求解的目标是要找到满足这两组约束（控制流约束和迁移约束）的最“精确”解

考虑的是在其他语句或块对于输入的影响和本次执行的输出对其他语句和块的影响

三种有用的分析

- **到达-定值**
  - 应用场景：检测未定义的变量
- **可用表达式**
  - 应用场景：检测全局公共子表达式
- **活跃变量**
  - 应用场景：可用于寄存器分配

本节提纲



- 代码优化的主要实现方式
- 数据流分析概述
- 数据流分析理论框架
- **到达-定值分析算法**

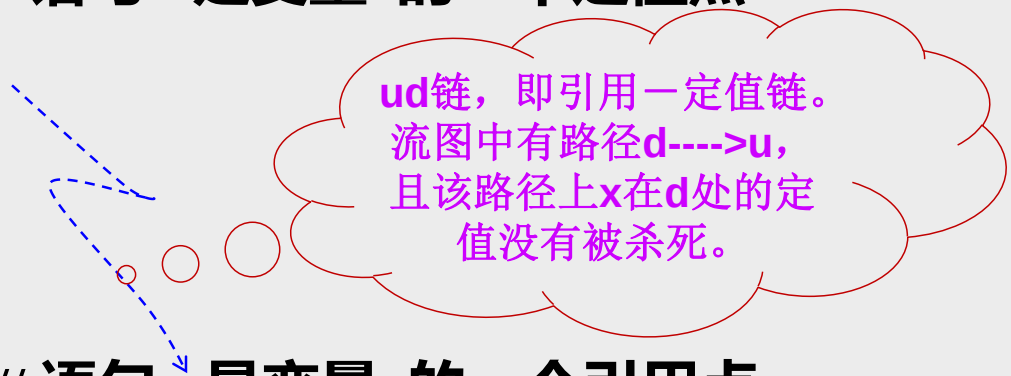
到达-定值分析

- 到达一个程序点的所有定值(gen)
- 定值的注销(kill)
  - 在一条执行路径上，对x的赋值注销先前对x的所有赋值
- 别名给到达-定值的计算带来困难，因此，本章其余部分仅考虑变量无别名的情况

到达-定值分析

- 定值与引用

d : x := y + z // 语句d 是变量x的一个定值点



u : w := x + v // 语句u 是变量x的一个引用点

- 变量x在d点的定值到达u点

到达-定值分析的用途

- **循环不变计算的检测**
  - 如果循环中含有赋值x=y+z，而y和z所有可能的定值都在循环外，那么y+z就是循环不变计算
- **常量合并**
  - 如果对变量x的某次使用只有一个定值到达，且该定值把一个常量赋给x，则可以用该常量替换x
- **错误检测**
  - 判定变量x在p点上是否未经定值就被引用

到达-定值分析

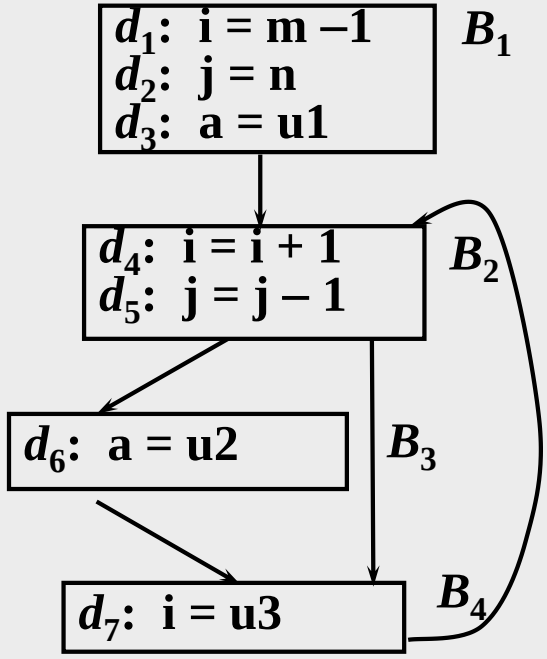
- *gen*和*kill*分别表示一个基本块生成和注销的定值

$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$
$kill\ [B_3] = \{d_3\}$

$gen\ [B_4] = \{d_7\}$
$kill\ [B_4] = \{d_1, d_4\}$



到达-定值分析

- 基本块的*gen*和*kill*是怎样计算的
  - 对三地址指令 *d*: u = v + w，它的状态传递函数是

    $f_d(x) = gen_d \cup (x - kill_d)$
  - 若：$f_1(x) = gen_1 \cup (x - kill_1)$，$f_2(x) = gen_2 \cup (x - kill_2)$

    则：$f_2(f_1(x)) = gen_2 \cup (gen_1 \cup (x - kill_1) - kill_2)$

    $= (gen_2 \cup (gen_1 - kill_2)) \cup (x - (kill_1 \cup kill_2))$
  - 若基本块*B*有*n*条三地址指令

    $f_B(x) = gen_B \cup (x - kill_B)$

    $kill_B = kill_1 \cup kill_2 \cup \ldots \cup kill_n$

    $gen_B = gen_n \cup (gen_{n-1} - kill_n) \cup (gen_{n-2} - kill_{n-1} - kill_n) \cup \ldots \cup (gen_1 - kill_2 - kill_3 - \ldots - kill_n)$

到达-定值分析

- 到达-定值的数据流等式
  - gen_B：B中能到达B的结束点的定值语句
  - $kill_B$：整个程序中决不会到达B结束点的定值
  - IN[B]：能到达B的开始点的定值集合
  - OUT[B]：能到达B的结束点的定值集合

  两组等式（根据*gen*和*kill*定义IN和OUT）
  - $\text{IN}[B] = \cup_{P\text{是}B\text{的前驱}} \text{OUT}[P]$
  - $\text{OUT}[B] = gen_B \cup (\text{IN}[B] - kill_B)$
  - $\text{OUT}[\text{ENTRY}] = \varnothing$
- 到达-定值方程组的迭代求解， 最终到达不动点

到达-定值的迭代计算算法

// 正向数据流分析

引入两个虚拟块：ENTRY、EXIT

(1) OUT[ENTRY] = Ø;

(2) for (除了ENTRY以外的每个块B) OUT[B] = Ø;

(3) while (任何一个OUT出现变化){

(4) for (除了ENTRY以外的每个块B) {

(5) IN[B] = $\cup_{\text{P是B的前驱}}$ OUT[P];

(6) $OUT[B] = gen_B \cup (IN[B] - kill_B)$;

(7) }}

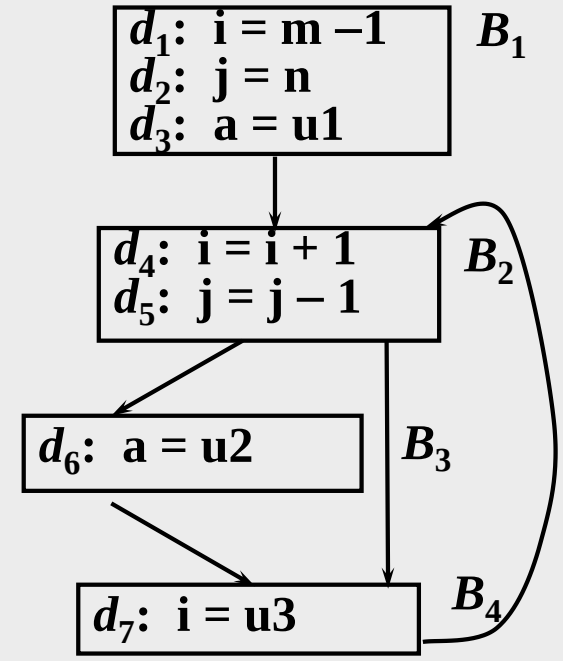
向量求解：集合并操作使用逻辑或，集合相减使用后者求补再逻辑与

到达-定值分析

$IN[B] = \cup_{P是B的前驱} OUT[P]$

$OUT[B] = gen_B \cup (IN[B] - kill_B)$

	IN [B]	OUT [B]
B_1		000 0000
B_2		000 0000
B_3		000 0000
B_4		000 0000



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

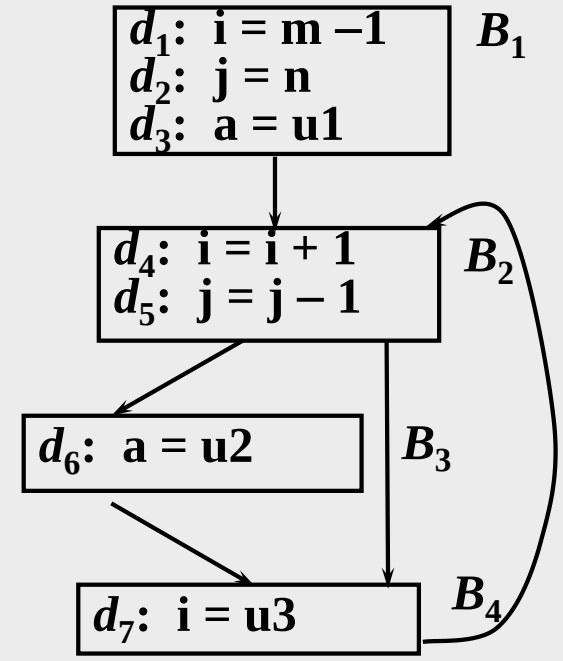
$gen\ [B_3] = \{d_6\}$ $gen\ [B_4] = \{d_7\}$
$kill\ [B_3] = \{d_3\}$ $kill\ [B_4] = \{d_1, d_4\}$

到达-定值分析

IN[*B*] = ∪ *P*是*B*的前驱 OUT[*P*]

OUT[*B*] = $gen_B \cup (IN[B] - kill_B)$

	IN [B]	OUT [B]
B_1	000 0000	000 0000
B_2		000 0000
B_3		000 0000
B_4		000 0000



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

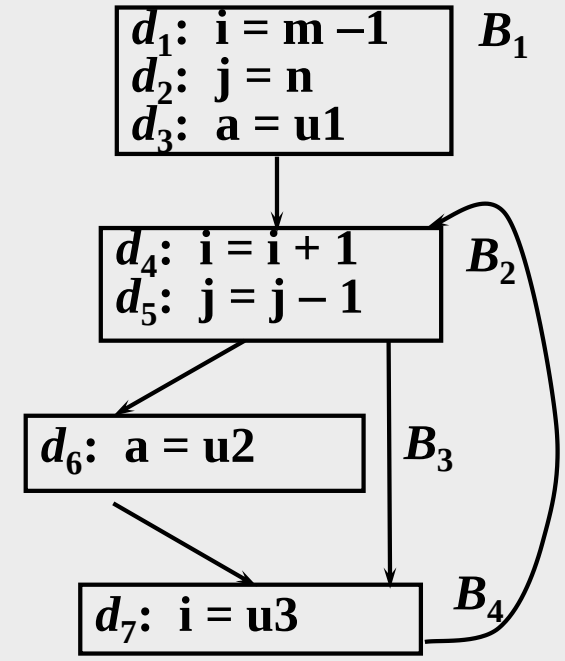
$gen\ [B_3] = \{d_6\}$ $gen\ [B_4] = \{d_7\}$
$kill\ [B_3] = \{d_3\}$ $kill\ [B_4] = \{d_1, d_4\}$

到达-定值分析

IN[B] = ∪ $_{P是B的前驱}$ OUT[P]

OUT[B] = $gen_B \cup (IN[B] - kill_B)$

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2		000 0000
B_3		000 0000
B_4		000 0000



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

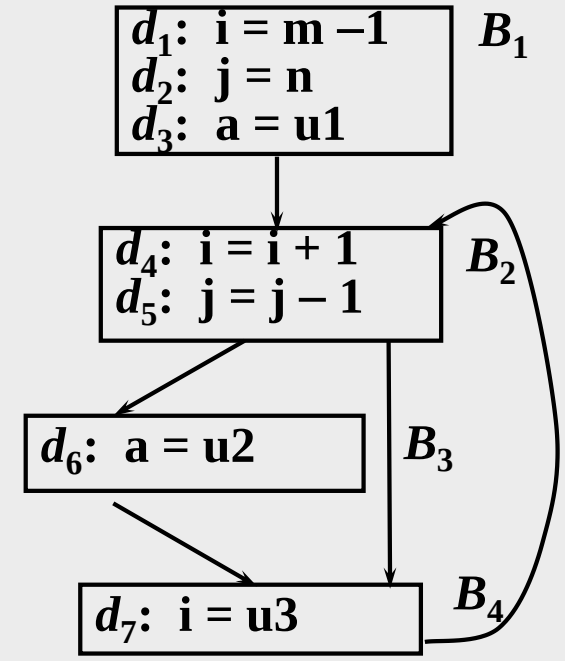
$gen\ [B_3] = \{d_6\}$ $gen\ [B_4] = \{d_7\}$
$kill\ [B_3] = \{d_3\}$ $kill\ [B_4] = \{d_1, d_4\}$

到达-定值分析

$IN[B] = \cup_{P是B的前驱} OUT[P]$

$OUT[B] = gen_B \cup (IN[B] - kill_B)$

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0000	000 0000
B_3		000 0000
B_4		000 0000



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

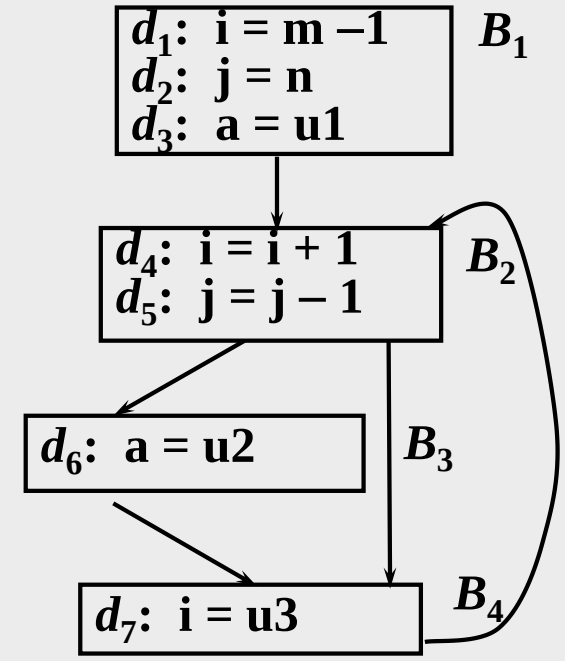
$gen\ [B_3] = \{d_6\}$ $gen\ [B_4] = \{d_7\}$
$kill\ [B_3] = \{d_3\}$ $kill\ [B_4] = \{d_1, d_4\}$

到达-定值分析

IN[B] = ∪ $_{P是B的前驱}$ OUT[P]

OUT[B] = $gen_B \cup (IN[B] - kill_B)$

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0000	001 1100
B_3		000 0000
B_4		000 0000



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

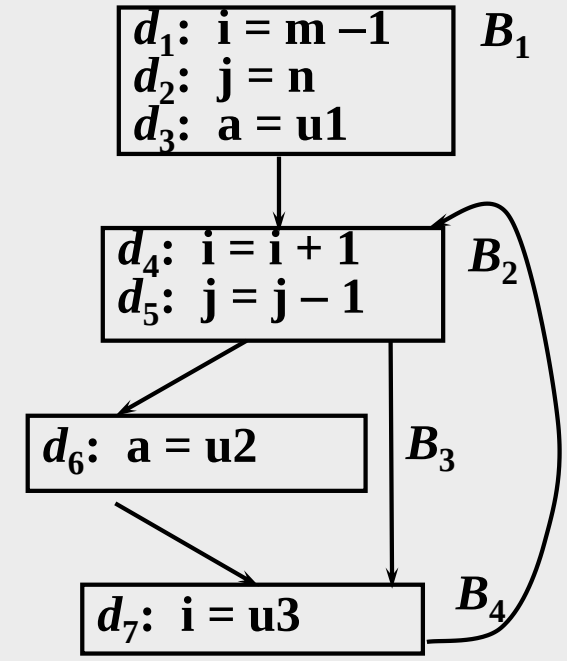
$gen\ [B_3] = \{d_6\}$ $gen\ [B_4] = \{d_7\}$
$kill\ [B_3] = \{d_3\}$ $kill\ [B_4] = \{d_1, d_4\}$

到达-定值分析

IN[*B*] = ∪ *P*是*B*的前驱 OUT[*P*]

OUT[*B*] = $gen_B \cup (IN[B] - kill_B)$

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0000	001 1100
B_3	001 1100	000 0000
B_4		000 0000



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

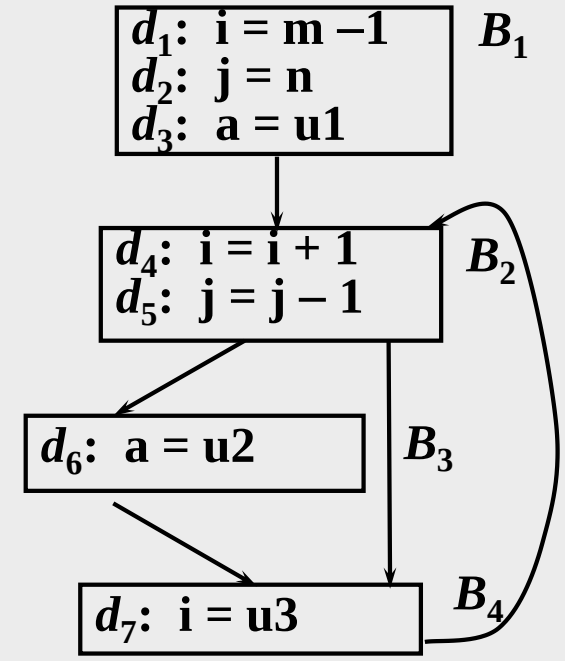
$gen\ [B_3] = \{d_6\}$ $gen\ [B_4] = \{d_7\}$
$kill\ [B_3] = \{d_3\}$ $kill\ [B_4] = \{d_1, d_4\}$

到达-定值分析

IN[B] = ∪ $_{P是B的前驱}$ OUT[P]

OUT[B] = $gen_B \cup (IN[B] - kill_B)$

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0000	001 1100
B_3	001 1100	000 1110
B_4		000 0000



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

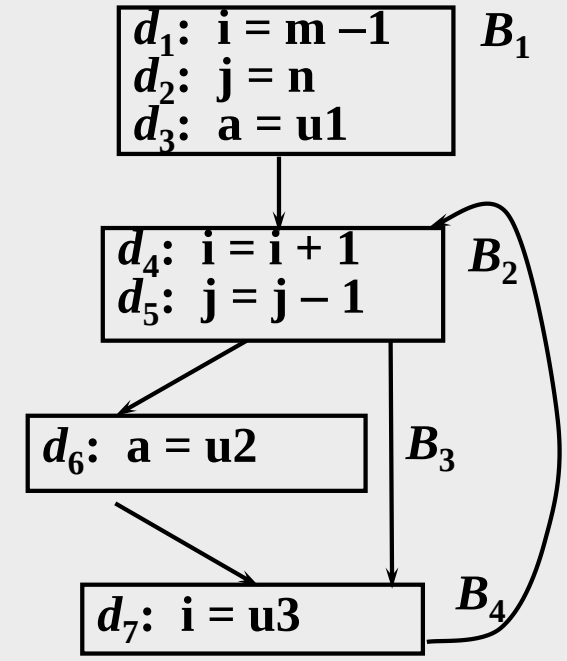
$gen\ [B_3] = \{d_6\}$ $gen\ [B_4] = \{d_7\}$
$kill\ [B_3] = \{d_3\}$ $kill\ [B_4] = \{d_1, d_4\}$

到达-定值分析

$IN[B] = \cup_{P是B的前驱} OUT[P]$

$OUT[B] = gen_B \cup (IN[B] - kill_B)$

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0000	001 1100
B_3	001 1100	000 1110
B_4	001 1110	000 0000



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

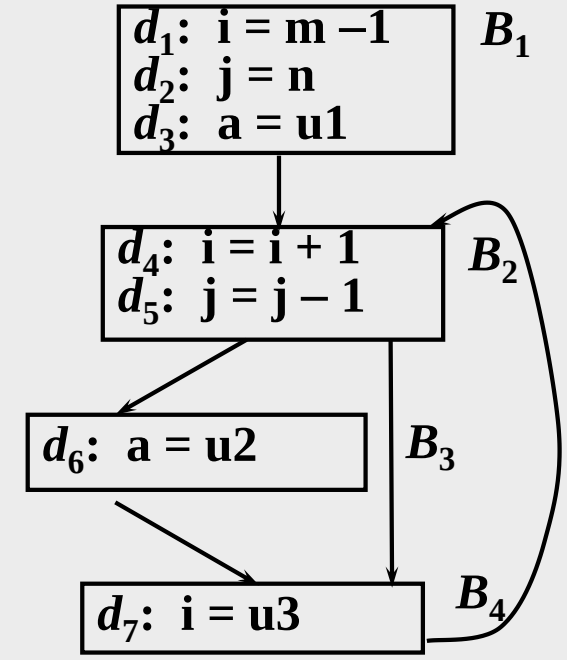
$gen\ [B_3] = \{d_6\}$ $gen\ [B_4] = \{d_7\}$
$kill\ [B_3] = \{d_3\}$ $kill\ [B_4] = \{d_1, d_4\}$

到达-定值分析

IN[B] = ∪ $_{P是B的前驱}$ OUT[P]

OUT[B] = $gen_B \cup (IN[B] - kill_B)$

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0000	001 1100
B_3	001 1100	000 1110
B_4	001 1110	001 0111



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

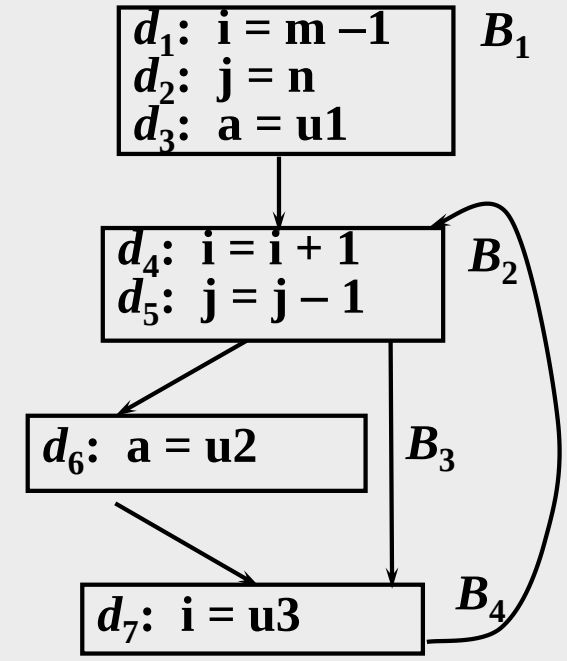
$gen\ [B_3] = \{d_6\}$ $gen\ [B_4] = \{d_7\}$
$kill\ [B_3] = \{d_3\}$ $kill\ [B_4] = \{d_1, d_4\}$

到达-定值分析

$IN[B] = \cup_{P是B的前驱} OUT[P]$

$OUT[B] = gen_B \cup (IN[B] - kill_B)$

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0111	001 1100
B_3	001 1100	000 1110
B_4	001 1110	001 0111



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$ $gen\ [B_4] = \{d_7\}$
$kill\ [B_3] = \{d_3\}$ $kill\ [B_4] = \{d_1, d_4\}$

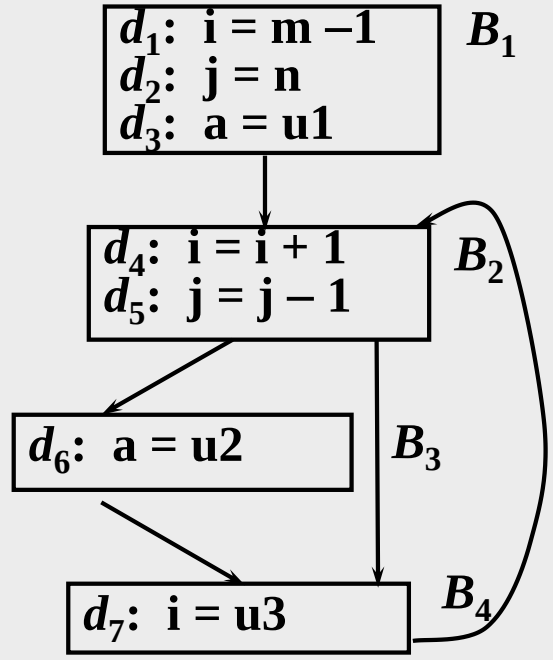
到达-定值分析

IN[B] = ∪ $_{P是B的前驱}$ OUT[P]

OUT[B] = $gen_B \cup (IN[B] - kill_B)$

	IN [B]	OUT [B]
B_1	000 0000	111 0000
B_2	111 0111	001 1110
B_3	001 1100	000 1110
B_4	001 1110	001 0111

不再继续演示迭代计算



$gen\ [B_1] = \{d_1, d_2, d_3\}$
$kill\ [B_1] = \{d_4, d_5, d_6, d_7\}$

$gen\ [B_2] = \{d_4, d_5\}$
$kill\ [B_2] = \{d_1, d_2, d_7\}$

$gen\ [B_3] = \{d_6\}$ $gen\ [B_4] = \{d_7\}$
$kill\ [B_3] = \{d_3\}$ $kill\ [B_4] = \{d_1, d_4\}$

到达-定值分析非向量计算方法

- 迭代计算
  - 计算次序，深度优先序，即 B1 -> B2 -> B3 -> B4
  - 初始值：for all B: IN[B] = ∅; OUT[B] = GEN[B]
  - 第一次迭代：

IN[B1] = ∅; // B1 无前驱结点

OUT[B1] = GEN[B1] ∪ (IN[B1]-KILL[B1]) = GEN[B1] = { d1, d2, d3 }

IN[B2] = OUT[B1] ∪ OUT[B4] = {d1, d2, d3 } ∪{ d7 } = { d1, d2, d3, d7 }

OUT[B2] = GEN[B2] ∪ (IN[B2]-KILL[B2]) = { d4, d5 } ∪ { d3 } = { d3, d4, d5 }

IN[B3] = OUT[B2] = { d3, d4, d5 }

OUT[B3] = { d6 } ∪ ({ d3, d4, d5 } – { d3 }) = { d4, d5, d6 }

IN[B4] = OUT[B3] ∪ OUT[B2] = { d3, d4, d5, d6 }

OUT[B4] = { d7 } ∪ ({ d3, d4, d5, d6 } – { d1, d4 }) = { d3, d5, d6, d7 }

到达-定值分析非向量计算方法

一第二次迭代

IN[B1] = ∅; // B1 无前驱结点

OUT[B1] = GEN[B1] ∪(IN[B1]-KILL[B1]) = GEN[B1] = { d1, d2, d3 }

IN[B2] = OUT[B1] ∪ OUT[B4] = { d1,d2,d3 } ∪ { d3, d5, d6, d7 } = {d1, d2, d3, d5, d6, d7}
OUT[B2] = GEN[B2] ∪(IN[B2]-KILL[B2]) = { d4, d5 } ∪{ d3, d5, d6 } = { d3, d4, d5, d6 }

IN[B3] = OUT[B2] = { d3, d4, d5, d6 }
OUT[B3] = { d6 } ∪ ({ d3, d4, d5, d6 } – { d3 }) = { d4, d5, d6 }

IN[B4] = OUT[B3] ∪ OUT[B2] = { d3, d4, d5, d6 }
OUT[B4] = { d7 } ∪ ({ d3, d4, d5, d6 } – { d1, d4 }) = { d3, d5, d6, d7 }

经过三次迭代后，IN[B]和OUT[B] 不再变化。

到达-定值分析

- **到达-定值数据流等式是正向的方程**

  $\mathbf{OUT}\ [B] = gen\ [B] \cup (\mathbf{IN}\ [B] - kill\ [B])$

  $\mathbf{IN}\ [B] = \cup_{P\text{是}B\text{的前驱}} \mathbf{OUT}\ [P]$

  某些数据流等式是反向的

- **到达-定值数据流等式的合流运算是求并集**

  $\mathbf{IN}\ [B] = \cup_{P\text{是}B\text{的前驱}} \mathbf{OUT}\ [P]$

  某些数据流等式的合流运算是求交集

- **对到达-定值数据流方程，迭代求它的最小解**

  某些数据流方程可能需要求最大解

《编译原理和技术》

机器无关的代码优化 II

谢谢！

《编译原理和技术》

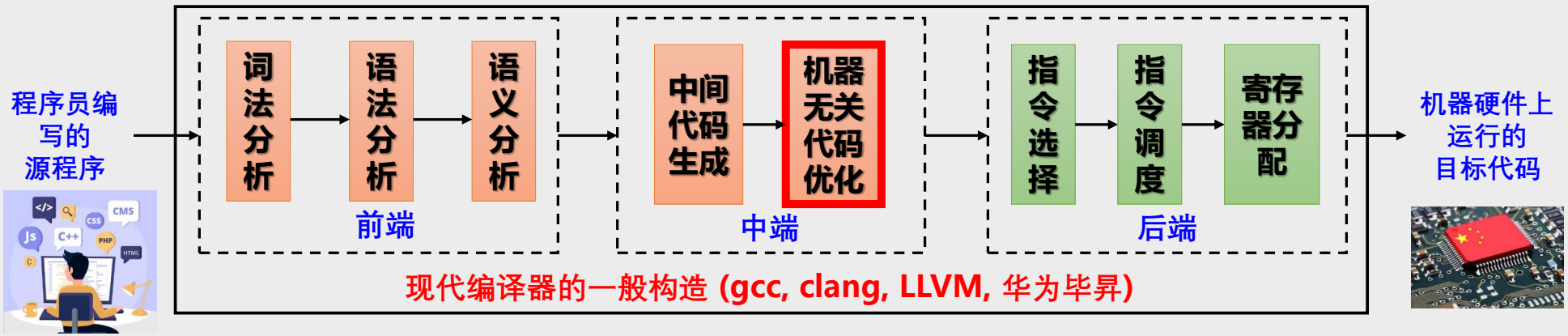
机器无关的代码优化 Ⅲ

——可用表达式分析

中科大计算机学院
李 诚

2022-11-02

本节提纲



- 可用表达式的定义和简单计算
- 可用表达式分析概述及算法介绍
- 可用表达式分析示例

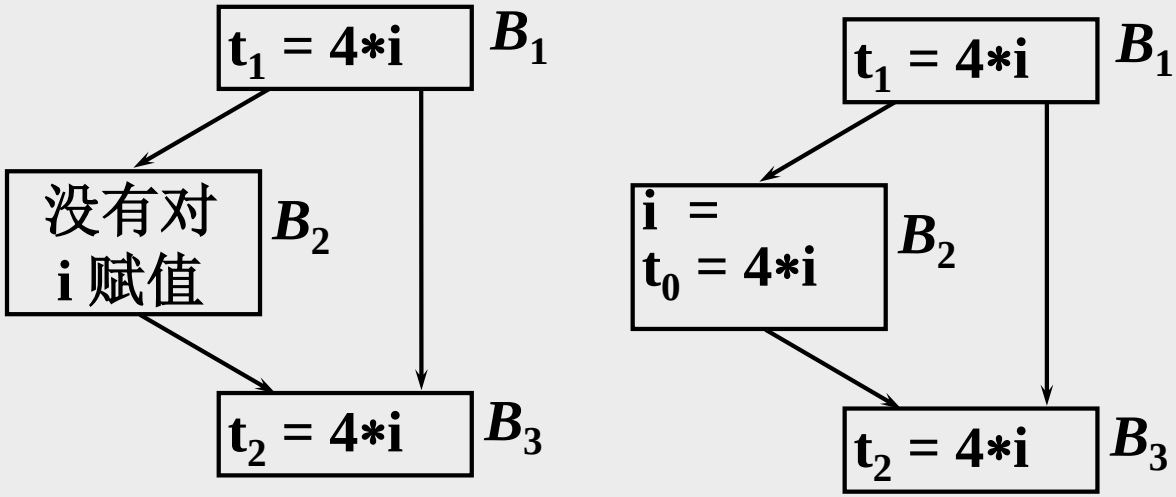
可用表达式

x = y + z	x = y + z	x = y + z
.	.	.
.	y = …	z = …
.	.	.
p	***p***	***p***
y + z 在*p*点	y + z 在*p*点	y + z 在*p*点
可用	不可用	不可用

可用表达式的应用

消除全局公共子表达式

- 例：下面两种情况下， 4*i在B_3的入口都可用



可用表达式

- **基本块生成的表达式：**

  基本块中语句d：x = y + z的前、后点分别为点p与点q。设在点p处可用表达式集合为S（基本块入口点处S为空集），那么经过语句d之后，在点q处可用表达式集合如下构成：

  (1) S = S ∪ { y+z }

  (2) S = S – { S 中所有涉及变量x的表达式 }

  注意，步骤(1)和(2)不可颠倒

可用表达式

- **基本块生成的表达式：**

  基本块中语句d：x = y + z的前、后点分别为点p与点q。设在点p处可用表达式集合为S（基本块入口点处S为空集），那么经过语句d之后，在点q处可用表达式集合如下构成：

  (1) S = S ∪ { y+z }

  (2) S = S – { S 中所有涉及变量x的表达式 }

  注意，步骤(1)和(2)不可颠倒，x可能就是y或z。

  如此处理完基本块中所有语句后，可以得到基本块生成的可用表达式集合S；

可用表达式

- **基本块生成的表达式：**

  基本块中语句d：x = y + z的前、后点分别为点p与点q。设在点p处可用表达式集合为S（基本块入口点处S为空集），那么经过语句d之后，在点q处可用表达式集合如下构成：

  (1) S = S ∪ { y+z }

  (2) S = S – { S 中所有涉及变量x的表达式 }

  注意，步骤(1)和(2)不可颠倒，x可能就是y或z。

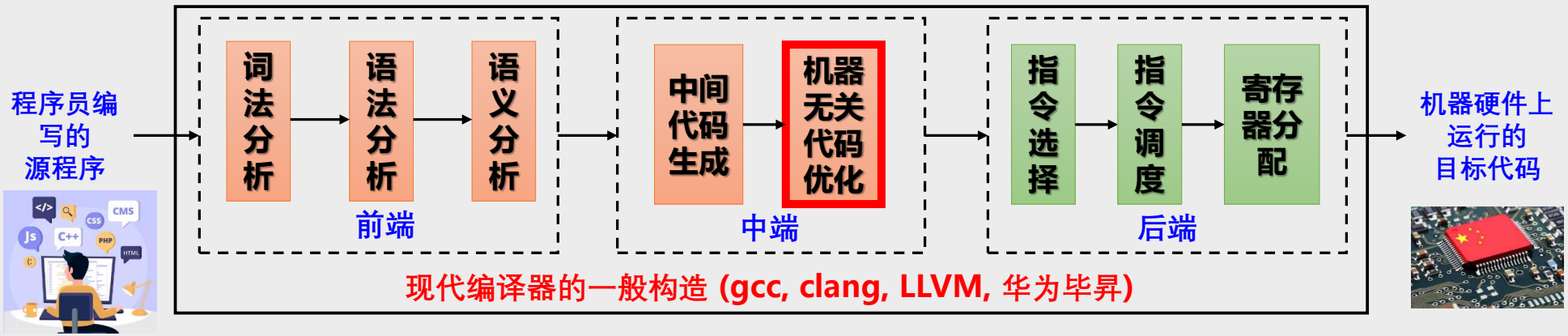
  如此处理完基本块中所有语句后，可以得到基本块生成的可用表达式集合S；

- **基本块杀死的表达式：所有其他类似y+z的表达式，基本块中对y或z定值，但基本块没有生成y+z。**

示例：基本块生成的表达式

语句	可用表达式
a = b + c	Ø
b = a – d	{ b + c }
c = b + c	{ a – d } // b+c被杀死
d = a – d	{ a – d } // b+c被杀死 Ø // a – d 被杀死

本节提纲



- 可用表达式的定义和简单计算
- 可用表达式分析概述及算法介绍
- 可用表达式分析示例

可用表达式分析

- 定义
  - 若到点p的每条执行路径都计算x op y，并且计算后没有对x或y赋值，那么称x op y在点p可用
  - e_gen_B：块B产生的可用表达式集合
  - e_kill_B：块B注销的可用表达式集合
  - IN [B]：块B入口的可用表达式集合
  - OUT [B]：块B出口的可用表达式集合

可用表达式分析

- **数据流等式**
  - **OUT [B] = $e\_gen_B \cup$ (IN [B] – e_kill_B)**
  - **IN [B] = $\cap_{P是B的前驱}$ OUT [P]**
  - OUT [**ENTRY**] = ∅
    - 在ENTRY的出口处没有可用表达式
- **同先前的主要区别**
  - 使用∩而不是∪作为这里数据流等式的汇合算符
  - 只有当一个表达式在B的所有前驱的结尾处都可用，那么它才会在B的开头可用
  - 求最大解而不是最小解

可用表达式数据流分析

- 迭代算法：

*U*是全体表达式集合

(1) OUT[ENTRY] = ø

(2) for(除ENTRY之外的每个基本块B) OUT[B]= *U*

(3) while(某个OUT值发生变化) {

(4) for(除ENTRY之外的每个基本块B){

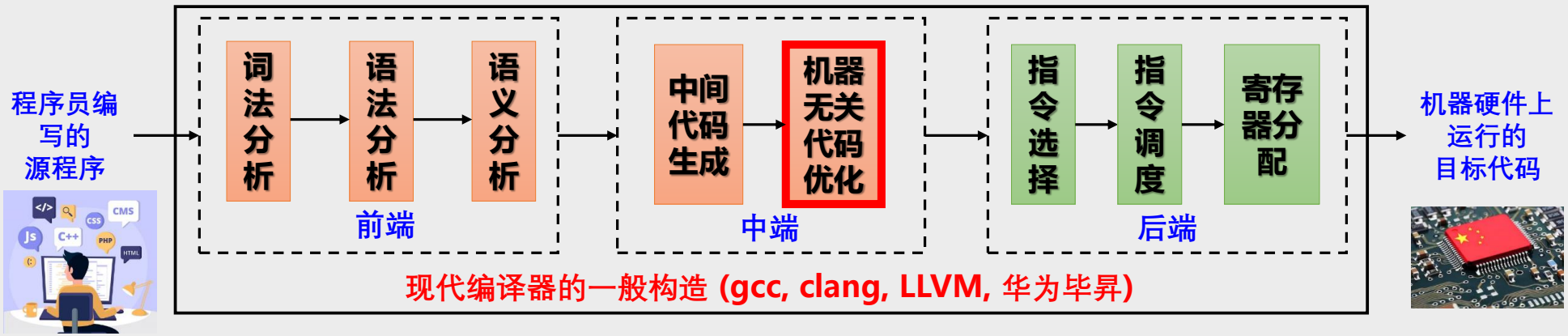
(5) IN[B] = ∩ P是B的前驱基本块(OUT[P])

(6) OUT[B] = $e\_gen_B \cup (IN[B] - e\_kill_B)$

} // end-of-for

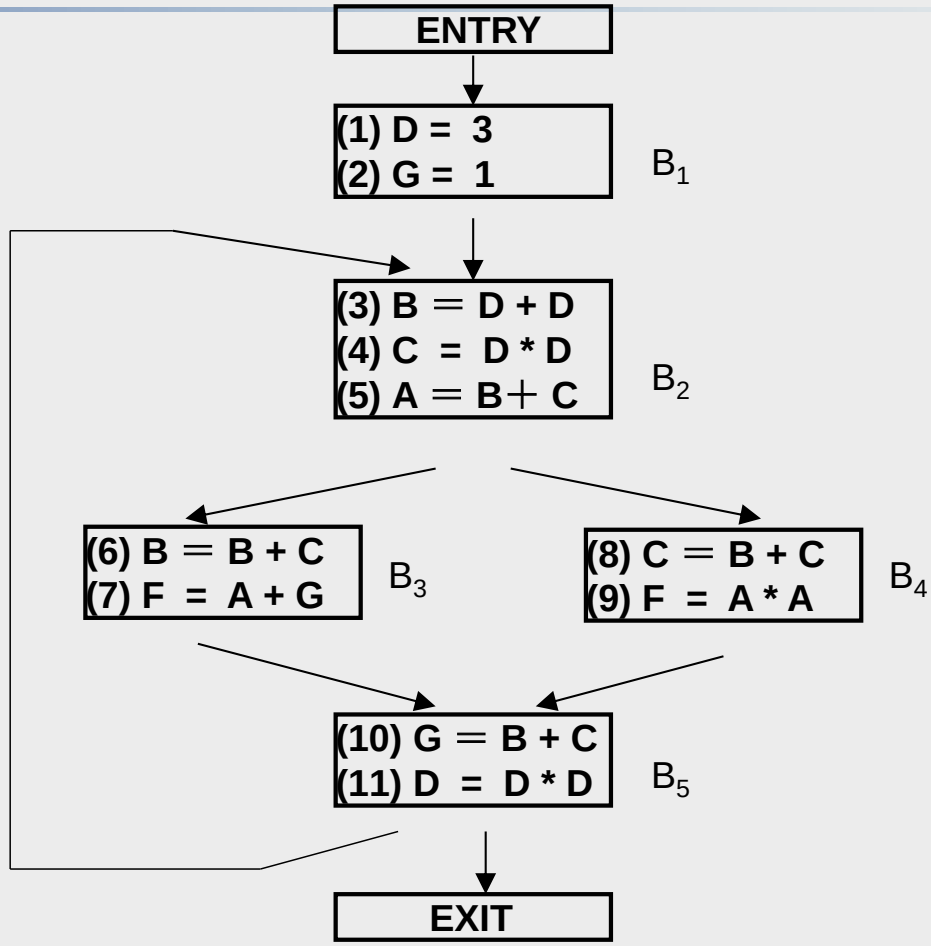
} // end-of-while

本节提纲



- 可用表达式的定义和简单计算
- 可用表达式分析概述及算法介绍
- **可用表达式分析示例**

示例：可用表达式分析


ENTRY
(1) D = 3
(2) G = 1
B1
(3) B = D + D
(4) C = D * D
(5) A = B+ C
B2
(6) B = B + C
(7) F = A + G
B3
(8) C = B + C
(9) F = A * A
B4
(10) G = B + C
(11) D = D * D
B5
EXIT

示例：可用表达式分析

基本块	前驱	后继
ENTRY	—	B_1
B_1	ENRTY	B_2
B_2	B_1 B_5	B_3 B_4
B_3	B_2	B_5
B_4	B_2	B_5
B_5	B_3 B_4	B_2 EXIT
EXIT	B_5	—

示例：可用表达式分析

基本块	e_gen	e_kill
ENTRY	Ø	Ø
B_1	{3, 1}	{ D+D, D*D, A+G }
B_2	{ D+D, D*D, B+C }	{ A*A, A+G }
B_3	{ A+G }	{ B+C }
B_4	{ A * A }	{ B+C }
B_5	{ B+C }	{ A+G, D*D, D+D }
EXIT	Ø	Ø
全部表达式*U*={ 3, 1, D+D, D*D, B+C, A+G, A*A }		

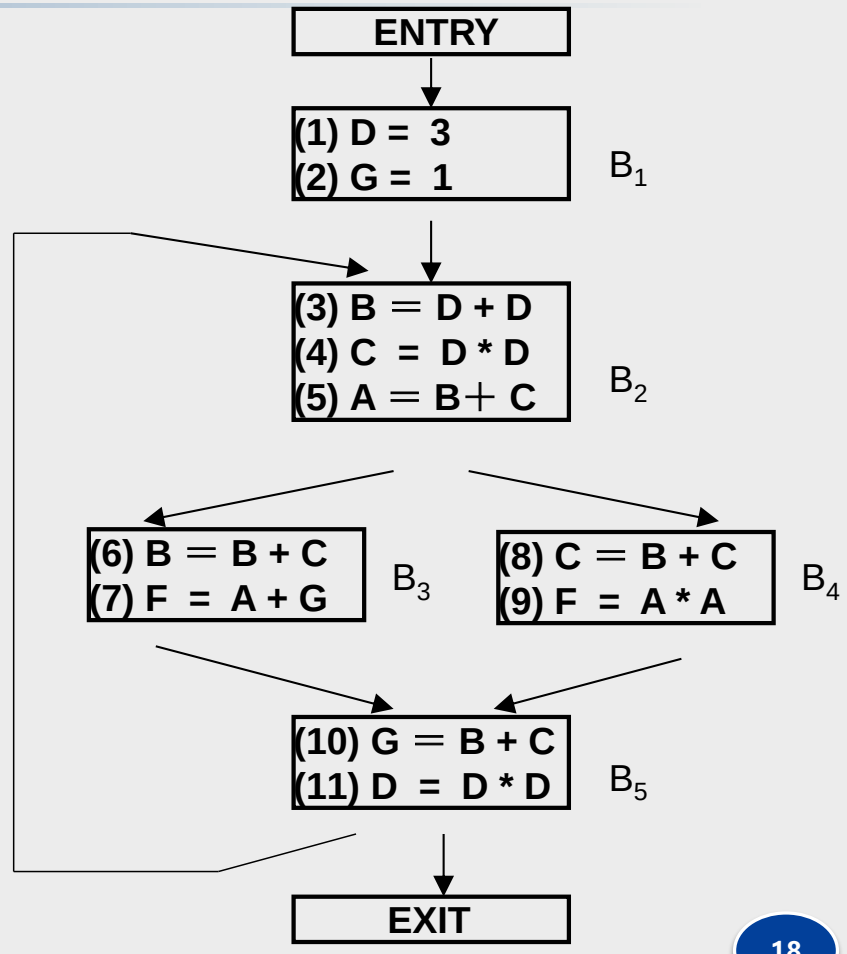
示例：可用表达式分析

基本块		e_kill
ENTRY		Ø
B_1		{ D+D, D*D, A+G }
B_2		{ A*A, A+G }
B_3	{ A+G }	{ B+C }
B_4	{ A * A }	{ B+C }
B_5	{ B+C }	{ A+G, D*D, D+D }
EXIT	Ø	Ø
全部表达式*U*={ 3, 1, D+D, D*D, B+C, A+G, A*A }		

- B2块的e_kill集合不包含B+C，因为虽然B和C的赋值改变了B+C的值，但是最后一个语句再次计算了B+C，这样B+C又成为可用表达式。生命力顽强，没有被kill掉。
- 从另一个视角来看，即便是e_kill中包含了B+C，OUT集合计算的时候也会被e_gen中的B+C覆盖掉。

示例：可用表达式分析

- 可用表达式的迭代计算
  - 深度优先序，即 B1 -> B2 -> B3 -> B4 -> B5 -> EXIT
  - 边界值：OUT[ENTRY] = ∅;
  - 初始化：for all NON-ENTRY B: OUT[B] = U ;



示例：可用表达式分析

- **第一次迭代：（all NON-ENTRY B）**

(1) IN[B1] = OUT[ENTRY] = ∅; // B1 前驱仅为ENTRY

OUT[B1] = e_gen[B1] ∪ (IN[B1] – e_kill [B1])

= e_gen[B1] = { 3, 1 } //变化

(2) IN[B2] = OUT[B1] ∩OUT[B5]

= { 3, 1 } ∩ U = { 3, 1 }

OUT[B2] = e_gen[B2] ∪ (IN[B2] – e_kill [B2])

= { D+D, D*D, B+C } ∪ ({ 3, 1 } – {A*A, A+G })

= {3, 1, D+D, D*D, B+C } //变化

示例：可用表达式分析

- **第一次迭代：（all NON-ENTRY B）**

(3) IN[B3] = OUT[B2]

= {3, 1, D+D, D*D, B+C }

OUT[B3] = e_gen[B3] ∪ (IN[B3] – e_kill[B3])

= { A+G } ∪ ({ 3, 1, D+D, D*D, B+C } – { B+C })

= { 3, 1, D+D, D*D, A+G } //变化

(4) IN[B4] = OUT[B2]

= {3, 1, D+D, D*D, B+C }

OUT[B4] = e_gen[B4] ∪ (IN[B4] – e_kill[B4])

= { A * A } ∪({ 3, 1, D+D, D*D, B+C } – { B+C })

= { 3, 1, D+D, D*D, A * A } //变化

示例：可用表达式分析

- **第一次迭代：（all NON-ENTRY B）**

(5) IN[B5] = OUT[B3] ∩ OUT[B4]

= { 3, 1, D+D, D*D, A+G } ∩ { 3, 1, D+D, D*D, A * A }

= { 3, 1, D+D, D*D }

OUT[B5] = e_gen[B5] ∪ (IN[B5] – e_kill[B5])

= {B+C} ∪ ({3,1,D+D, D*D} – {A+G, D*D, D+D})

= { 3, 1, B+C } //变化

(6) IN[EXIT] = OUT[B5] = { 3, 1, B+C }

OUT[EXIT] = e_gen[EXIT] ∪ (IN[EXIT] –e_kill [EXIT])

= ⌀ ∪ ({ 3, 1, B+C } – ⌀)

= { 3, 1, B+C } //变化

示例：可用表达式分析

- **第二次迭代：（all NON-ENTRY B）**

(1) IN[B1] = OUT[ENTRY] = ∅;

OUT[B1] = e_gen[B1] ∪ (IN[B1] – e_kill[B1])

= e_gen[B1] = { 3, 1 } // 不变

(2) IN[B2] = OUT[B1] ∩ OUT[B5]

= { 3, 1 } ∩ { 3, 1, B+C } = { 3, 1 } // 不变

OUT[B2] = e_gen[B2] ∪ (IN[B2] – e_kill[B2])

= { D+D, D*D, B+C } ∪ ({ 3, 1 } – {A*A, A+G })

= {3, 1, D+D, D*D, B+C } // 不变

示例：可用表达式分析

- **第二次迭代：（all NON-ENTRY B）**

(3) IN[B3] = OUT[B2]

= {3, 1, D+D, D*D, B+C } //不变

OUT[B3] = e_gen[B3] ∪ (IN[B3] – e_kill[B3])

= { A+G } ∪ ({ 3, 1, D+D, D*D, B+C } – { B+C })

= { 3, 1, D+D, D*D, A+G } //不变

(4) IN[B4] = OUT[B2]

= {3, 1, D+D, D*D, B+C } //不变

OUT[B4] = e_gen[B4] ∪ (IN[B4] – e_kill[B4])

= { A * A } ∪({ 3, 1, D+D, D*D, B+C } – { B+C })

= { 3, 1, D+D, D*D, A * A } //不变

示例：可用表达式分析

- **第二次迭代：（all NON-ENTRY B）**

(5) IN[B5] = OUT[B3] ∩ OUT[B4]

= { 3, 1, D+D, D*D, A+G } ∩{ 3, 1, D+D, D*D, A * A }

= { 3, 1, D+D, D*D } //不变

OUT[B5] = e_gen[B5] ∪ (IN[B5] – e_kill[B5])

= {B+C} ∪ ({3,1,D+D, D*D} – {A+G, D*D, D+D})

= { 3, 1, B+C } //不变

(6) IN[EXIT] = OUT[B5] = { 3, 1, B+C } //不变

OUT[EXIT] = e_gen[EXIT] ∪ (IN[EXIT] –e_kill[EXIT])

= ⌀ ∪ ({ 3, 1, B+C } – ⌀)

= { 3, 1, B+C } //不变

《编译原理和技术》

机器无关的代码优化 III

谢谢！

《编译原理和技术》

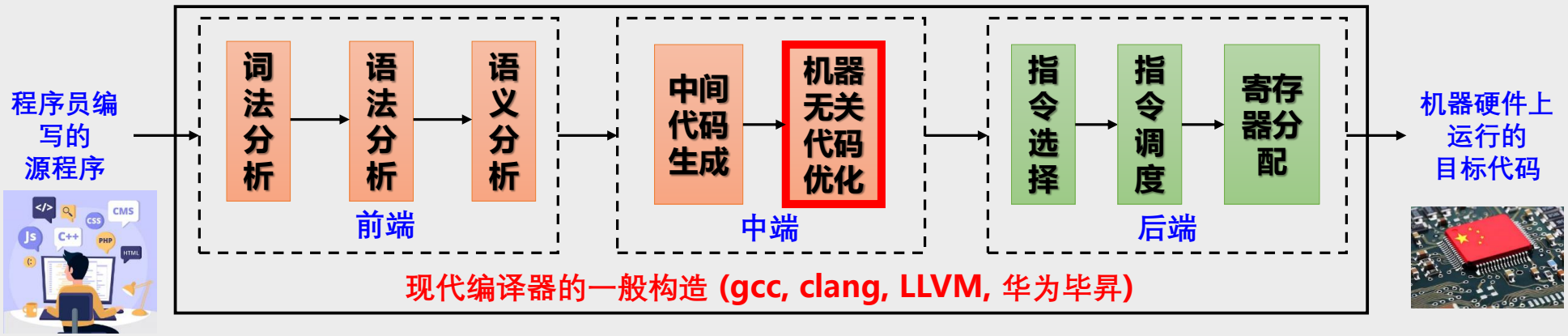
机器无关的代码优化 IV

——活跃变量分析

中科大计算机学院
李 诚

2022-11-07

本节提纲



- 活跃变量定义及应用
- 活跃变量分析算法
- 示例驱动的分析流程

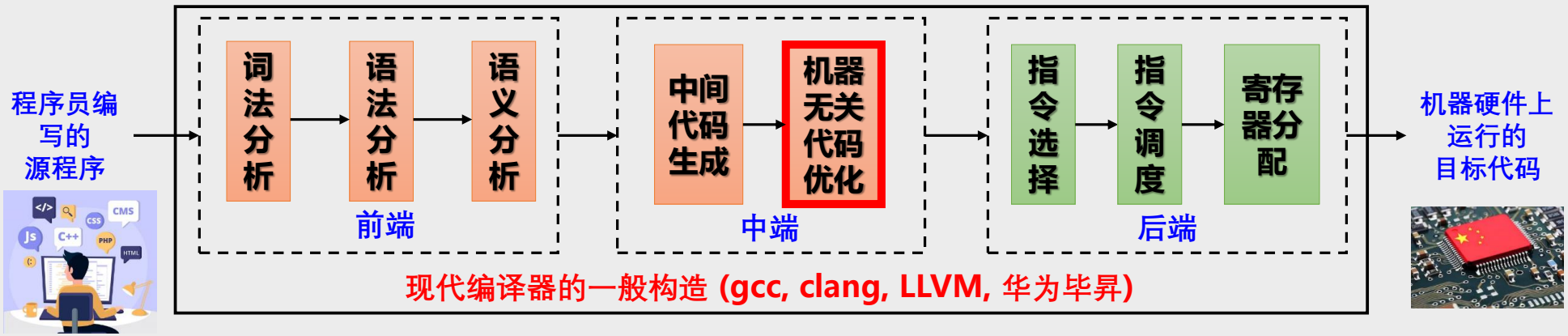
活跃变量

- **定义：**
  - 对于变量x和程序点p，如果x的值在*p*点开始的某条执行路径上被引用，则说x在*p*点活跃（live），否则称x在*p*点已经死亡（dead）

活跃变量分析的应用

- **为基本块分配寄存器**
  - 如果所有寄存器都被占用，且还需要申请一个寄存器，则应该考虑使用已经存放死亡值的寄存器
  - 如果一个值在基本块结尾处是死的，就不必在结尾处保存这个值了

本节提纲



- 活跃变量定义及应用
- 活跃变量分析算法
- 示例驱动的分析流程

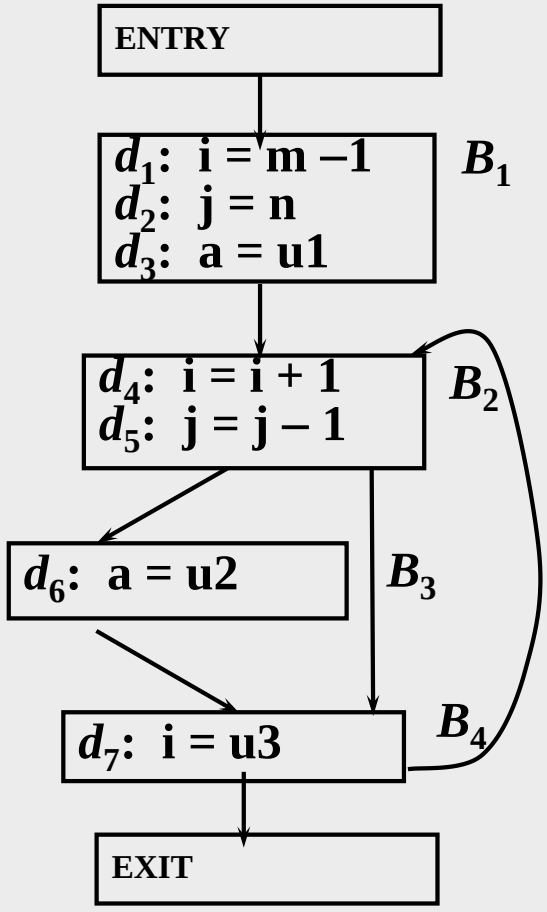
活跃变量分析

相关定义

- IN[B]：块B开始点的活跃变量集合
- OUT[B]：块B结束点的活跃变量集合
- use_B：块B中有引用，且在引用前在B中没有被定值的变量集
- def_B：块B中有定值，且该定值前在B中没有被引用的变量集

use与def的计算

- 例
  - $use[B_1]$ = { m, n, u1 }
  - $def[B_1]$ = {i, j, a}
  - **$use[B_2]$ = { i, j }**
  - **$def[B_2]$ = {}**
  - $use[B_3]$ = { u2}
  - $def[B_3]$ = {a}
  - $use[B_4]$ = { u3 }
  - $def[B_4]$ = {i}



求解方程组

- **活跃变量分析逆向数据流等式**
  - IN [EXIT] = ∅
    - 边界条件：程序出口处没有活跃变量
  - **OUT[*B*] = ∪ $_{S是B的后继}$ IN [*S*]**
  - **IN [*B*] = use_B ∪ (OUT [*B*] – def_B)**
    - 入口处活跃：1）在B中重定值之前被使用；2)离开时活跃且没有在B中被定值
- **和到达-定值等式之间的联系与区别**
  - 都以集合并算符作为它们的汇合算符
  - 信息流动方向相反，IN和OUT的作用相互交换
  - *use*和*def*分别取代*gen*和*kill*
  - 仍然需要最小解

活跃变量的迭代计算算法

输入：流图G，其中每个基本块B的use和def都已计算

输出：IN[B]和OUT[B]

IN[EXIT] = Ø;

for (除了EXIT以外的每个块B) IN[B] = Ø;

while (某个IN值出现变化) {

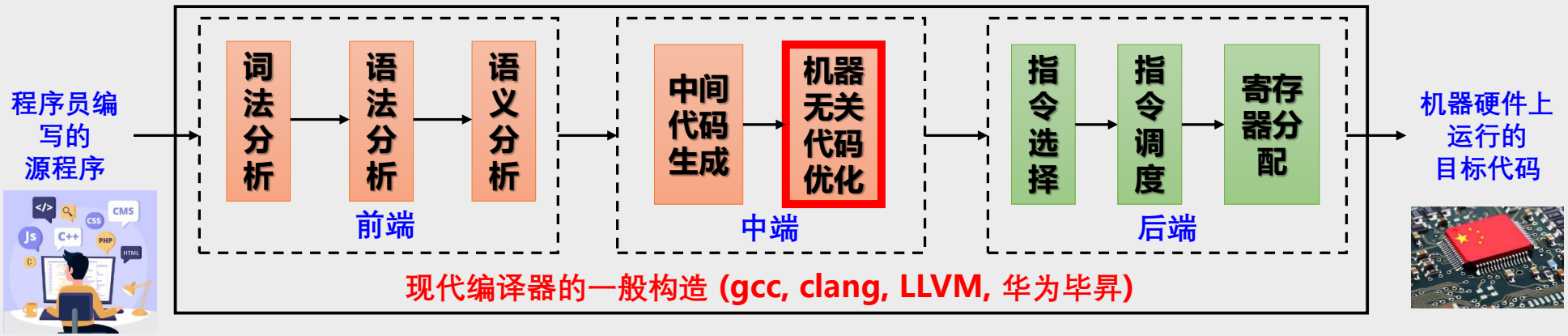
for (除了EXIT以外的每个块B) {

$\mathrm{OUT}[B] = \cup_{S\text{是}B\text{的后继}} \mathrm{IN}[S]$

$\mathrm{IN}[B] = use_B \cup (\mathrm{OUT}[B] - def_B)$;

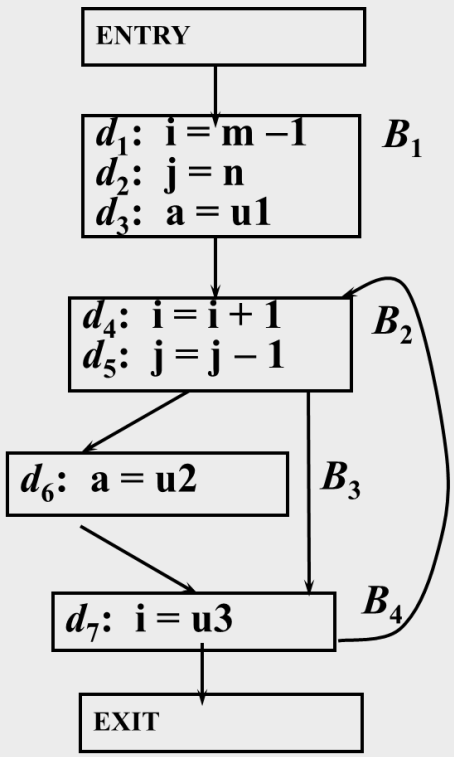
}}

本节提纲



- 活跃变量定义及应用
- **活跃变量分析算法**
- 示例驱动的分析流程

活跃变量分析-举例1



$use[B_1]$ = { m, n, u1 }
$def[B_1]$ = {i, j, a}
$use[B_2]$ = { i, j }
$def[B_2]$ = {}
$use[B_3]$ = { u2}
$def[B_3]$ = {a}
$use[B_4]$ = { u3 }
$def[B_4]$ = {i}

IN[EXIT] = Ø;

for (除了EXIT以外的每个块B) IN[B] = Ø;

while (某个IN值出现变化) {

for (除了EXIT以外的每个块B) {

$OUT[B] = \cup_{S是B的后继} IN[S]$

$IN[B] = use_B \cup (OUT[B] - def_B)$;

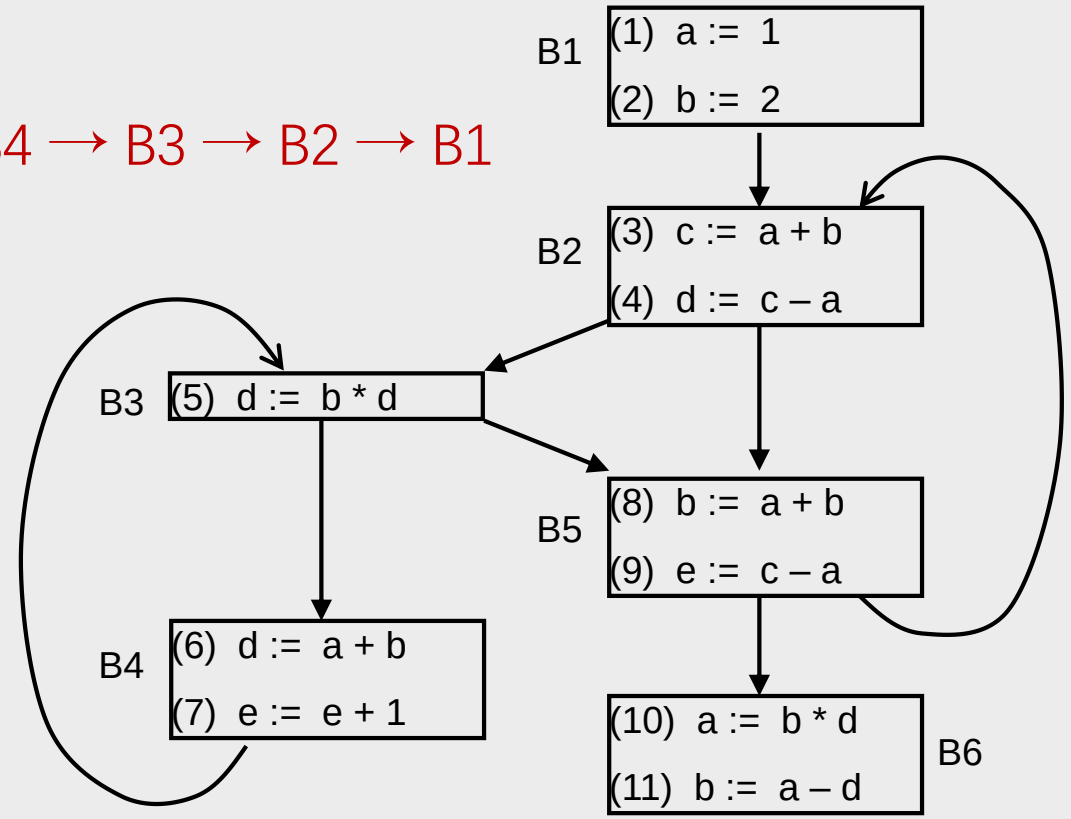
}}

	$OUT[B]^1$	$IN[B]^1$	$OUT[B]^2$	$IN[B]^2$	$OUT[B]^3$	$IN[B]^3$
B_4		u3	i, j, u2, u3	j, u2, u3	i, j, u2, u3	j, u2, u3
B_3	u3	u2, u3	j, u2, u3	j, u2, u3	j, u2, u3	j, u2, u3
B_2	u2, u3	i, j, u2, u3	j, u2, u3	i, j, u2, u3	j, u2, u3	i, j, u2, u3
B_1	i, j, u2, u3	m, n, u1, u2, u3	i, j, u2, u3	m, n, u1, u2, u3	i, j, u2, u3	m, n, u1, u2, u3

活跃变量分析-举例2

计算次序

* B6 → B5 → B4 → B3 → B2 → B1



基本块出口活跃变量

- 各基本块USE和DEF如下，

  USE[B1] = { } ; DEF[B1] = { a, b }

  USE[B2] = { a, b } ; DEF[B2] = { c, d }

  USE[B3] = { b, d } ; DEF[B3] = { }

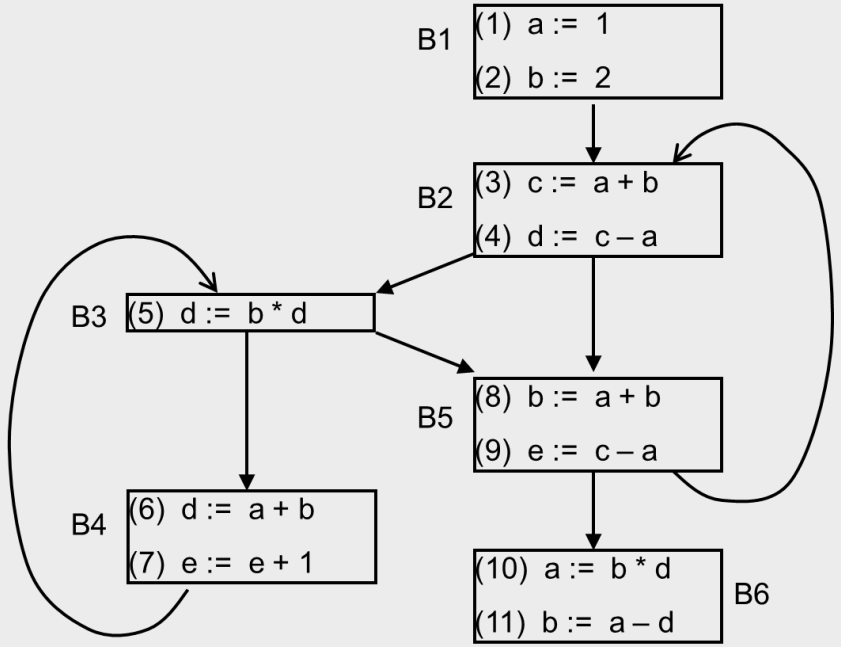
  USE[B4] = { a, b, e } ; DEF[B4] = { d }

  USE[B5] = { a, b, c } ; DEF[B5] = { e }

  USE[B6] = { b, d } ; DEF[B6] = { a }

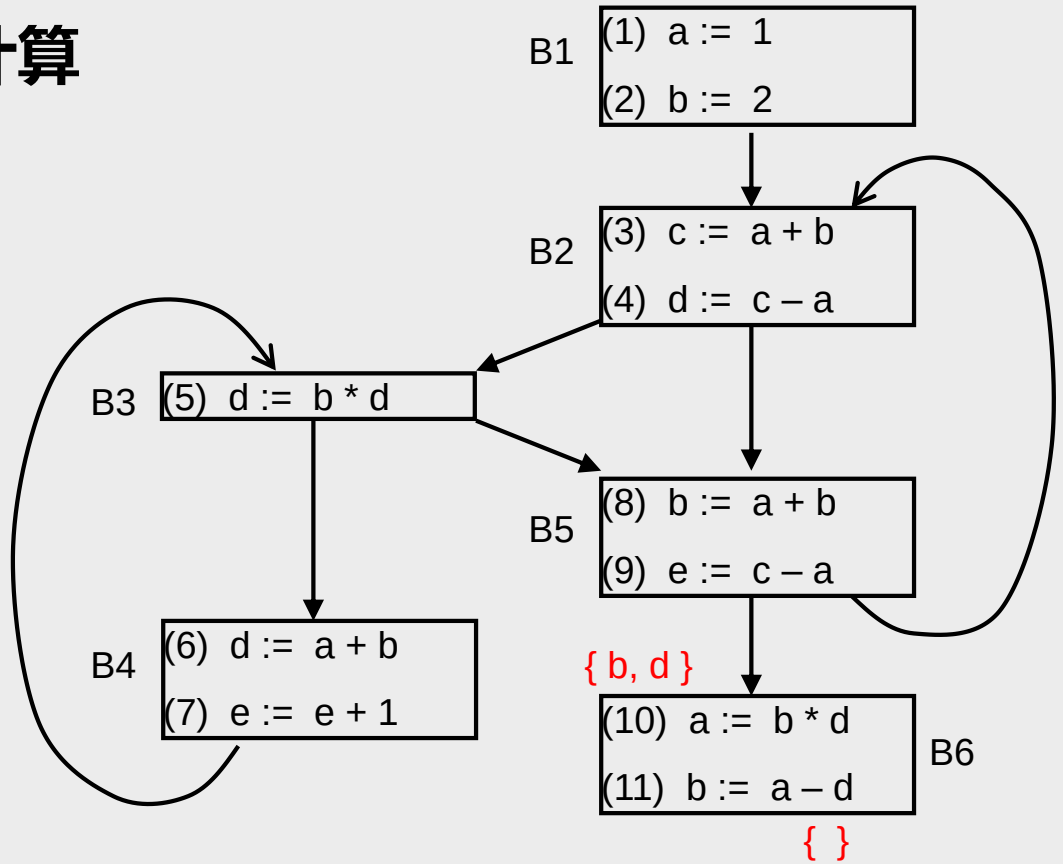
- 初始值，all B, IN[B] = { },

  OUT[B6]={ }//出口块



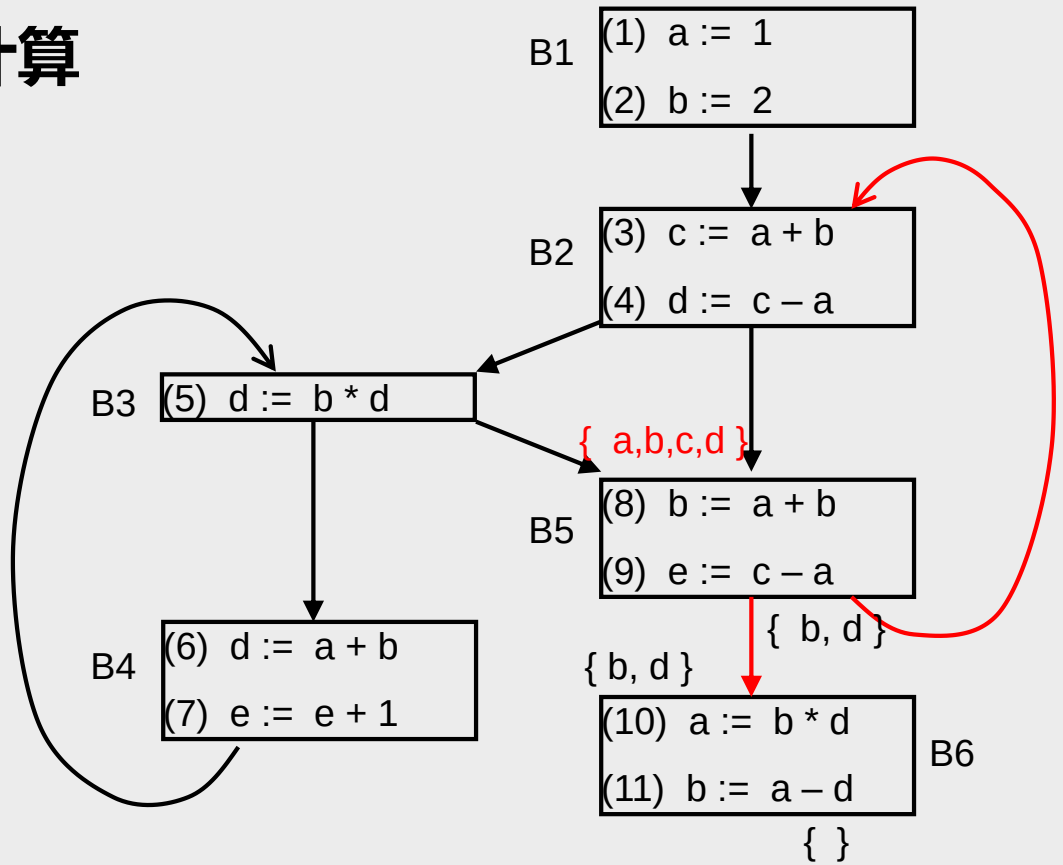
基本块出口活跃变量

- 第一次迭代计算



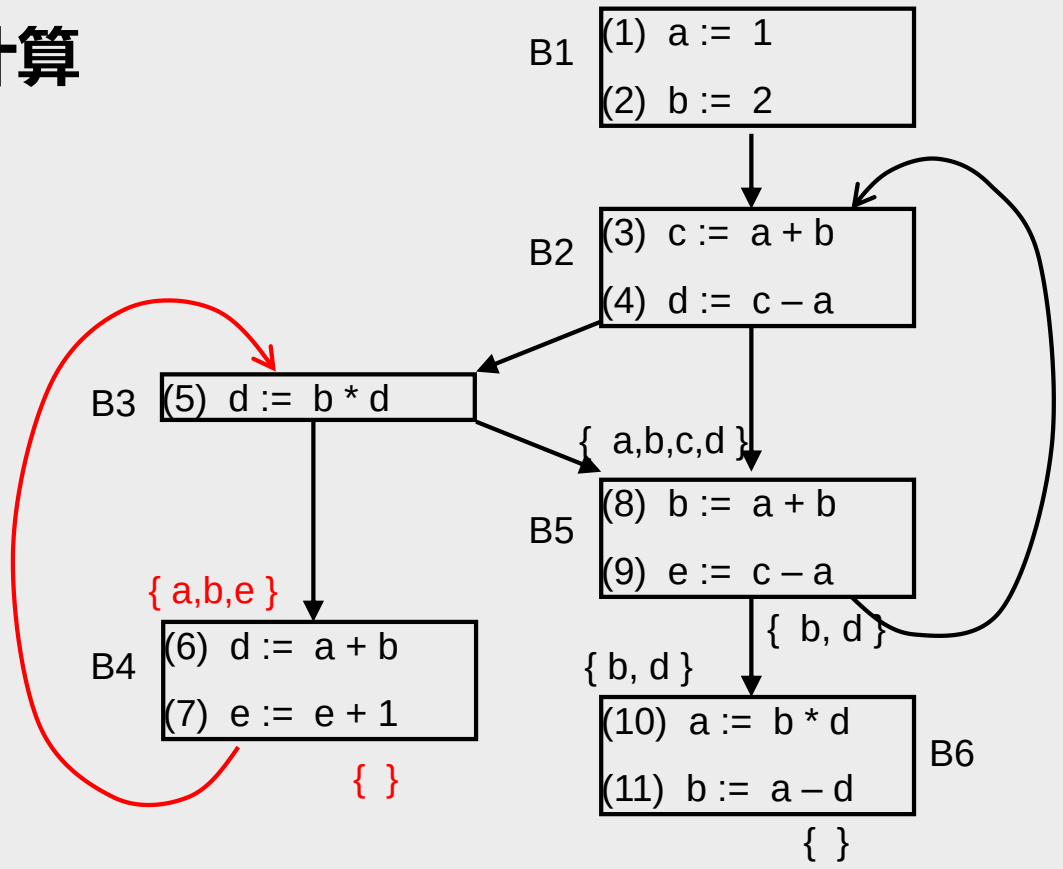
基本块出口活跃变量

- 第一次迭代计算



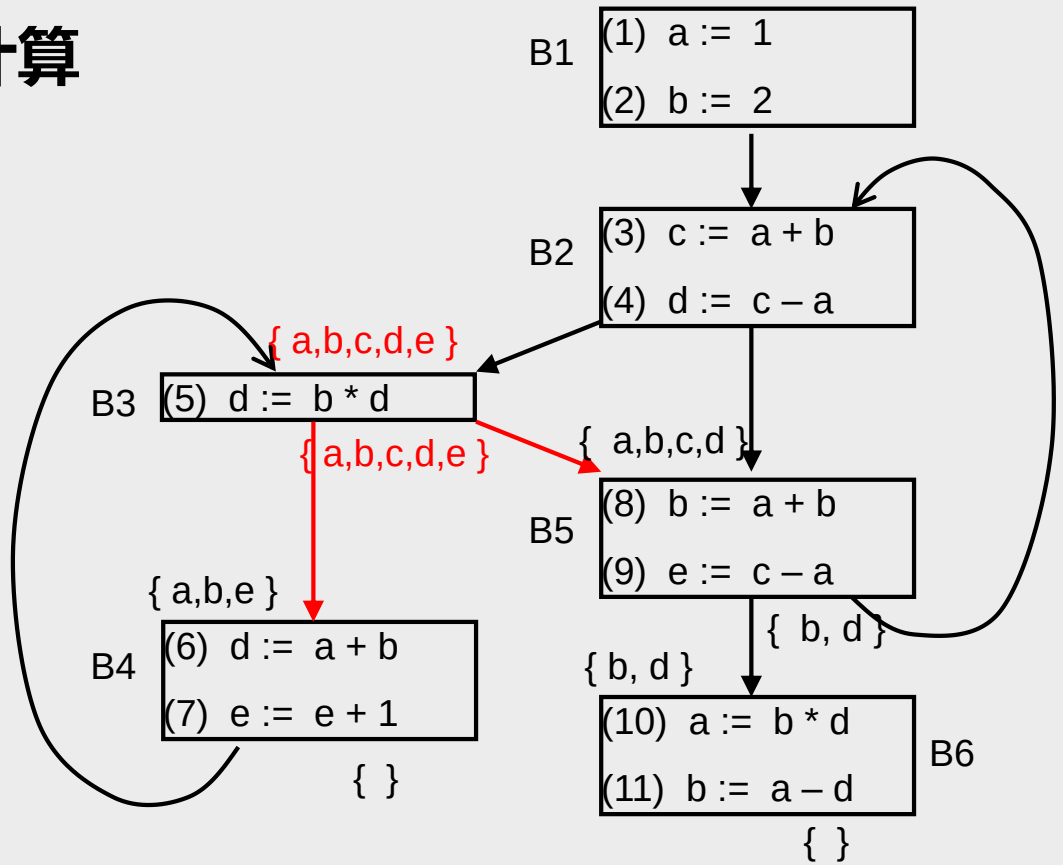
基本块出口活跃变量

- 第一次迭代计算



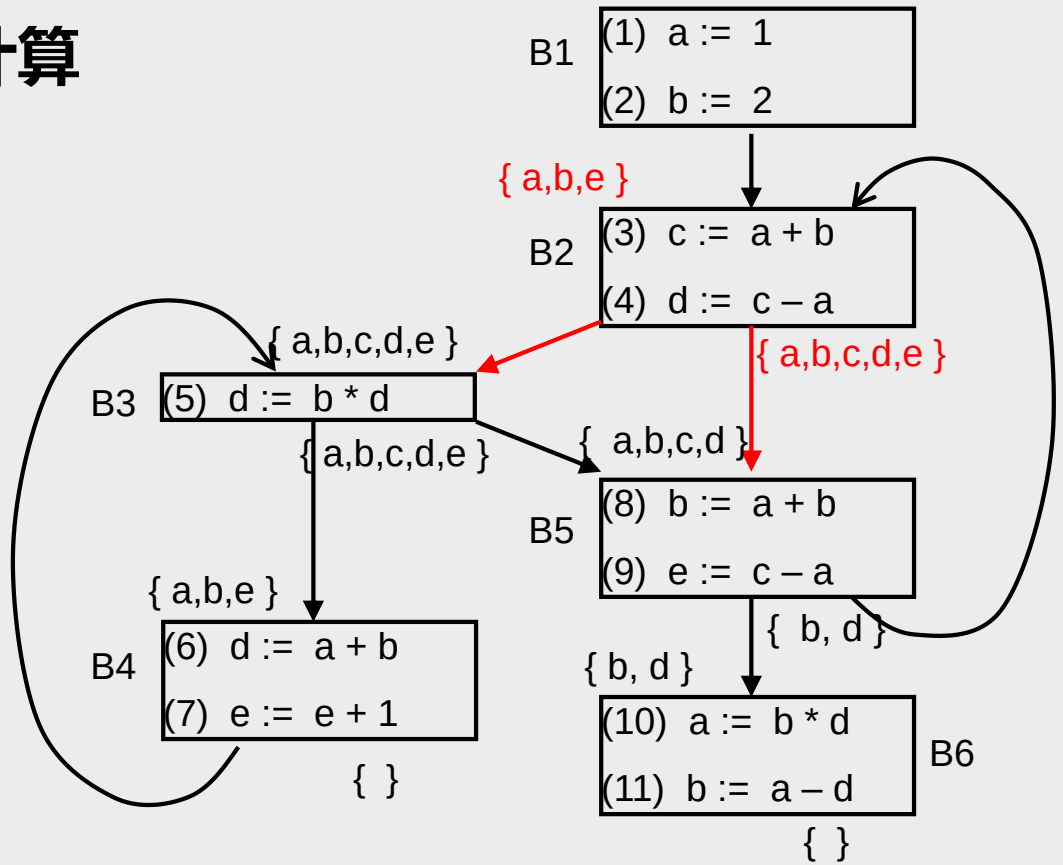
基本块出口活跃变量

- 第一次迭代计算



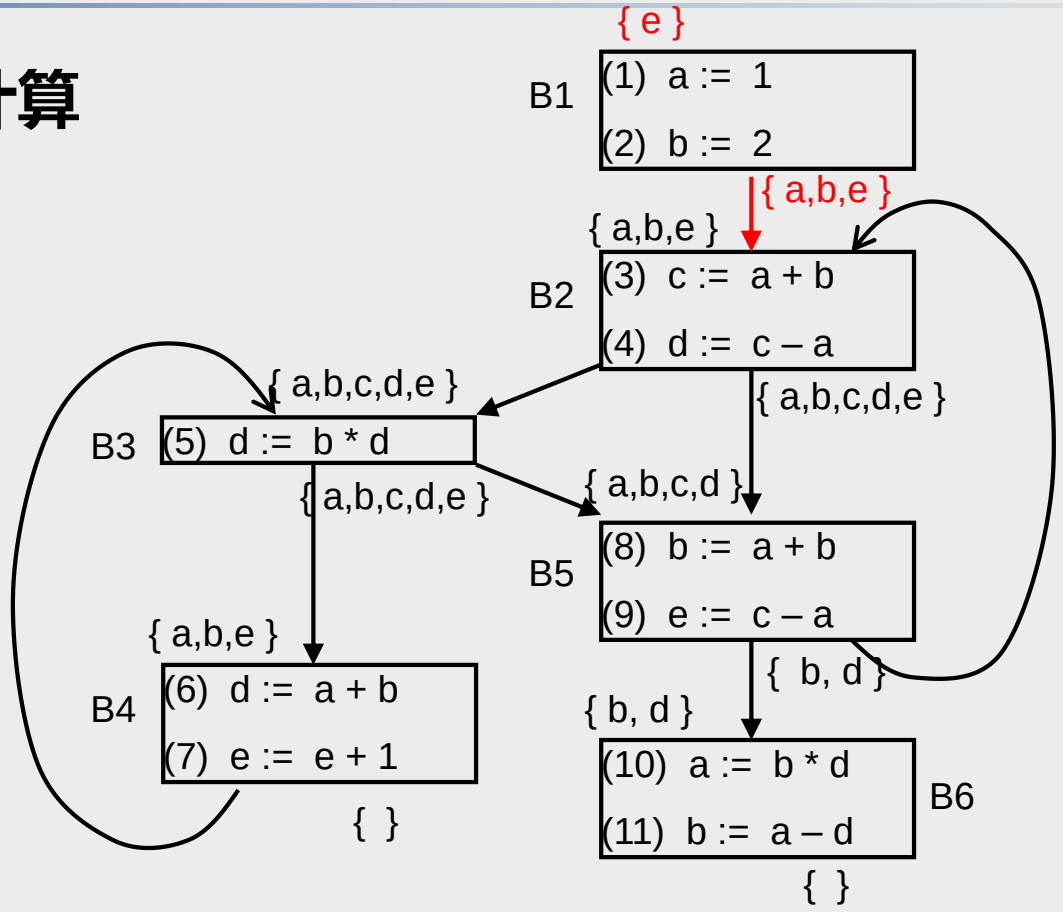
基本块出口活跃变量

- 第一次迭代计算



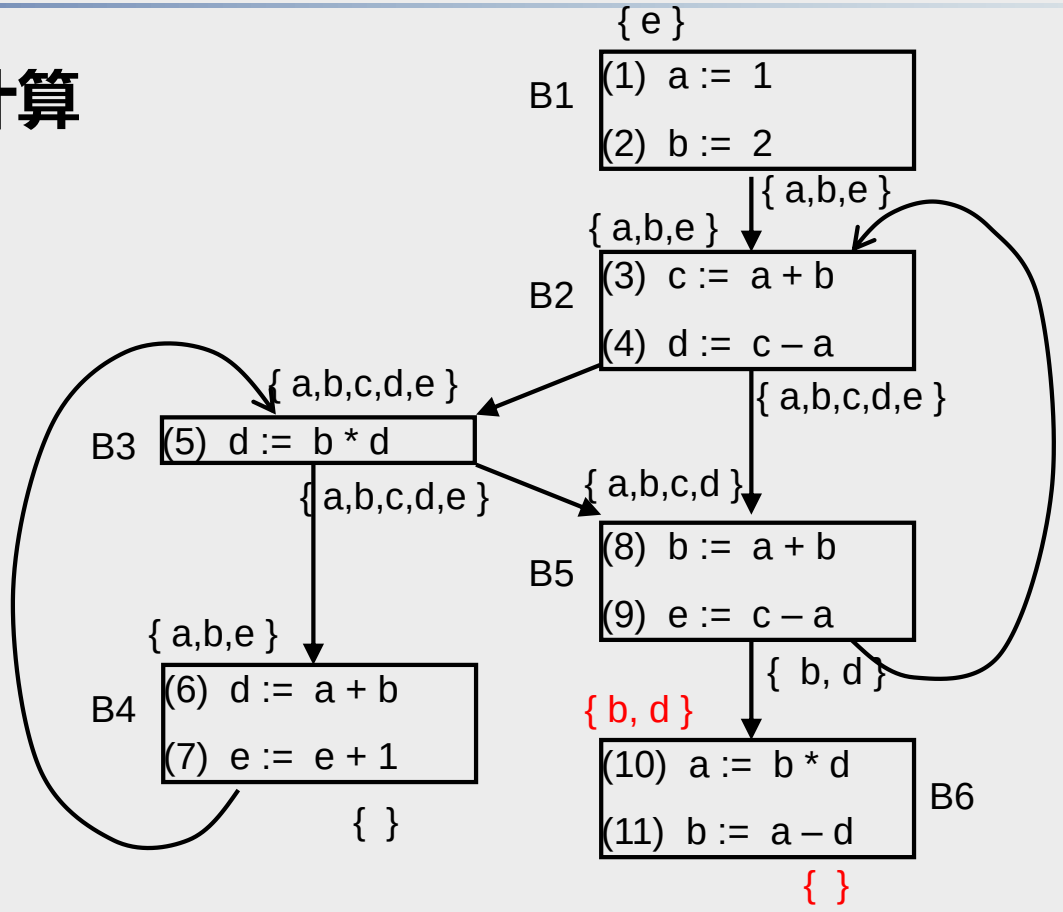
基本块出口活跃变量

- 第一次迭代计算



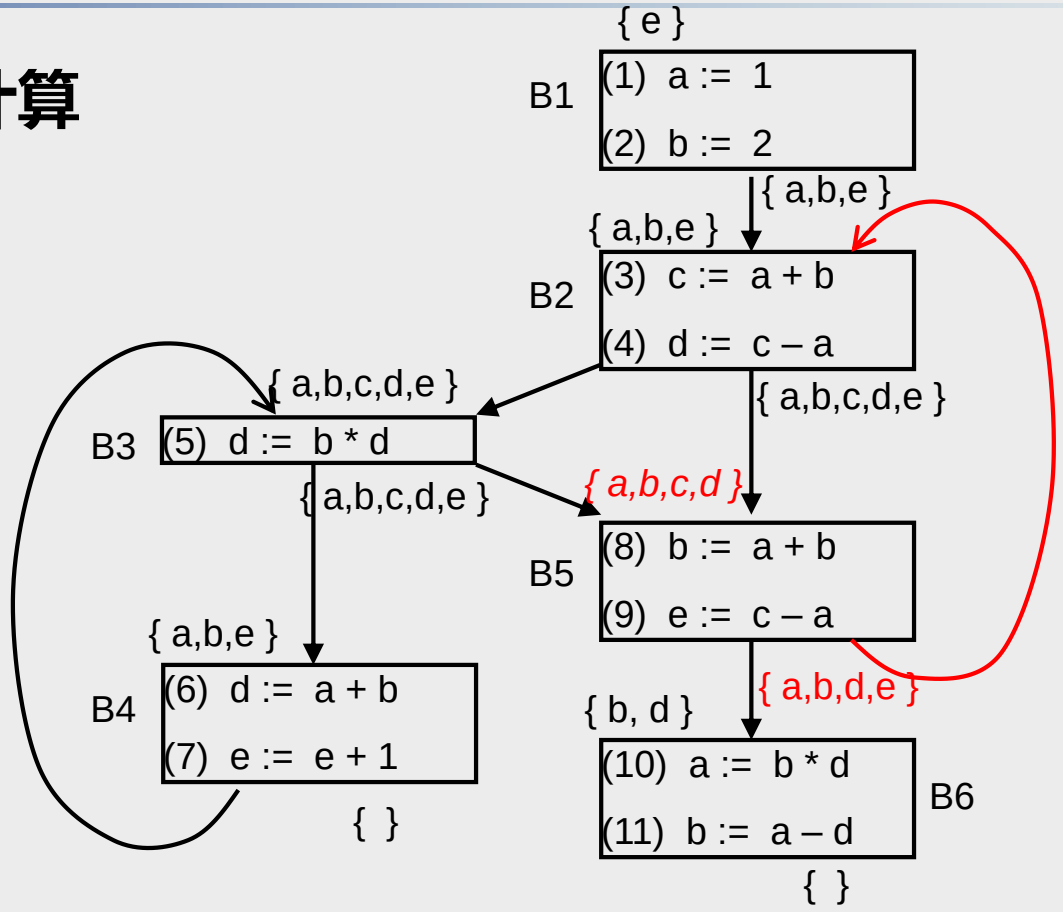
基本块出口活跃变量

- 第二次迭代计算



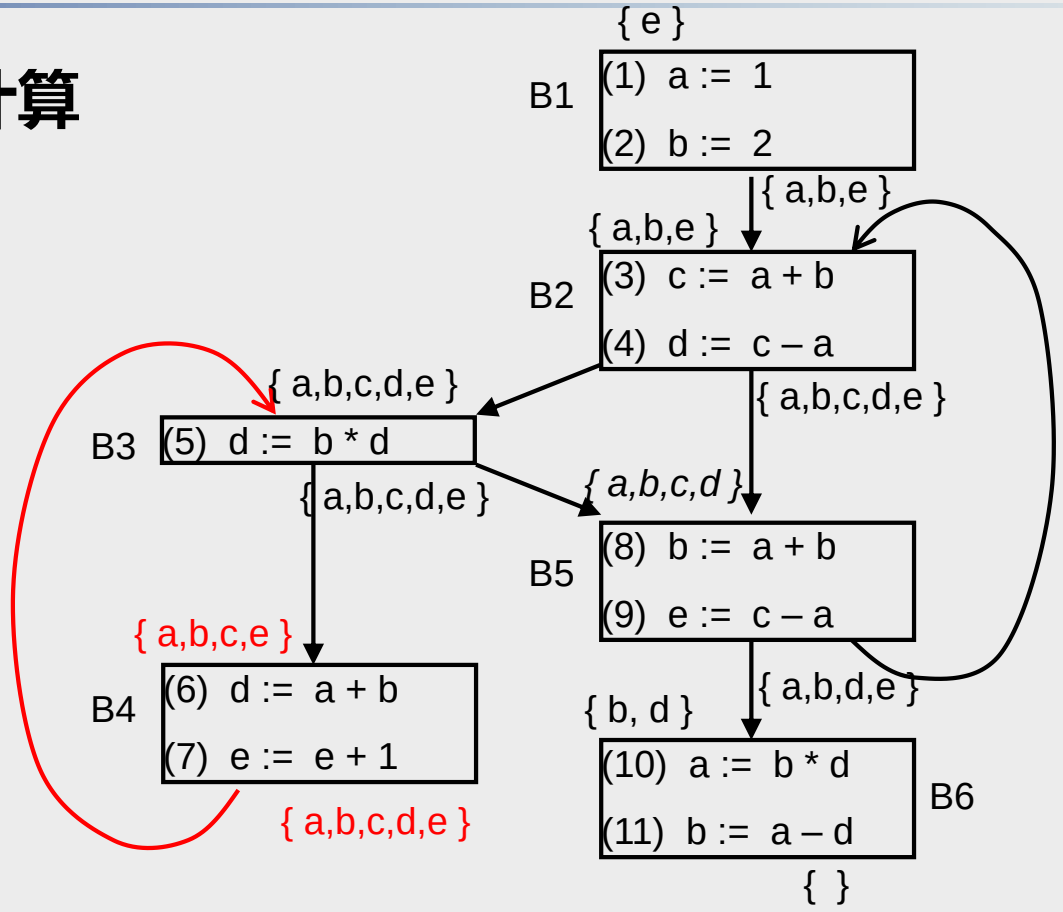
基本块出口活跃变量

- 第二次迭代计算



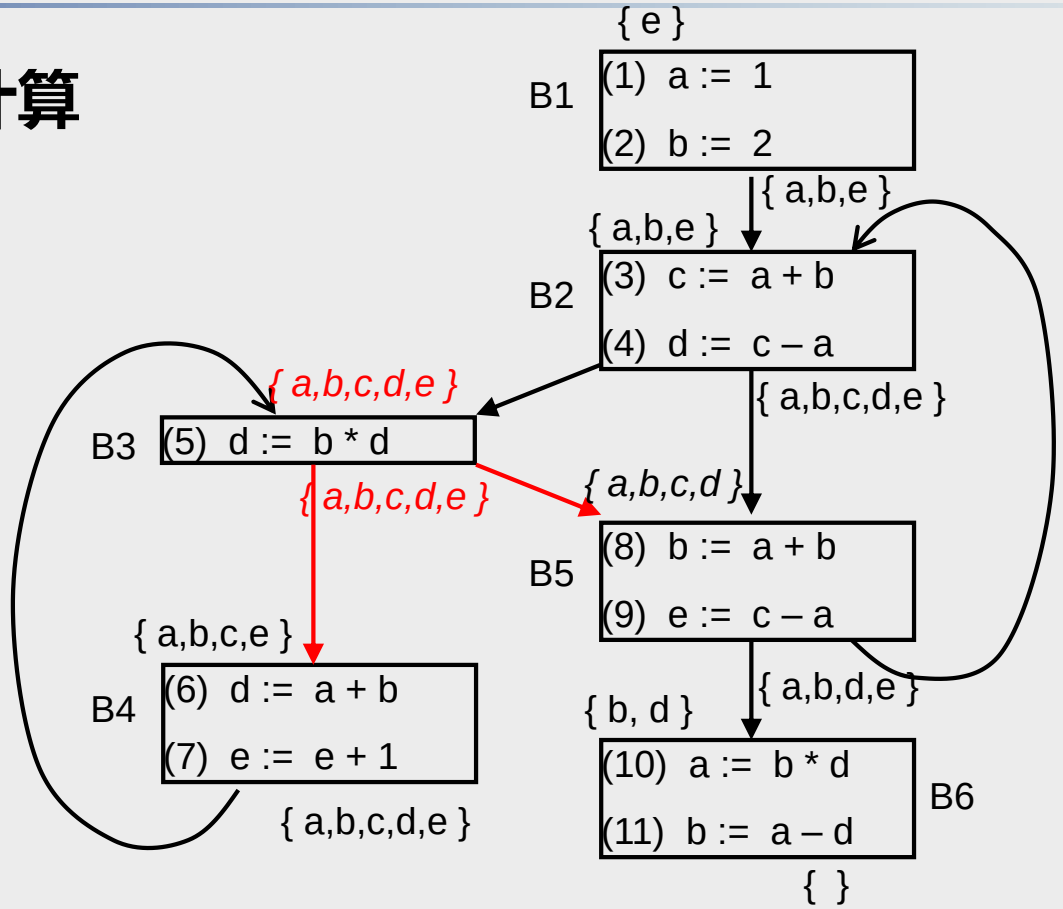
基本块出口活跃变量

- 第二次迭代计算



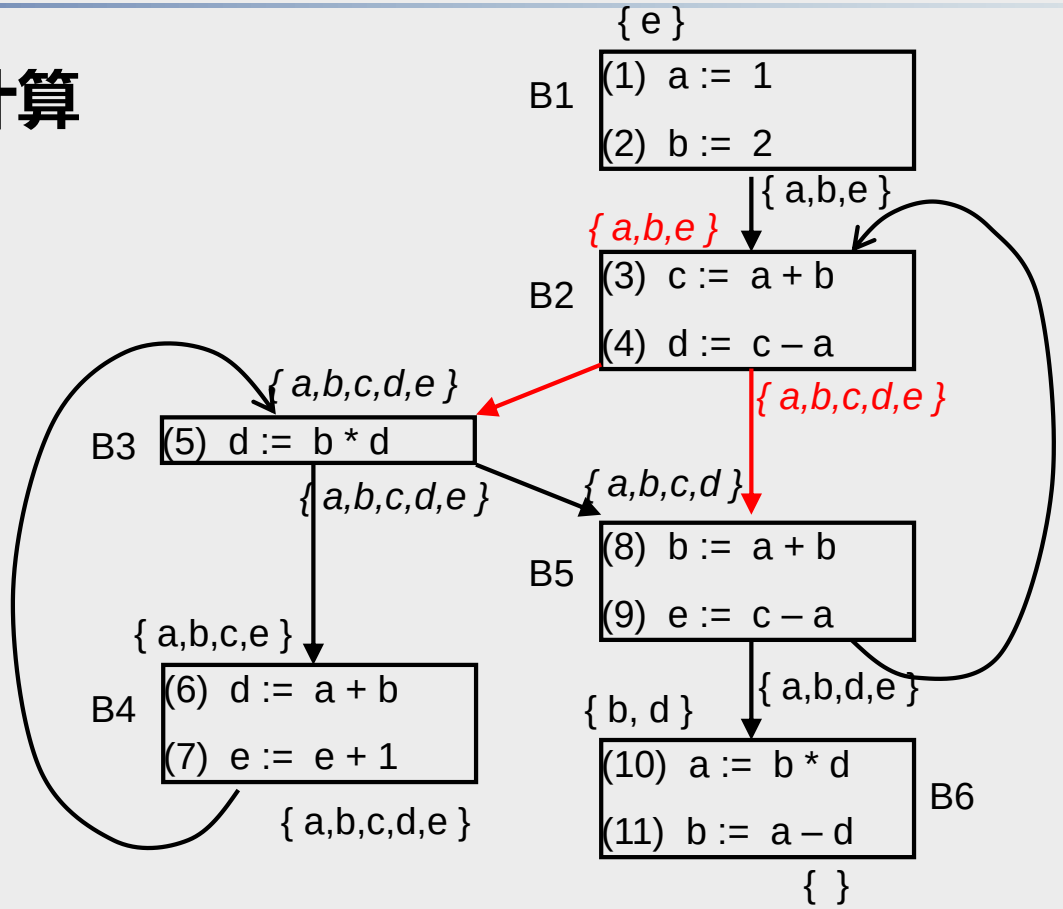
基本块出口活跃变量

- 第二次迭代计算



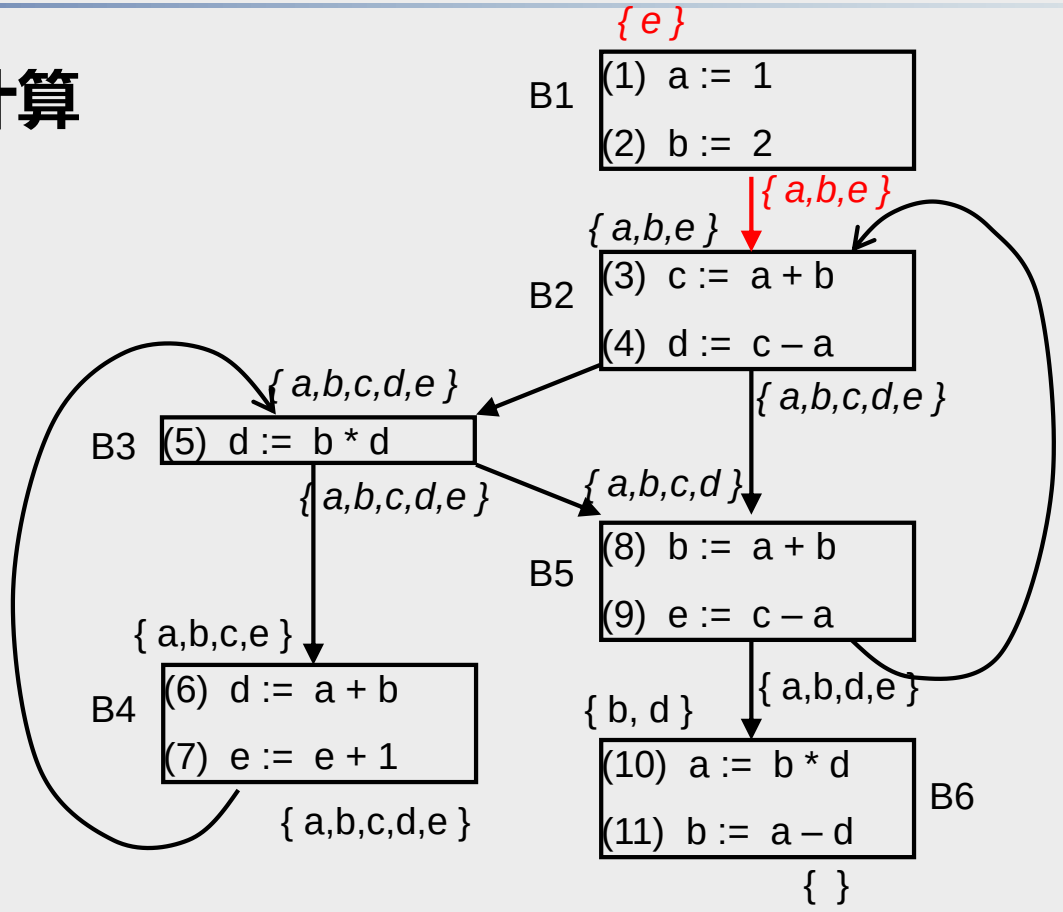
基本块出口活跃变量

- 第二次迭代计算



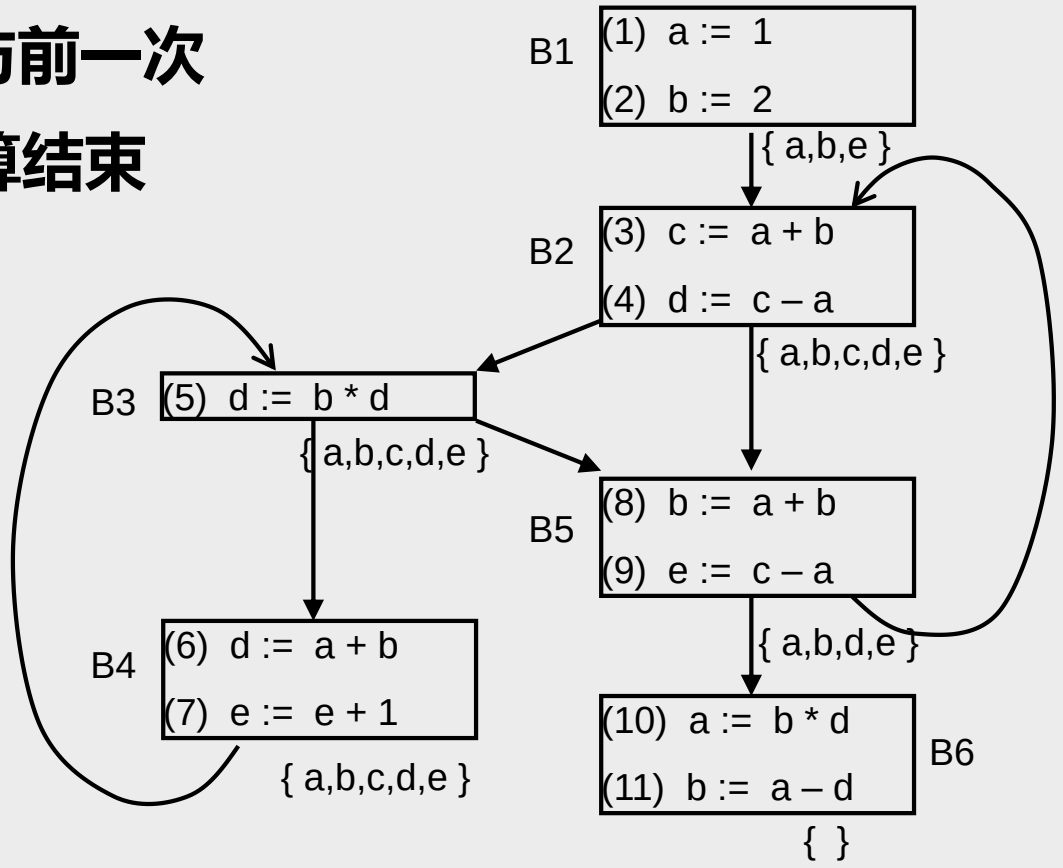
基本块出口活跃变量

- 第二次迭代计算



基本块出口活跃变量

- 第三次迭代与前一次结果一样，计算结束



《编译原理和技术》

机器无关的代码优化 IV

谢谢！

《编译原理和技术》

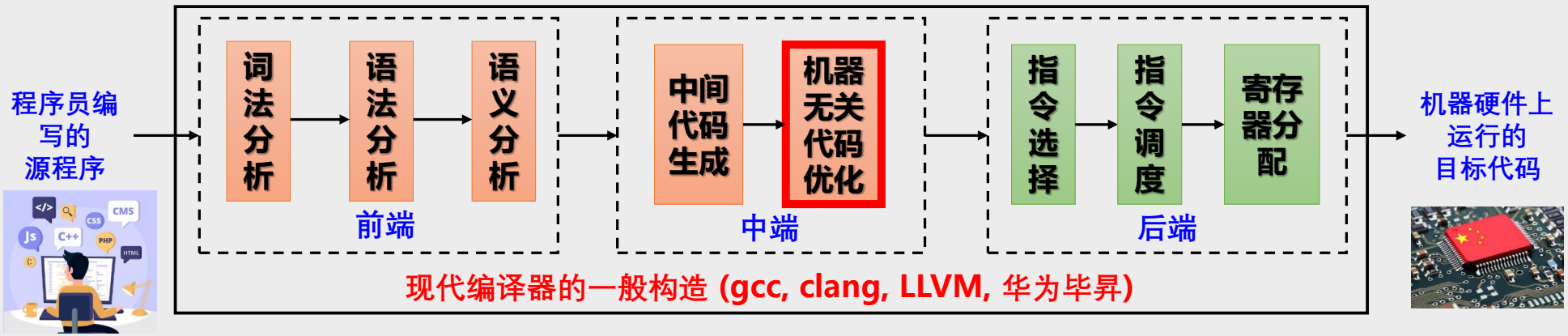
机器无关的代码优化 V

——基本块内的优化

中科大计算机学院
李 诚

2022-11-07

本节提纲



- 基本块的DAG表示
- 删除局部公共子表达式
- 删除死代码
- 数组引用的表示
- 从DAG到基本块的重组

优化的实现方式

- **局部视角-基本块的优化**
  - DAG表示
- **全局视角-跨基本块的优化**
  - 数据流分析

基本块的DAG(有向无环图)表示

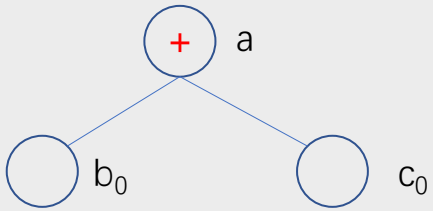
基本块DAG的构造方式

- 每个变量有一个对应的DAG结点表示其初值
- 每条语句s都对应一个内部结点N
  - 结点N的标号是s中的运算符
  - 有一组变量被关联到N，表示s是在此基本块中最晚对这些变量定值的语句
  - N的子结点是基本块中在s之前，最后一个对s所使用的某个运算分量进行定值的语句对应的结点。如果某个运算分量在基本块中在s之前没有被定值，则这个分量对应的子结点就是其初始值对应的结点，用下标0区分
- 某些结点是输出结点，在出口处活跃

基本块的DAG(有向无环图)表示

考虑如下基本块：

a = b + c
b = a – d
c = b + c
d = a - d



- 每个变量有一个对应的DAG结点表示**其初值**
- 每条语句s都对应一个**内部结点N**
  - 结点N的标号是s中的运算符
  - 有一组变量被关联到N，表示s是在此基本块中最晚对这些变量定值的语句
  - N的子结点是基本块中在s之前，最后一个对s所使用的某个运算分量进行定值的语句对应的结点。如果某个运算分量在基本块中在s之前没有被定值，则这个分量对应的子结点就是其初始值对应的结点，用下标0区分

基本块的DAG(有向无环图)表示

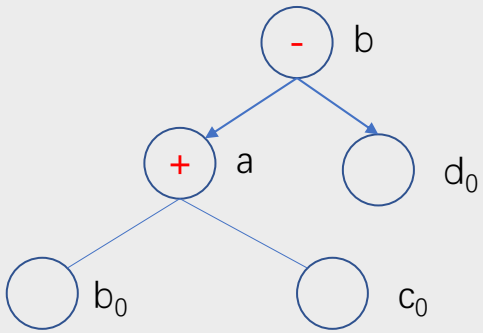
考虑如下基本块：

a = b + c

b = a – d

c = b + c

d = a - d



- **每个变量有一个对应的DAG结点表示其初值**
- **每条语句s都对应一个内部结点N**
  - 结点N的标号是s中的运算符
  - 有一组变量被关联到N，表示s是在此基本块中最晚对这些变量定值的语句
  - N的子结点是基本块中在s之前，最后一个对s所使用的某个运算分量进行定值的语句对应的结点。如果某个运算分量在基本块中在s之前没有被定值，则这个分量对应的子结点就是其初始值对应的结点，用下标0区分

基本块的DAG(有向无环图)表示

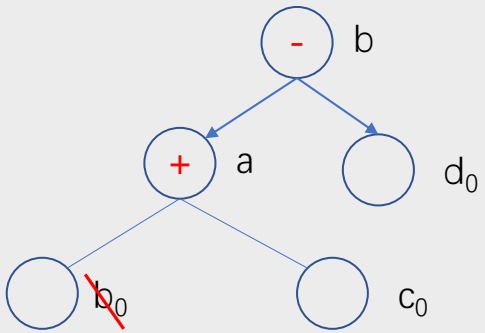
考虑如下基本块：

a = b + c

b = a – d

c = b + c

d = a - d



- **每个变量有一个对应的DAG结点表示其初值**
- **每条语句s都对应一个内部结点N**
  - 结点N的标号是s中的运算符
  - 有一组变量被关
  - N的子结点是基

注1：在为语句x= y+z构造结点N的时候，如果x已经被关联到某结点M上，那么需要从M的关联变量中删除变量x

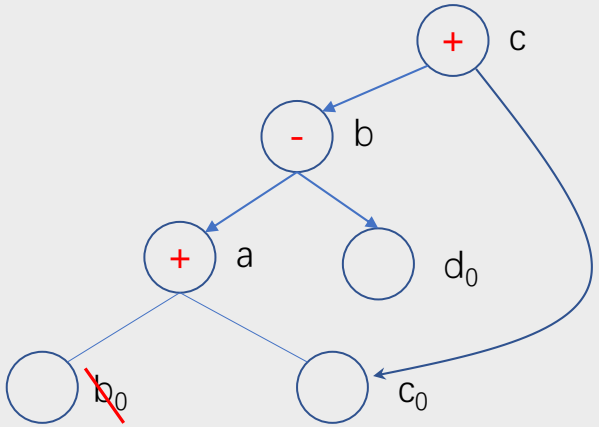
句

定值的语句对应的结点。如果某个运算分量在基本块中在s之前没有被定值，则这个分量对应的子结点就是其初始值对应的结点，用下标0区分

基本块的DAG(有向无环图)表示

考虑如下基本块：

a = b + c
b = a – d
c = b + c
d = a - d

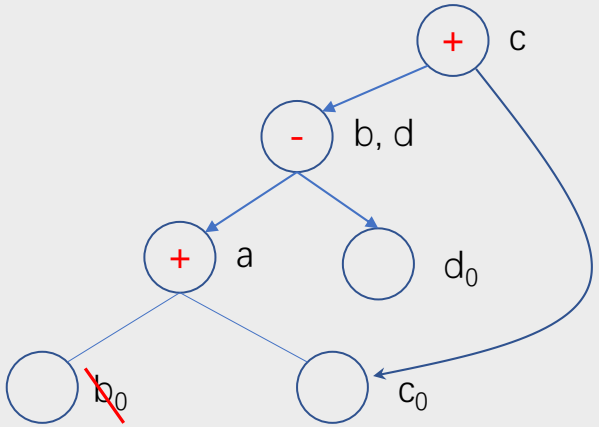


- **每个变量有一个对应的DAG结点表示其初值**
- **每条语句s都对应一个内部结点N**
  - 结点N的标号是s中的运算符
  - 有一组变量被关联到N，表示s是在此基本块中最晚对这些变量定值的语句
  - N的子结点是基本块中在s之前，最后一个对s所使用的某个运算分量进行定值的语句对应的结点。如果某个运算分量在基本块中在s之前没有被定值，则这个分量对应的子结点就是其初始值对应的结点，用下标0区分

基本块的DAG(有向无环图)表示

考虑如下基本块：

a = b + c
b = a – d
c = b + c
d = a - d



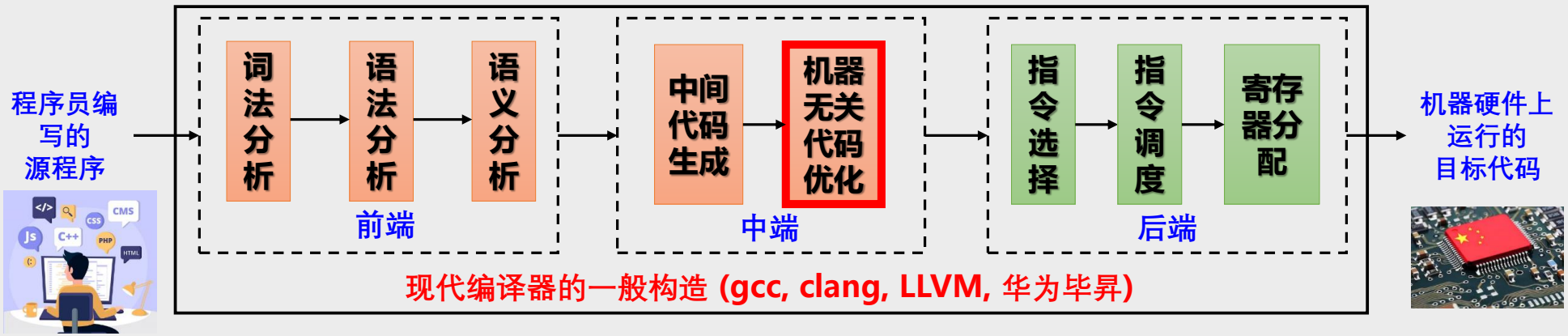
- **每个变量有一个对应的DAG结点表示其初值**
- **每条语句s都对应一个内部结点N**
  - 结点N的标号是s中的运算符
  - 有一组变量被
  - N的子结点是

语句

行定值的语句对应的结点。如果某个运算分量在基本块中在s之前没有被定值，则这个分量对应的子结点就是其初始值对应的结点，用下标0区分

注2：对于x=y+z语句，可对y+z进行值编码，并在该基本快DAG中查询是否已经存在一个结点表示y+z，如果有，就不需要在DAG中加入新的结点，而是在已存在的结点的关联变量中增加x

本节提纲



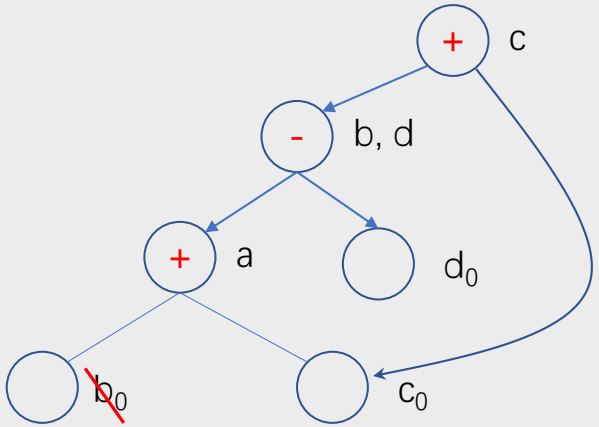
- 基本块的DAG表示
- 删除局部公共子表达式
- 删除死代码
- 数组引用的表示
- 从DAG到基本块的重组

删除局部公共子表达式

考虑如下基本块：

a = b + c
b = a – d
c = b + c
d = a - d

假设b在基本块出口处不活跃



a = b + c
d = a – d
c = d + c

删除局部公共子表达式

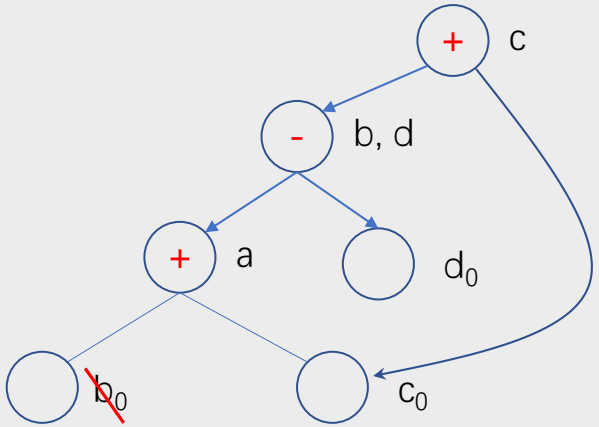
- 当完成基本块优化后，就可以根据优化得到的DAG生成新的等价的三地址代码
- 如果结点有多个关联的活跃变量，就必须引入复制语句，为每个变量赋予正确的值

删除局部公共子表达式

考虑如下基本块：

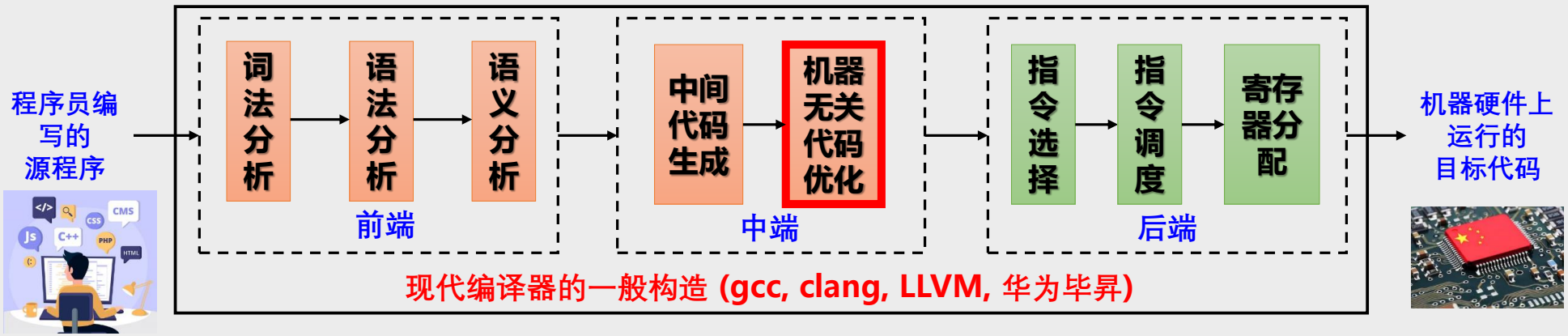
a = b + c
b = a – d
c = b + c
d = a - d

假设b和d在基本块出口处都活跃



a = b + c
d = a – d
b = d
c = d + c

本节提纲



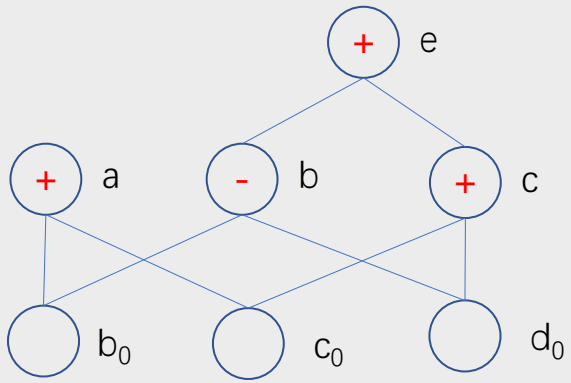
- 基本块的DAG表示
- 删除局部公共子表达式
- 删除死代码
- 数组引用的表示
- 从DAG到基本块的重组

DAG与删除死代码

- 活跃变量是指其值可能会在以后被使用的变量
- 在DAG上删除死代码，可进行如下操作
  - 1.删除所有没有关联活跃变量的根结点
  - 2.重复1操作

考虑如下基本块：

a = b + c
b = b – d
c = c + d
e = b + c



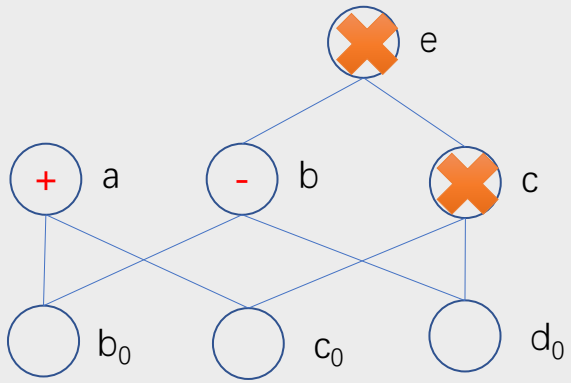
假设a和b是活跃变量，但c和e不是

DAG与删除死代码

- 活跃变量是指其值可能会在以后被使用的变量
- 在DAG上删除死代码，可进行如下操作
  - 1.删除所有没有关联活跃变量的根结点
  - 2.重复1操作

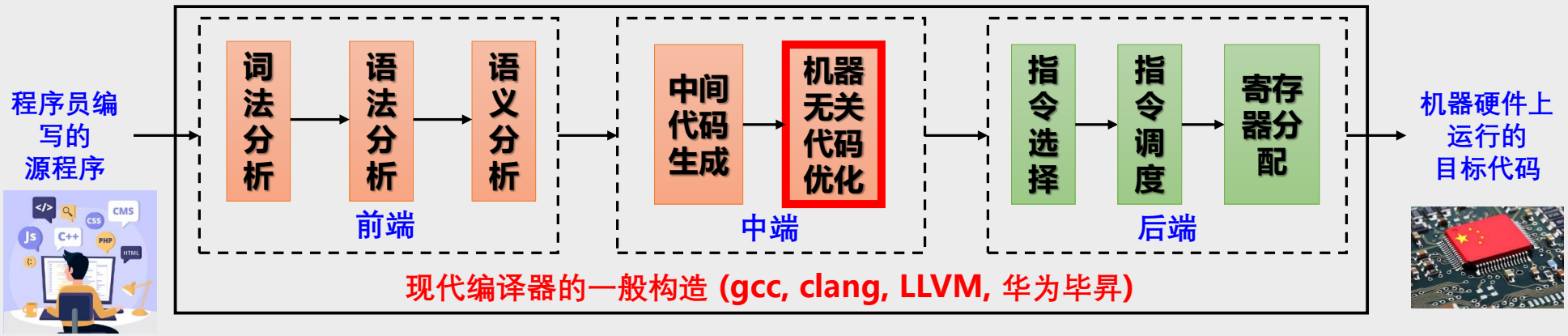
考虑如下基本块：

a = b + c
b = b – d
c = c + d
e = b + c



假设a和b是活跃变量，但c和e不是

本节提纲



- 基本块的DAG表示
- 删除局部公共子表达式
- 删除死代码
- **数组引用的表示**
- **从DAG到基本块的重组**

数组引用的表示

考虑如下基本块：

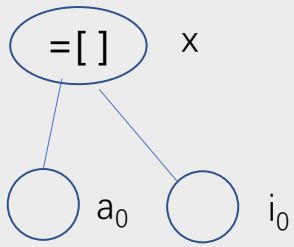
x = a[i]

a[j] = y

z = a[i]

在构造DAG时，如何避免将a[i]误判为公共子表达式

- 对于形如x=a[i]的三地址指令创建一个运算符为=[ ]的结点
- 该结点的子结点为a和i
- 该结点的关联变量是x



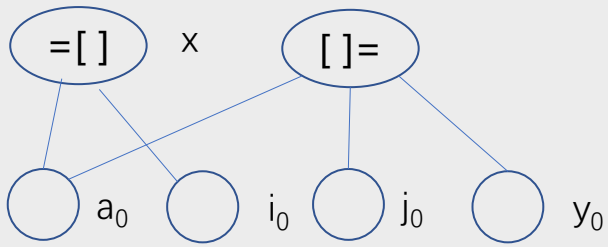
数组引用的表示

考虑如下基本块：

x = a[i]
a[j] = y
z = a[i]

在构造DAG时，如何避免将a[i]误判为公共子表达式

- 对于形如a[j]=y的三地址指令创建一个运算符为[ ]=的结点
- 该结点的子结点为a, j和y
- 该结点没有关联变量



数组引用的表示

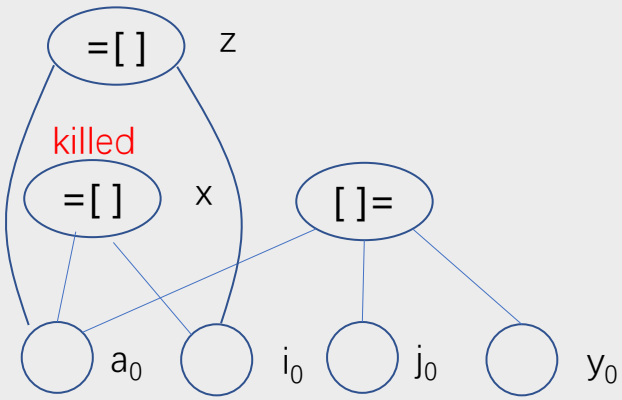
考虑如下基本块：

x = a[i]

a[j] = y

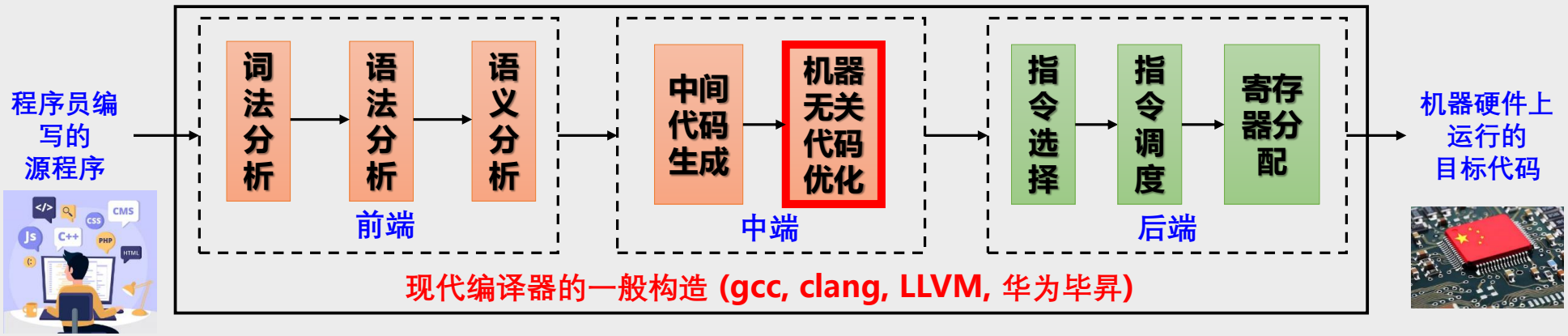
z = a[i]

在构造DAG时，如何避免将a[i]误判为公共子表达式



- 对于形如a[j]=y的三地址指令创建一个运算符为[ ]=的结点
- 该结点的子结点为a, j和y
- 该结点没有关联变量
- 该结点将杀死所有已经建立的、其值依赖于a_0的结点
- 被杀死的结点不能再关联定值变量，也就不能成为公共子表达式

本节提纲



- 基本块的DAG表示
- 删除局部公共子表达式
- 删除死代码
- 数组引用的表示
- **从DAG到基本块的重组**

从DAG到基本块的重组

- **当完成基本块优化后，就可以根据优化得到的DAG生成新的等价的三地址代码**
- **对每个具有若干关联定值变量的结点，构造一个三地址指令来计算其中某个变量的值**
  - 倾向于把计算得到的结果赋给一个在基本块出口处活跃的变量(如果没有全局活跃变量的信息作为依据，就要假设所有变量在基本块出口处活跃，不包含编译器为处理表达式而生成的临时变量)
  - 如果结点有多个附加活跃变量，必须用复制语句

《编译原理和技术》

机器无关的代码优化 V

谢谢！

《编译原理和技术》

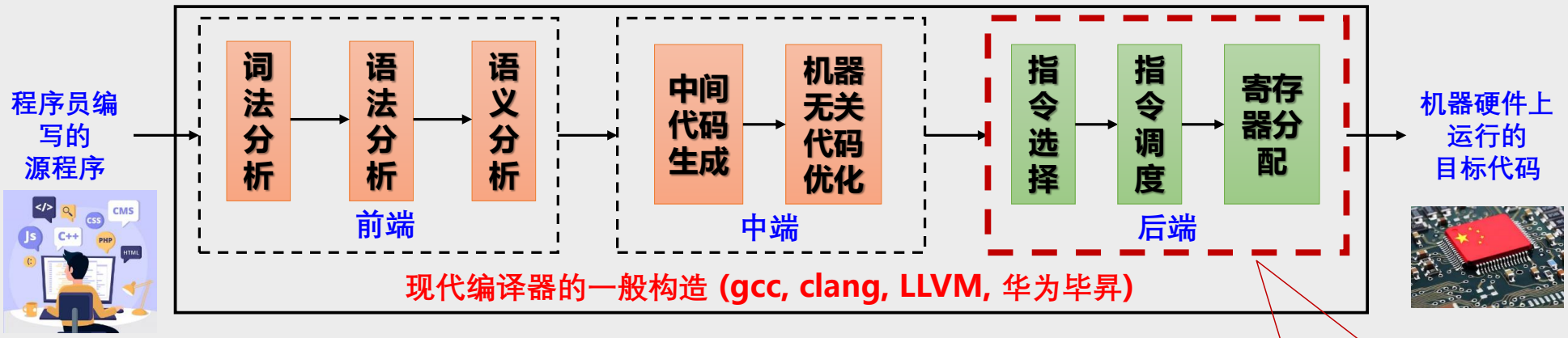
运行时刻环境 I

——概述及存储空间组织管理

中科大计算机学院
李 诚

2022-11-09

本节提纲



- 运行时环境概述
- 存储空间的组织与分配

目标程序需要一个运行环境！

运行时环境

- **编译器为目标程序创建并管理一个运行时环境，目标程序运行在该环境中。**
  - 对象存储位置和空间的分配
  - 访问变量的机制
  - 过程间的连接
  - 参数传递机制

运行时涉及的主要内容

- 运行时存储空间组织管理概述
- 活动树与栈式空间分配
- 调用序列与返回序列
- 非局部数据的访问

由编译器、操作系统、目标机器共同完成
编译器视角：目标程序运行在逻辑地址空间
操作系统视角：将逻辑地址转换为物理地址
目标机器视角：真正执行指令，访问数据，同时限制了存储空间的组织和数据的访问

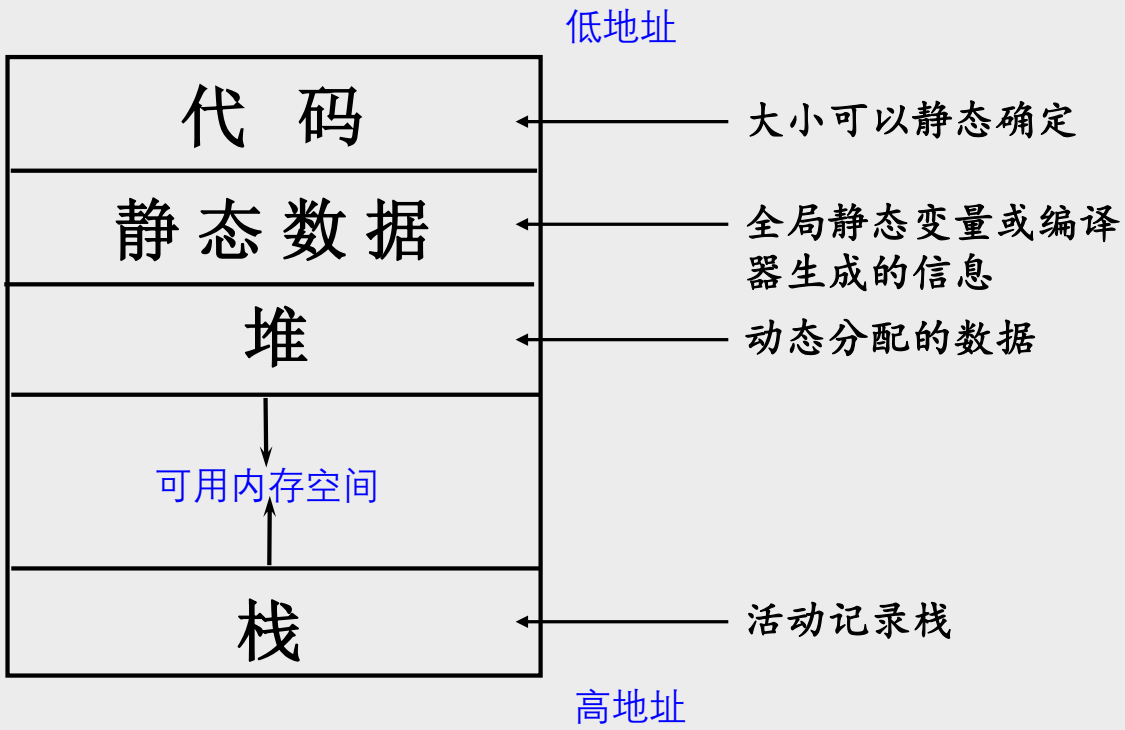
存储分配的策略

- 编译器必须为源程序中出现的一些**数据对象分配运行时的存储空间**
  - 静态存储分配
  - 动态存储分配
- 对于那些在编译时刻就可以确定大小的数据对象，可以在编译时刻就为它们分配存储空间，这样的策略成为**静态存储分配**
  - 比较简单

存储分配的策略

- 如果不能在编译时刻完全确定数据对象的大小，就要采用动态存储分配的策略。即，在编译时刻仅产生各种必要的信息，而在运行时刻，再动态地分配存储空间。
  - 栈式存储分配
  - 堆式存储分配
- 静态和动态这两个概念分别对应编译时刻和运行时刻

程序的存储分配


低地址
代 码
静 态 数 据
堆
可用内存空间
栈
高地址
大小可以静态确定
全局静态变量或编译器生成的信息
动态分配的数据
活动记录栈

影响存储分配策略的语言特征

- 过程能否递归
- 当控制从过程的活动返回时, 局部变量的值是否要保留
- 过程能否访问非局部变量
- 过程调用的参数传递方式
- 过程能否作为参数被传递
- 过程能否作为结果值传递
- 存储块能否在程序控制下被动态地分配
- 存储块是否必须被显式地释放

过程的存储组织与分配

- **过程**
  - FORTRAN的子例程(subroutine)
  - PASCAL的过程/函数(procedure/function)
  - C的函数
- **过程的激活（调用）与终止（返回）**
- **过程的执行需要：**
  - 代码段＋活动记录（过程运行所需的额外信息，如参数，局部数据，返回地址等）

局部存储分配

- 基本概念：作用域与生存期
- 活动记录的常见布局
  - 字节寻址、类型、次序、对齐
- 程序块：同名情况的处理

局部存储分配

- 名字的作用域
  - 一个声明起作用的程序部分称为该声明的作用域
  - 即使一个名字在程序中只声明一次，该名字在程序运行时也可能表示不同的数据对象

如下图代码中的n

```
int f(int n){
if (n<0) error("arg<0");
else if (n==0) return 1;
else return n*f(n-1);
}
```

局部存储分配

环境和状态

- 环境把名字映射到左值，而状态把左值映射到右值（即名字到值有两步映射）
- 赋值改变状态，但不改变环境
- 过程调用改变环境
- 如果环境将名字x映射到存储单元s，则说x被绑定到s

局部存储分配

- 静态概念和动态概念的对应

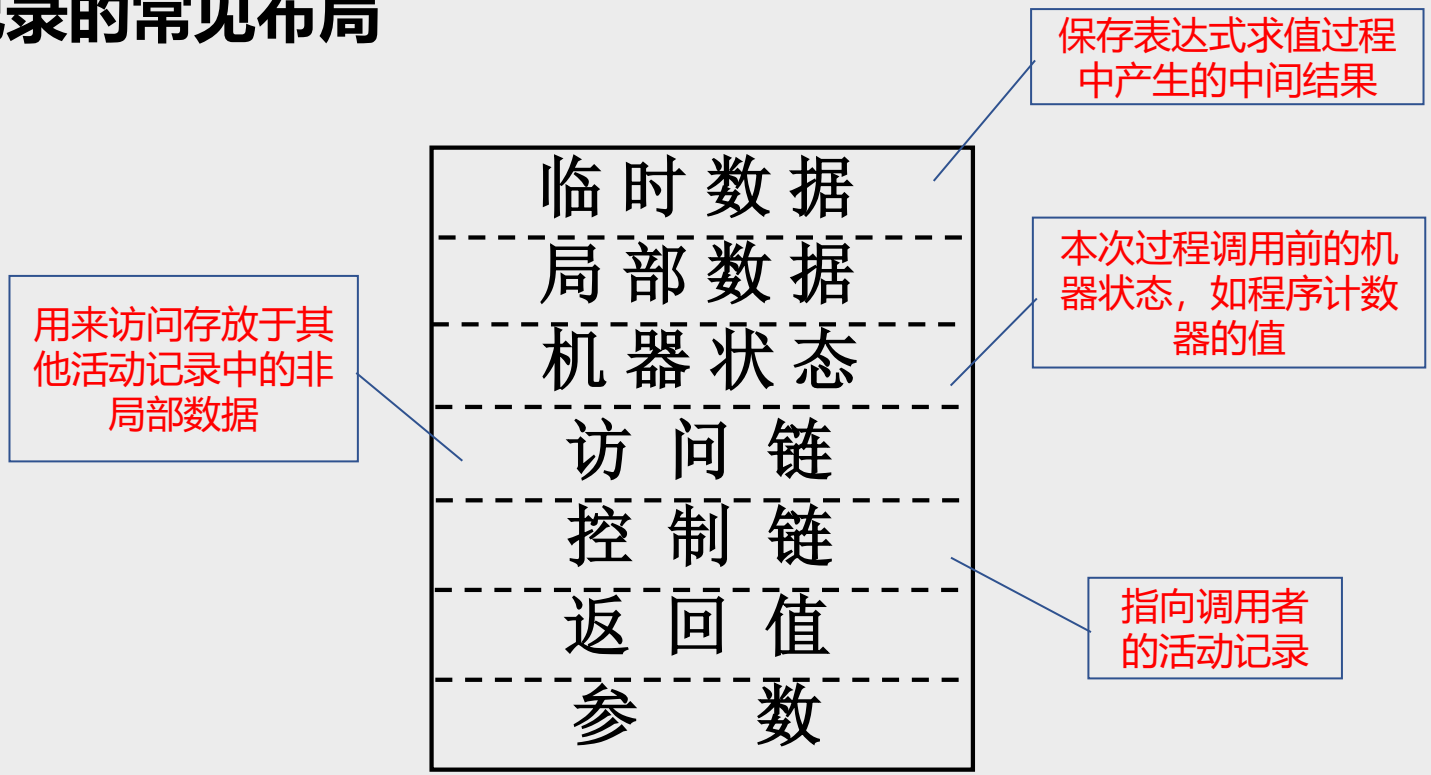
静 态 概 念	动 态 对 应
过程的定义	过程的活动
名字的声明	名字的绑定
声明的作用域	绑定的生存期

活动记录

- 使用过程(或函数、方法)作为用户自定义动作的单元的语言，其编译器通常以过程为单位分配存储空间
- 过程体的每次执行成为该过程的一个活动
- 编译器为每一个活动分配一块连续存储区域，用来管理此次执行所需的信息，这片区域称为活动记录(activation record)

活动记录

活动记录的常见布局



局部存储分配

局部数据的布局

- 字节是可编址内存的最小单位
- 变量所需的存储空间可以根据其类型而静态确定
- 一个过程所声明的局部变量，按这些变量声明时出现的次序，在局部数据域中依次分配空间
- 局部数据的地址可以用相对于活动记录中某个位置的地址来表示
- 数据对象的存储布局还有一个对齐问题

局部存储分配

- 例 在SPARC/Solaris工作站上下面两个结构体的size分别是24和16，为什么不一样?

```
typedef struct  _a{
        char   c1;
        long   i;
        char   c2;
        double f;
}a;
```

```
typedef struct  _b{
        char c1;
        char c2;
        long i;
        double f;
}b;
```

对齐：char : 1, long : 4, double : 8

局部存储分配

- 例 在SPARC/Solaris工作站上下面两个结构体的size分别是24和16，为什么不一样?

```
typedef struct  _a{
        char   c1;     0
        long   i;      4
        char   c2;     8
        double f;      16
}a;
```

```
typedef struct  _b{
        char c1;      0
        char c2;      1
        long i;       4
        double f;     8
}b;
```

对齐：char : 1, long : 4, double : 8

局部存储分配

- 例 在x86/Linux机器的结果和SPARC/Solaris工作站不一样，是20和16。

```
typedef struct  _a{
        char   c1;     0
        long   i;      4
        char   c2;     8
        double f;      12
}a;
```

```
typedef struct  _b{
        char c1;     0
        char c2;     1
        long i;      4
        double f;    8
}b;
```

对齐：char : 1, long : 4, double : 4

局部存储分配

- **程序块**
  - 本身含有局部变量声明的语句
  - 可以嵌套
  - 最接近的嵌套作用域规则
  - 并列程序块不会同时活跃
  - 并列程序块的变量可以重叠分配

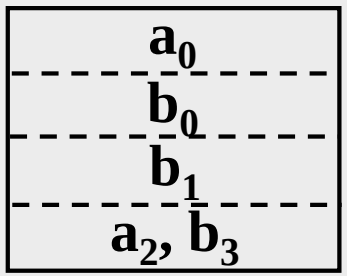
局部存储分配

```
main() / * 例 */
{                                                    /* begin of B0 */
    int a = 0;
    int b = 0;
    {                                                /* begin of B1 */
        int b = 1;
        {                                            /* begin of B2 */
            int a = 2;
        }                                            /* end of B2 */
        {                                            /* begin of B3 */
            int b = 3;
        }                                            /* end of B3 */
    }                                                /* end of B1 */
}                                                    /* end of B0 */
```

局部存储分配

```
main() / * 例 */
{ /* begin of B0 */
      int a = 0;
      int b = 0;
      { /* begin of B1 */
            int b = 1;
            {/* begin of B2 */
                int a = 2;
            }/* end of B2 */
            {/* begin of B3 */
                int b = 3;
            }/* end of B3 */
      }/* end of B1 */
}/* end of B0 */
```

声　明	作 用 域
int a = 0;	$B_0 - B_2$
int b = 0;	$B_0 - B_1$
int b = 1;	$B_1 - B_3$
int a = 2;	B_2
int b = 3;	B_3



重叠分配存储单元

静态分配

- 名字在程序被编译时绑定到存储单元，不需要运行时的任何支持
- 静态分配给语言带来限制
  - 递归过程不被允许
  - 数据对象的长度和它在内存中位置的限制，必须是在编译时可以知道的
  - 数据结构不能动态建立

静态分配

- 例 C程序的外部变量、静态局部变量以及程序中出现的常量都可以静态分配
- 声明在函数外面
  - 外部变量 —— 静态分配
  - 静态外部变量 —— 静态分配
- 声明在函数里面
  - 静态局部变量 —— 也是静态分配
  - 自动变量 —— 不能静态分配

动态分配

- **程序对数据区的需求在编译时是完全未知的，这是因为每个数据对象所需数据区的大小和数目在编译时是未知的。**
- **主要有两种策略**
  - 栈式存储：与过程调用返回有关，涉及过程的局部变量以及过程活动记录
  - 堆存储：关系到部分生存周期较长的数据

《编译原理和技术》

运行时刻环境 I

——概述及存储空间组织管理

谢谢！

《编译原理和技术》

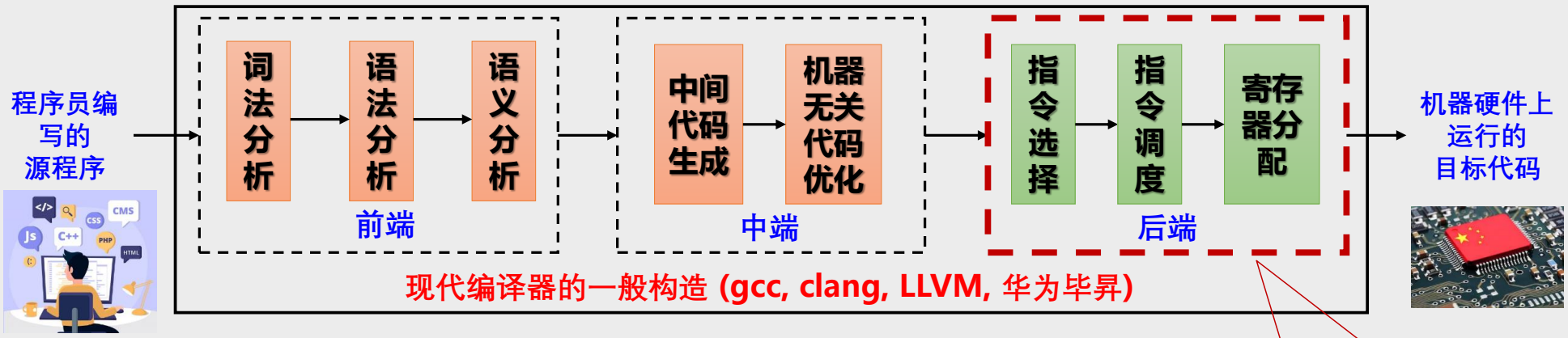
运行时刻环境 II

——空间的栈式分配

中科大计算机学院
李 诚

2022-11-09

本节提纲



- 活动树与运行栈
- 调用序列与返回序列

目标程序需要一个运行环境！

栈式存储分配

- 对于支持过程、函数和方法的语言，其编译器通常会用栈的形式来管理其运行时刻存储
- 当一个过程被调用时，该过程的活动记录被压入栈中；当过程结束时，记录被弹出
- 这种安排不仅允许活跃时段不交叠的多个过程调用共享空间；而且可以使得过程的非局部变量的相对地址总是固定的，和调用序列无关

活动树

- 用来描述程序运行期间控制进入和离开各个活动的情况的树
- 树中的每个结点对应于一个活动。根结点是启动程序执行的main过程的活动
- 对于过程p，其子结点对应于被p的这次活动调用的各个过程的活动。按照调用次序，自左向右地显示它们。一个子结点必须在其右兄弟结点的活动开始之前结束。

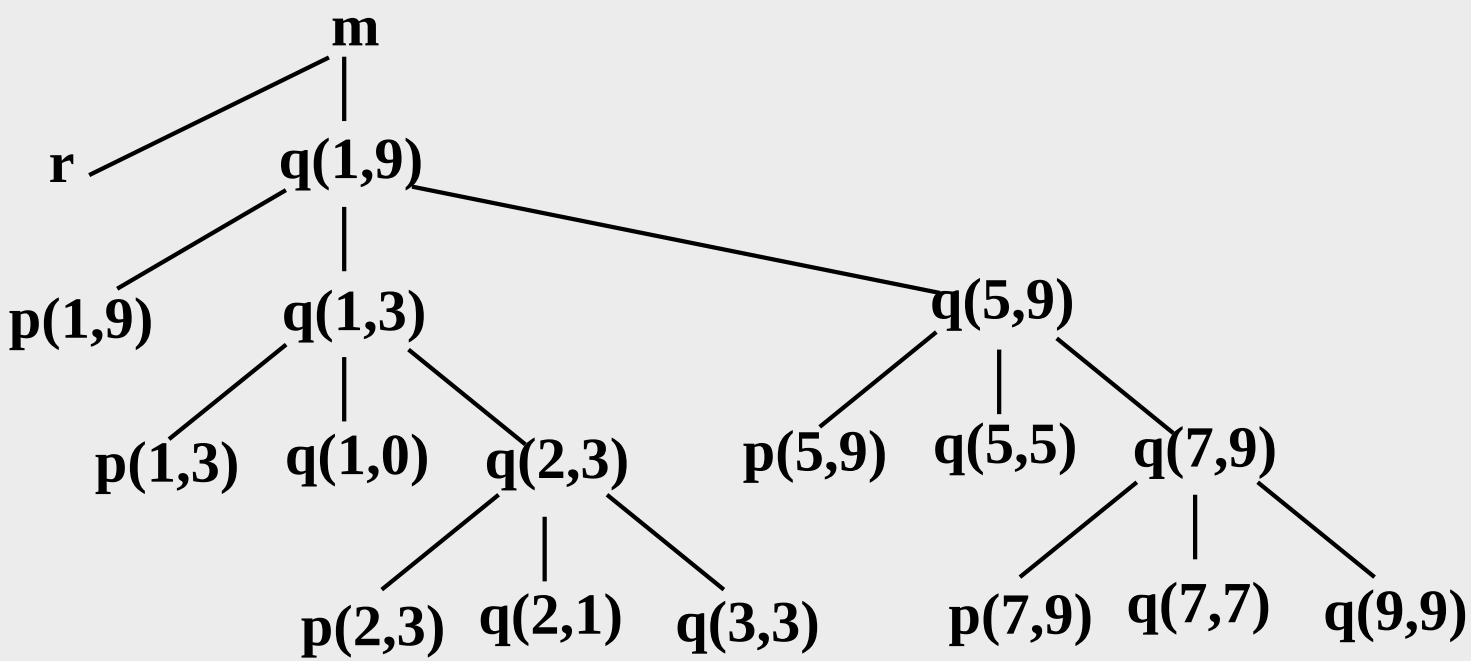
例：一个快排程序的介绍

```
int a[11];
void readArray() /*将9个数读入a[1],…,a[9]中*/
{ int i; …}
int partition(int m, int n)
{/*选择一个分割值v, 划分a[m,…,n], 使得a[m… p-1]小于v, a[p]=v, a[p+1…n]大于v, 返回p*/
…}
void quicksort(int m, int n)
{  int i;
    if(n>m){
         i = partition(m, n);
         quicksort(m, i-1);
         quicksort(i+1, n);}
main(){
    readArray();
   a[0]  = -9999;
   a[10] = 9999;
   quicksort(1,9);}
```

活动树和运行栈

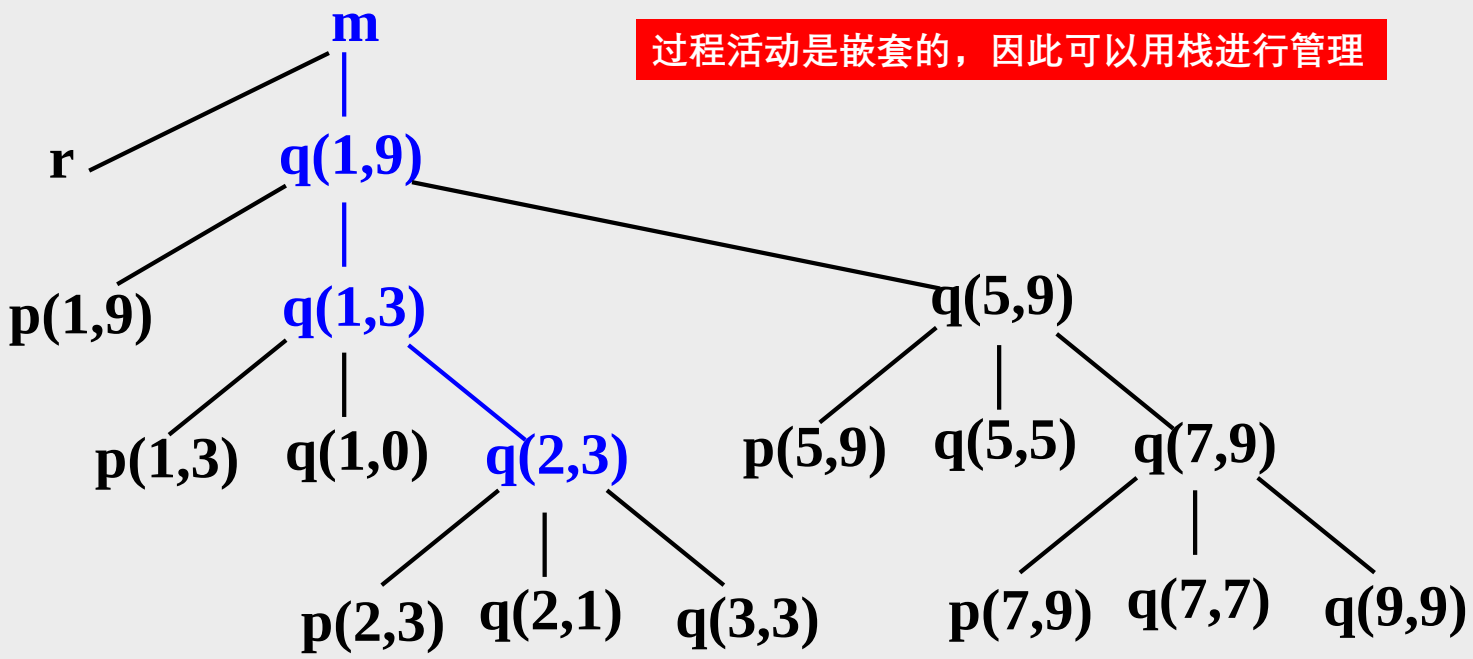
活动树

- 用树来描绘控制进入和离开活动的方式



活动树和运行栈

- 当前活跃着的过程活动可以保存在一个栈中
  - 例 控制栈的内容：m, q (1, 9), q (1, 3), q (2, 3)



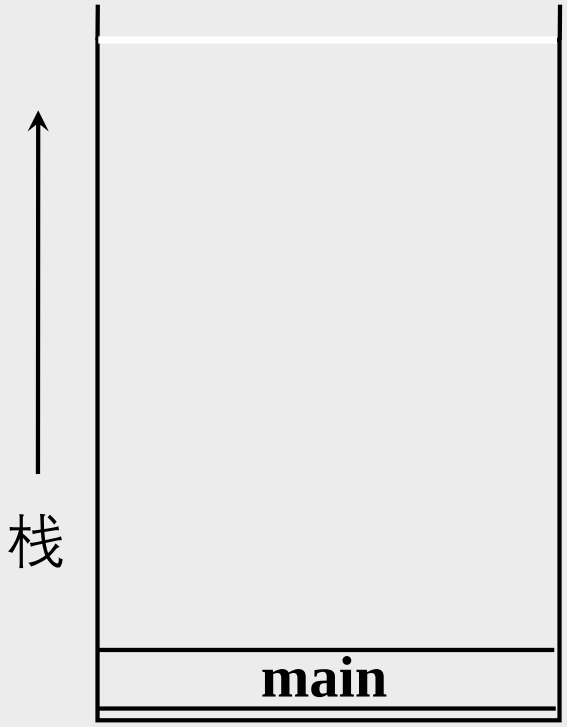
过程活动是嵌套的，因此可以用栈进行管理

活动树和运行栈

- 运行栈：把控制栈中的信息拓广到包括过程活动所需的所有局部信息（即活动记录）

活动树和运行栈

- 运行栈：把控制栈中的信息拓广到包括过程活动所需的所有局部信息（即活动记录）

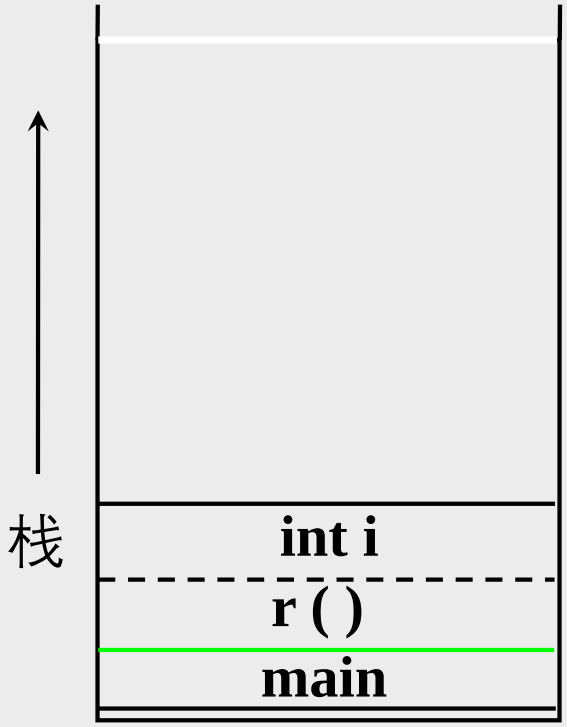


函数调用关系树

main

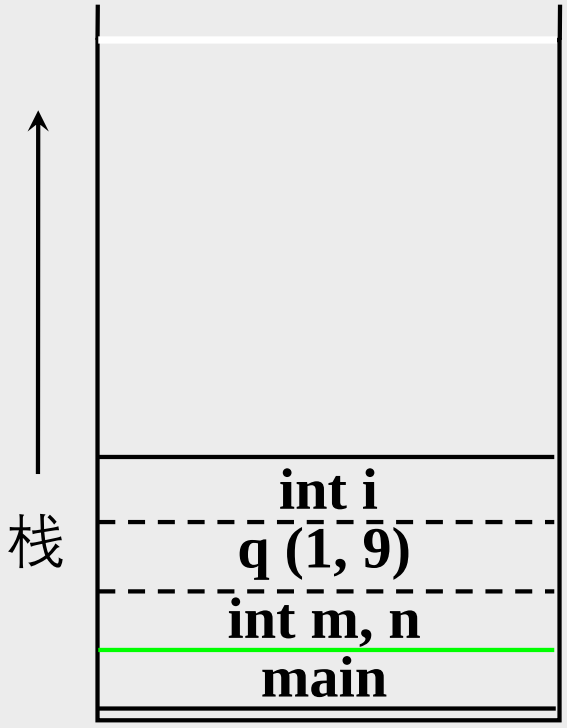
活动树和运行栈

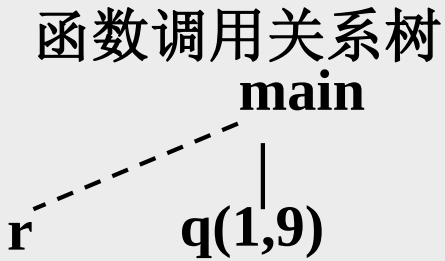
- 运行栈：把控制栈中的信息拓广到包括过程活动所需的所有局部信息（即活动记录）



活动树和运行栈

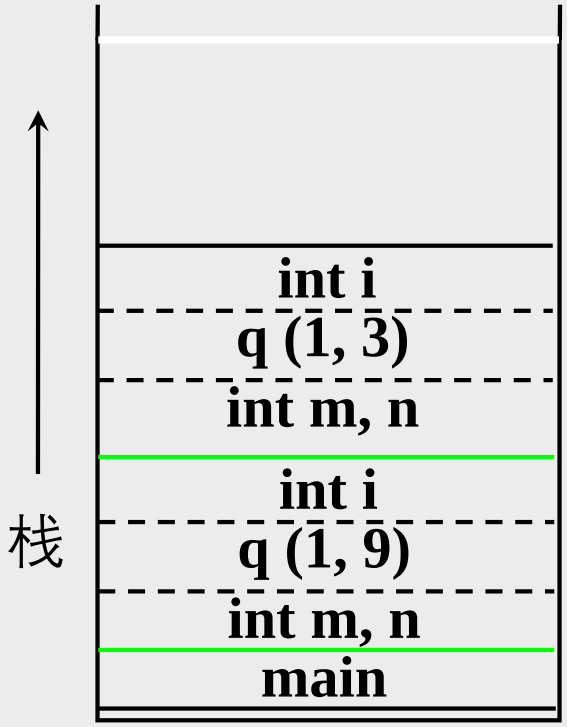
- 运行栈：把控制栈中的信息拓广到包括过程活动所需的所有局部信息（即活动记录）

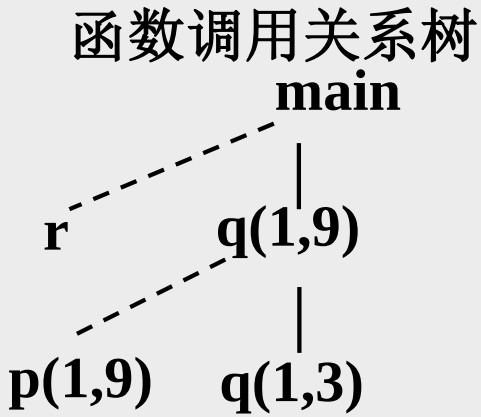




活动树和运行栈

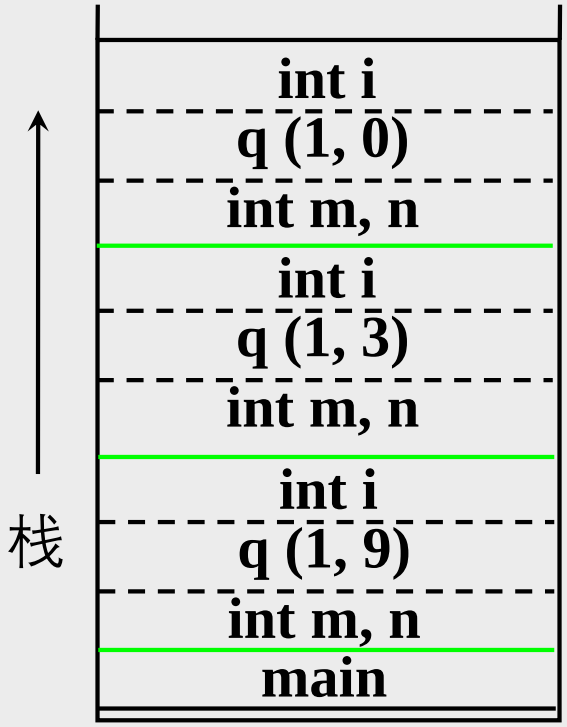
- 运行栈：把控制栈中的信息拓广到包括过程活动所需的所有局部信息（即活动记录）

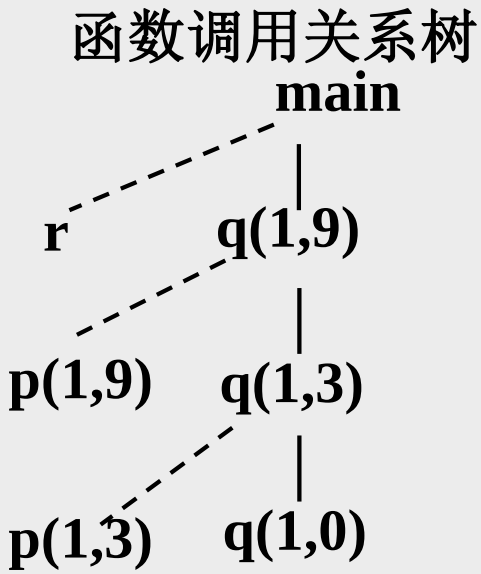




活动树和运行栈

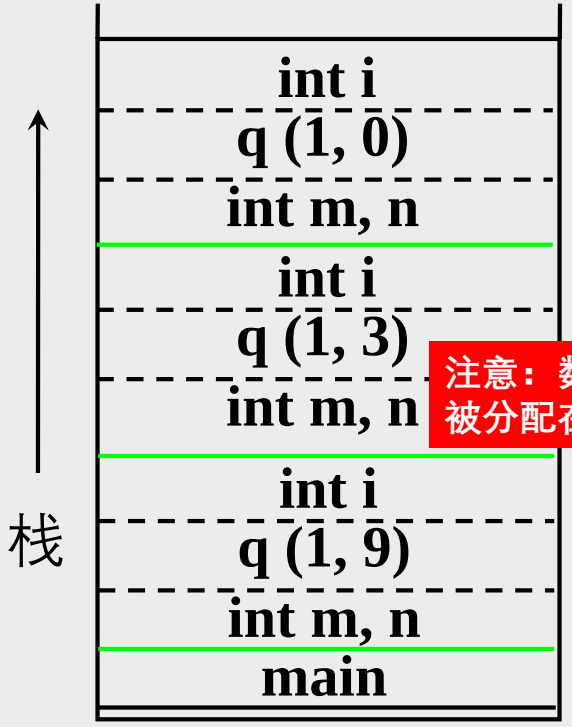
- 运行栈：把控制栈中的信息拓广到包括过程活动所需的所有局部信息（即活动记录）

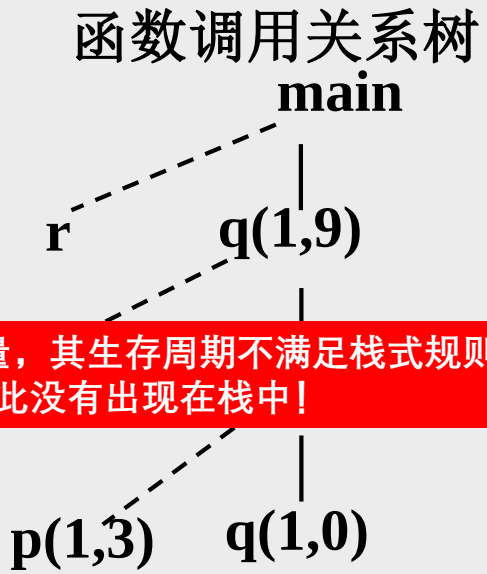




活动树和运行栈

- 运行栈：把控制栈中的信息拓广到包括过程活动所需的所有局部信息（即活动记录）



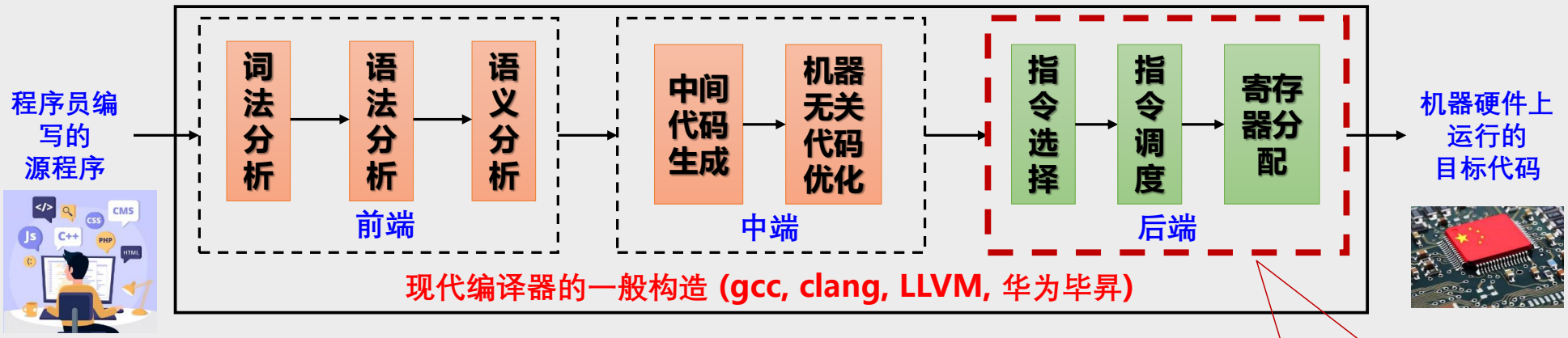


注意：数组a是全局变量，其生存周期不满足栈式规则，被分配在静态区域，因此没有出现在栈中！

活动树与控制栈

- 每个活跃的活动**都有**一个位于控制栈中的活动记录
- 活动树的根结点的记录位于**栈底**
- 程序控制所在的活动的记录位于**栈顶**
- 栈中全部活动记录的序列对应在活动树中到达当前控制所在的活动结点的**路径**

本节提纲



- 活动树与运行栈
- 调用序列与返回序列

目标程序需要一个运行环境！

对运行栈的管理

- **代码序列**
  - 过程调用和过程返回都需要执行一些代码来管理活动记录栈，保存或恢复机器状态等
- **过程调用序列(calling sequence)**
  - 过程调用时执行的分配活动记录，把信息填入它的域中，使被调用过程可以开始执行的代码
- **过程返回序列(return sequence)**
  - 被调用过程返回时执行的恢复机器状态，释放被调用过程活动记录，使调用过程能够继续执行的代码
- **调用序列和返回序列常常都分成两部分，分处于调用过程和被调用过程的活动记录中**

对运行栈的管理

- **代码序列**
  - 过程调用和过程返回都需要执行一些代码来管理活动记录栈，保存或恢复机器状态等
- **过程调用序列(calling sequence)**
  - 过程调用时执行的分配活动记录，把信息填入它的域中，使被调用过程可以开始执行的代码
- **过程返回序列(return sequence)**
  - 被调用过程返回时执行的恢复机器状态，释放被调用过程活动记录，使调用过程能够继续执行的代码
- **调用序列和返回序列常常都分成两部分，分处于调用过程和被调用过程的活动记录中**

对运行栈的管理

- **代码序列**
  - 过程调用和过程返回都需要执行一些代码来管理活动记录栈，保存或恢复机器状态等
- **过程调用序列(calling sequence)**
  - 过程调用时执行的分配活动记录，把信息填入它的域中，使被调用过程可以开始执行的代码
- **过程返回序列(return sequence)**
  - 被调用过程返回时执行的恢复机器状态，释放被调用过程活动记录，使调用过程能够继续执行的代码
- **调用序列和返回序列常常都分成两部分，分处于调用过程和被调用过程的活动记录中**

对运行栈的管理

- 即使是同一种语言，过程调用序列、返回序列和活动记录中各域的排放次序，也会因实现而异
- 设计这些序列和活动记录的一些原则
  - 调用者与被调用者之间交流的数据放在被调用者活动记录的开始处，尽量靠近调用者的活动记录
    - 参数域紧邻调用者活动记录
    - 返回值在参数域之上

对运行栈的管理

- 即使是同一种语言，过程调用序列、返回序列和活动记录中各域的排放次序，也会因实现而异
- 设计这些序列和活动记录的一些原则
  - 固定长度的域放在活动记录的中间
    - 控制链
    - 访问链
    - 机器状态

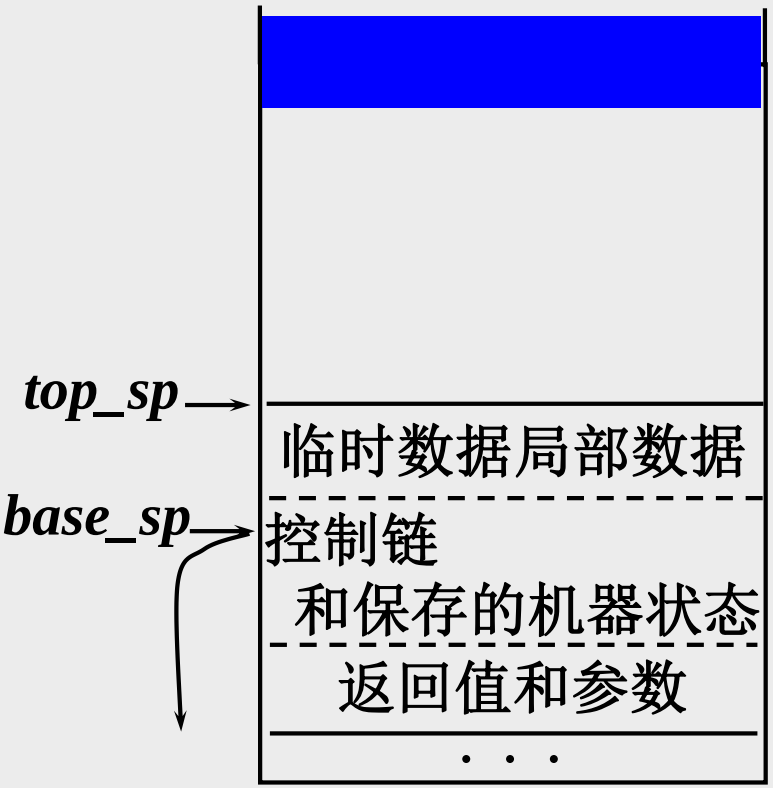
对运行栈的管理

- 即使是同一种语言，过程调用序列、返回序列和活动记录中各域的排放次序，也会因实现而异
- 设计这些序列和活动记录的一些原则
  - 不能编译时刻确定大小的数据一般放在活动记录的末端
    - 局部动态数组
    - 临时数据

对运行栈的管理

- 即使是同一种语言，过程调用序列、返回序列和活动记录中各域的排放次序，也会因实现而异
- 设计这些序列和活动记录的一些原则
  - 以活动记录中间的某个位置作为基地址(控制链)来活动记录中的内容

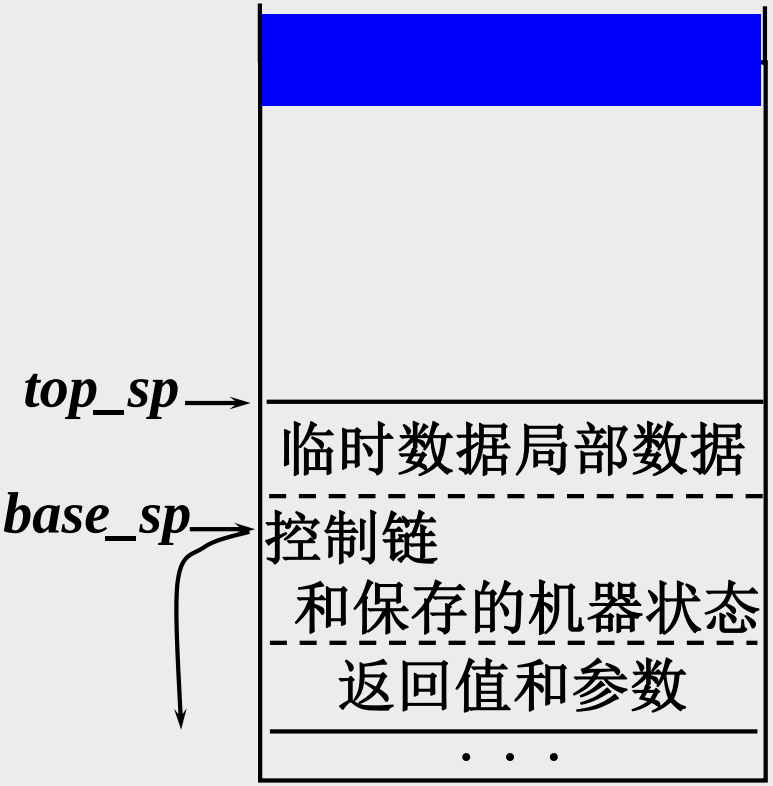
全局栈式存储分配



- **top_sp**: 栈顶寄存器，如esp、rsp。指向栈顶活动记录的末端
- **base_sp**: 基址寄存器，如ebp、rbp。指向栈顶活动记录中控制链所在位置。

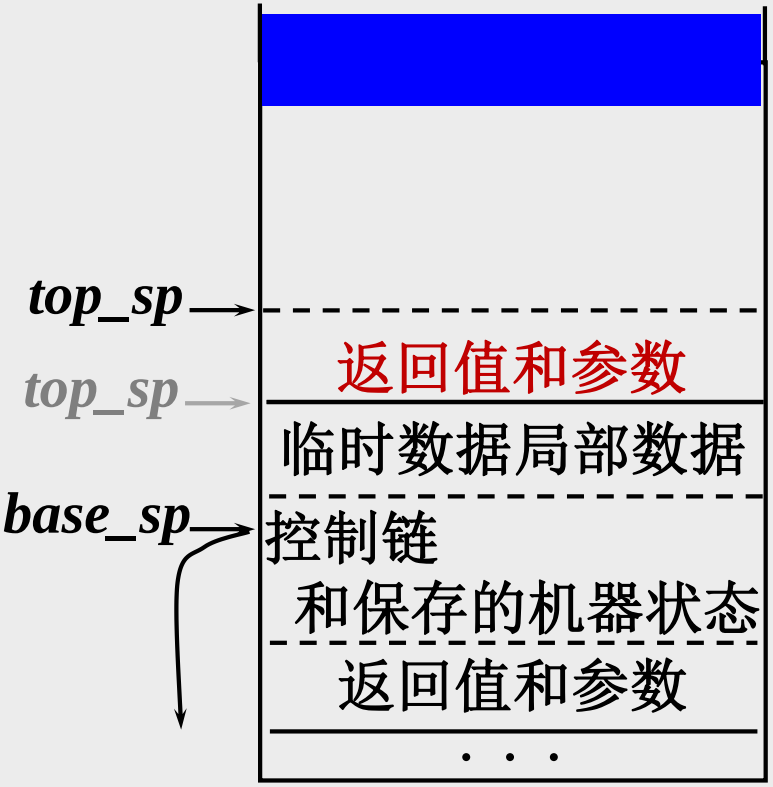
全局栈式存储分配

- 过程p调用过程q的调用序列(**栈往上增长**)



全局栈式存储分配

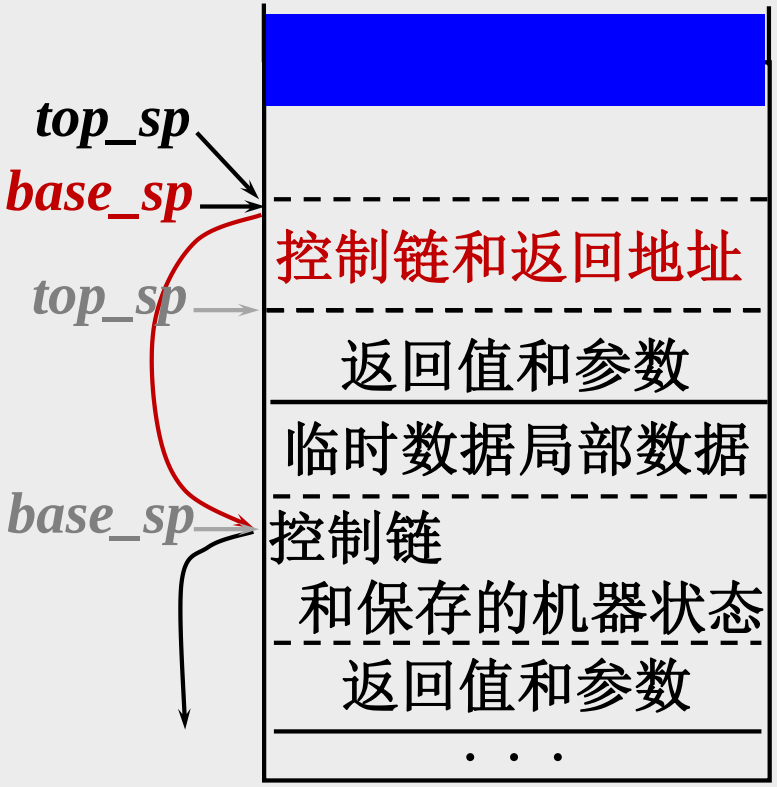
- 过程p调用过程q的调用序列(栈往上增长)



(1) p计算实参，依次放入栈顶，并在栈顶留出放返回值的空间。top_sp的值在此过程中被改变

全局栈式存储分配

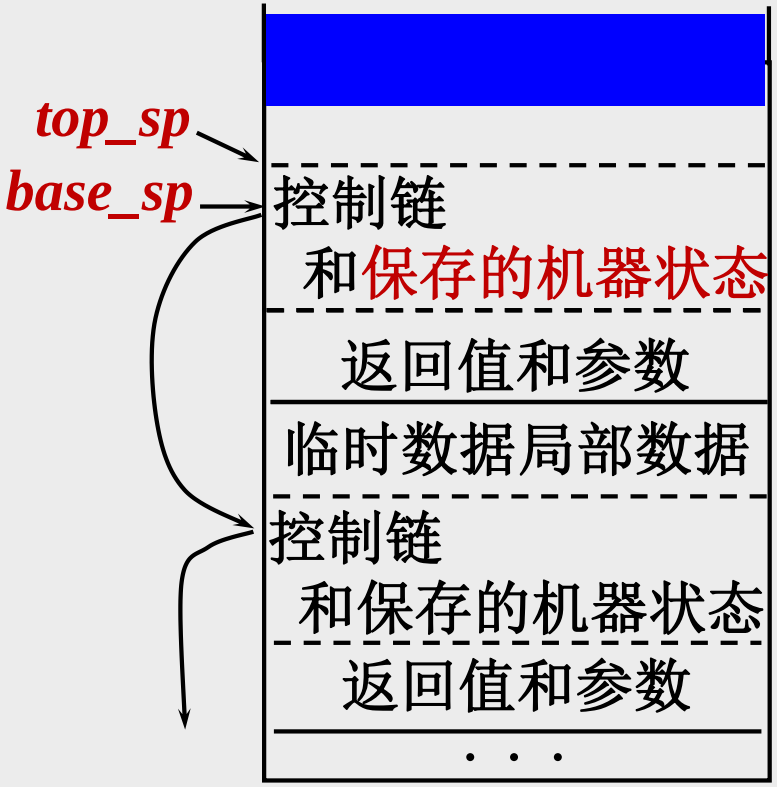
- 过程p调用过程q的调用序列



(2) p把返回地址和当前base_sp的值存入q的活动记录中，建立q的控制链，改变base_sp的值

全局栈式存储分配

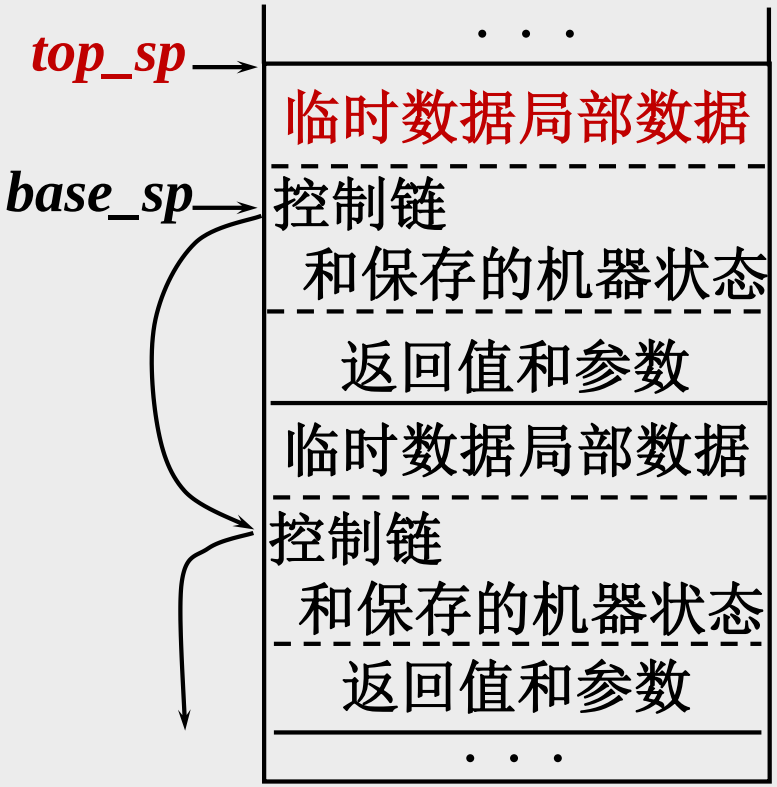
- 过程p调用过程q的调用序列



(3) q保存寄存器的值和其它机器状态信息

全局栈式存储分配

- 过程p调用过程q的调用序列

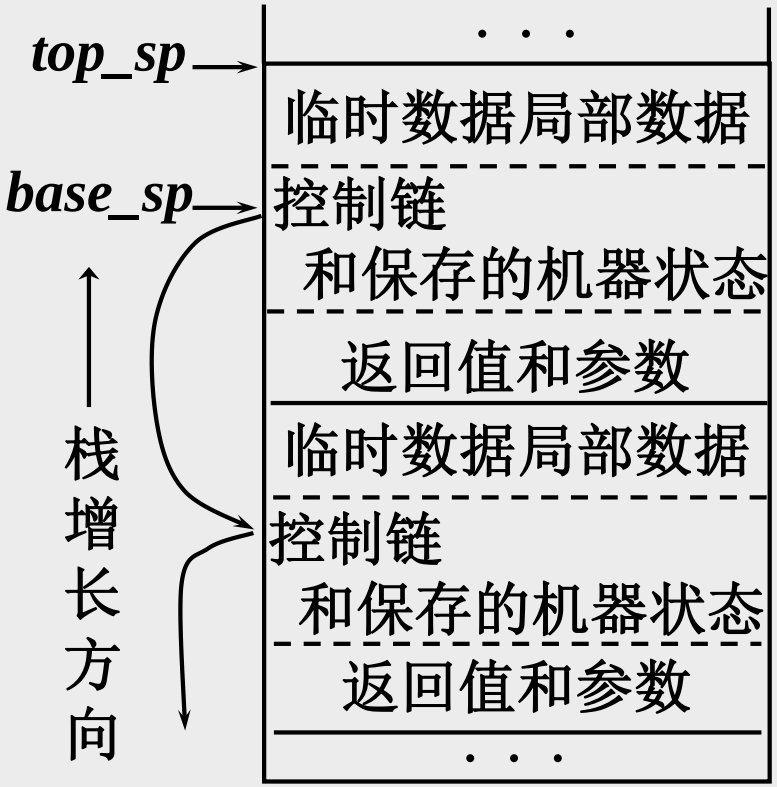


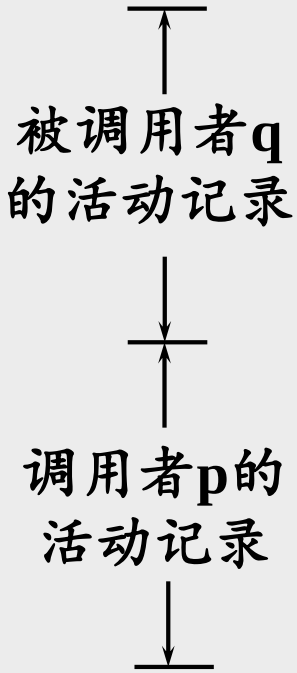
base_sp不变，指向活动记录中间

(4) q根据局部数据域和临时数据域的大小减小top_sp的值，初始化它的局部数据，并开始执行过程体

全局栈式存储分配

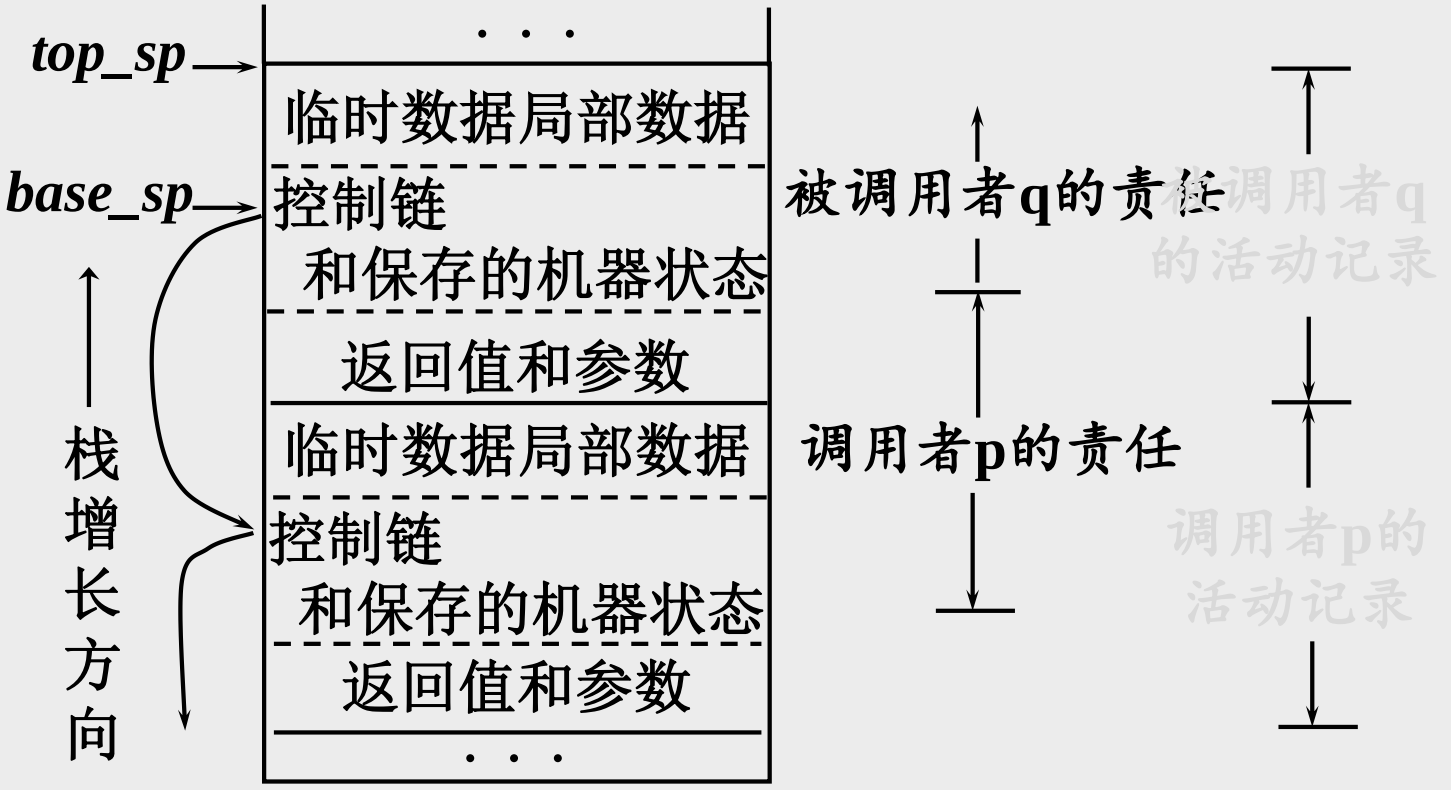
- 调用者p和被调用者q之间的任务划分





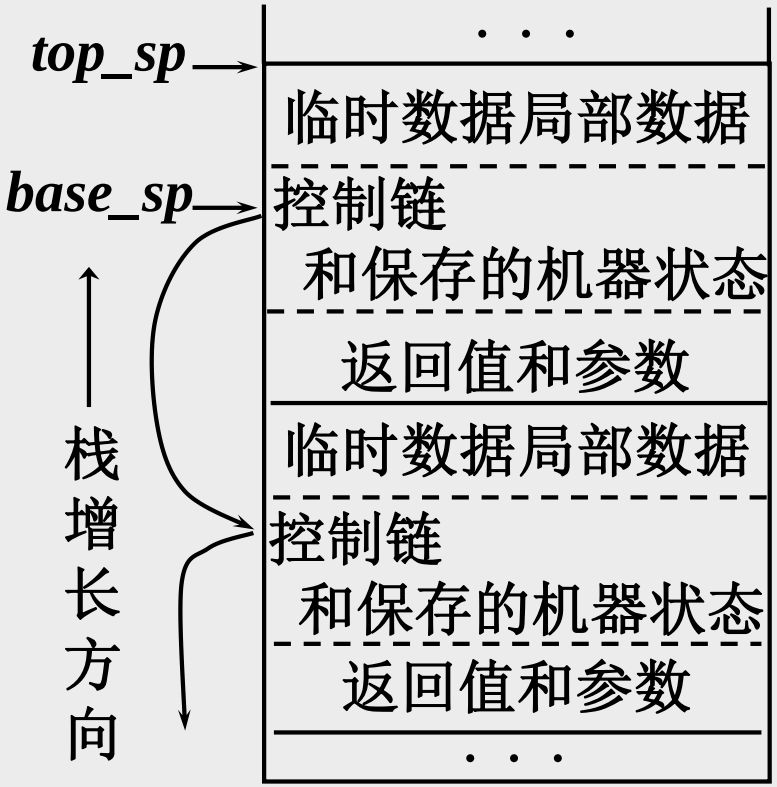
全局栈式存储分配

- 调用者p和被调用者q之间的任务划分



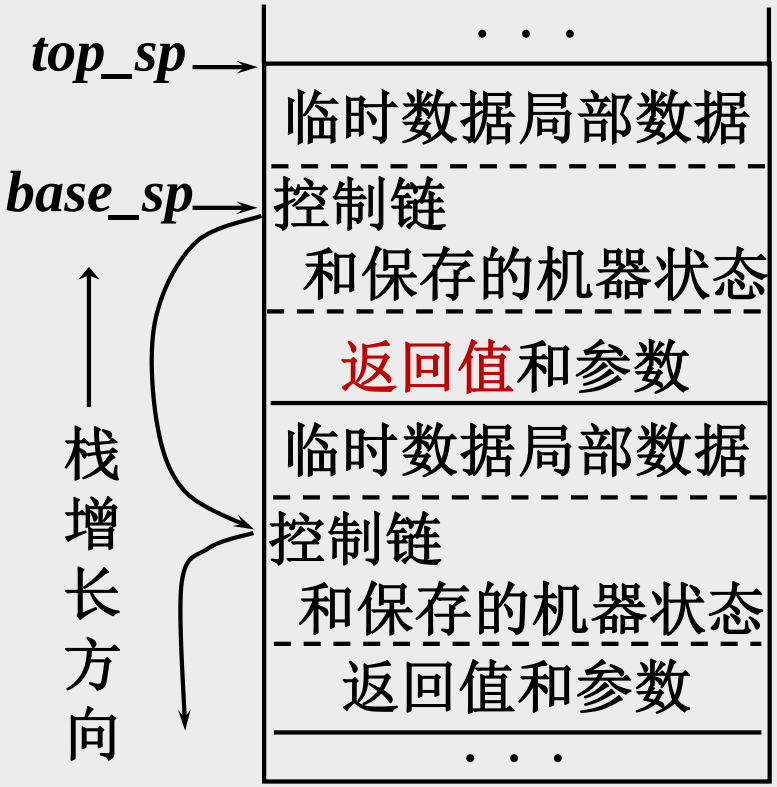
全局栈式存储分配

- 过程p调用过程q的返回序列



全局栈式存储分配

- 过程p调用过程q的返回序列

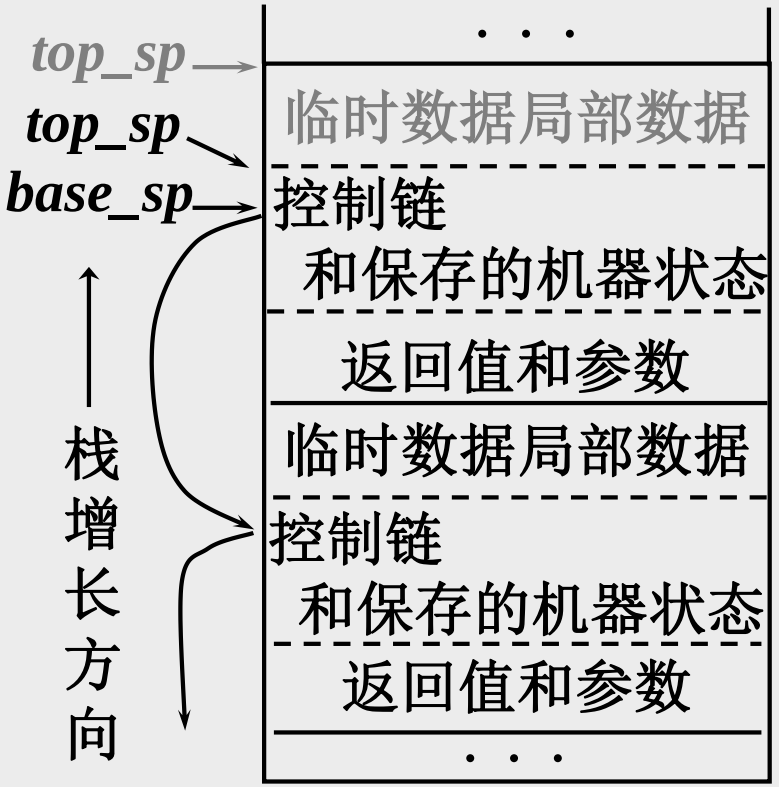


(1) q把返回值置入邻近p的活动记录的地方

引申：参数个数可变场合难以确定存放返回值的位置，因此通常用寄存器传递返回值

全局栈式存储分配

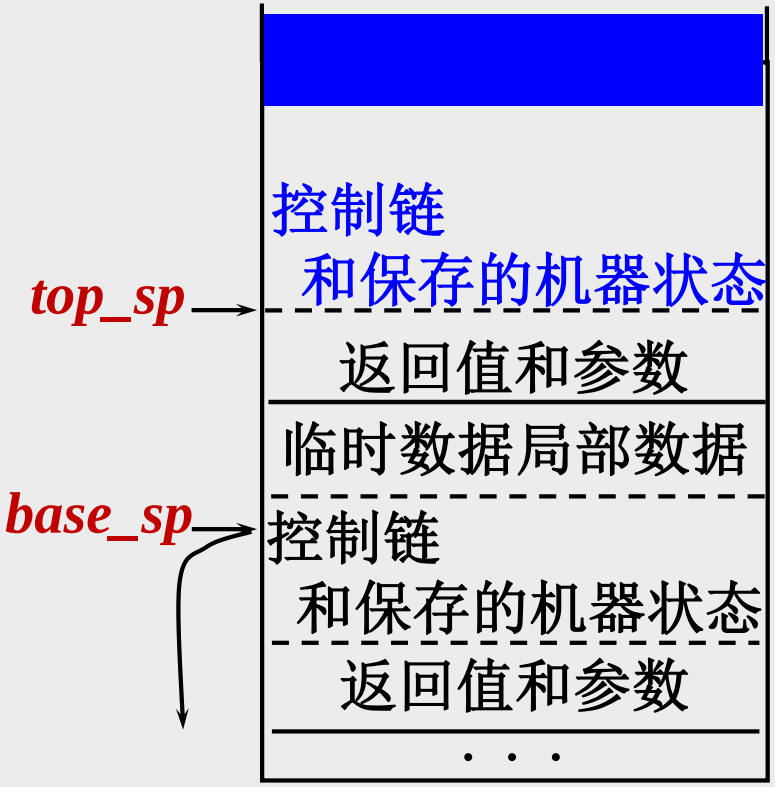
- 过程p调用过程q的返回序列



(2) q对应调用序列的步骤(4)，增加top_sp的值

全局栈式存储分配

- 过程p调用过程q的返回序列



(3) q恢复寄存器(包括base_sp)和机器状态，返回p

控制权转到p

全局栈式存储分配

- 过程p调用过程q的返回序列

(4) p根据参数个数与类型和返回值类型调整top_sp，然后取出返回值

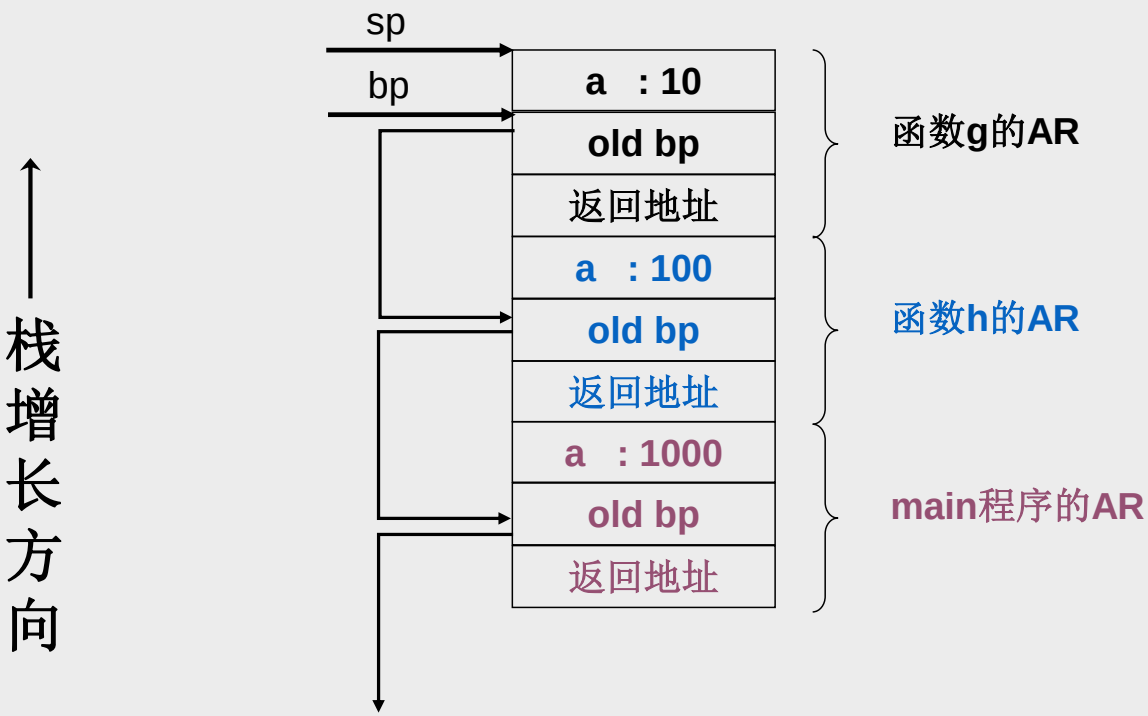
全局栈式存储分配

有C程序如下：

```
void g() { int a ; a = 10 ; }
void h() { int a ; a = 100; g();  }
main()
{  int a = 1000; h(); }
```

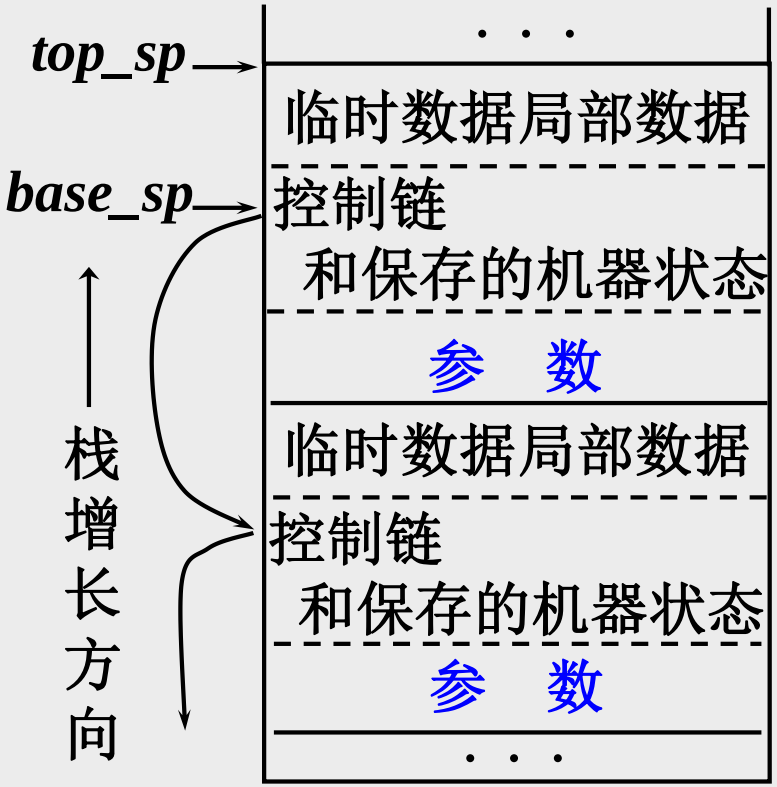
全局栈式存储分配

- 过程g被调用时，活动记录栈的（大致）内容



全局栈式存储分配

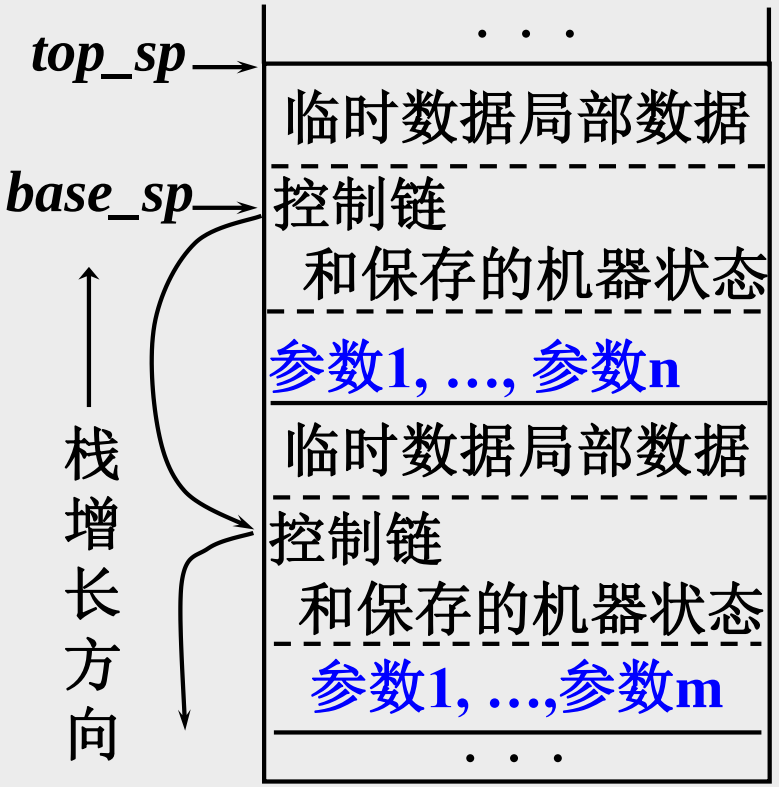
- 过程的参数个数可变的情况



(1) 函数返回值改成用寄存器传递

全局栈式存储分配

- 过程的参数个数可变的情况

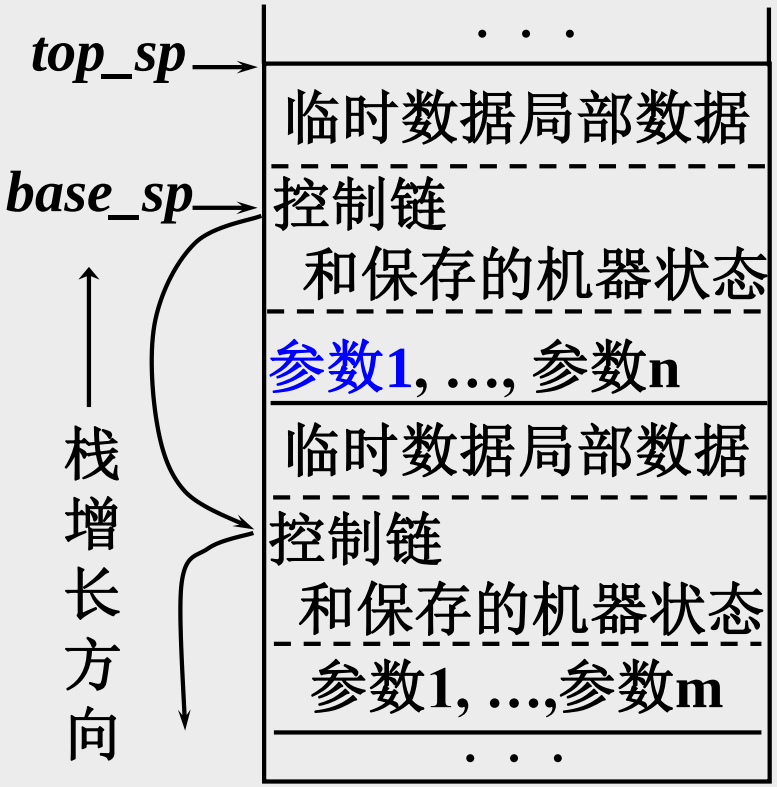


(2) 编译器产生将实参表达式逆序计算并将结果进栈的代码

自上而下依次是参数1, …, 参数n

全局栈式存储分配

- 过程的参数个数可变的情况

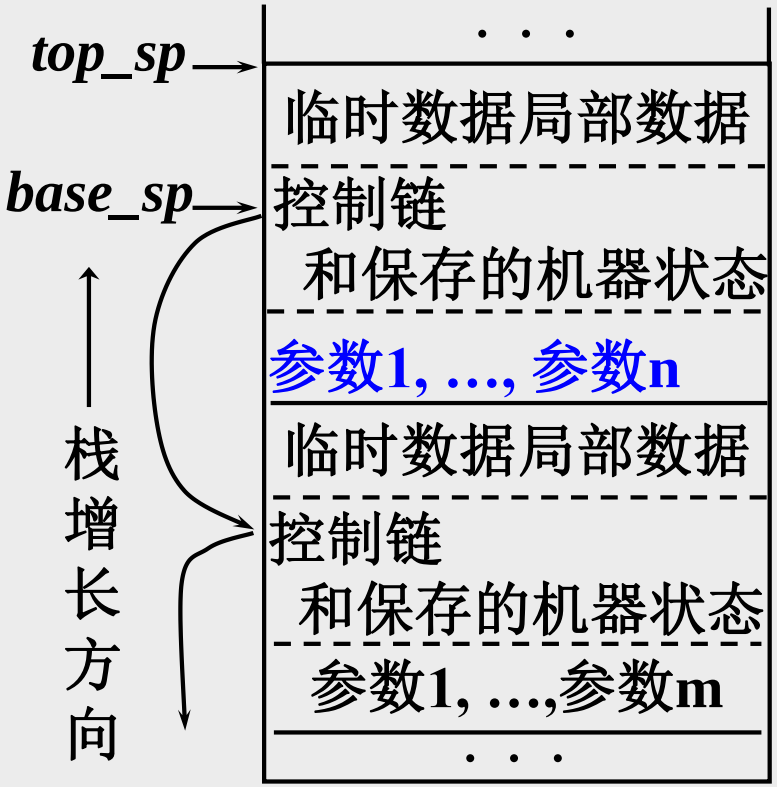


(3) 被调用函数能准确地知道第一个参数的位置

But why?

全局栈式存储分配

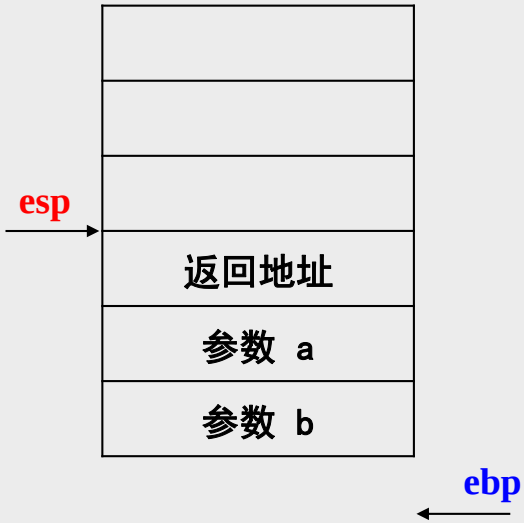
- 过程的参数个数可变的情况



(4) 被调用函数根据第一个参数到栈中取第二、第三个参数等等

有参函数的活动记录

```
void func( int a , int b )
{
    int c , d;
    c = a;
    d = b;
}
```




```
.file   "ar.c"
        .text
.globl func
        .type   func,@function
func:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    8(%ebp), %eax
        movl    %eax, -4(%ebp)
        movl    12(%ebp), %eax
        movl    %eax, -8(%ebp)
        leave
        ret
```

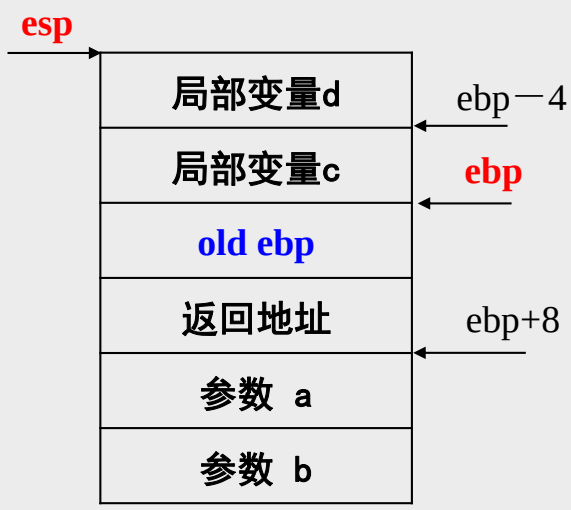
有参函数的活动记录

```
void func( int a , int b )
{
    int c , d;
    c = a;
    d = b;
}
```


```
.file   "ar.c"
        .text
.globl func
        .type   func,@function
func:
        pushl   %ebp //老基地址压栈
        movl    %esp, %ebp //基地址指针=栈顶指针
        subl    $8, %esp
        movl    8(%ebp), %eax
        movl    %eax, -4(%ebp)
        movl    12(%ebp), %eax
        movl    %eax, -8(%ebp)
        leave
        ret
```

有参函数的活动记录

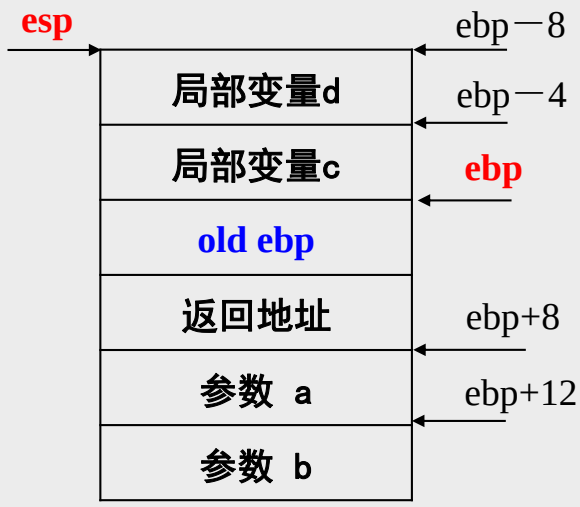
```
void func( int a , int b )
{
    int c , d;
    c = a;
    d = b;
}
```




```
.file   "ar.c"
        .text
.globl func
        .type   func,@function
func:
        pushl   %ebp //老基地址压栈
        movl    %esp, %ebp //基地址指针=栈顶指针
        subl    $8, %esp //分配c,d局部变量空间
        movl    8(%ebp), %eax //将a值放进寄存器
        movl    %eax, -4(%ebp) //将a值赋给c
        movl    12(%ebp), %eax
        movl    %eax, -8(%ebp)
        leave
        ret
```

有参函数的活动记录

```
void func( int a , int b )
{
    int c , d;
    c = a;
    d = b;
}
```



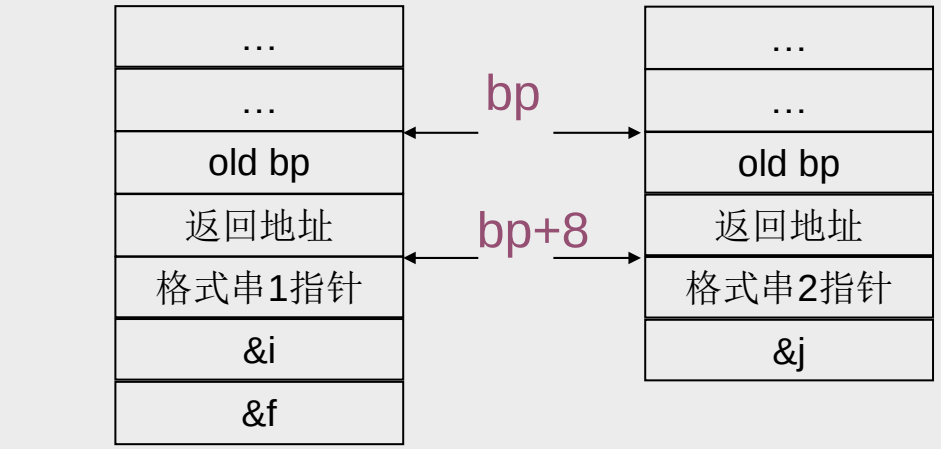

```
.file   "ar.c"
        .text
.globl func
        .type   func,@function
func:
        pushl   %ebp //老基地址压栈
        movl    %esp, %ebp //基地址指针=栈顶指针
        subl    $8, %esp //分配c,d局部变量空间
        movl    8(%ebp), %eax //将a值放进寄存器
        movl    %eax, -4(%ebp) //将a值赋给c
        movl    12(%ebp), %eax //将b值放进寄存器
        movl    %eax, -8(%ebp) //将b值赋给d
        leave
        ret
```

scanf的可变参数

有如下C程序:

```
main()
{
 int i ; float f;  int j ;
 scanf(“%d%f”, &i, &f); //第一次调用时3个参数
 scanf(“%d”, &j ); //第二次调用时 2个参数
 return 0;
}
```

scanf的可变参数



scanf的第一次调用时AR　　scanf的第二次调用时AR

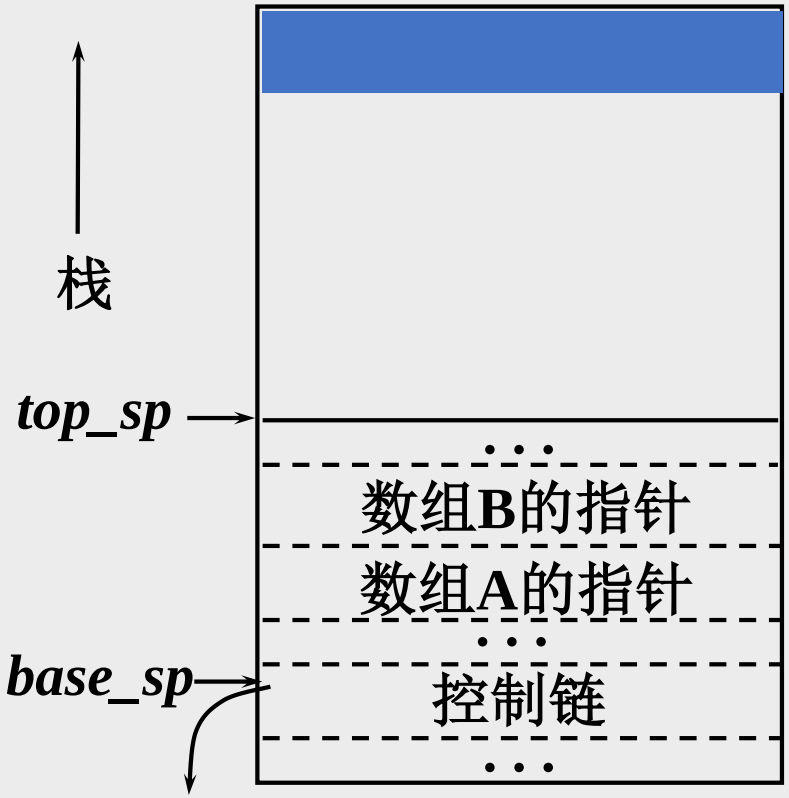
由于C语言采用逆序传递参数，格式串参数将被放在AR中的“固定”位置，即bp+8。而由此参数即可确定待输入值的参数（变量）的个数。从而适应参数个数变化的情况。

全局栈式存储分配

- **栈上可变长数据**
  - 数据对象**的长度在编译时不能确定的情况**
  - 但仅仅为该活动运行过程使用
  - 避免垃圾回收开销
  - **例：局部数组的大小要等到过程激活时才能确定**
- **如何在栈上布局可变长的数组?**
  - 先分配存放数组指针的单元，对数组的访问通过指针间接访问
  - 运行时，这些指针指向分配在栈顶的数组存储空间

全局栈式存储分配

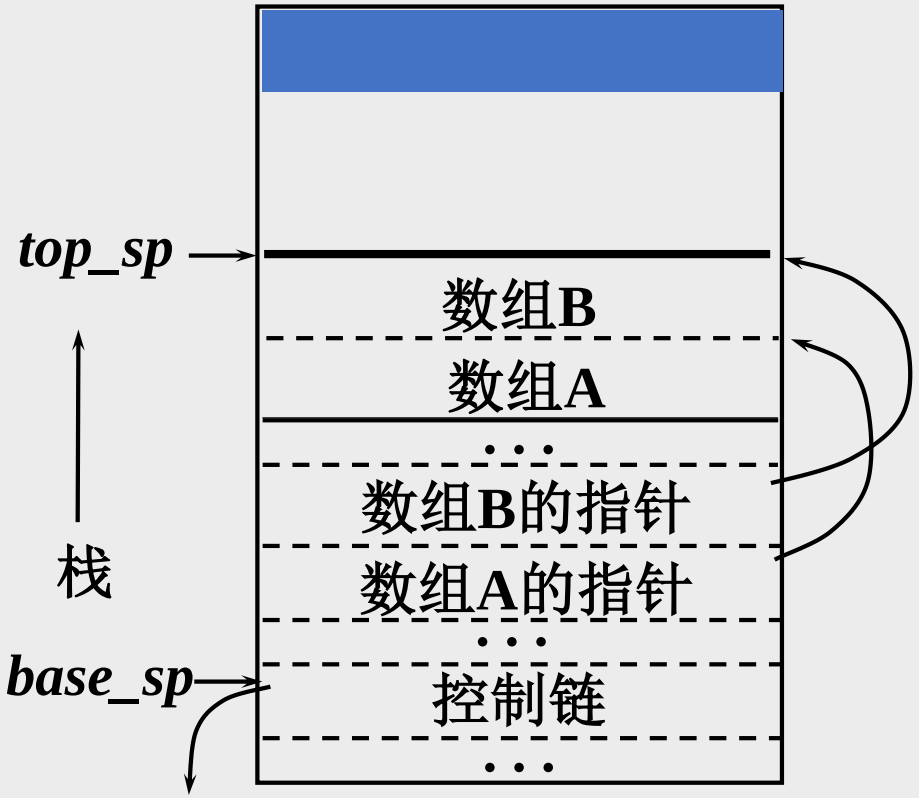
- 访问动态分配的数组



(1) 编译时，在活动记录中为这样的数组分配存放数组指针的单元

全局栈式存储分配

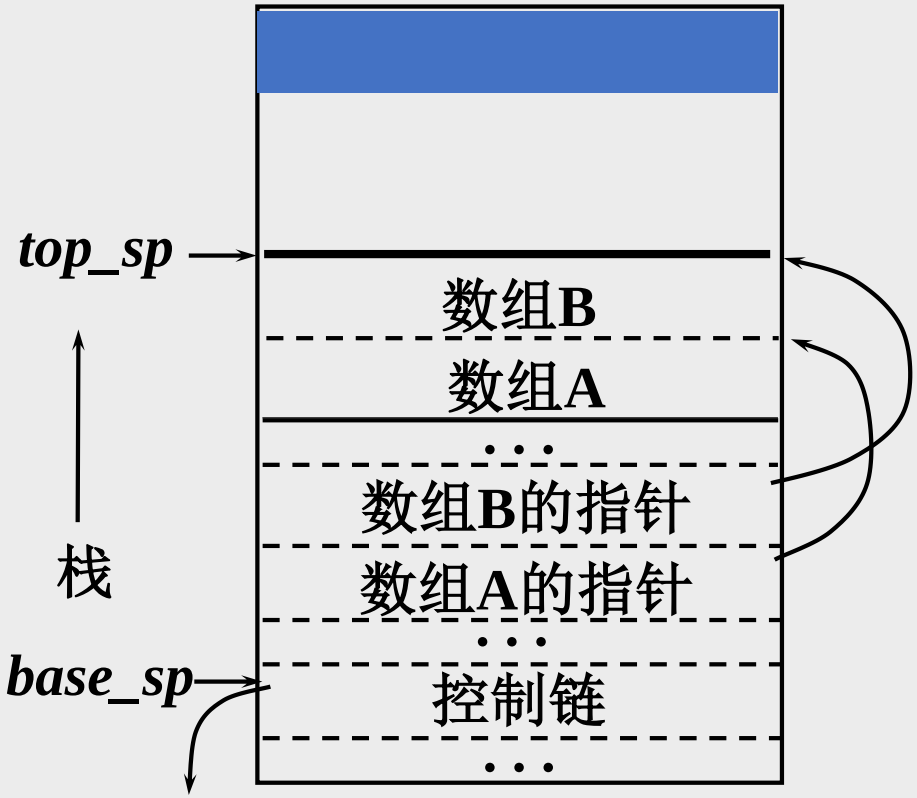
- 访问动态分配的数组



(2) 运行时，这些指针指向分配在栈顶的数组存储空间

全局栈式存储分配

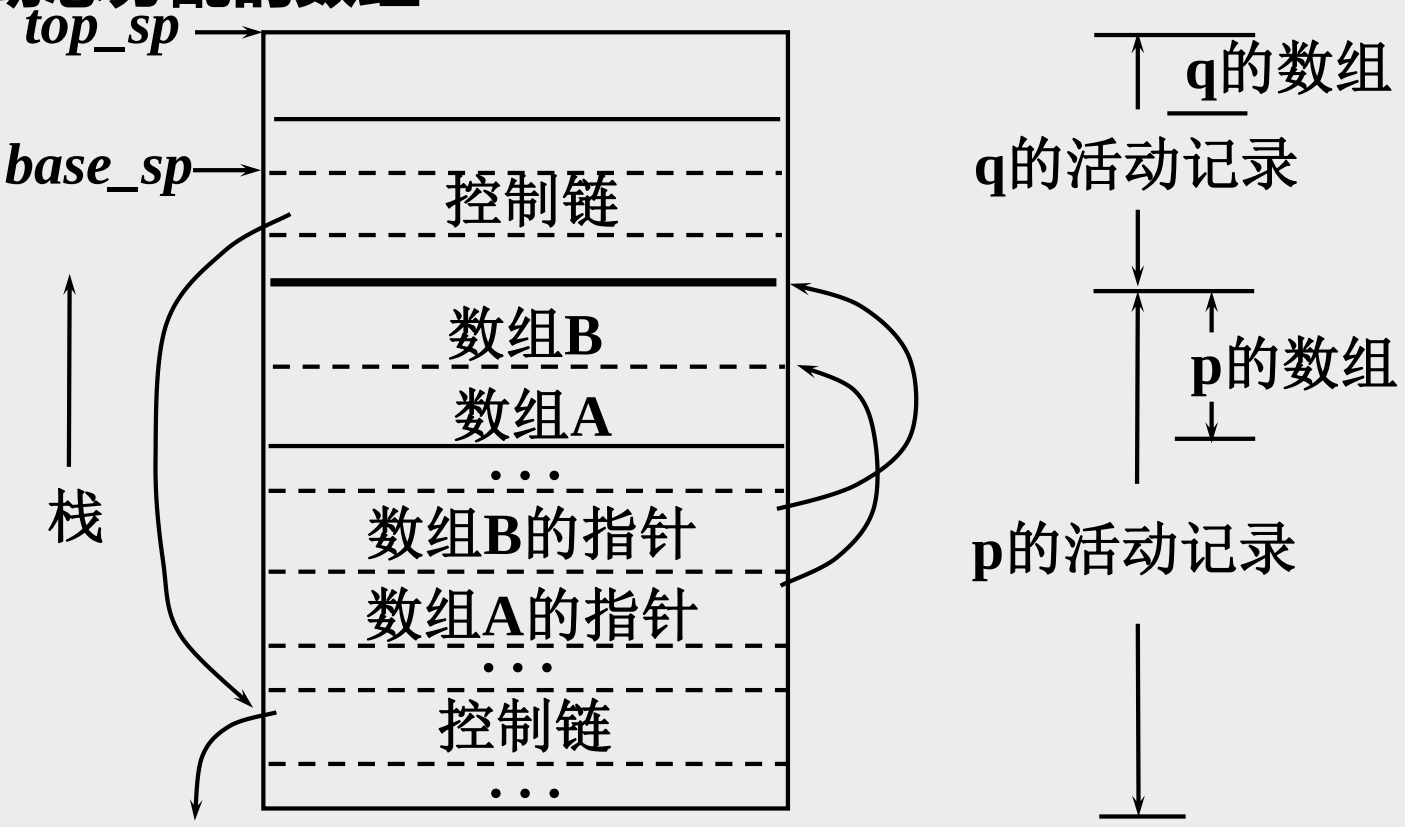
- 访问动态分配的数组



(3) 运行时，对数组A和B的访问都要通过相应指针来间接访问

全局栈式存储分配

- 访问动态分配的数组



全局栈式存储分配

- 悬空引用
  - 引用某个已被释放的存储单元

例：main中引用p指向的对象

```
main( ) {                    |    int * dangle ( ) {
    int *q;                  |        int j = 20;
    q = dangle ( );          |        return &j;
}                            |    }
```

《编译原理和技术》

运行时刻环境 II

——空间的栈式分配

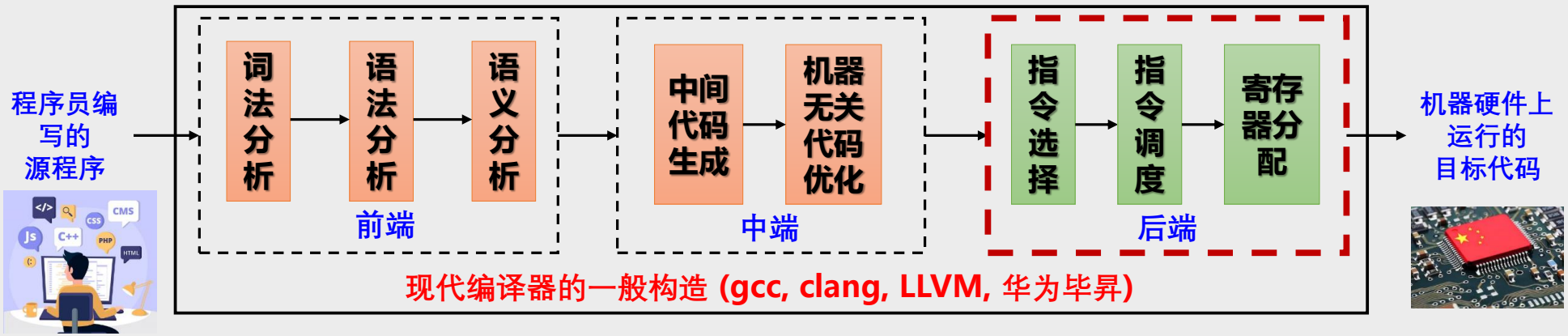
谢谢！

《编译原理和技术》

目标代码生成 I

中科大计算机学院
李 诚
2022-11-14

本节提纲



- 代码生成器任务概述
- 一个简单的目标机器模型
- 指令选择
- 寄存器选择

目标：语义正确+资源有效利用

代码生成器的主要任务及问题

指令选择

- 为中间表示(IR)语句选择适当的目标机器指令
- 如果不考虑效率，则十分简单
- 例：

  三地址语句 x = y + z 对应的目标代码为

```
LD R0, y /*将y的值加载到寄存器R0中*/
ADD R0, R0, z /*将z加到R0上*/
ST x, R0   /*将R0的值保存到x中*/
```

代码生成器的主要任务及问题

指令选择

- 为中间表示(IR)语句选择适当的目标机器指令
- 例：三地址语句x = y + z; m = x + n对应的目标代码为

```
LD R0, y /*将y的值加载到寄存器R0中*/
ADD R0, R0, z /*将z加到R0上*/
ST x, R0   /*将R0的值保存到x中*/
LD R0, x /*将x的值加载到寄存器R0中*/
ADD R0, R0, n /*将n加到R0上*/
ST m, R0   /*将R0的值保存到m中*/
```

代码生成器的主要任务及问题

指令选择

- 为中间表示(IR)语句选择适当的目标机器指令
- 例：三地址语句x = y + z; m = x + n对应的目标代码为

```
LD R0, y /*将y的值加载到寄存器R0中*/
ADD R0, R0, z /*将z加到R0上*/
ST x, R0   /*将R0的值保存到x中*/
LD R0, x /*将x的值加载到寄存器R0中*/
ADD R0, R0, n /*将n加到R0上*/
ST m, R0   /*将R0的值保存到m中*/
```

冗余指令

代码生成器的主要任务及问题

- 同一中间表示代码可以由多组指令序列来实现，但不同实现之间的效率差别是很大的
  - 例：语句a = a + 1可以有两种实现方式

```
LD    R0, a          // R0 = a
ADD   R0, R0,  #1 //  R0 = R0 + 1
ST    a, R0          //  a = R0
```

```
INC a
```

- 因此，生成高质量代码需要知道指令代价。

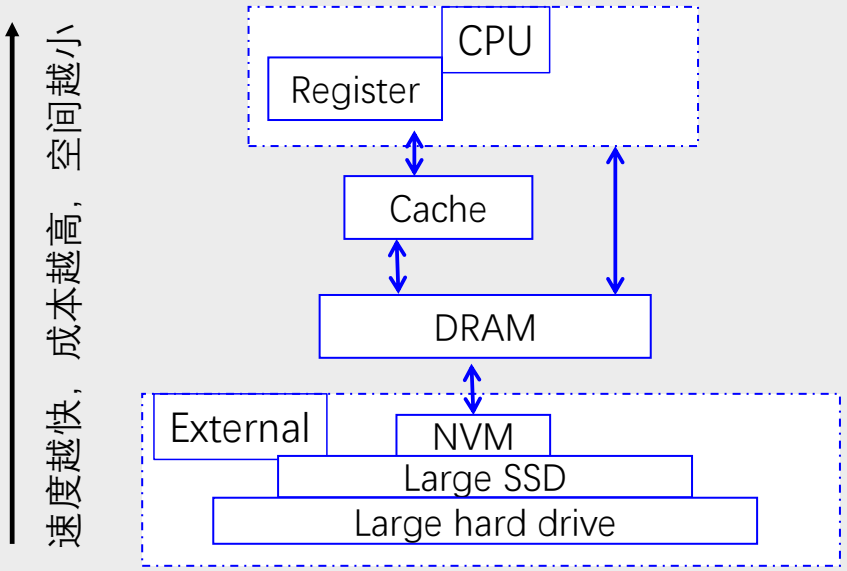
代码生成器的主要任务及问题

寄存器分配和指派

- 在每个程序点上决定将哪些值放在哪些寄存器中
- 在寄存器中获得运算分量比内存要快，但是寄存器数量十分有限
- 高效利用寄存器可以减少CPU等待的时间

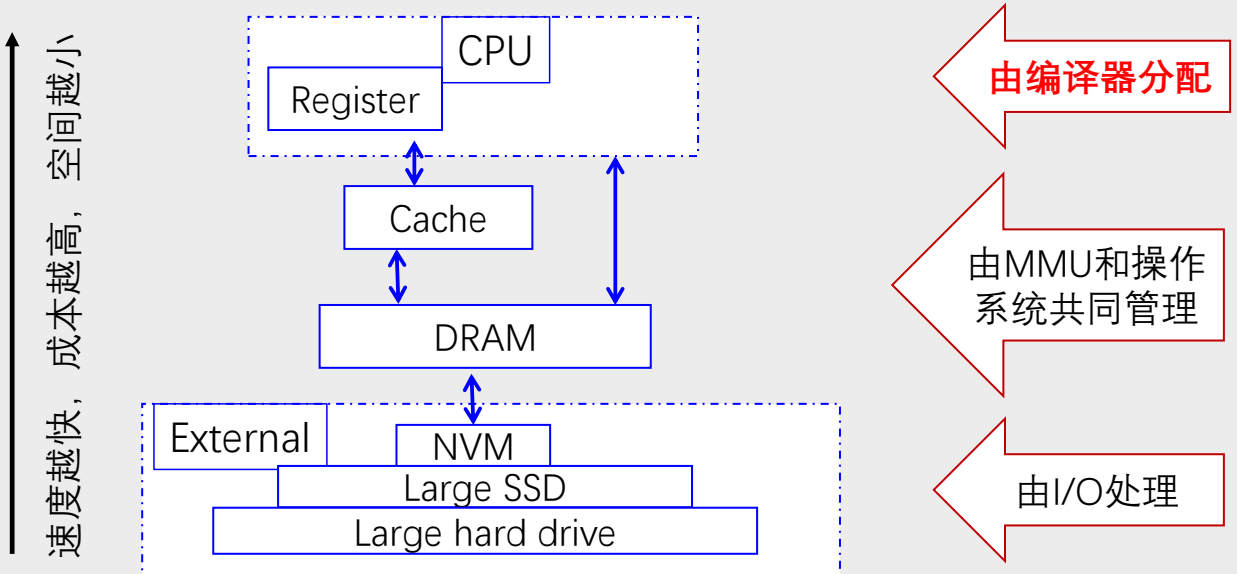
代码生成器的主要任务及问题

- 除了考虑指令的代价和序列长度外，我们还需要考虑运算对象和结果如何存储的问题。

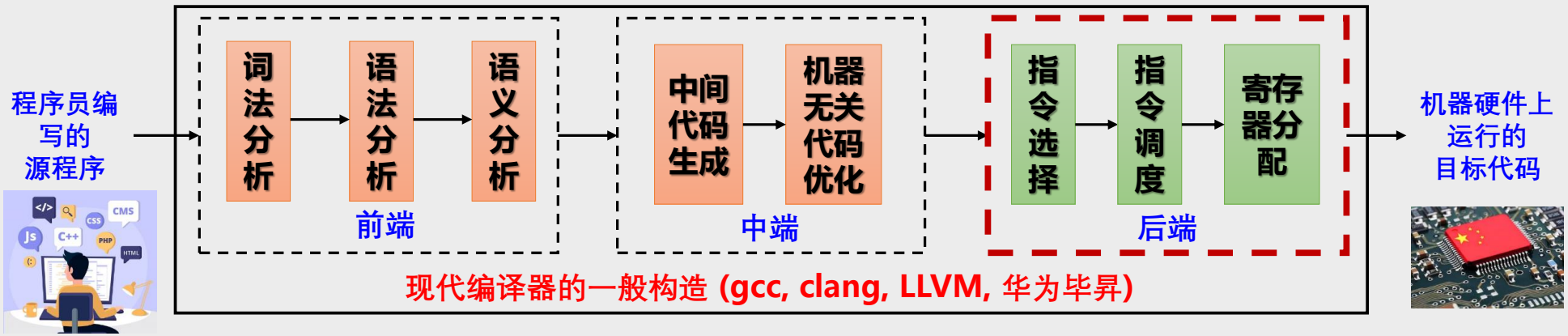


代码生成器的主要任务及问题

- 除了考虑指令的代价和序列长度外，我们还需要考虑运算对象和结果如何存储的问题。



本节提纲



- 代码生成器任务概述
- 一个简单的目标机器模型
- 指令选择
- 寄存器选择

目标：语义正确+资源有效利用

一个简单的目标机器模型

三地址机器模型

- 目标机器指令集(也可以称为目标语言)包含LD、ST、运算、跳转等指令
- 内存按照字节寻址
- 假设有n个通用寄存器R0, R1, …, Rn-1
- 假设所有运算分量都是整数
- 指令之前可能有一个标号

目标机器指令集

- **加载指令　LD dst, addr**
  - LD R0, x
  - LD R1, R2
- **保存指令 ST x, R**
- **运算指令　OP dst, src1, src2**
- **跳转指令**
  - 无条件跳转　　BR L
  - 条件跳转　　Bcond r, L
    - 例：　BLTZ r, L (LTZ 是 less than zero的缩写)

寻址模式

- **变量名 a**
  - 例：LD R1, a

    R1 = contents(a) 其中contents(a) 表示a位置中存放的内容
- **a(r)：数组访问**
  - a是一个变量，r是一个寄存器
  - 例：LD R1, a(R2)

    R1 = contents(a + contents(R2))
- **c(r)：沿指针取值**
  - c是一个整数，r是一个寄存器
  - 例：LD R1, 100(R2)

    R1 = contents(100 + contents(R2))

寻址模式——间接寻址

- ***r**
  - 在寄存器r的内容所表示的位置上存放的内存位置
  - 例：LD R1, *R2

    R1 = contents(contents(contents(R2)))
- ***c(r)**
  - 在寄存器r中内容加上c后所表示的位置上存放的内存位置
  - 例：LD R1, *100(R2)

    R1 = contents(contents(100 + contents(R2)))

寻址模式——直接数寻址

- **#c**
  - c是一个常数
  - 例：LD R1, #100

    R1 = 100

指令的代价

- 在上述简单的目标机器上，指令代价简化为

1 + 指令的源和目的寻址模式(addressing mode)的附加代价

- 寄存器寻址模式附加代价为0
- 涉及内存位置或者常数的寻址方式代价为1

指令	代价	
LD R0，R1	1	寄存器
LD R0，M	2	寄存器+内存
LD R1，*100(R2)	2	寄存器+内存

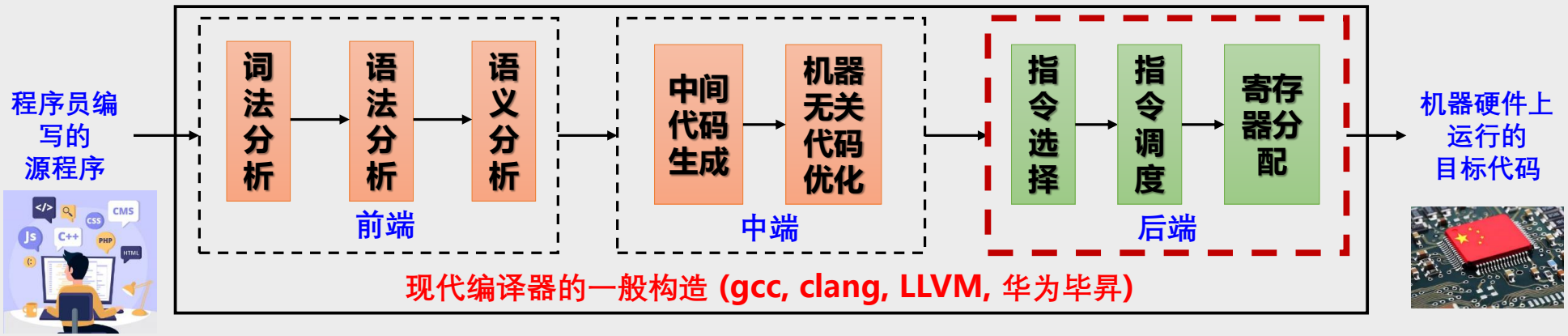
指令的代价——举例

三地址代码	x = y – z	b = a[i]	if x < y goto L

目标指令序列	LD r1 , y
	LD r2 , z
	SUB r1,r1,r2
	ST x, r1
		LD r1,i
		MUL r1,r1,8
		LD r2, a(r1)
		ST b, r2
			LD r1, x
			LD r2, y
			SUB r1,r1,r2
			BLTZ r1, M

代价	2+2+1+2 = 7	2+2+2+2=8	2+2+1+2=7

本节提纲



- 代码生成器任务概述
- 一个简单的目标机器模型
- 指令选择
- 寄存器选择

目标：语义正确+资源有效利用

运算语句的目标代码

- 三地址指令
  - x = y – z
- 目标代码
  - LD R1, y　　// R1 = y
  - LD R2, z　　// R2 = z
  - SUB R1, R1, R2　　// R1 = R1 – R2
  - ST x, R1　　// x = R1

数组寻址语句的目标代码

- 三地址指令
  - b = a[i]
  - a是一个实数数组，每个实数占8个字节
- 目标代码
  - LD R1, i　　// R1 = i
  - MUL R1, R1, 8　　// R1 = R1 * 8
  - LD R2, a(R1)　　// R2 = contents(a + contents(R1))
  - ST b, R2　　// b = R2

数组寻址语句的目标代码

- 三地址指令
  - a[j] = c
  - a是一个实数数组，每个实数占8个字节
- 目标代码
  - LD R1, c　　// R1 = c
  - LD R2, j　　// R2 = j
  - MUL R2, R2, 8　　// R2 = R2 * 8
  - ST a(R2), R1　　// contents(a + contents(R2)) = R1

数组寻址语句的目标代码

- 三地址指令
  - x = *p
- 目标代码
  - LD R1, p　　// R1 = p
  - LD R2, 0(R1)　　// R2 = contents(0 + contents(R1))
  - ST x, R2　　// x = R2

数组寻址语句的目标代码

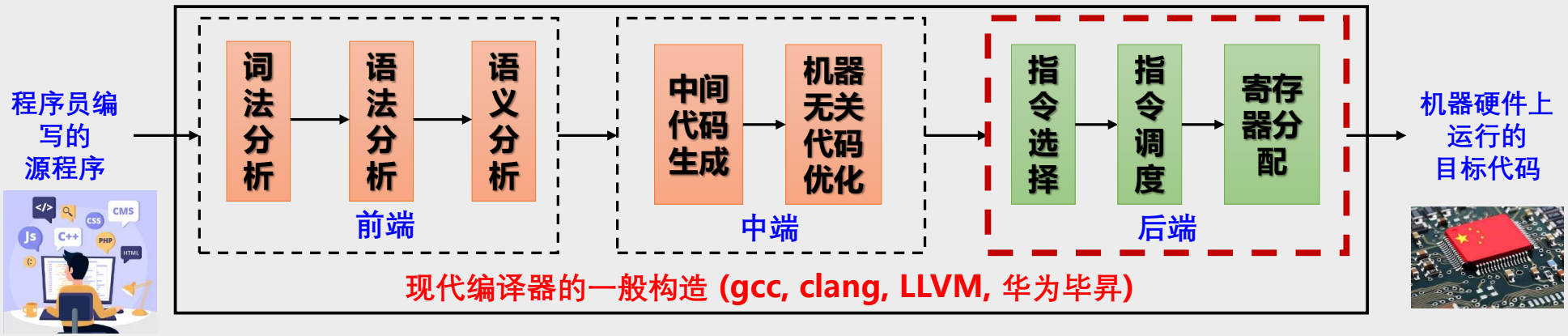
- 三地址指令
  - *p = y
- 目标代码
  - LD R1, p　　// R1 = p
  - LD R2, y　　// R2 = y
  - ST 0(R1), R2　　// contents(0 + contents(R1)) = R2

条件跳转语句的目标代码

- 三地址指令
  - if x < y goto L
- 目标代码
  - LD R1, x　　// R1 = x
  - LD R2, y　　// R2 = y
  - SUB R1, R1, R2　　// R1 = R1 – R2
  - BLTZ R1, M　　// if R1 < 0 jump to M

M是标号为L的三地址指令所产生的目标代码中的第一条指令的标号

本节提纲



- 代码生成器任务概述
- 一个简单的目标机器模型
- 指令选择
- **寄存器选择**

目标：语义正确+资源有效利用

寄存器及地址描述符

- **寄存器描述符 (register descriptor)**
  - 记录每个寄存器当前存放的是哪些变量的值
- **地址描述符 (address descriptor)**
  - 记录运行时每个名字的当前值存放在哪个或者哪些位置
  - 该位置可能是寄存器、栈单元、内存地址或者是它们的某个集合
  - 这些信息可以存放在该变量名对应的符号表条目中

三地址语句的目标代码生成

- **对每个形如x = y op z的三地址指令I**
  - 调用寄存器选择函数getReg(I)来为x、y、z选择寄存器Rx, Ry, Rz
  - 如果Ry中存放的不是y，则生成指令“LD Ry, y’”, y’是存放y的内存位置之一
  - 对于Rz和z的处理与上一步骤类似
  - 最后，生成目标指令“OP Rx, Ry, Rz”

基本块的收尾处理

- **对于一个在基本块出口处可能活跃的变量x，如果它的地址描述符表明它的值没有存放在x的内存位置上，则生成指令“ST x, R”**
  - R是在基本块结尾处存放x值的寄存器

管理寄存器和地址描述符

- **当生成加载、保存和其他指令时，必须同时更新寄存器和地址描述符**
  - 对于LD R, x指令
    - 修改R的寄存器描述符，使之只包含x
    - 修改x的地址描述符，把R作为新增位置加入到x的位置集合中
    - 从任何不同于x的地址描述符中删除R

管理寄存器和地址描述符

- **当生成加载、保存和其他指令时，必须同时更新寄存器和地址描述符**
  - 对于OP Rx, Ry, Rz指令
    - 修改Rx的寄存器描述符，使之只包含x
    - 从任何不同于Rx的寄存器描述符中删除x
    - 修改x的地址描述符，使之只包含位置Rx
    - 从任何不同于x的地址描述符中删除Rx

管理寄存器和地址描述符

- 当生成加载、保存和其他指令时，必须同时更新寄存器和地址描述符
  - 对于ST x, R指令
    - 修改x的地址描述符，使之包含自己的内存位置

管理寄存器和地址描述符

- **当生成加载、保存和其他指令时，必须同时更新寄存器和地址描述符**
  - 对于x = y指令，假设总是为x和y分配同一个寄存器，如果需要生成“LD Ry, y’”，则
    - 修改Ry的寄存器描述符，使之只包含y
    - 修改y的地址描述符，把Ry作为新增位置加入y的位置集合中
    - 从任何不同于y的地址描述符中删除Ry
    - 修改Ry的寄存器描述符，使之也包含x
    - 修改x的地址描述符，使之只包含Ry

寄存器分配选择——举例

基本块三地址代码如下：

t = a – b

u = a – c

v = t + u

a = d

d = v + u

t、u、v为临时变量

a、b、c、d在出口处活跃

R1	R2	R3

a	b	c	d	t	u	v
a	b	c	d			

寄存器分配选择——举例

基本块三地址代码如下：

t = a – b

u = a – c

v = t + u

a = d

d = v + u

t、u、v为临时变量

a、b、c、d在出口处活跃

LD R1, a

LD R2, b

SUB R2, R1, R2

R1	R2	R3

a	b	c	d	t	u	v
a	b	c	d			

寄存器分配选择——举例

基本块三地址代码如下：

t = a – b

u = a – c

v = t + u

a = d

d = v + u

t、u、v为临时变量

a、b、c、d在出口处活跃

LD R1, a

LD R2, b

SUB R2, R1, R2

R1	R2	R3
a	b	

a	b	c	d	t	u	v
a, R1	b, R2	c	d			

寄存器分配选择——举例

基本块三地址代码如下：

t = a – b

u = a – c

v = t + u

a = d

d = v + u

t、u、v为临时变量

a、b、c、d在出口处活跃

LD R1, a

LD R2, b

SUB R2, R1, R2

R1	R2	R3
a	t	

a	b	c	d	t	u	v
a, R1	b	c	d	R2		

寄存器分配选择——举例

基本块三地址代码如下：

t = a – b

u = a – c

v = t + u

a = d

d = v + u

t、u、v为临时变量

a、b、c、d在出口处活跃

LD R3, c

SUB R1, R1, R3

R1	R2	R3
a	t	

a	b	c	d	t	u	v
a, R1	b	c	d	R2		

寄存器分配选择——举例

基本块三地址代码如下：

t = a – b

u = a – c

v = t + u

a = d

d = v + u

t、u、v为临时变量

a、b、c、d在出口处活跃

```
LD R3, c
SUB R1, R1, R3
```

R1	R2	R3
a	t	c

a	b	c	d	t	u	v
a, R1	b	c, R3	d	R2		

寄存器分配选择——举例

基本块三地址代码如下：

t = a – b

u = a – c

v = t + u

a = d

d = v + u

t、u、v为临时变量

a、b、c、d在出口处活跃

LD R3, c

SUB R1, R1, R3

R1	R2	R3
u	t	c

a	b	c	d	t	u	v
a	b	c, R3	d	R2	R1	

寄存器分配选择——举例

基本块三地址代码如下：

t = a – b

u = a – c

v = t + u

a = d

d = v + u

t、u、v为临时变量

a、b、c、d在出口处活跃

ADD R3, R2, R1

R1	R2	R3
u	t	c

a	b	c	d	t	u	v
a	b	c, R3	d	R2	R1	

寄存器分配选择——举例

基本块三地址代码如下：

t = a – b

u = a – c

v = t + u

a = d

d = v + u

t、u、v为临时变量

a、b、c、d在出口处活跃

ADD R3, R2, R1

R1	R2	R3
u	t	v

a	b	c	d	t	u	v
a	b	c	d	R2	R1	R3

寄存器分配选择——举例

基本块三地址代码如下：

t = a – b

u = a – c

v = t + u

a = d

d = v + u

t、u、v为临时变量

a、b、c、d在出口处活跃

LD R2, d

R1	R2	R3
u	t	v

a	b	c	d	t	u	v
a	b	c	d	R2	R1	R3

寄存器分配选择——举例

基本块三地址代码如下：

t = a – b

u = a – c

v = t + u

a = d

d = v + u

t、u、v为临时变量

a、b、c、d在出口处活跃

LD R2, d

R1	R2	R3
u	a, d	v

a	b	c	d	t	u	v
R2	b	c	d, R2		R1	R3

寄存器分配选择——举例

基本块三地址代码如下：

t = a – b

u = a – c

v = t + u

a = d

d = v + u

t、u、v为临时变量

a、b、c、d在出口处活跃

ADD R1, R3, R1

R1	R2	R3
u	a, d	v

a	b	c	d	t	u	v
R2	b	c	d, R2		R1	R3

寄存器分配选择——举例

基本块三地址代码如下：

t = a – b

u = a – c

v = t + u

a = d

d = v + u

exit

t、u、v为临时变量

a、b、c、d在出口处活跃

ST a, R2

ST d, R1

R1	R2	R3
d	a	v

a	b	c	d	t	u	v
R2	b	c	R1			R3

寄存器分配选择——举例

基本块三地址代码如下：

t = a – b

u = a – c

v = t + u

a = d

d = v + u

exit

t、u、v为临时变量

a、b、c、d在出口处活跃

ST a, R2

ST d, R1

R1	R2	R3
d	a	v

a	b	c	d	t	u	v
a, R2	b	c	d, R1			R3

寄存器选择函数的设计

- 函数*getReg*返回保存x = y *op* z的x值的场所L
  - 如果名字y在R中，这个R不含其它名字的值,并且在执行x = y *op* z后y不再有下次引用，那么返回这个R作为L
  - 否则，如果有的话，返回一个空闲寄存器
  - 否则，如果x在块中有下次引用，或者*op*是必须用寄存器的算符，那么找一个已被占用的寄存器R(可能产ST M, R 指令，并修改 M的描述)
  - 否则，如果x在基本块中不再引用，或者找不到适当的被占用寄存器，选择x的内存单元作为L

《编译原理和技术》

目标代码生成 I

谢谢！

《编译原理和技术》

目标代码生成 II

——寄存器分配

中科大计算机学院
李 诚

2022-11-21

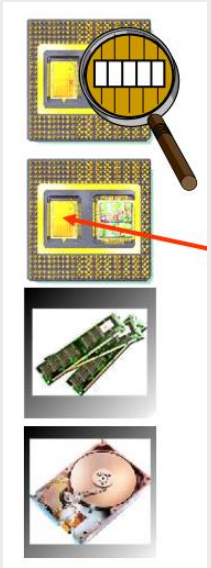
程序如何在计算机上执行的?

寄存器

缓存

内存

磁盘



计算单元
(龙芯CPU)

存储单元
(长鑫内存、国科微固态硬盘)

程序如何在计算机上执行的?

		读取速度(指令周期)	容量(字节)
寄存器		1	8K
缓存		3	40M
内存		20	512G
磁盘		5M	10T

计算单元
(龙芯CPU)

存储单元
(长鑫内存、国科微固态硬盘)

程序如何在计算机上执行的?

		读取速度(指令周期)	容量(字节)
寄存器		1	8K
缓存		3	40M
内存		20	512G
磁盘	代码 数据	5M	10T

计算单元
(龙芯CPU)

存储单元
(长鑫内存、国科微固态硬盘)

程序如何在计算机上执行的?

代码

计算单元
(龙芯CPU)

		读取速度(指令周期)	容量(字节)
寄存器		1	8K
缓存	代码	3	40M
内存	代码	20	512G
磁盘	代码 数据	5M	10T

存储单元
(长鑫内存、国科微固态硬盘)

程序如何在计算机上执行的?

代码

			读取速度(指令周期)	容量(字节)
寄存器		数据	1	8K
缓存	代码	数据	3	40M
内存	代码	数据	20	512G
磁盘	代码	数据	5M	10T

计算单元
(龙芯CPU)

存储单元
(长鑫内存、国科微固态硬盘)

寄存器资源管理十分重要

- 寄存器容量和个数十分有限
  - 受限于电源功耗等因素
- 为了编程简单，高级语言假设可以使用无限多个寄存器

架构	32位
ARM	15
Intel x86	8
MIPS	32
RISC-V	16/32
LoongArch	32+32

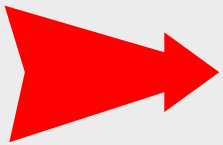
龙芯

程序片段:

```
t1 = 0
t2 = t1- 5
t3 = t1 + t2
t4 = t2 * t3
t5 = t3 – t4
… …
```

t1
t2
.
.
.
t∞

竞争寄存器的使用



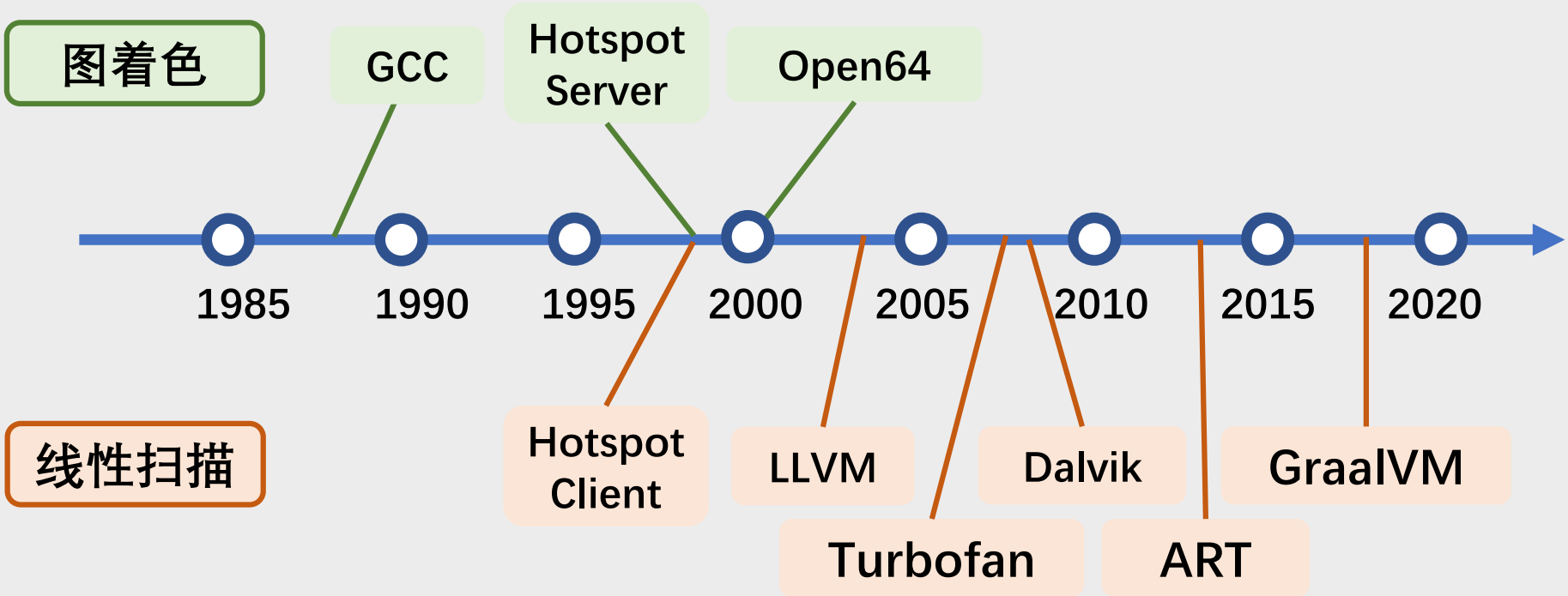
R1
R2
.
.
.
R32

寄存器分配

- **任务目标：**
  - 在不改变程序行为的前提下，将同一个寄存器分配给多个变量
- **约束条件：**
  - 同一时刻，一个寄存器只能被一个变量占用
  - 寄存器占满后，新的使用申请将选择一个寄存器，移出其所存储的变量，放回内存（成为**Spill**，产生较大的开销）
  - 换入换出寄存器的开销尽可能小

寄存器分配算法的演进

分配效果非常好、但运行时间长、常见于传统编译器。



算法运行时间很短，分配效果接近图着色、常见于现代编译器。

举例——寄存器线性扫描分配

```
        e = d + a
        f = b + c
        f = f + b
        if e == 0 goto _L0
        d = e + f
        goto _L1
_L0: d = e – f
_L1: g = d
```

a

b

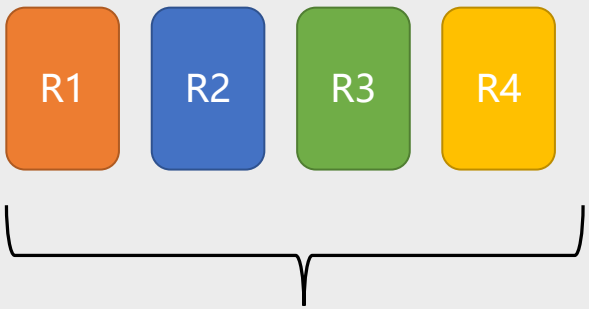
c

d

e

f

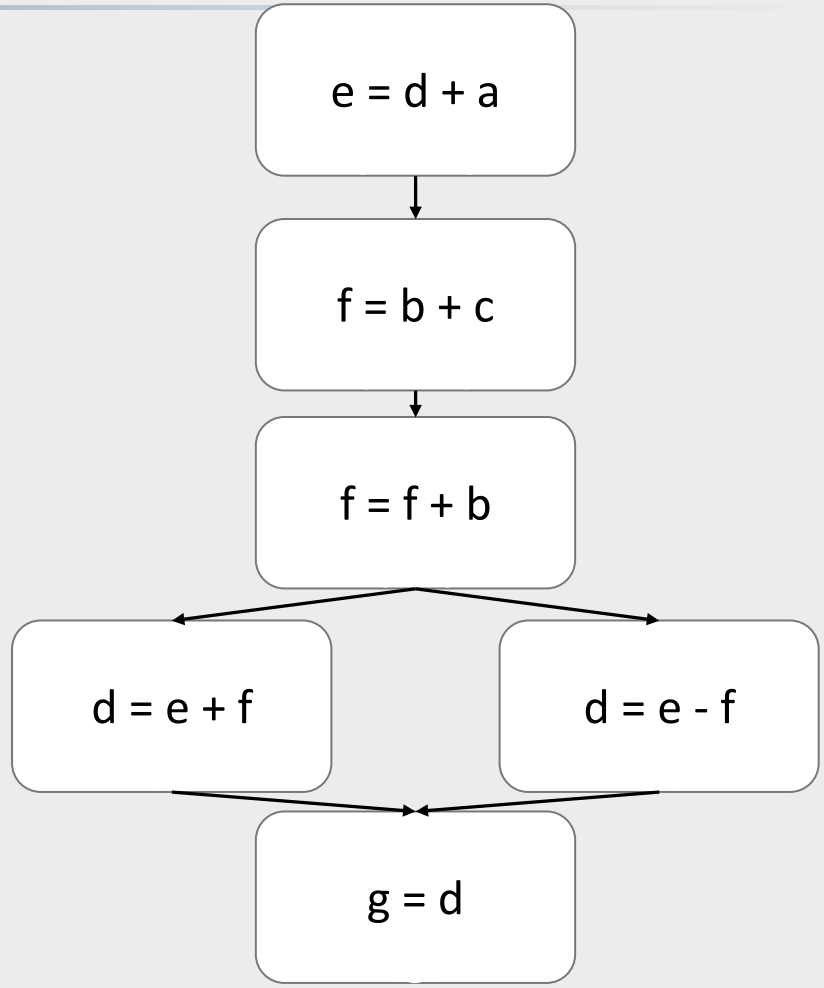
g



仅有4个可用的寄存器

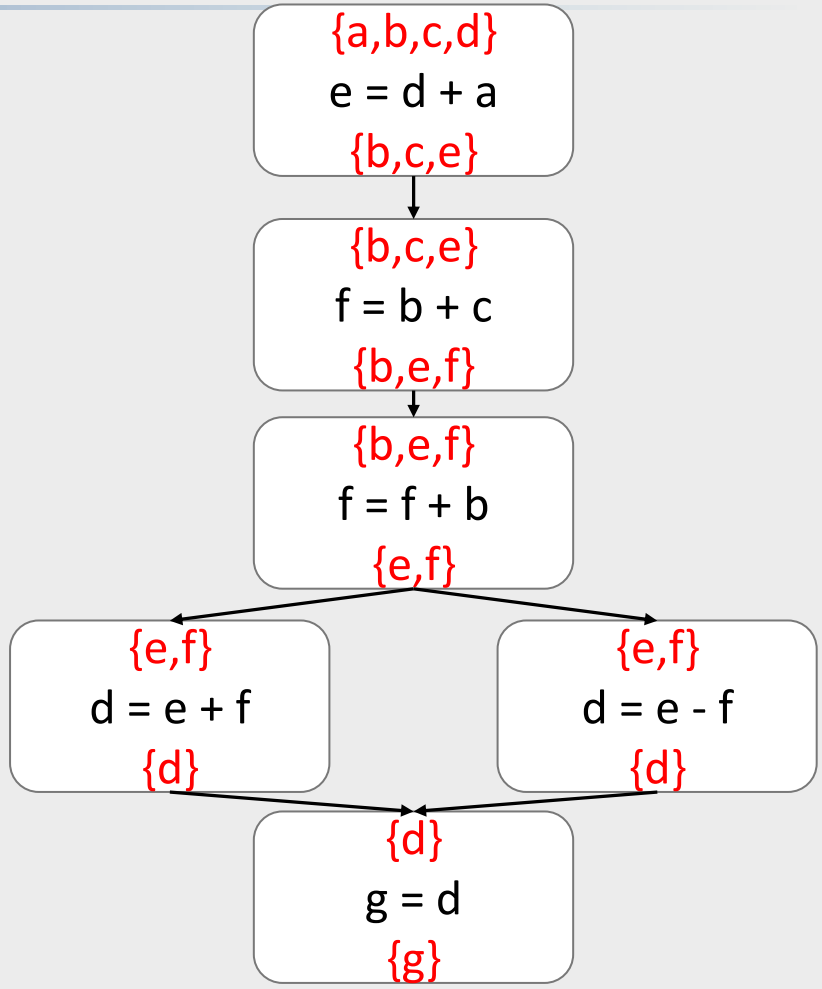
寄存器线性扫描分配——变量存活区间

```
        e = d + a
        f = b + c
        f = f + b
        if e == 0 goto _L0
        d = e + f
        goto _L1
_L0: d = e – f
_L1: g = d
```

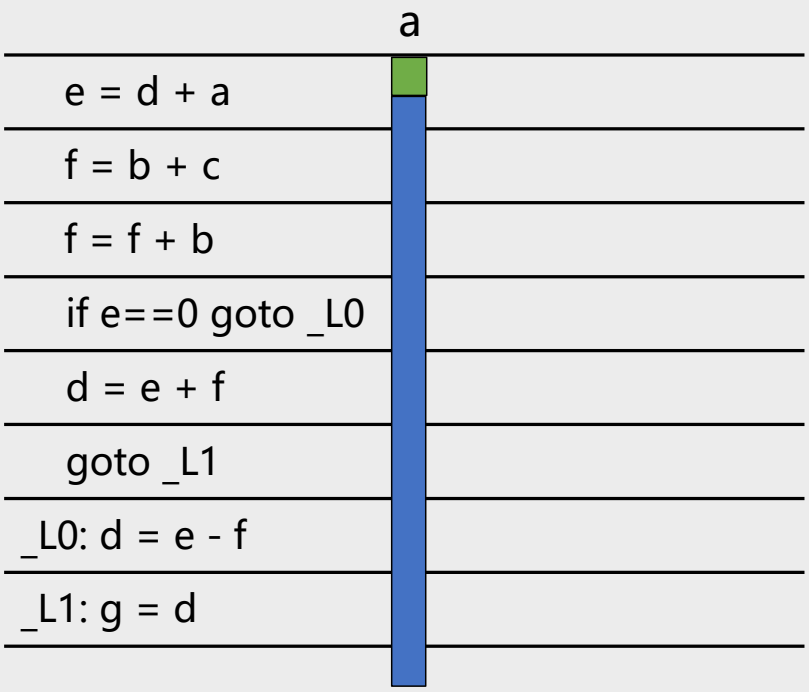


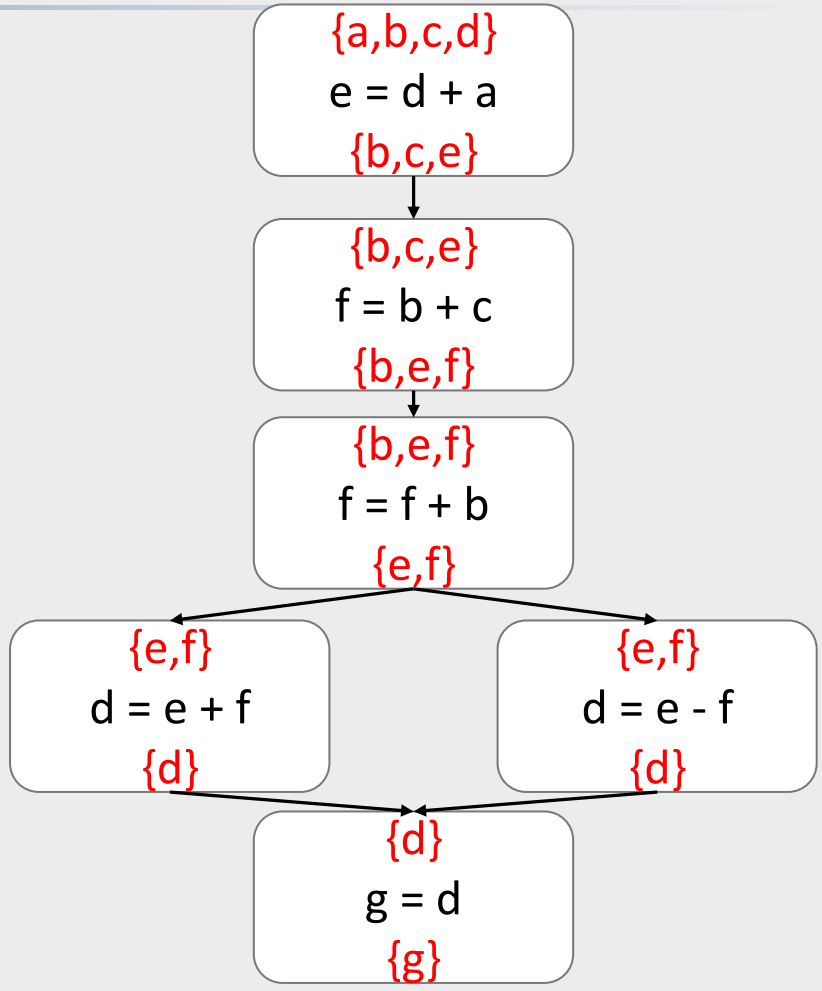
寄存器线性扫描分配——变量存活区间

```
        e = d + a
        f = b + c
        f = f + b
        if e == 0 goto _L0
        d = e + f
        goto _L1
_L0: d = e – f
_L1: g = d
```

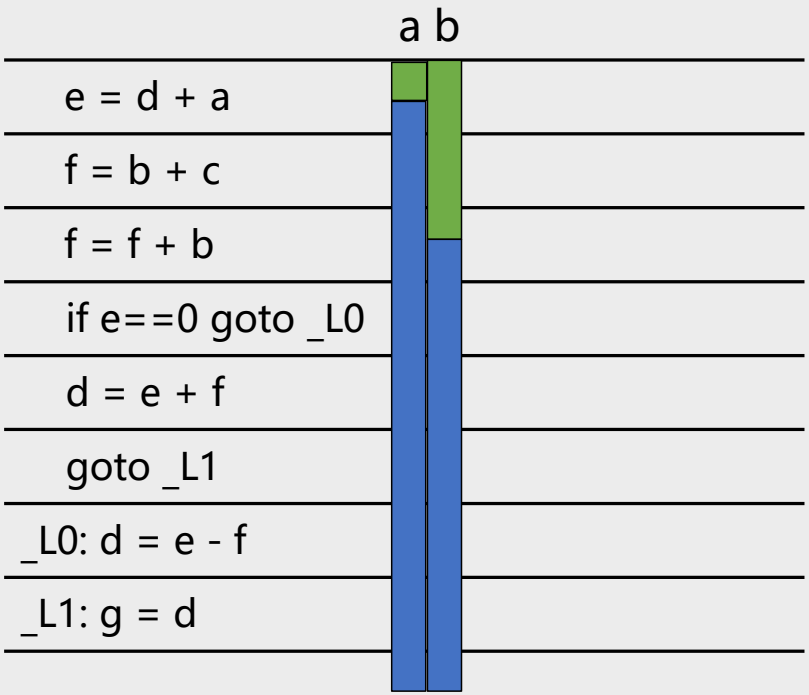


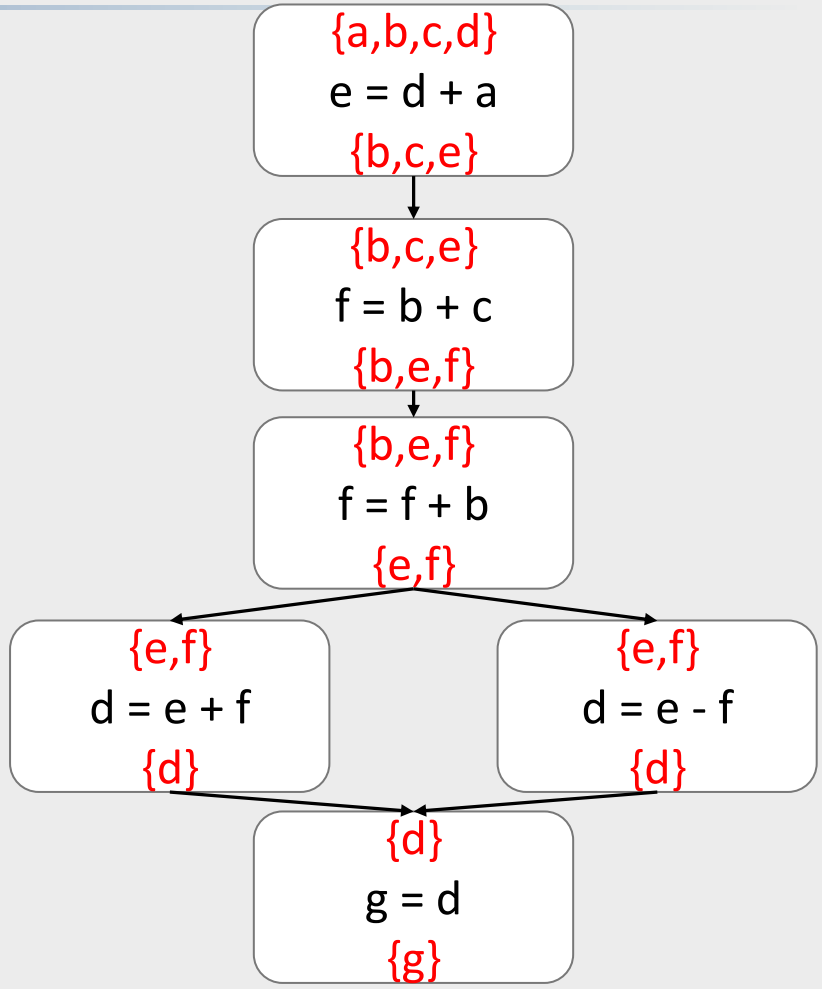
寄存器线性扫描分配——变量存活区间



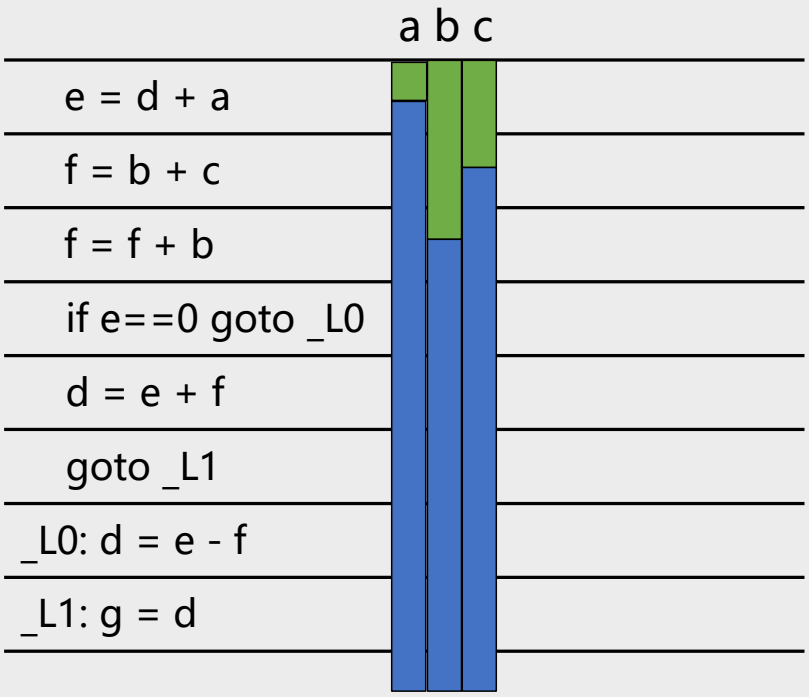


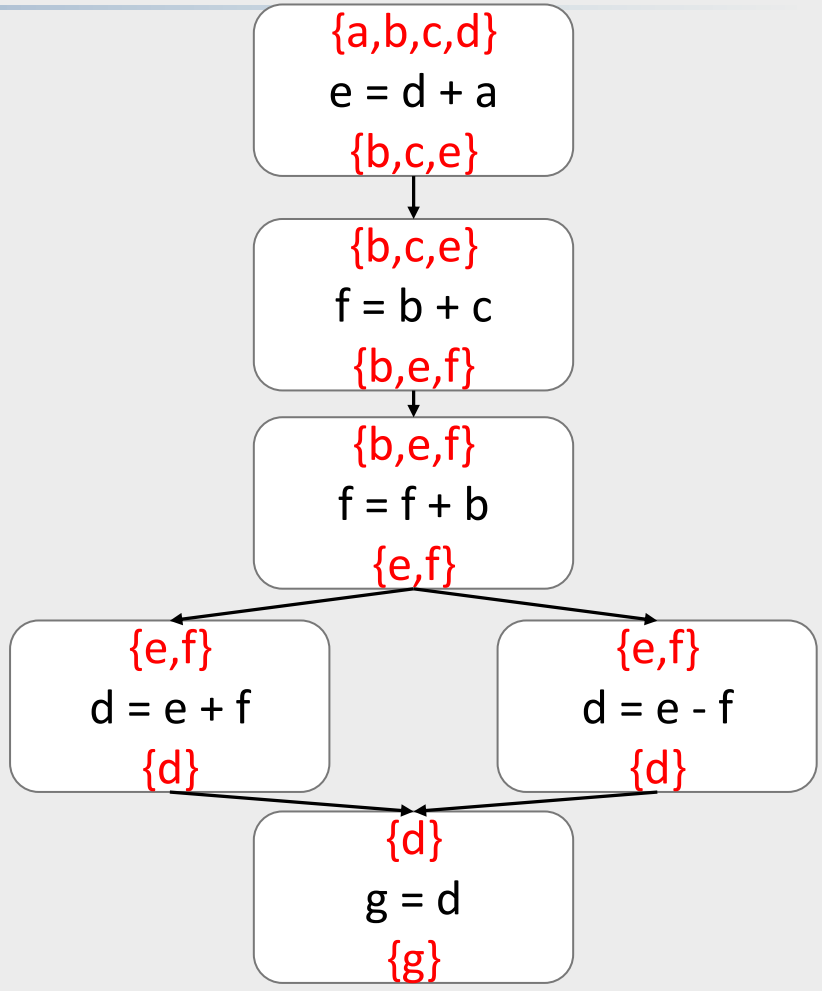
寄存器线性扫描分配——变量存活区间



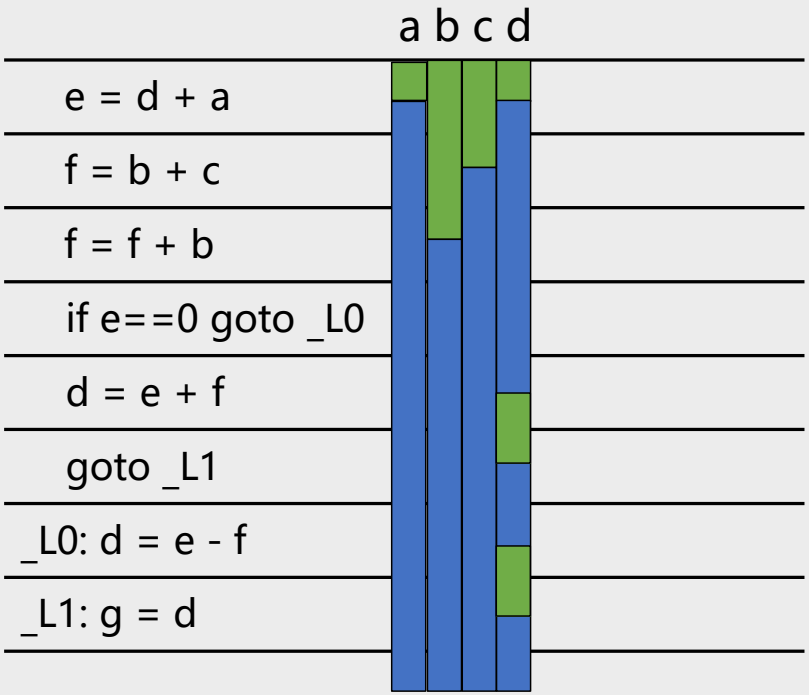


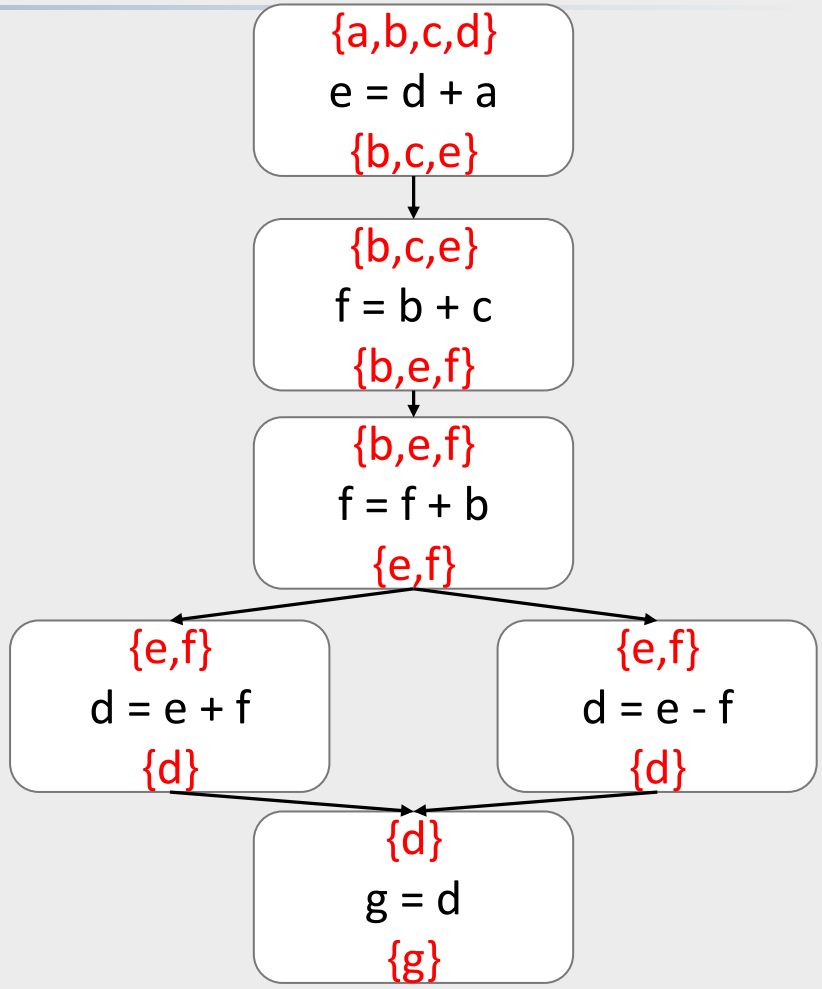
寄存器线性扫描分配——变量存活区间



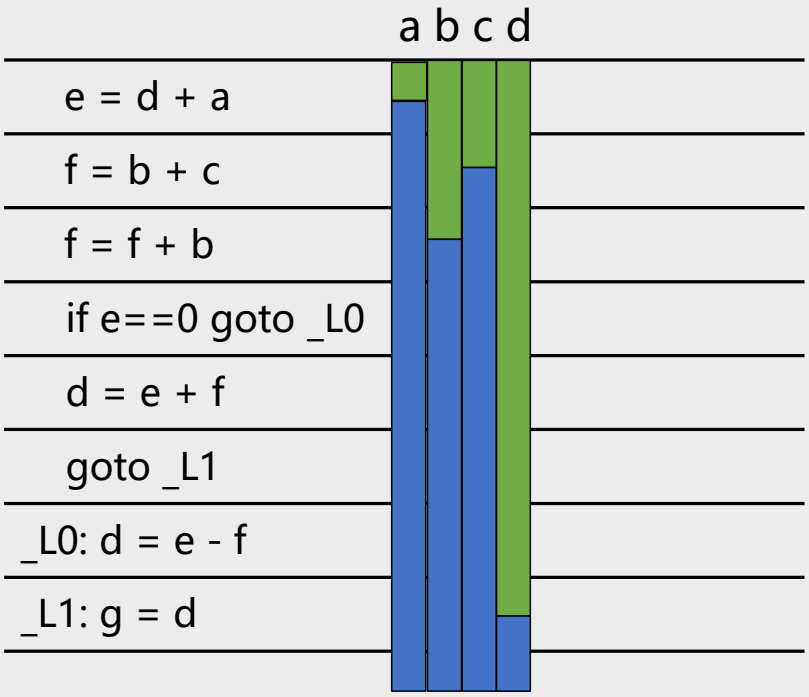


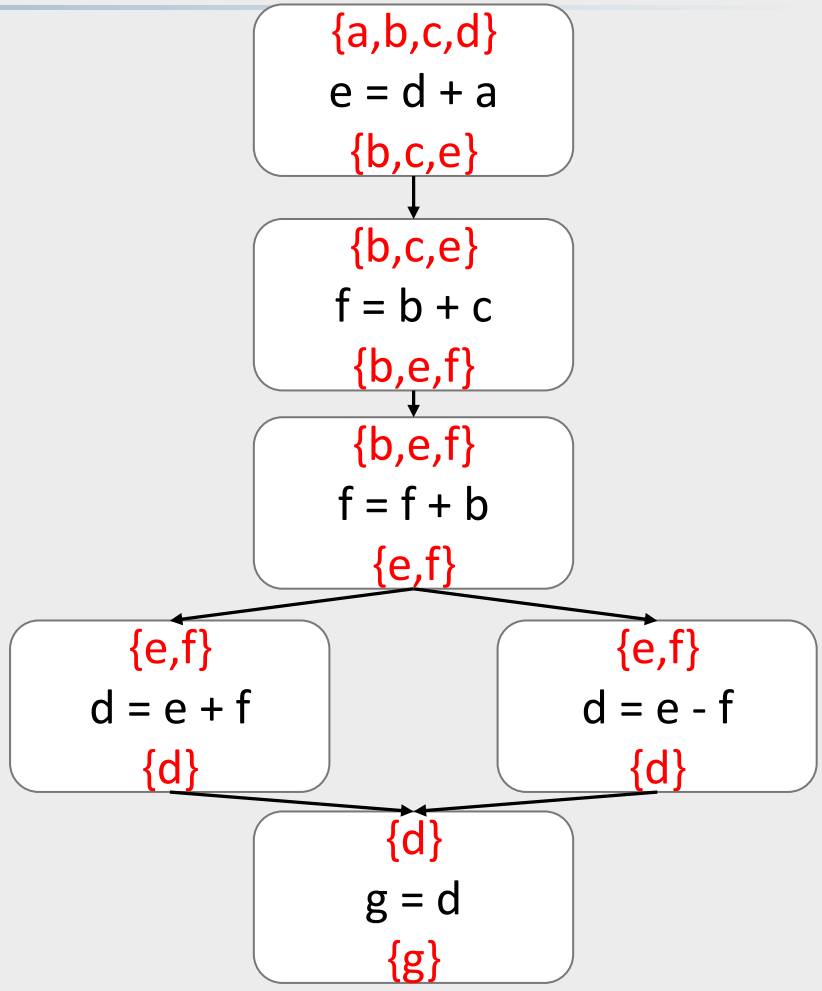
寄存器线性扫描分配——变量存活区间



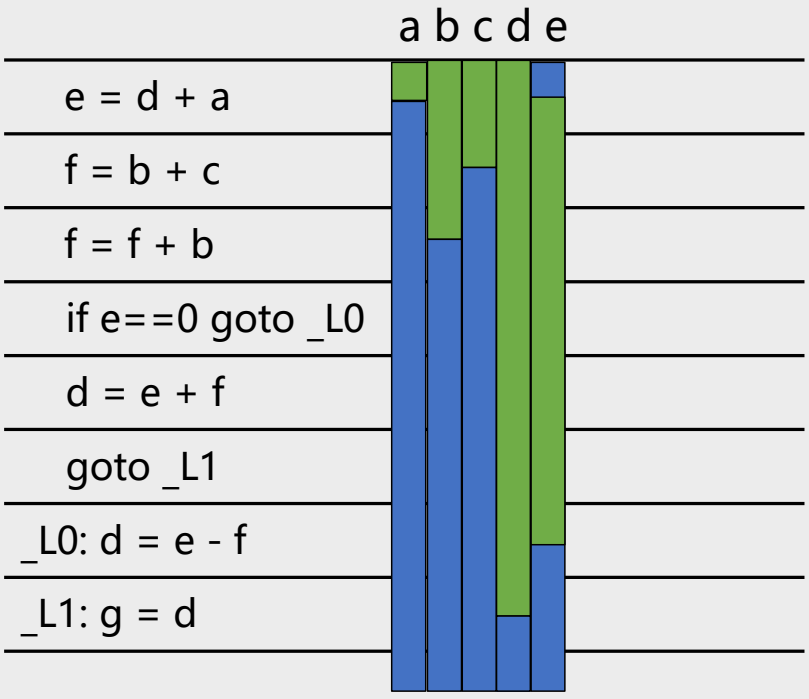


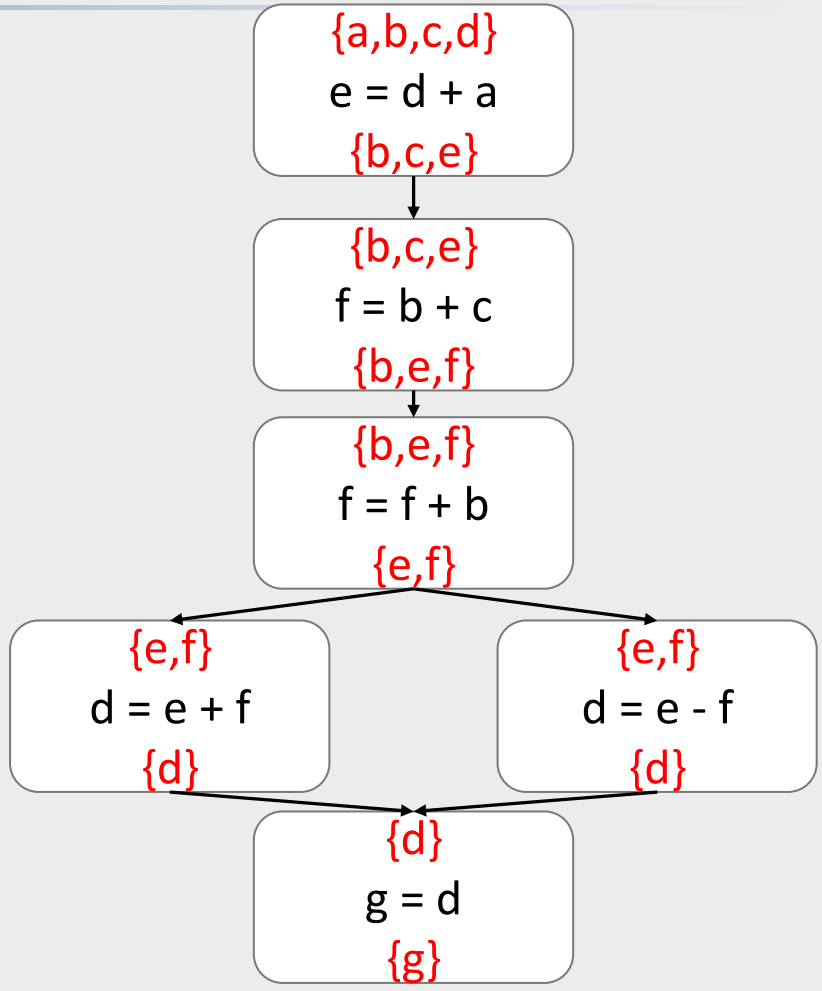
寄存器线性扫描分配——变量存活区间



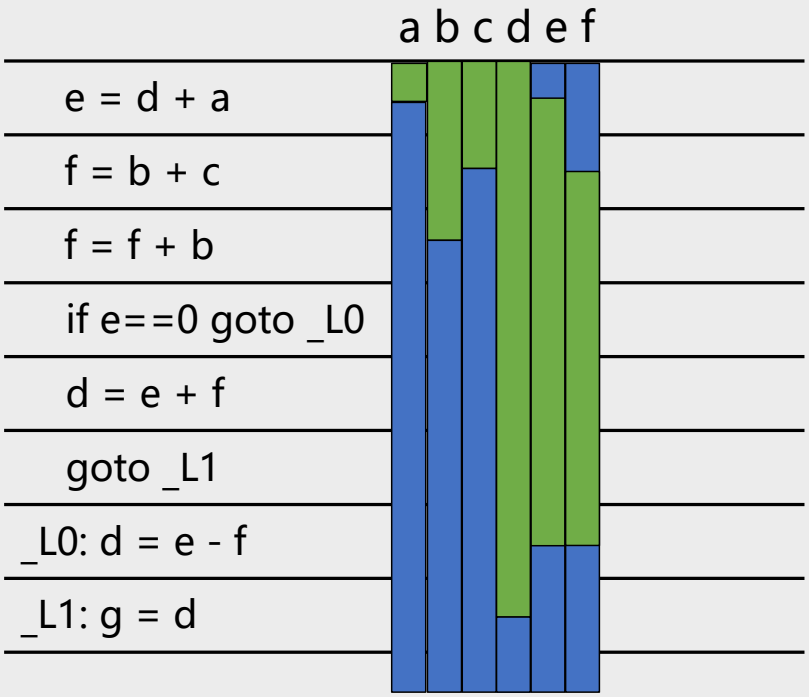


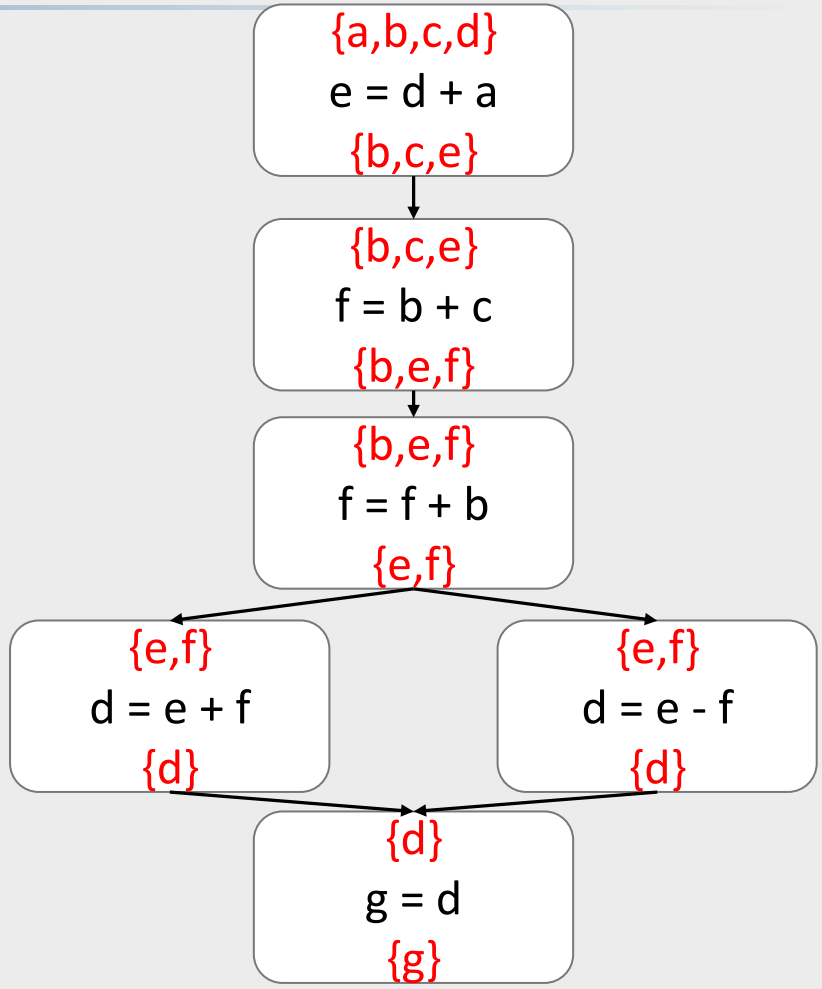
寄存器线性扫描分配——变量存活区间



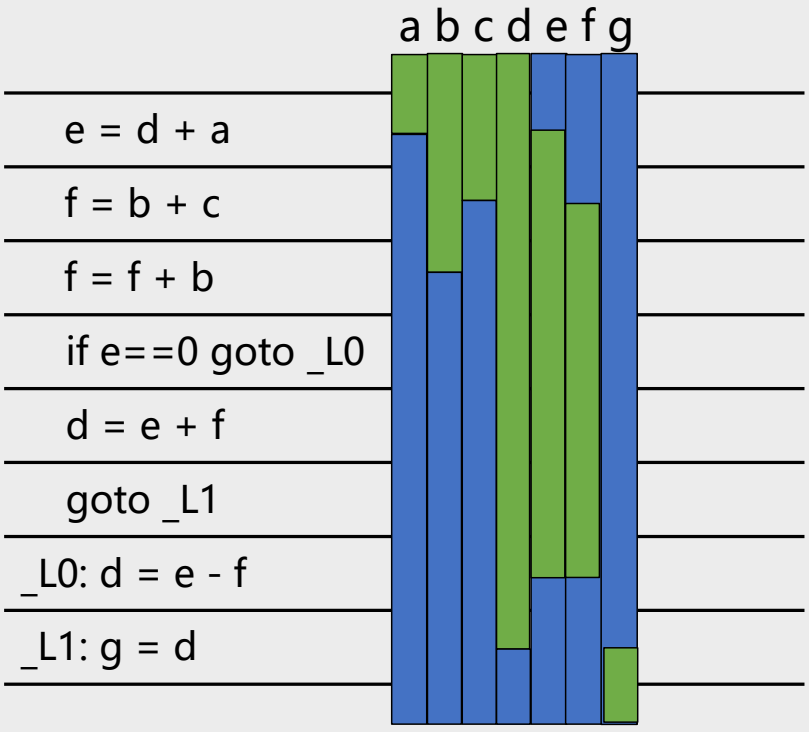


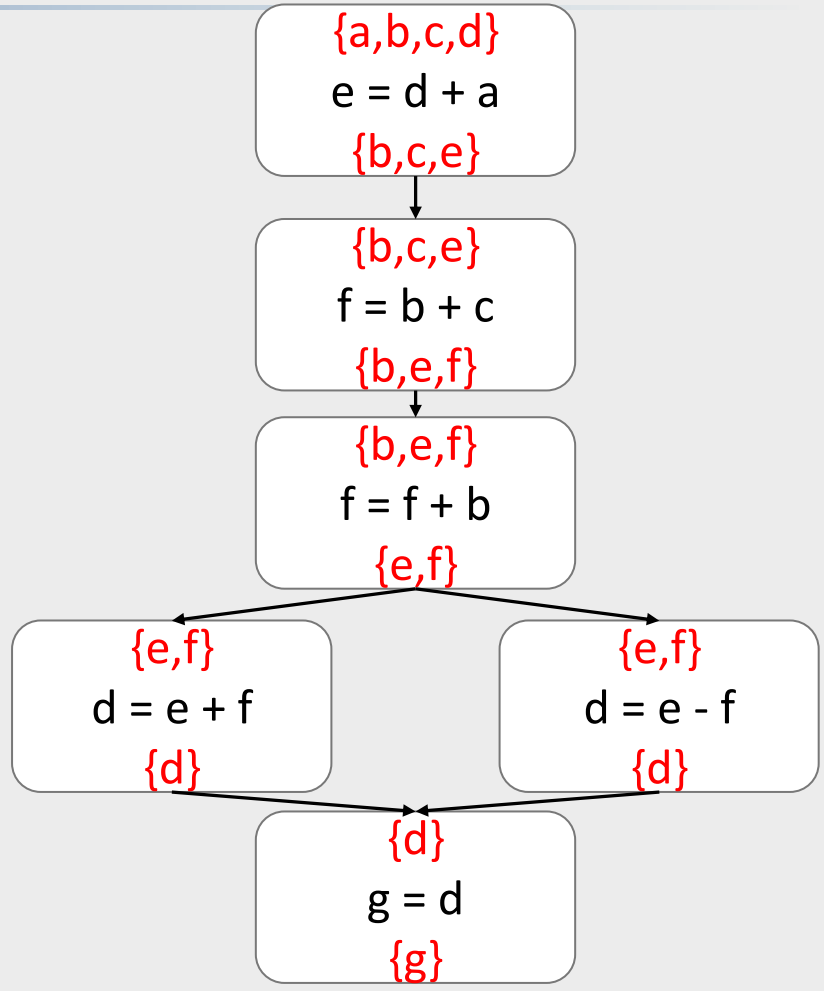
寄存器线性扫描分配——变量存活区间



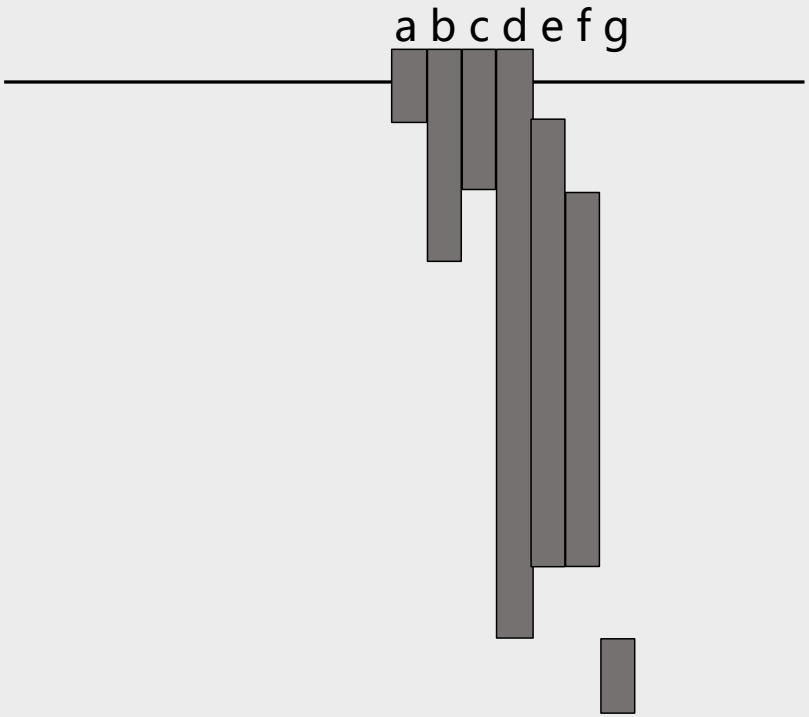


寄存器线性扫描分配——变量存活区间

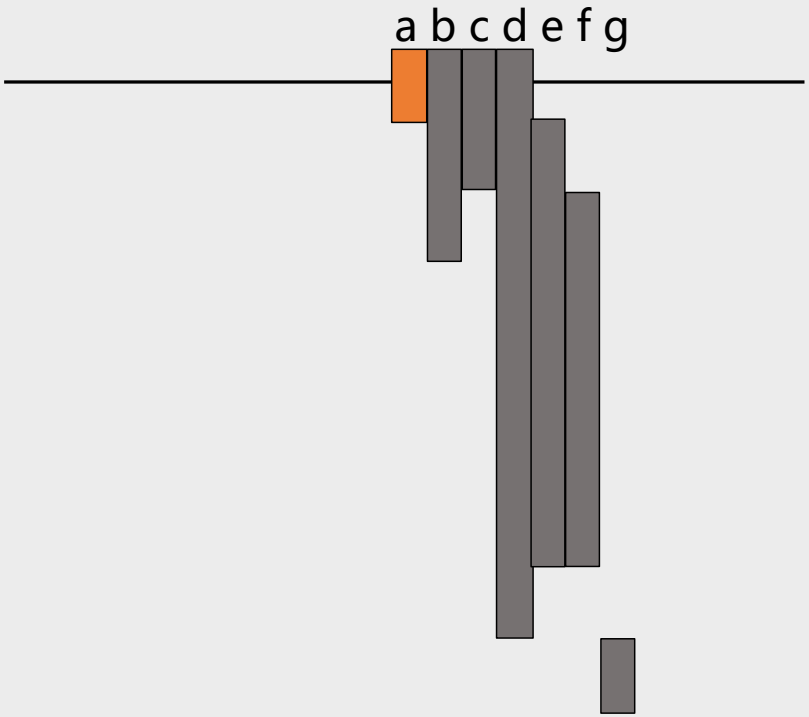




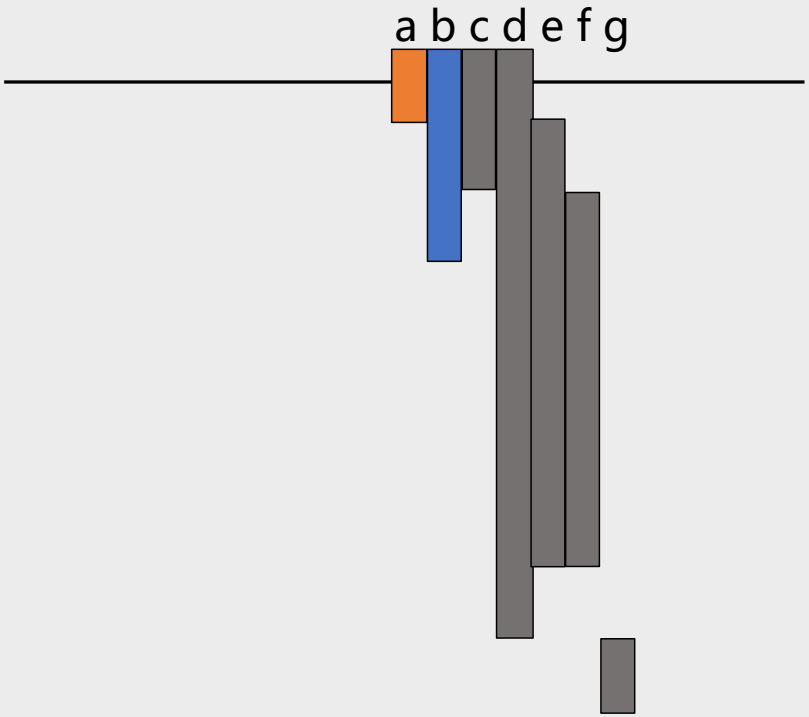
寄存器线性扫描——贪心分配策略



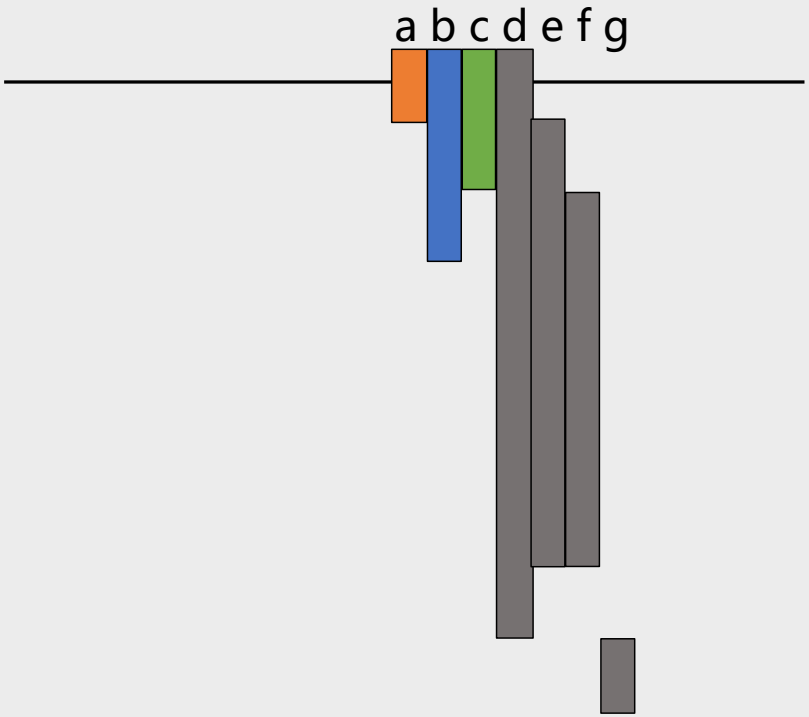
寄存器线性扫描——贪心分配策略



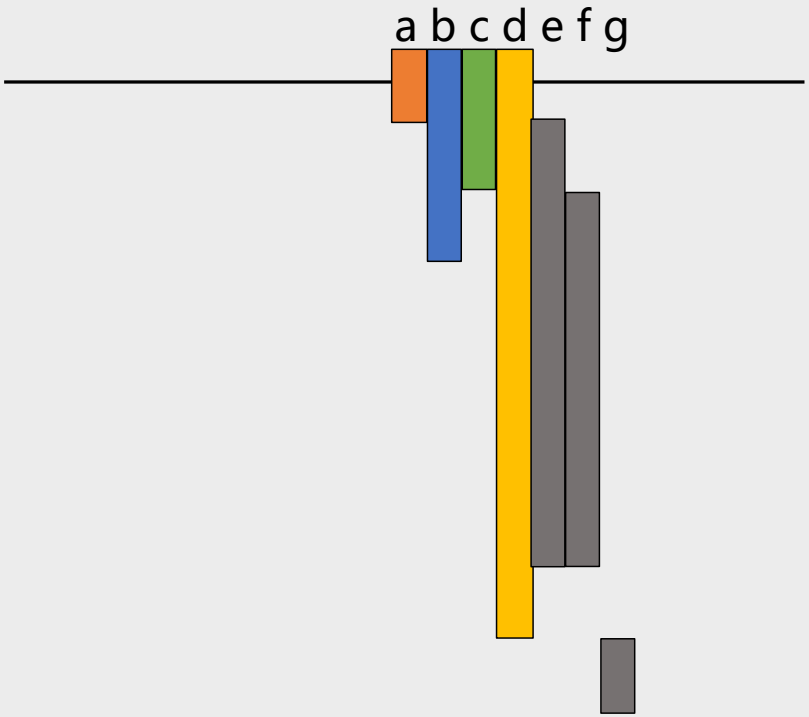
寄存器线性扫描——贪心分配策略



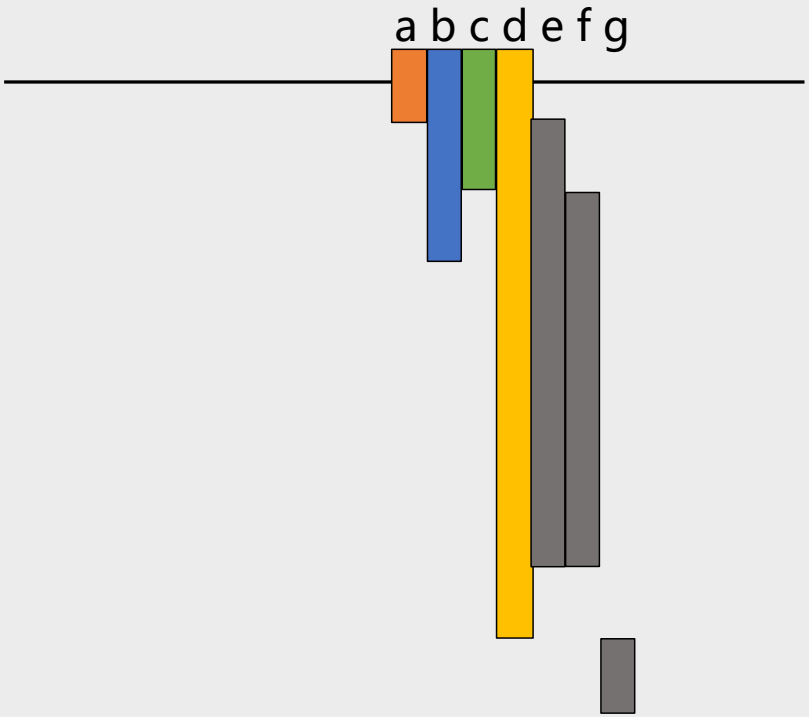
寄存器线性扫描——贪心分配策略



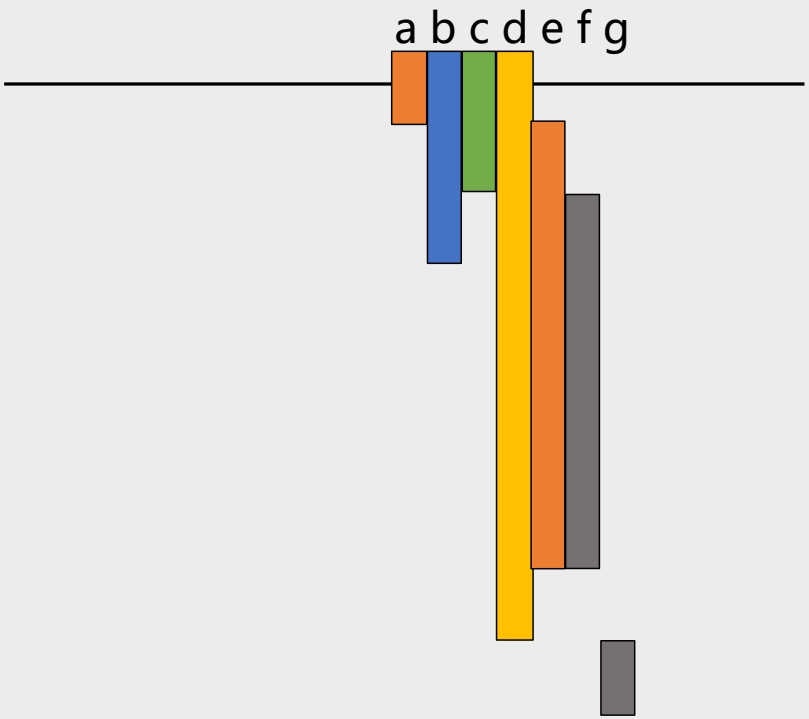
寄存器线性扫描——贪心分配策略



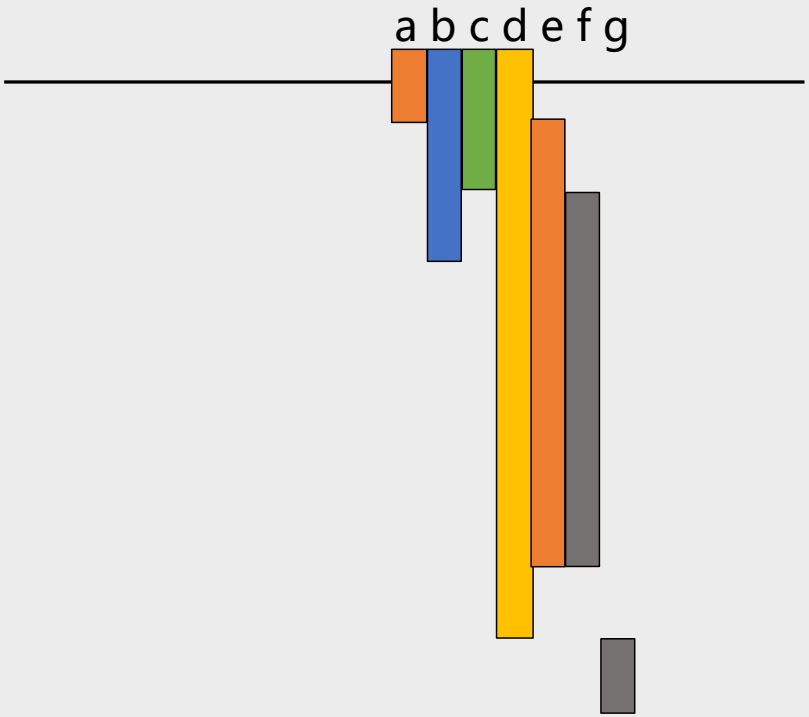
寄存器线性扫描——贪心分配策略



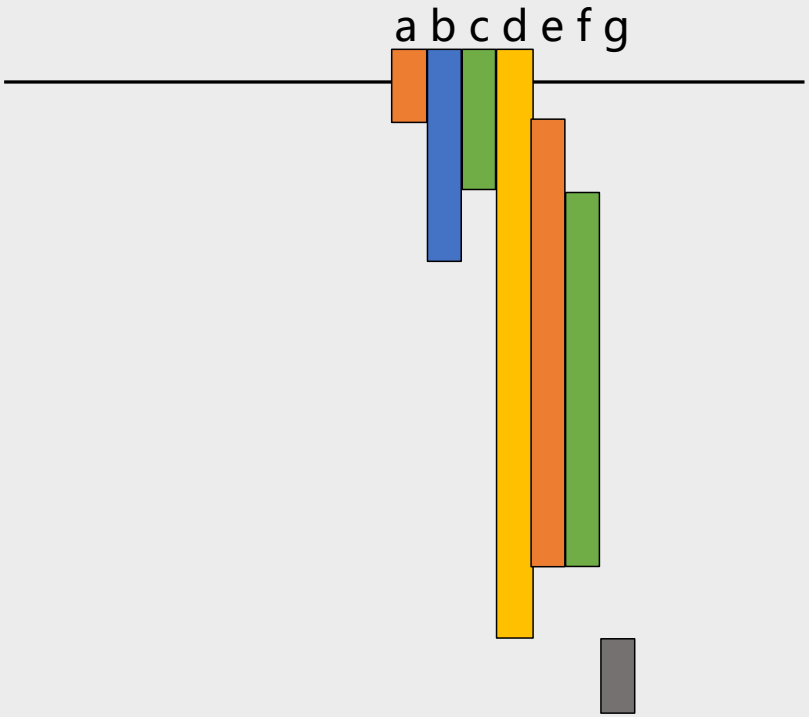
寄存器线性扫描——贪心分配策略



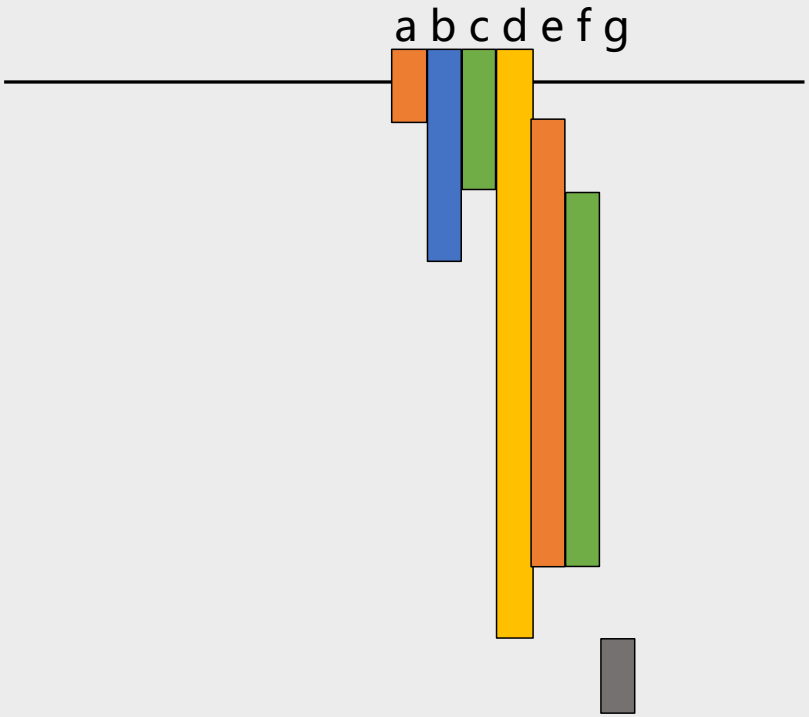
寄存器线性扫描——贪心分配策略



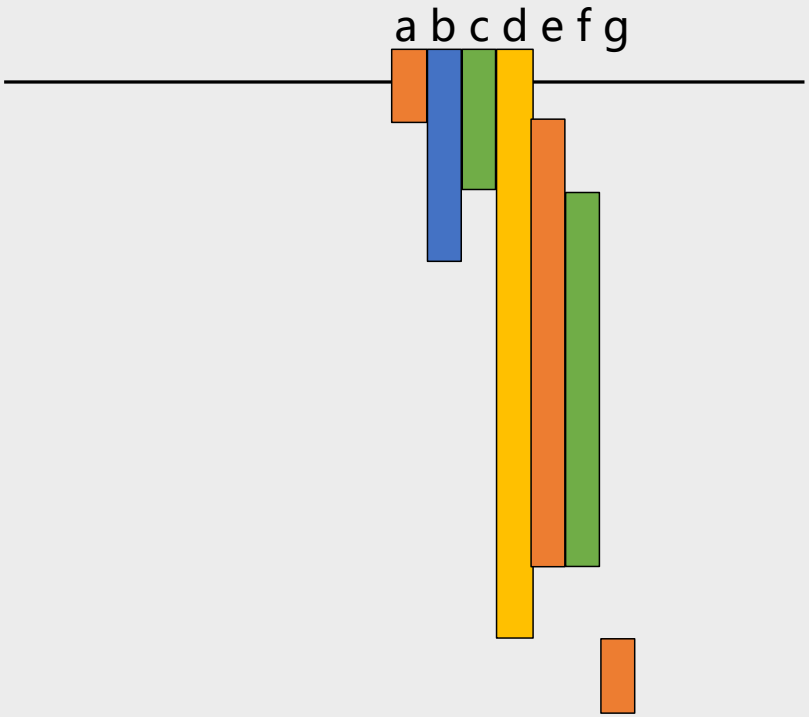
寄存器线性扫描——贪心分配策略



寄存器线性扫描——贪心分配策略



寄存器线性扫描——贪心分配策略



本节小结

- 寄存器是宝贵的计算机资源，需要合理利用和分配
- 寄存器分配主要有线性扫描和图着色两类算法
  - 前者比后者性能更好，应用更加广泛
- 线性扫描需要借助于变量存活区间的分析
  - 需要数据流分析的抽象

拓展与思考

- **问题：如何计算变量的活跃区间?**
  - 请预习数据流分析和活跃变量分析（第9章第9.2节）
- **延伸阅读：**
  - 线性扫描算法：
    - Linear Scan Register Allocation for the Java HotSpot™ Client Compiler
  - 图着色算法：
    - Register allocation & spilling via graph coloring

《编译原理和技术》

目标代码生成 II

——寄存器分配

谢谢！

《编译原理和技术》

循环

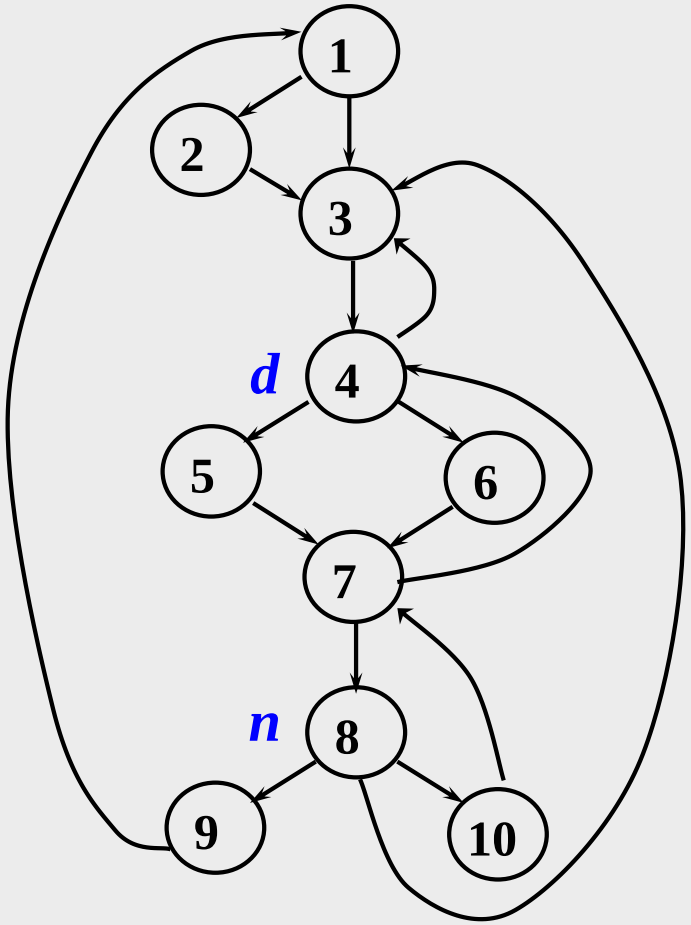
中科大计算机学院
李 诚
2022-11-21

本节提纲

- **标识循环并对循环专门处理的重要性**
  - 程序执行的大部分时间消耗在循环上，改进循环性能的优化会对程序执行产生显著影响
  - 循环也会影响程序分析的运行时间
  - 循环对数据流分析带来了诸多挑战
- **本节主要内容**
  - 支配信息相关定义和计算方法
  - 静态单赋值形式转换及支配信息的使用
  - 循环的其他定义

支配结点(Dominators)

- d是n的支配结点:
  - 若从初始结点起，每条到达*n*的路径都要经过*d*，写成*d dom n*
- 结点是它本身的支配结点
- 循环的入口是循环中所有结点的支配结点

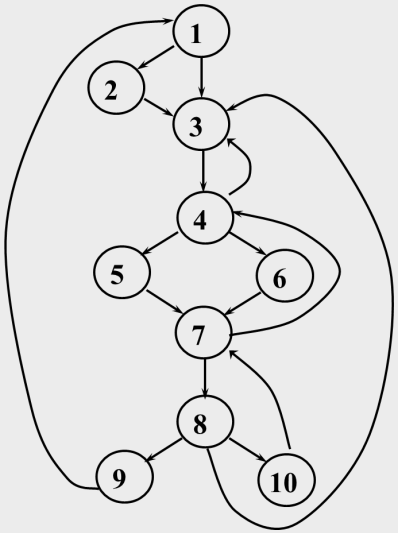


支配结点(Dominators)的数据流分析

- **目标：**寻找支配结点
- **输入：**流图G，G的结点集N
- **输出：**每个N中的结点n，支配n的所有结点的集合D(n)
- **边界条件：**OUT[Entry] = {Entry}, 初始化：OUT[B] = N
- **正向分析**

```
while(某个OUT值变化){
        for(除Entry外的块B){
                IN[B] = ∩ P是B的前驱 (OUT[P]) //交集
                OUT[B]  =  {B} ∪ IN[B]
        }
}
```

支配结点(Dominators)-举例

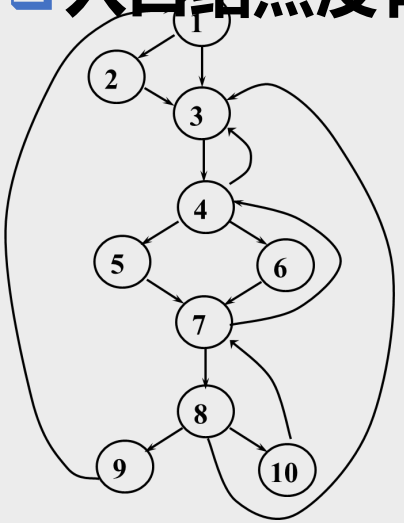


IN[B] = ∩ P是B的前驱 (OUT[P])
OUT[B] = {B} ∪ IN[B]

结点	计算过程	支配n的结点	支配对象
1	{1}	1	1-10
2	{2} U D(1)	1, 2	2
3	{3} U (D(1) ∩ D(2) ∩ D{4} ∩ D{8})	1, 3	3-10
4	{4} U (D(3) ∩ D(7))	1, 3, 4	4-10
5	{5} U D(4)	1, 3, 4, 5	5
6	{6} U D(4)	1, 3, 4, 6	6
7	{7} U (D(5) ∩ D(6) ∩ D(10))	1, 3, 4, 7	7-10
8	{8} U D(7)	1, 3, 4, 7, 8	8-10
9	{9} U D(8)	1, 3, 4, 7, 8, 9	9
10	{10} U D(8)	1, 3, 4, 7, 8, 10	10

直接支配结点

- 流图中的结点n，严格支配n的结点集为D(n) – {n}，该集合中与n最接近的结点成为n的直接支配结点（immediate dominator），记为IDom(n)
- 入口结点没有直接支配结点

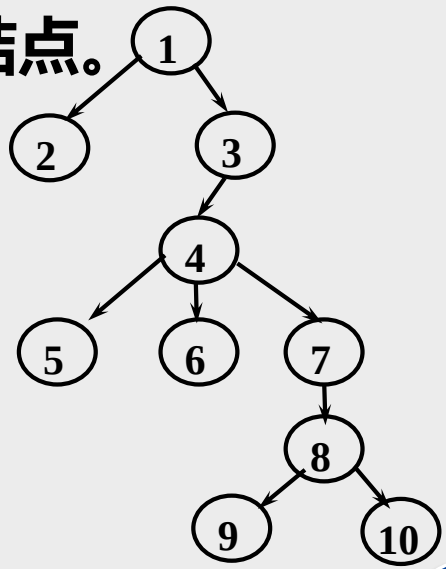


	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	B_{10}
D(n)	{1}	{1,2}	{1,3}	{1,3,4}	{1,3,4,5}	{1,3,4,6}	{1,3,4,7}	{1,3,4,7,8}	{1,3,4,7,8,9}	{1,3,4,7,8,10}
IDom(n)	-	1	1	3	4	4	4	7	8	8

支配结点树

- 编码了流图支配性信息的树，叫做支配结点树(dominator tree)。
- 入口结点为根结点，且每个结点n只支配它在树中的后代结点。
- 对于非入口结点n，IDom(n)是其父结点，D(n)中的各个结点，是在该支配树中，根结点到n之间的路径上的那些结点。

	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	B_9	B_{10}
D(n)	{1}	{1,2}	{1,3}	{1,3,4}	{1,3,4,5}	{1,3,4,6}	{1,3,4,7}	{1,3,4,7,8}	{1,3,4,7,8,9}	{1,3,4,7,8,10}
IDom(n)	-	1	1	3	4	4	4	7	8	8



支配边界

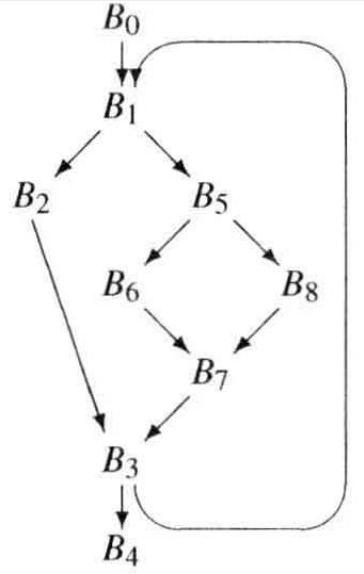
- **流图中结点n，其支配边界(dominator frontier)，记为DF(n)，每一个在DF(n)中的结点m，满足如下性质：**
  - n支配m的一个前驱结点，即$q \in preds(m) \wedge n \in D(q)$
  - n并不严格支配m，即$m \notin D(n) - \{n\}$

支配边界

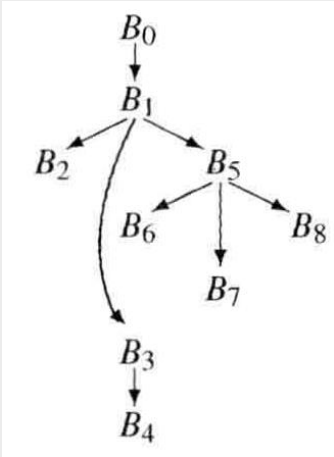
- **流图中结点n，其支配边界(dominator frontier)，记为DF(n)，每一个在DF(n)中的结点m，满足如下性质：**
  - n支配m的一个前驱结点，即$q \in preds(m) \wedge n \in D(q)$
  - n并不严格支配m，即$m \notin D(n) - \{n\}$
- **计算方法**

```
for all nodes n in a CFG (流图)
    DF(n) = ∅ //初始化

for all nodes n in that CFG
    for each predecessor p of n
        runner = p
        while runner != IDom(n)
            DF(runner) = DF(runner) U {n}
            runner = IDom(runner)
```

支配边界计算-举例



流图

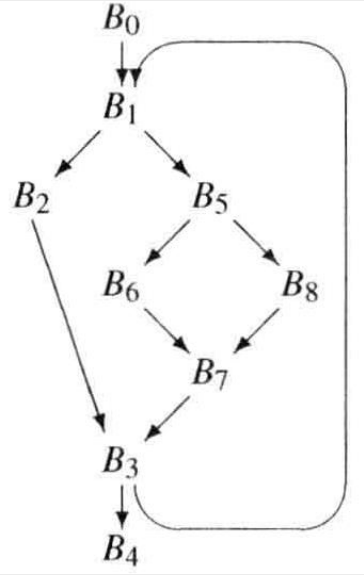


支配者树
IDom信息

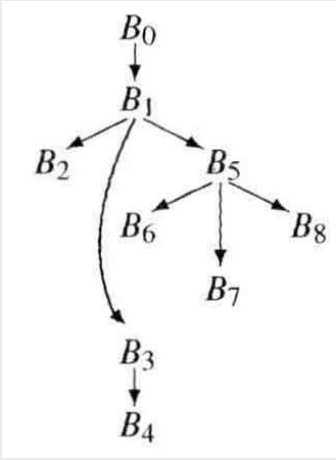
```
for all nodes n in a CFG (流图)
    DF(n) = ∅ //初始化

for all nodes n in that CFG
    for each predecessor p of n
        runner = p
        while runner != IDom(n)
            DF(runner) = DF(runner) U {n}
            runner = IDom(runner)
```

算法

支配边界计算-举例



流图



支配结点树
IDom信息

for all nodes *n* in a CFG (流图)
  DF(*n*) = ∅ //初始化

for all nodes *n* in that CFG
  for each predecessor *p* of *n*
    runner = *p*
    while *runner* != IDom(*n*)
      DF(*runner*) = DF(*runner*) U {*n*}
      runner = IDom(*runner*)

算法

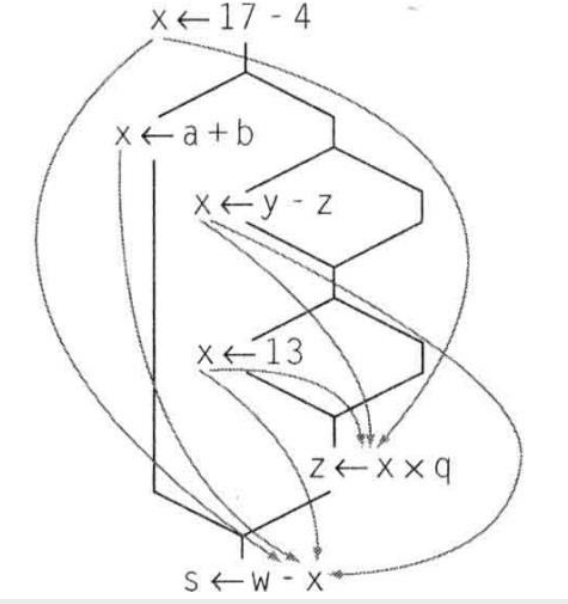
	步骤	B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8
DF(n)	初始	∅	∅	∅	∅	∅	∅	∅	∅	∅
	最终结果	∅	{B_1}	{B_3}	{B_1}	∅	{B_3}	{B_7}	{B_3}	{B_7}

本节提纲

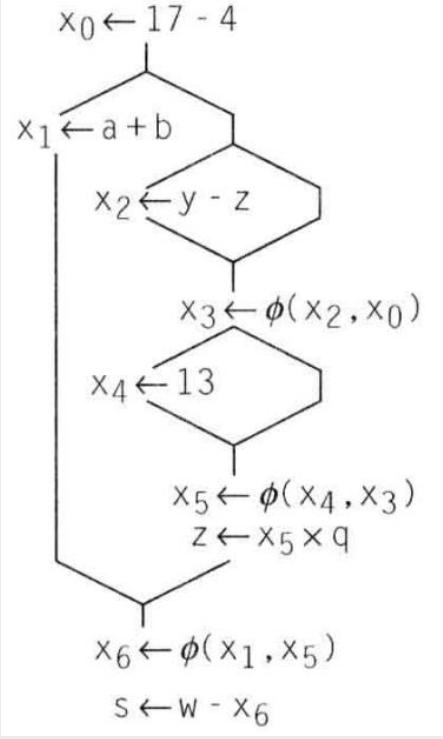
- 标识循环并对循环专门处理的重要性
  - 程序执行的大部分时间消耗在循环上，改进循环性能的优化会对程序执行产生显著影响
  - 循环也会影响程序分析的运行时间
  - 循环对数据流分析带来了诸多挑战
- 本节主要内容
  - 支配信息相关定义和计算方法
  - 静态单赋值形式转换及支配信息的使用
  - 循环的其他定义

静态单赋值形式

- SSA格式的IR可以同时对数据流和控制流进行编码



原始代码
箭头代表了定值可达信息



将x转换为SSA格式之后

静态单赋值形式转换

- **静态单赋值形式的两条必要规则**
  - 过程中每个定义都有唯一名字
  - 每个使用处都引用了一个定义
- **因此，需要在汇合处将不同路径上的静态单赋值形式名调和为一个名字！具体有两个步骤**
  - 插入ϕ指令或者函数。如果基本块B定义了y，则要求在DF(B)集合中包含的每个结点起始处插入$y \leftarrow \phi(y, y)$
  - 重命名。

静态单赋值形式的优化

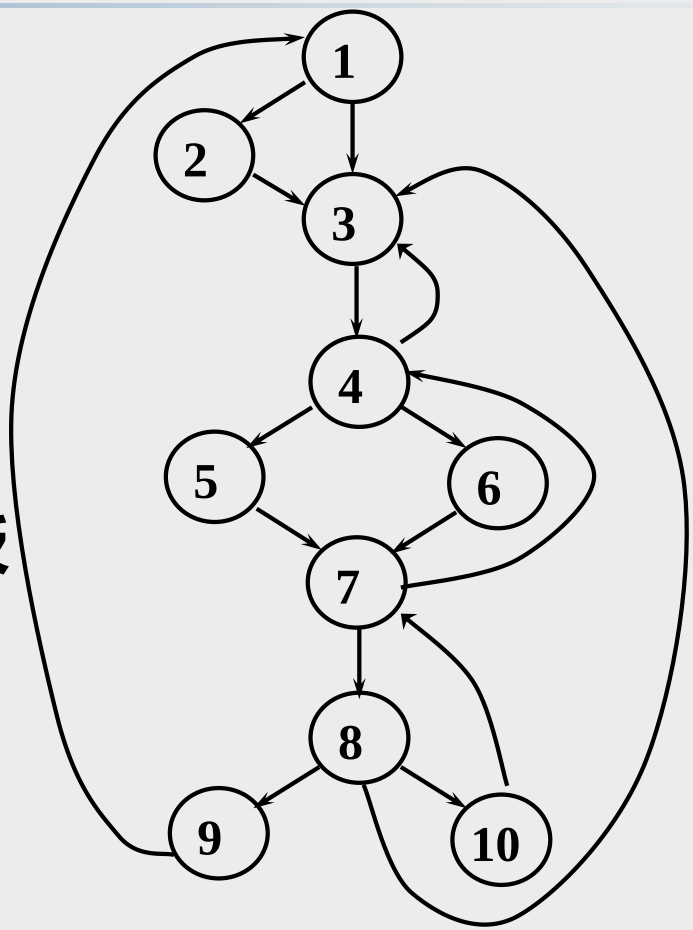
- 问题：引入了非必要的phi指令，如何删除?
- 可以延伸阅读
  - 第9.3.3章，《编译器设计-第2版》

本节提纲

- 标识循环并对循环专门处理的重要性
  - 程序执行的大部分时间消耗在循环上，改进循环性能的优化会对程序执行产生显著影响
  - 循环也会影响程序分析的运行时间
  - 循环对数据流分析带来了诸多挑战
- 本节主要内容
  - 支配信息相关定义和计算方法
  - 静态单赋值形式转换及支配信息的使用
  - 循环的其他定义

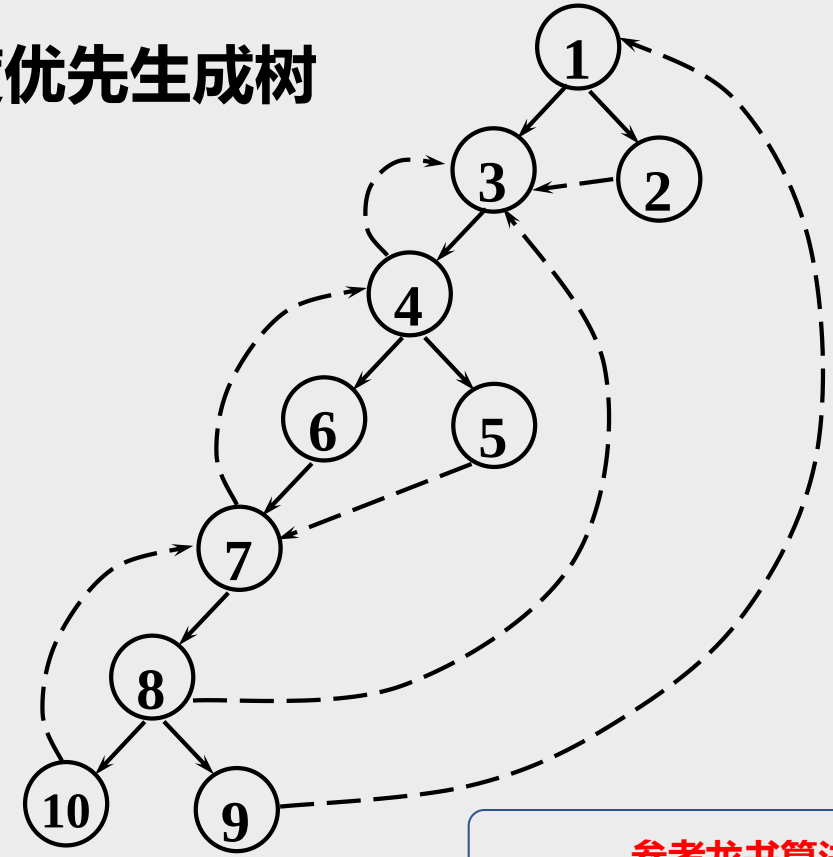
深度优先排序

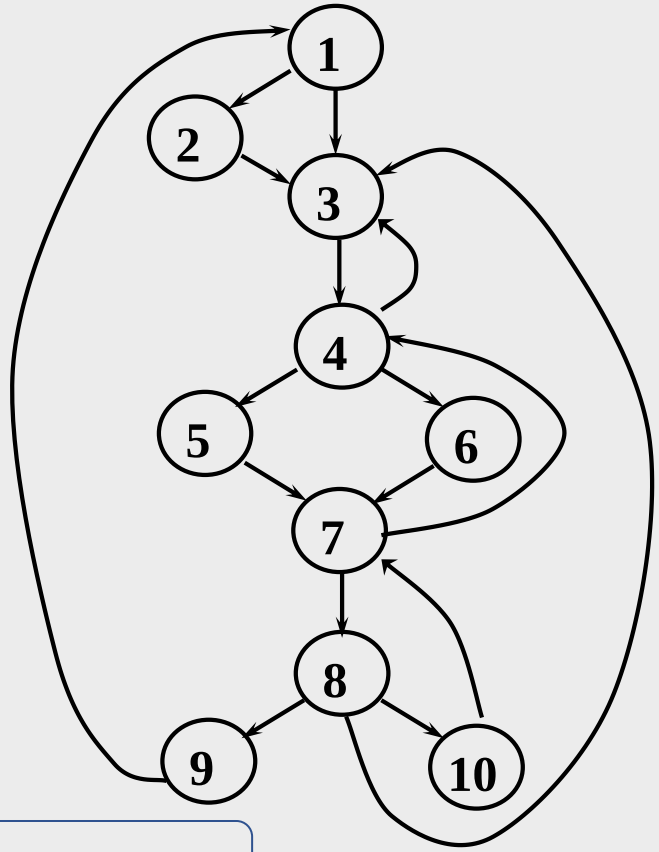
- 深度优先搜索
  - 先序遍历（先左后右）
    - 1,3,4,6,7,8,10,9,5,2
  - 后序遍历
    - 10,9,8,7,6,5,4,3,2,1
- 深度优先排序正好与后序遍历相反
  - 1,2,3,4,5,6,7,8,9,10
  - 先右后左



深度优先表示

- 深度优先生成树





参考龙书算法9.41

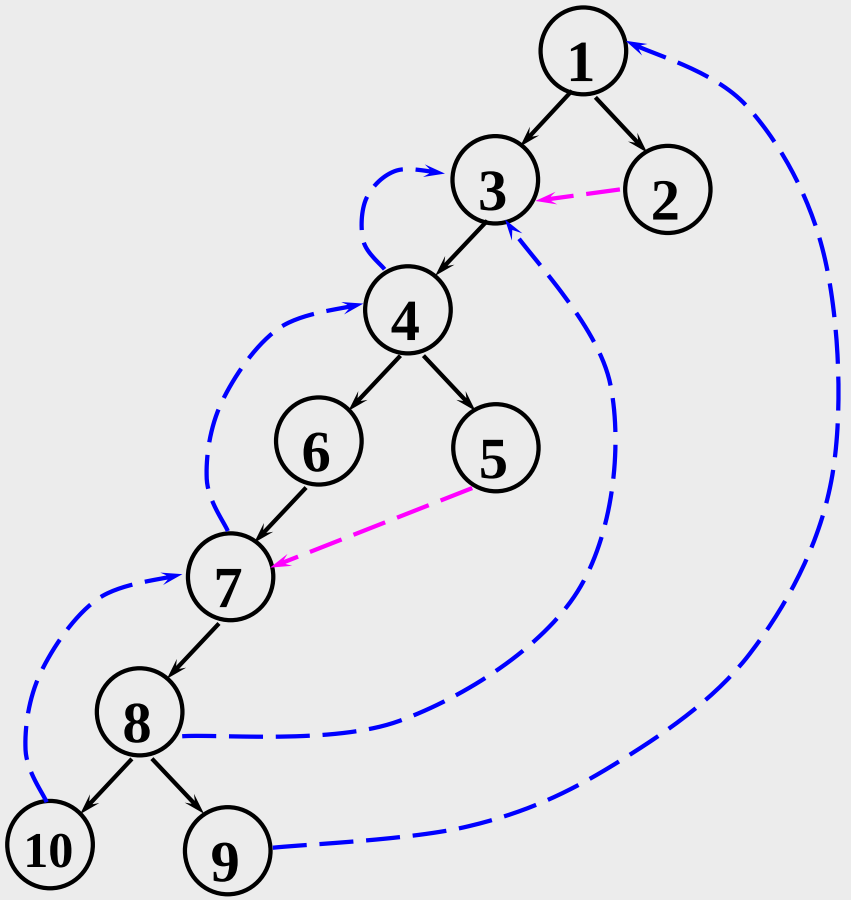
深度优先生成树中的边

- **前进边**
  - 深度优先生成树的边
- **后撤边**
  - 指向祖先节点

  4 → 3、7 → 4、10 → 7、

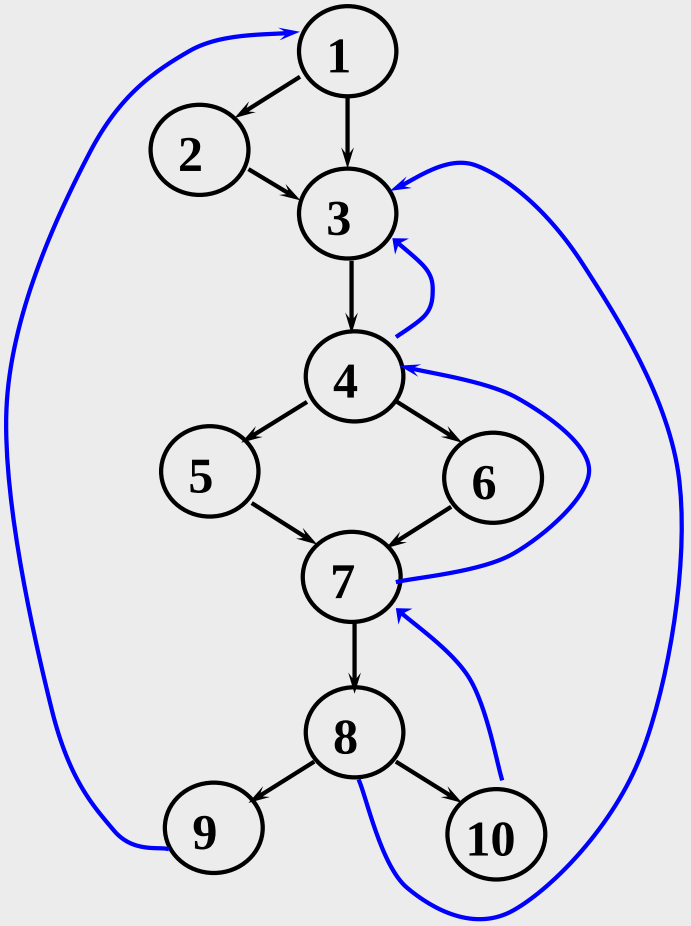
  8 → 3和9 → 1
- **交叉边**
  - 在树中互不为祖先

  2 → 3和5 → 7



回边和可归约性

- 回边
  - 如果有$a\ dom\ b$，那么边$b \rightarrow a$叫做回边



回边和可归约性

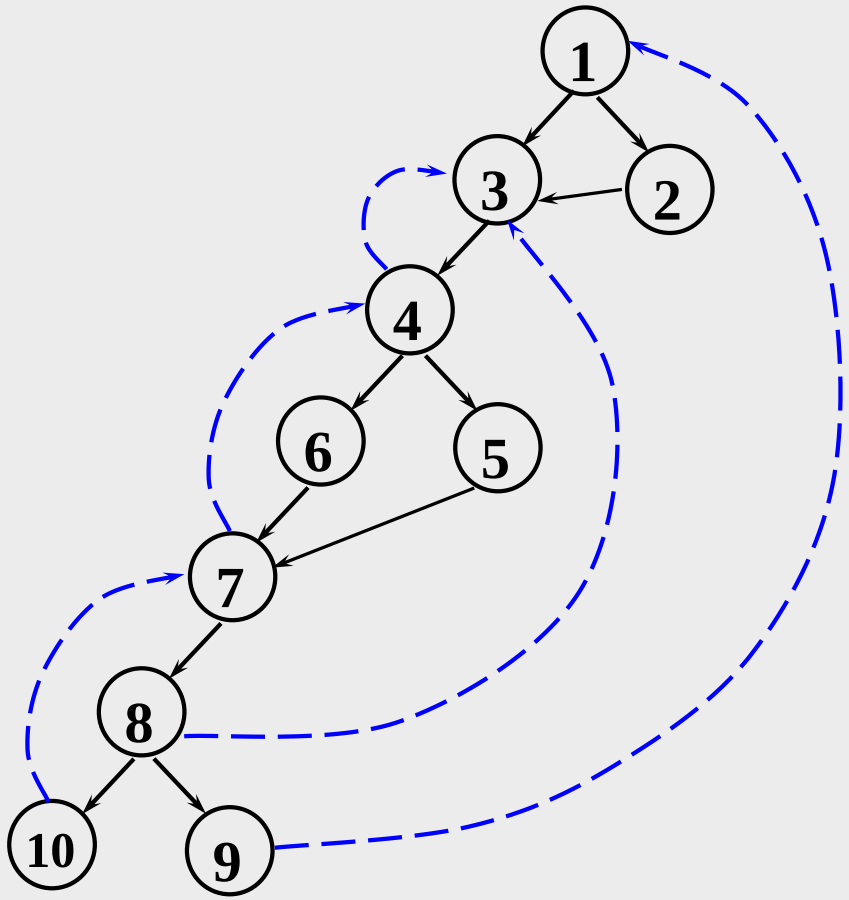
- 回边
  - 如果有$a\ dom\ b$，那么边$b \rightarrow a$叫做回边
  - 如果流图可归约，则后撤边正好就是回边

后撤边集合

4 → 3、7 → 4、10 → 7、8 → 3和9 → 1

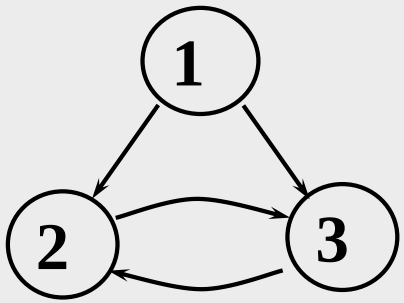
回边集合

4 → 3、7 → 4、10 → 7、8 → 3和9 → 1



回边和可归约性

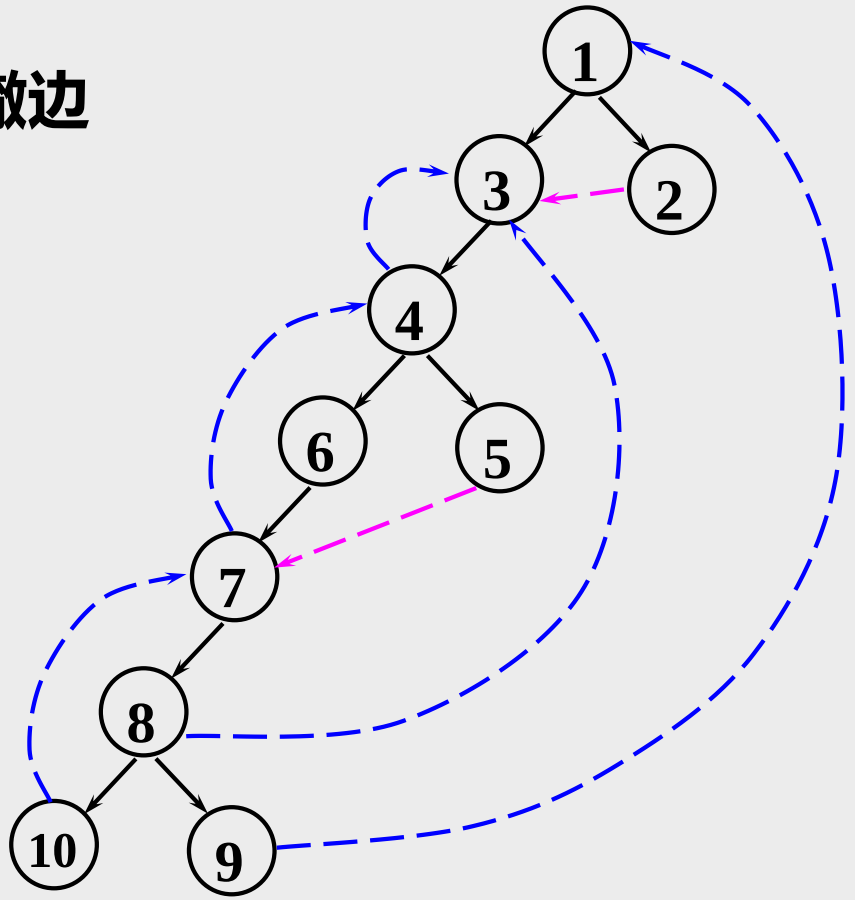
- 可归约流图
  - 如果把一个流图中所有回边删掉后，剩余的图无环
- 例：不可归约流图
  - 开始结点是1
  - 2 →3和3 →2都不是回边
  - 该图不是无环的
  - 从结点2和3两处都能进入由它们构成的环



流图的深度

- 给定深度优先生成树，深度depth是在无环路径上的后撤边数目中的最大值
  - 深度不大于流图中循环嵌套的层数
  - 该例深度为3

    路径10 → 7 → 4 → 3



自然循环

- **在源程序中，循环可以用多种形式描述**
  - for, while, goto等
- **但从程序分析的角度来看，循环的代码形式并不重要，重要的是它是否具有一些易于优化的性质。**
  - 如果循环的入口结点唯一，那么我们就可以假设某些初始条件在循环的每一次迭代开头成立。

自然循环

- **自然循环的性质**
  - 有唯一的入口结点，叫做首结点，首结点支配该循环中所有结点
  - 至少存在一条回边进入该循环首结点
- **回边$n \to d$确定的自然循环**
  - d加上不经过d能到达n的所有结点
  - 结点d是该循环的首结点

自然循环

- 回边10 → 7

  循环{7, 8, 10}

- 回边7→ 4

  循环{4, 5, 6, 7, 8, 10}

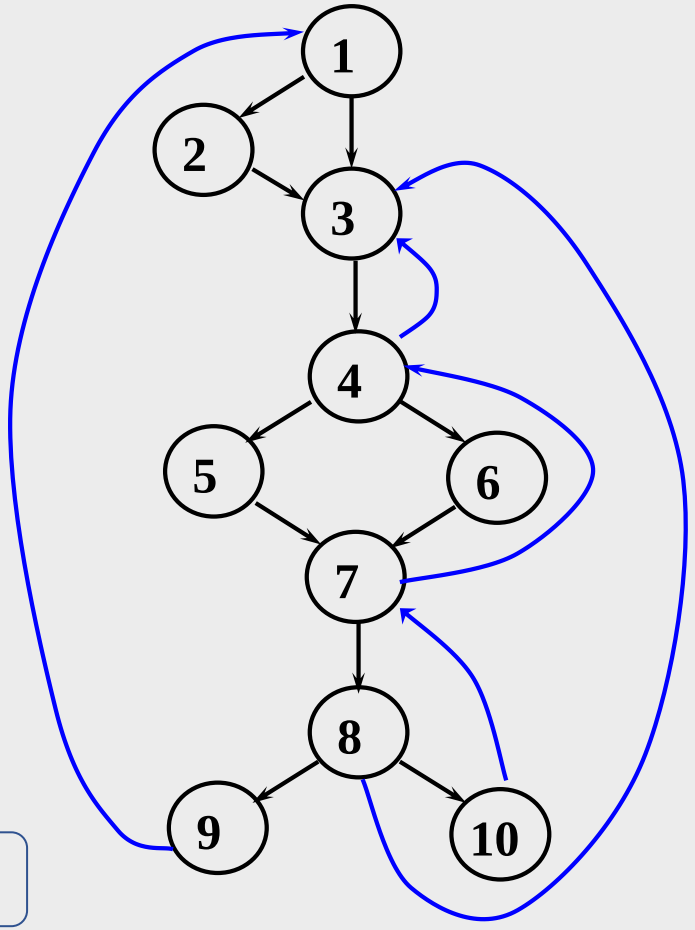
- 回边4 → 3和8 → 3

  循环{3, 4, 5, 6, 7, 8, 10}

- 回边9 → 1

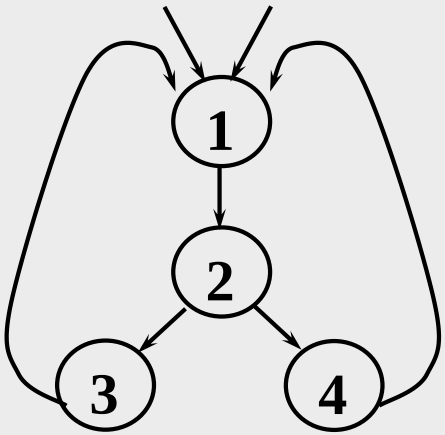
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

参考龙书算法9.46



流图中的循环

- 内循环
  - 若一个循环的结点集合是另一个循环的结点集合的子集
  - 两个循环有相同的首结点，但并非一个结点集是另一个的子集，则看成一个循环



《编译原理和技术》

循环

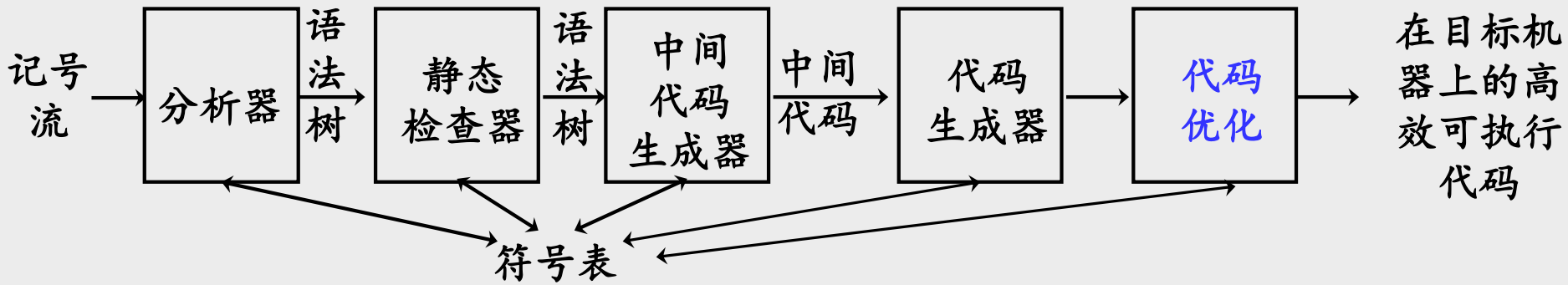
谢谢！

《编译原理和技术》

机器相关的代码优化

中科大计算机学院
李 诚
2022-11-23

本节提纲



- 流水线并行的例子
- 指令调度与数据依赖分析
- 数据依赖指导下的指令调度

Credit this slides to Stanford CS143

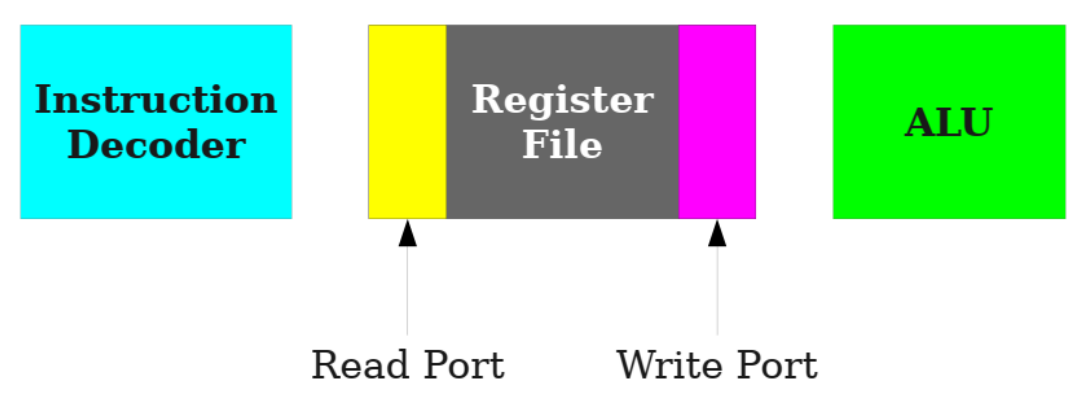
面向目标机器的代码优化 – almost final

- **目标： 优化生成的机器代码，与机器无关的优化不同，这一层级的信息是IR层无法获取的。**
- **Critical step in most compilers, but often very messy:**
  - Techniques developed for one machine may be completely useless on another.
  - Techniques developed for one language may be completely useless with another.

处理器流水线

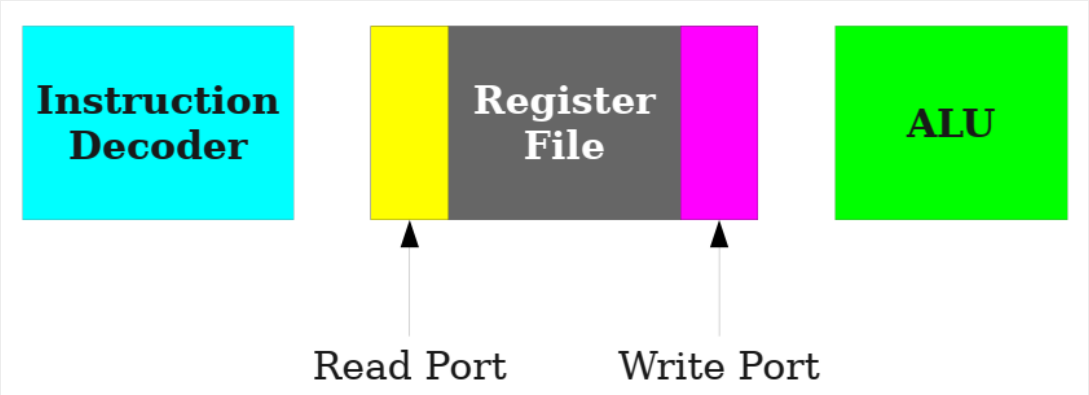
```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t5, $t3, $t4      # $t5 = $t3 + $t4
add $t8, $t6, $t7      # $t8 = $t6 + $t7
```

处理器流水线




```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t5, $t3, $t4      # $t5 = $t3 + $t4
add $t8, $t6, $t7      # $t8 = $t6 + $t7
```

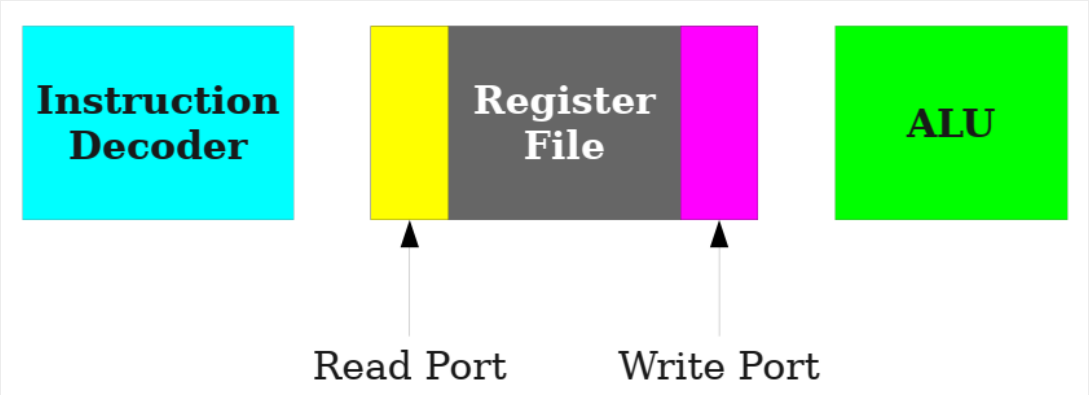
处理器流水线



ID	RR	ALU	RW

```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t5, $t3, $t4      # $t5 = $t3 + $t4
add $t8, $t6, $t7      # $t8 = $t6 + $t7
```

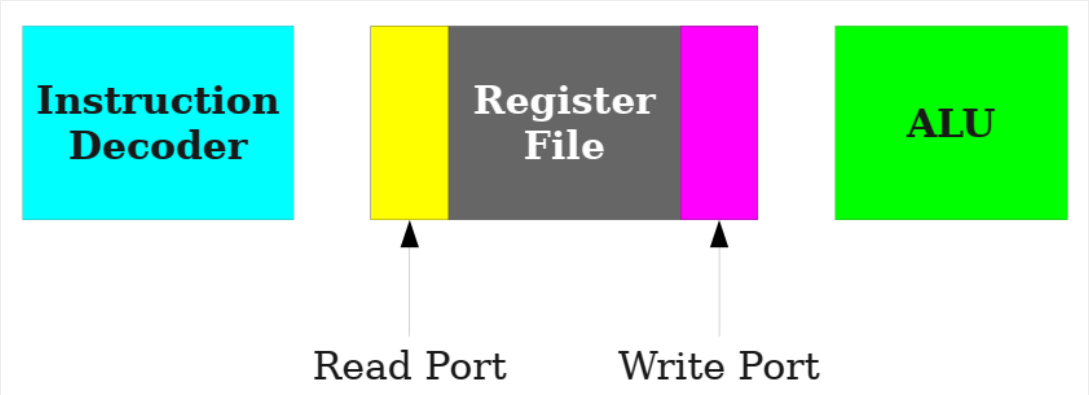
处理器流水线



ID	RR	ALU	RW

```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t5, $t3, $t4      # $t5 = $t3 + $t4
add $t8, $t6, $t7      # $t8 = $t6 + $t7
```

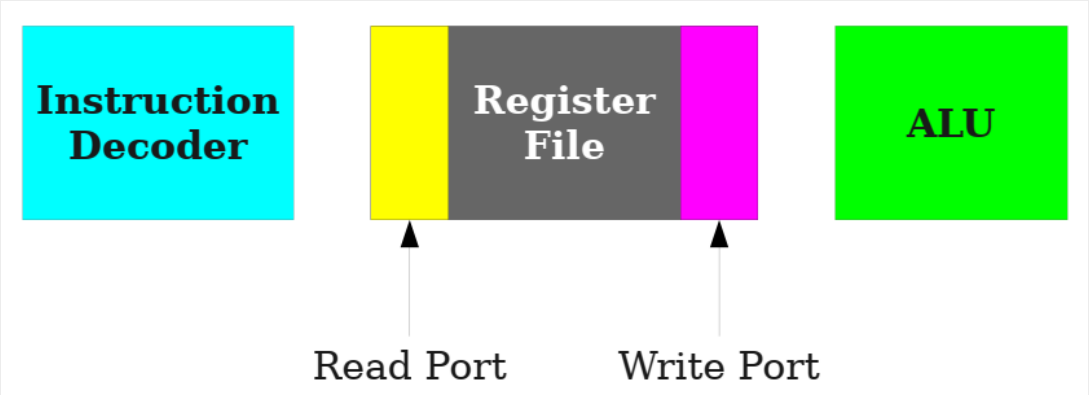
处理器流水线



ID	RR	ALU	RW

```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t5, $t3, $t4      # $t5 = $t3 + $t4
add $t8, $t6, $t7      # $t8 = $t6 + $t7
```

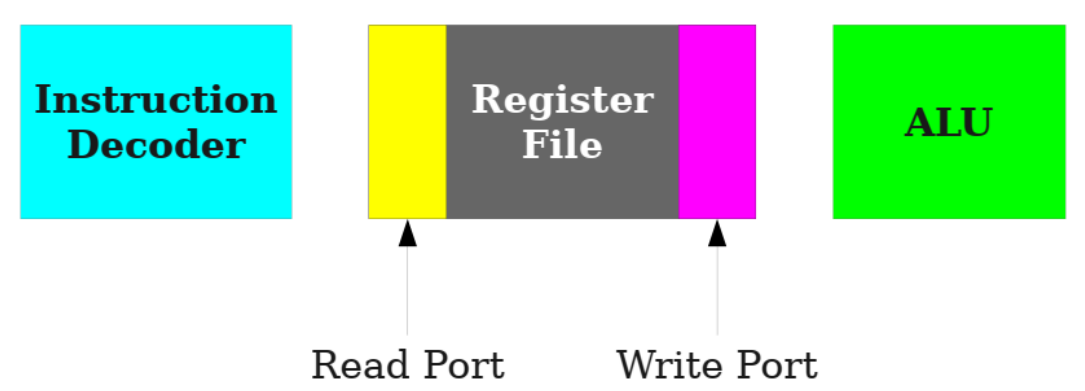
处理器流水线



ID	RR	ALU	RW

```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t5, $t3, $t4      # $t5 = $t3 + $t4
add $t8, $t6, $t7      # $t8 = $t6 + $t7
```

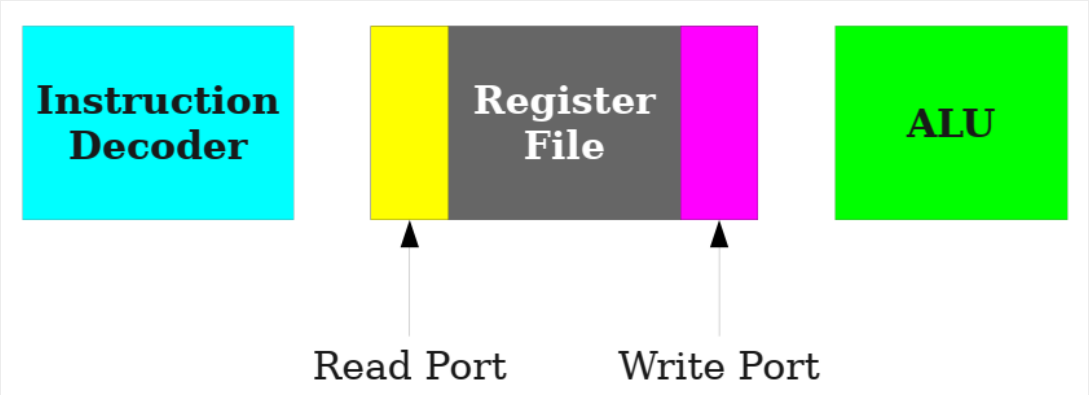
处理器流水线



ID	RR	ALU	RW

```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t5, $t3, $t4      # $t5 = $t3 + $t4
add $t8, $t6, $t7      # $t8 = $t6 + $t7
```

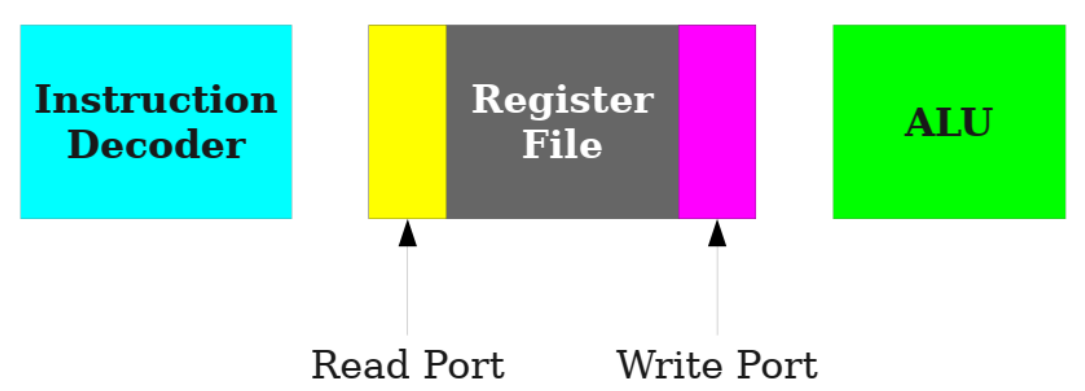
处理器流水线



ID	RR	ALU	RW

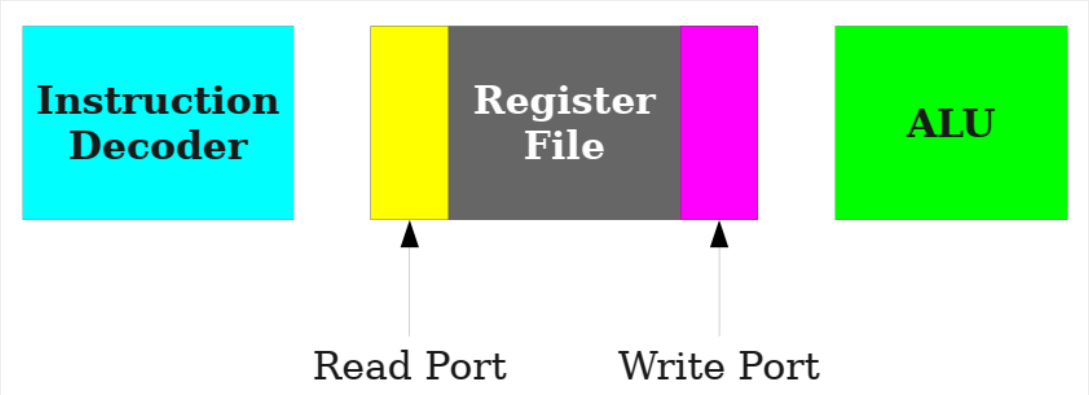
```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t5, $t3, $t4      # $t5 = $t3 + $t4
add $t8, $t6, $t7      # $t8 = $t6 + $t7
```

复杂的流水线并行

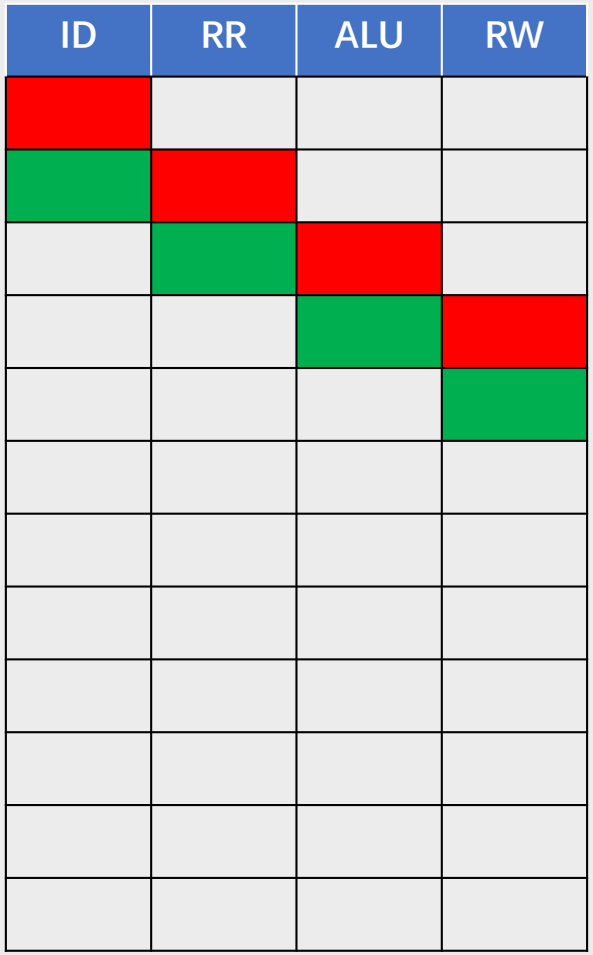



```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t4, $t3, $t2      # $t4 = $t3 + $t2
add $t7, $t5, $t6      # $t7 = $t5 + $t6
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```

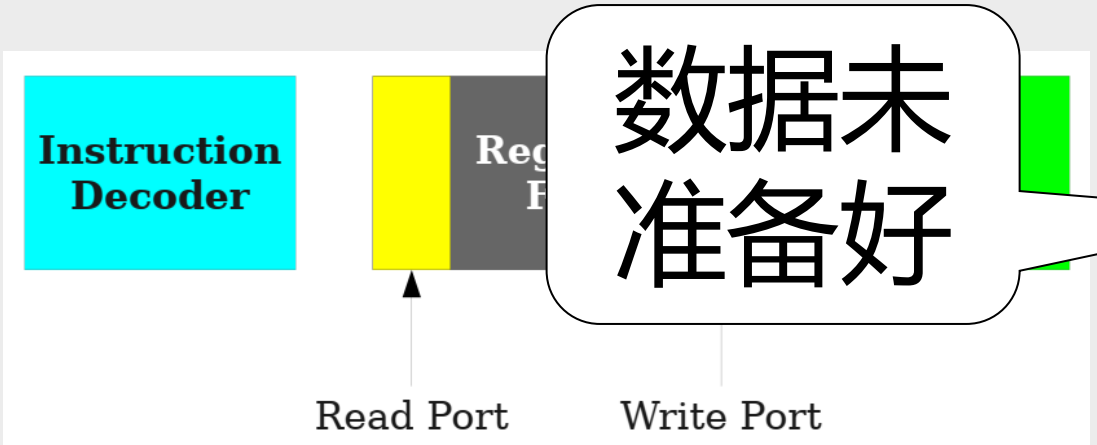
复杂的流水线并行

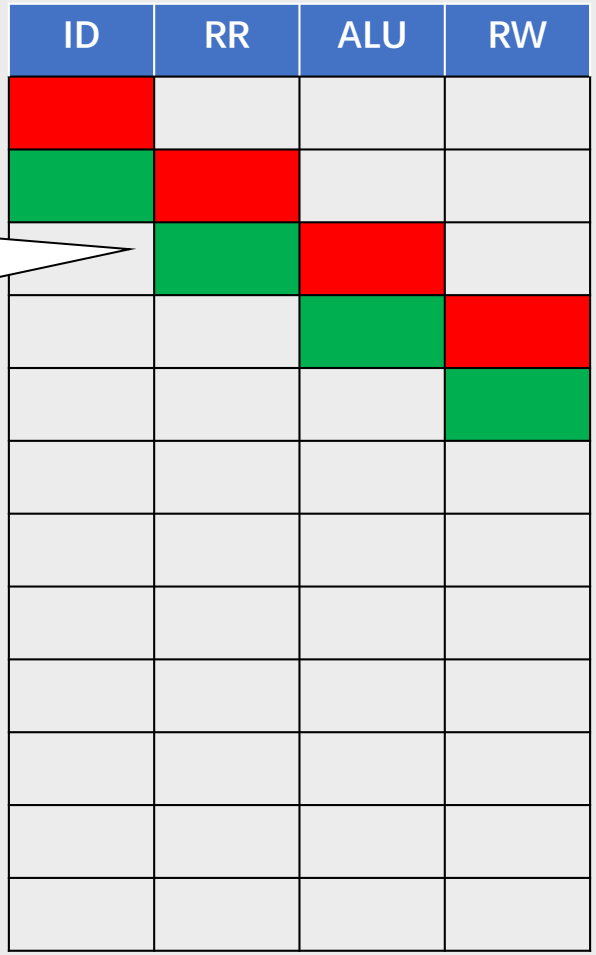



```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t4, $t3, $t2      # $t4 = $t3 + $t2
add $t7, $t5, $t6      # $t7 = $t5 + $t6
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```



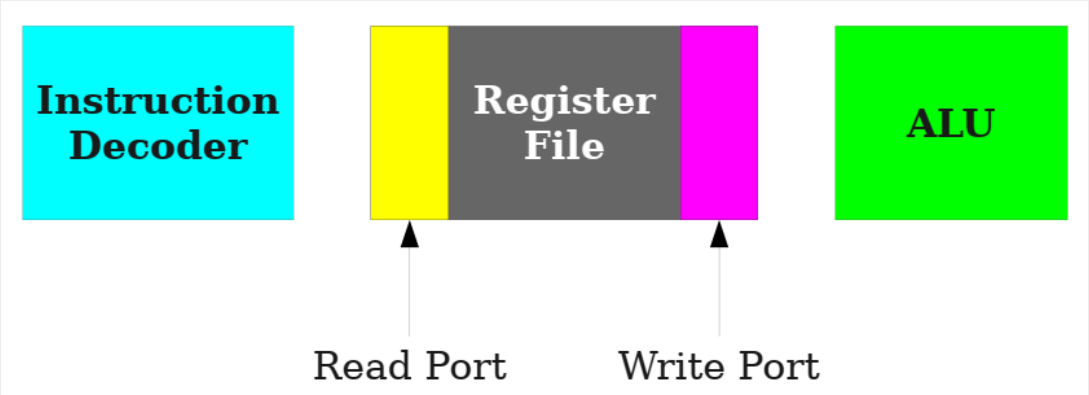
复杂的流水线并行



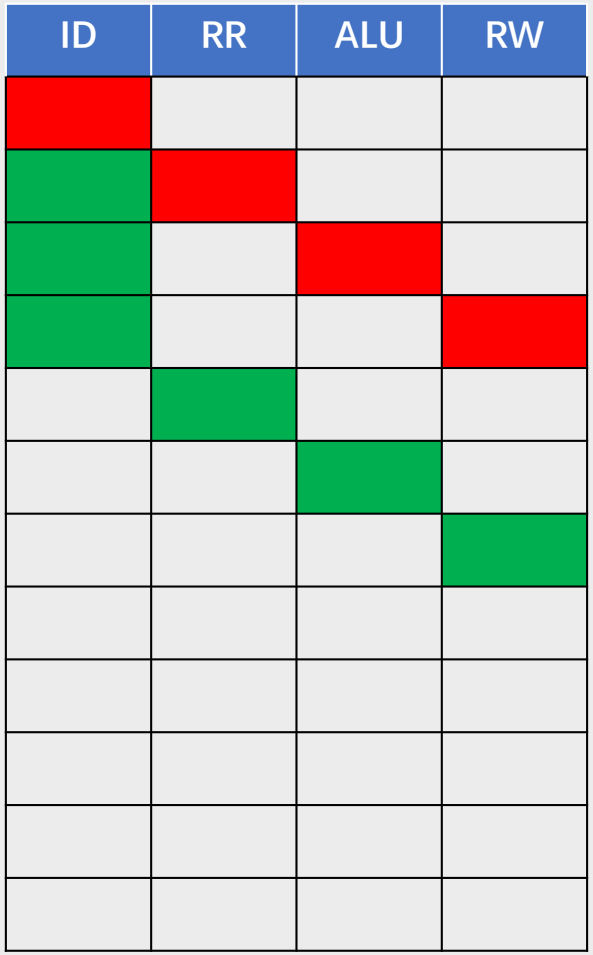



```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t4, $t3, $t2      # $t4 = $t3 + $t2
add $t7, $t5, $t6      # $t7 = $t5 + $t6
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```

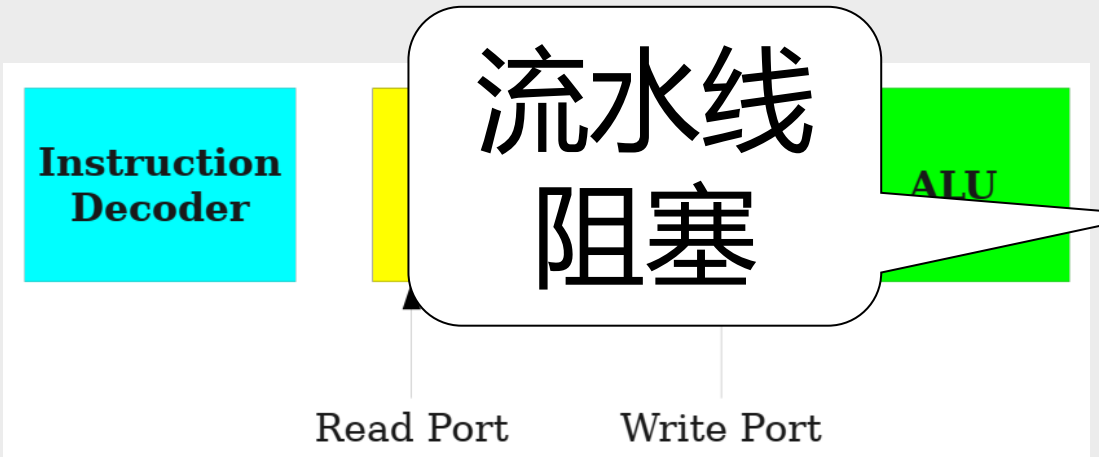
复杂的流水线并行

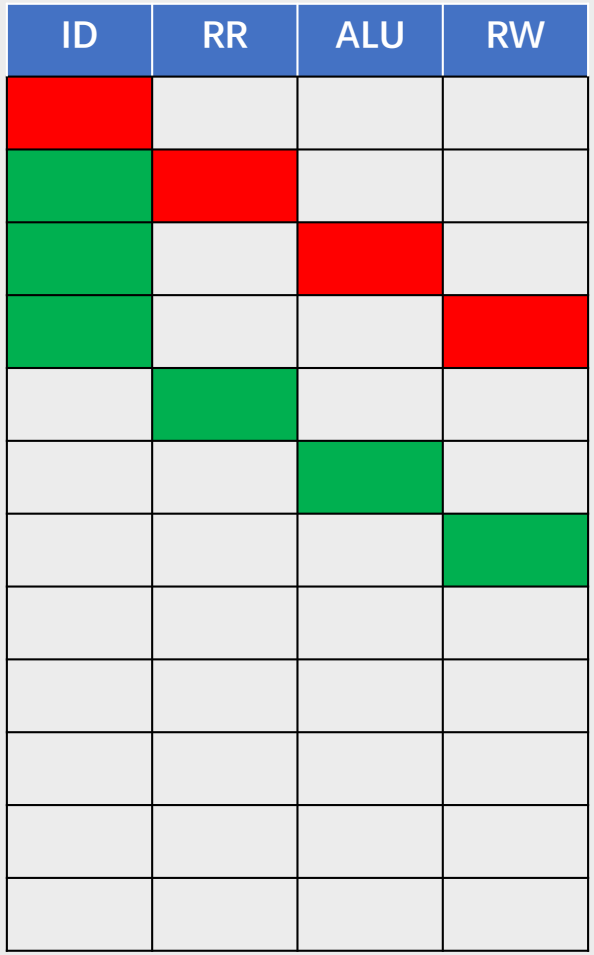



```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t4, $t3, $t2      # $t4 = $t3 + $t2
add $t7, $t5, $t6      # $t7 = $t5 + $t6
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```



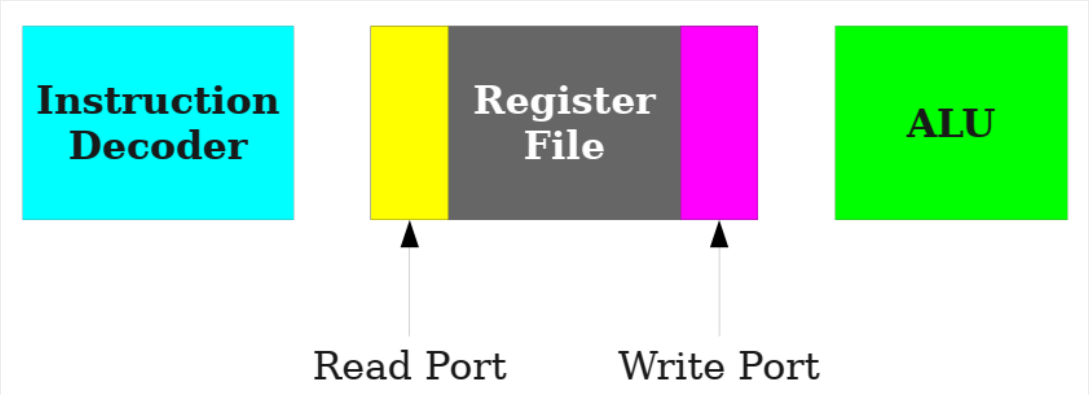
复杂的流水线并行



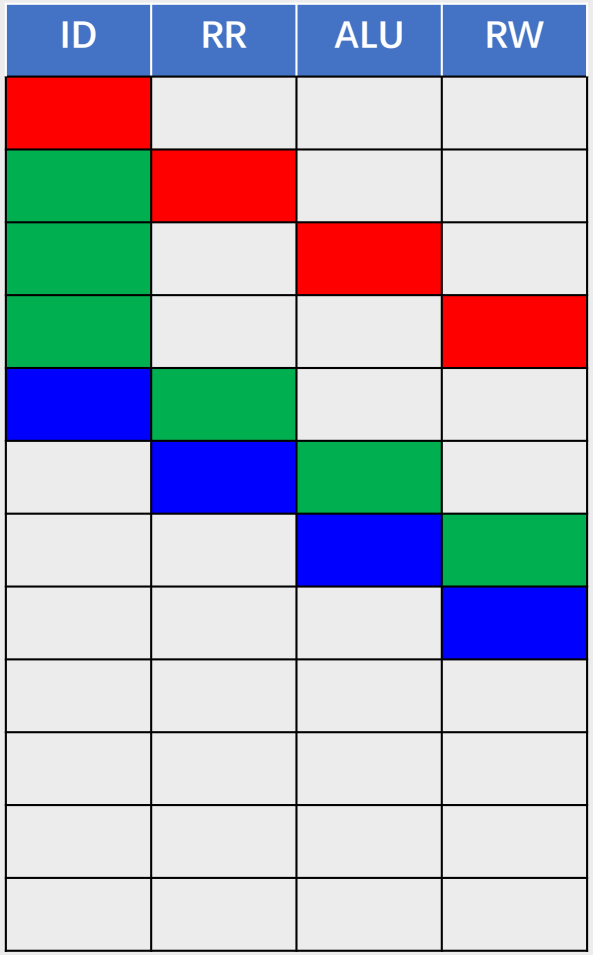



```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t4, $t3, $t2      # $t4 = $t3 + $t2
add $t7, $t5, $t6      # $t7 = $t5 + $t6
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```

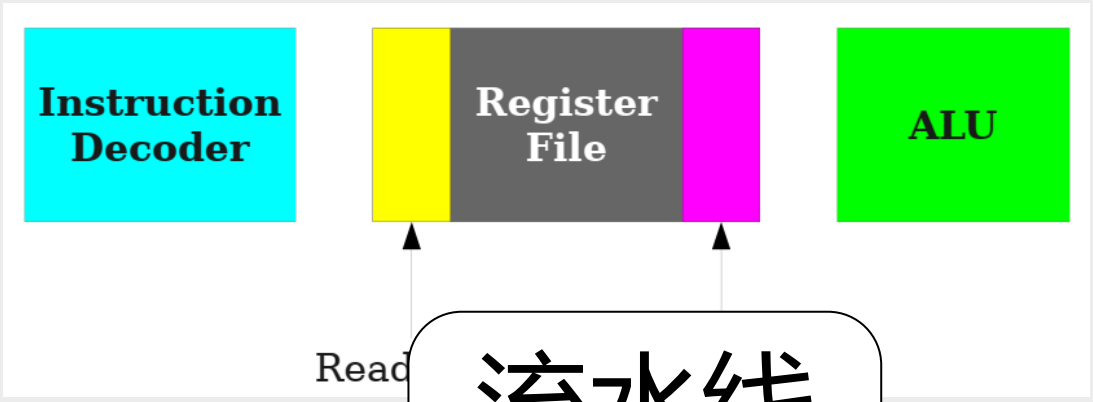
复杂的流水线并行




```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t4, $t3, $t2      # $t4 = $t3 + $t2
add $t7, $t5, $t6      # $t7 = $t5 + $t6
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```



复杂的流水线并行



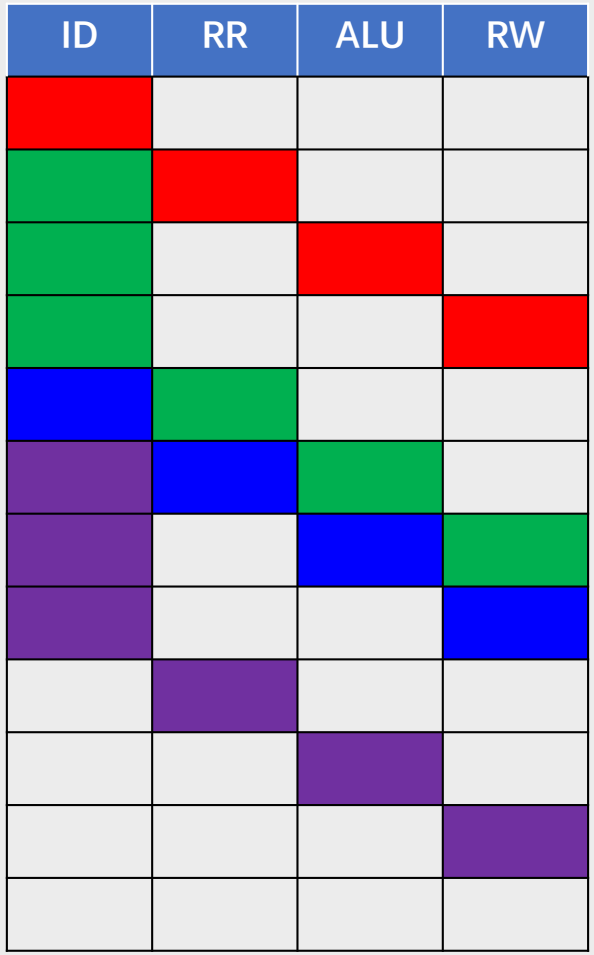
流水线阻塞

add $t2, $t0, $t + $t1

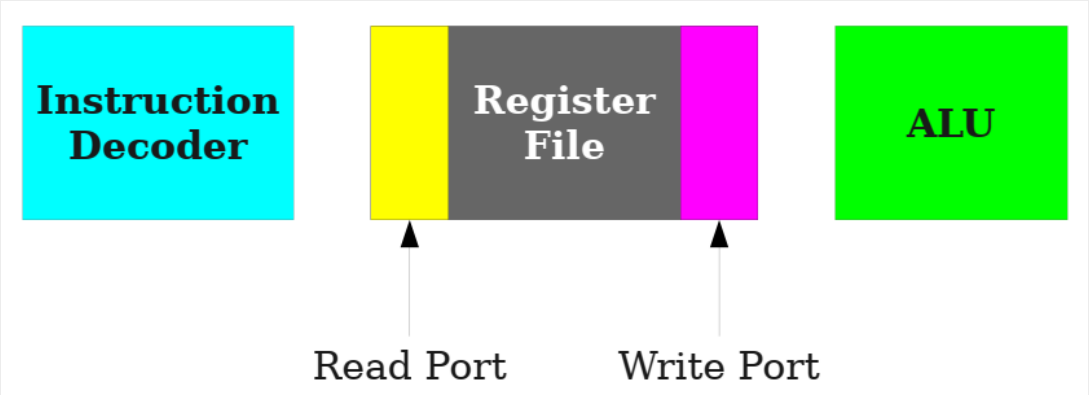
add $t4, $t3, $t + $t2

add $t7, $t5, $t6 # $t7 = $t5 + $t6

add $t0, $t0, $t7 # $t0 = $t0 + $t7



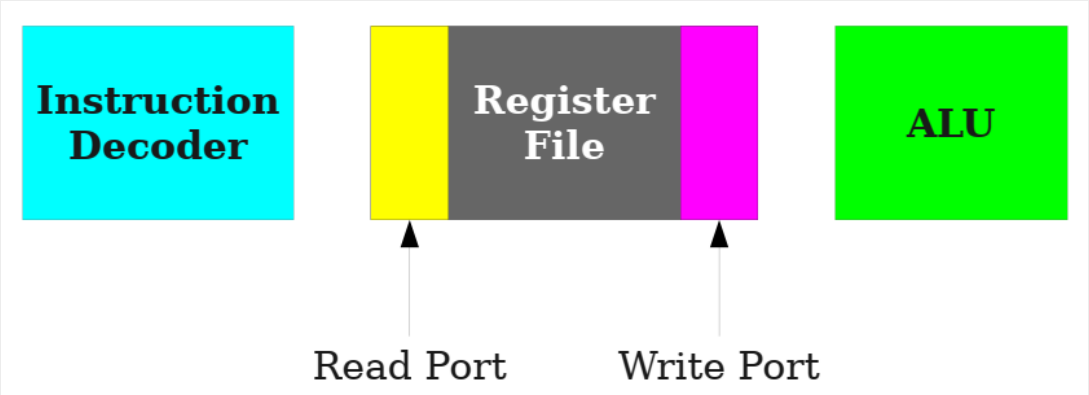
可能的优化




```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t4, $t3, $t2      # $t4 = $t3 + $t2
add $t7, $t5, $t6      # $t7 = $t5 + $t6
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```

ID	RR	ALU	RW

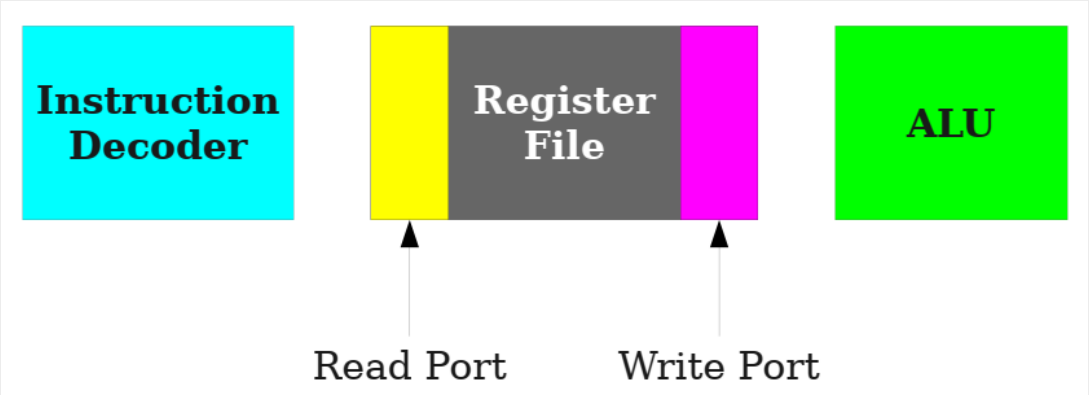
可能的优化




```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t7, $t5, $t6      # $t7 = $t5 + $t6
add $t4, $t3, $t2      # $t4 = $t3 + $t2
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```

ID	RR	ALU	RW

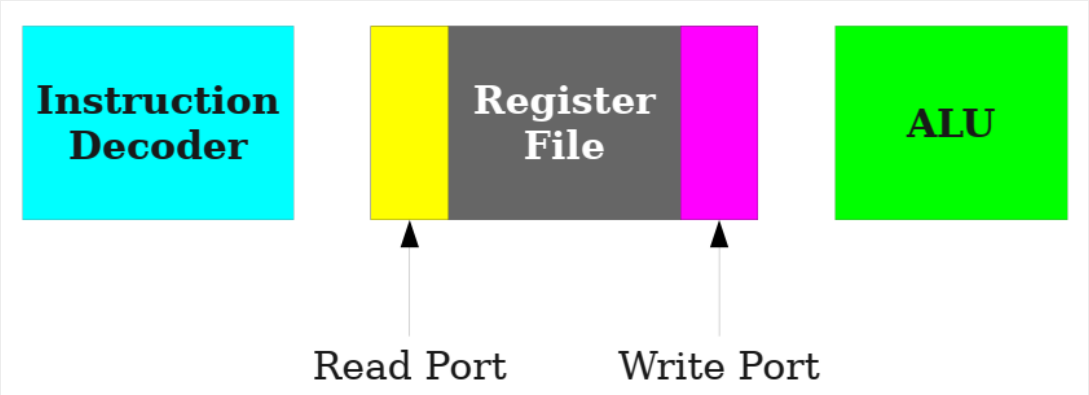
可能的优化



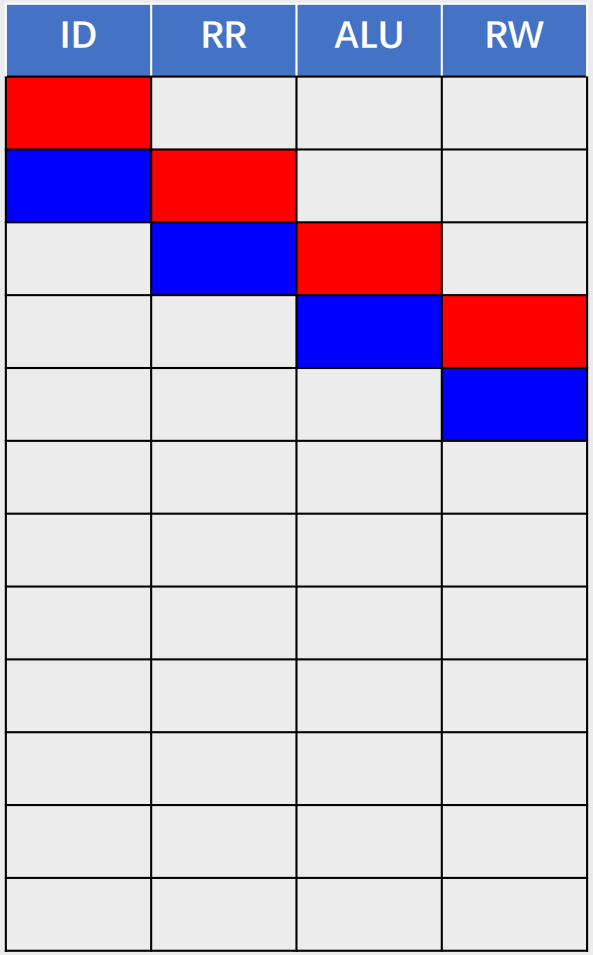

```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t7, $t5, $t6      # $t7 = $t5 + $t6
add $t4, $t3, $t2      # $t4 = $t3 + $t2
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```

ID	RR	ALU	RW

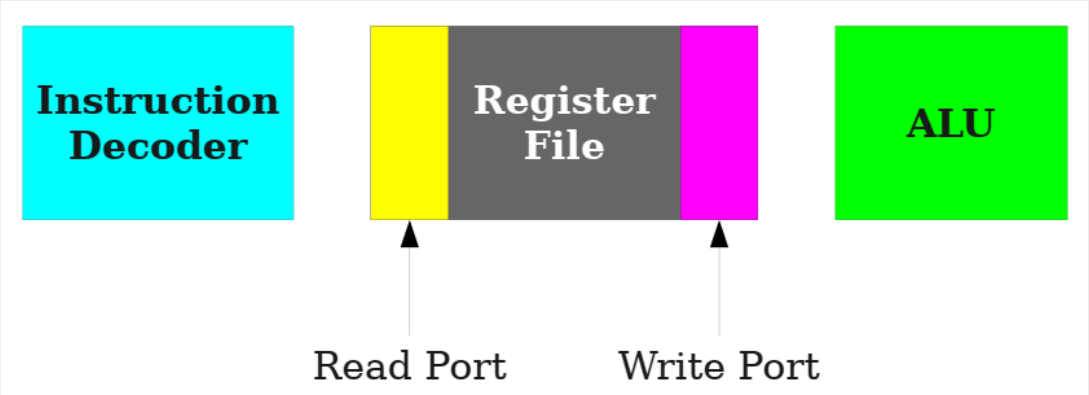
可能的优化



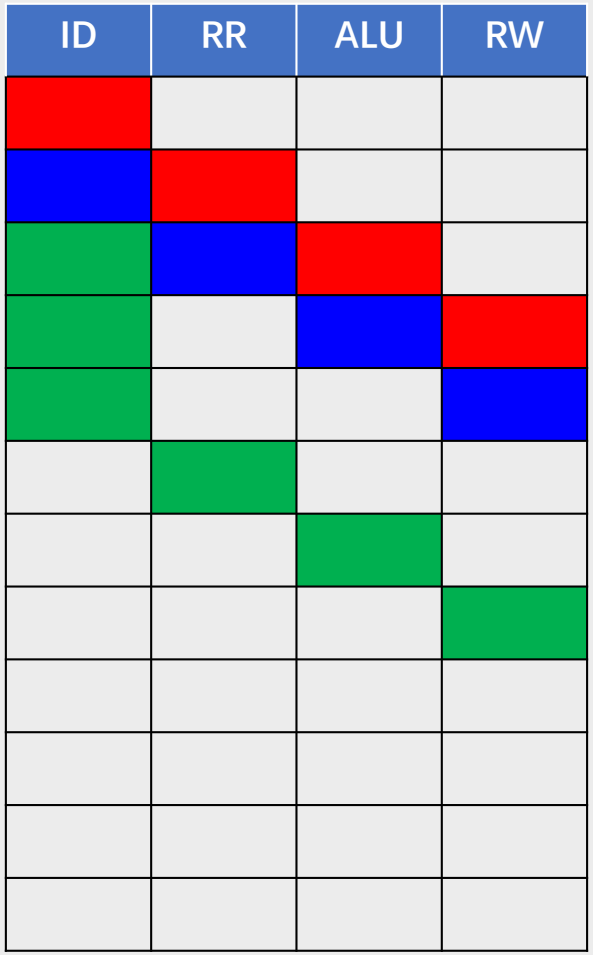

```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t7, $t5, $t6      # $t7 = $t5 + $t6
add $t4, $t3, $t2      # $t4 = $t3 + $t2
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```



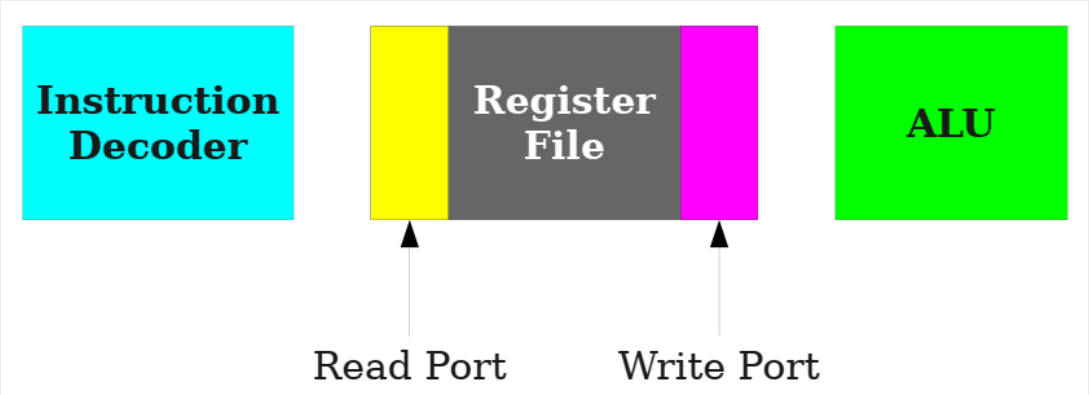
可能的优化



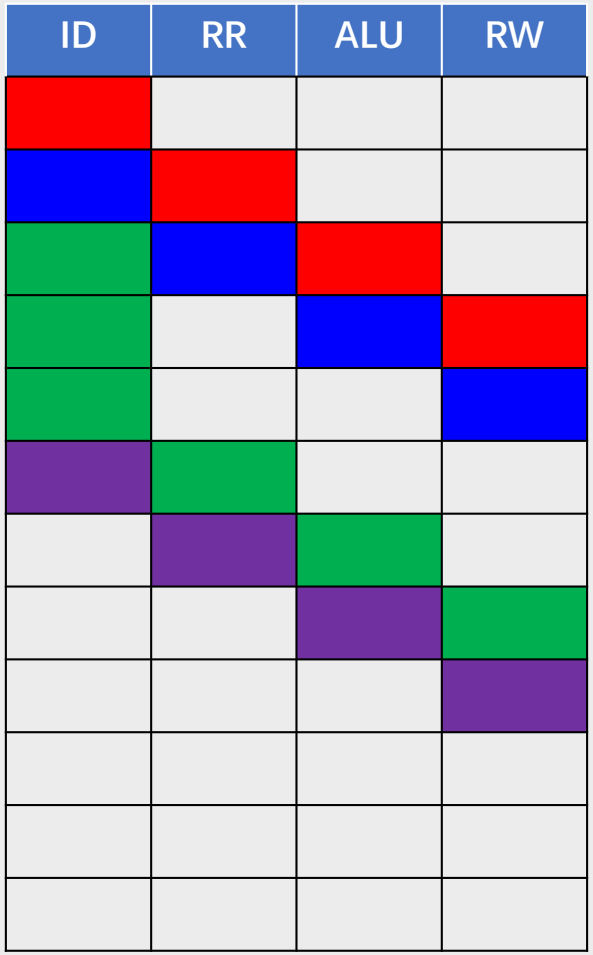

```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t7, $t5, $t6      # $t7 = $t5 + $t6
add $t4, $t3, $t2      # $t4 = $t3 + $t2
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```



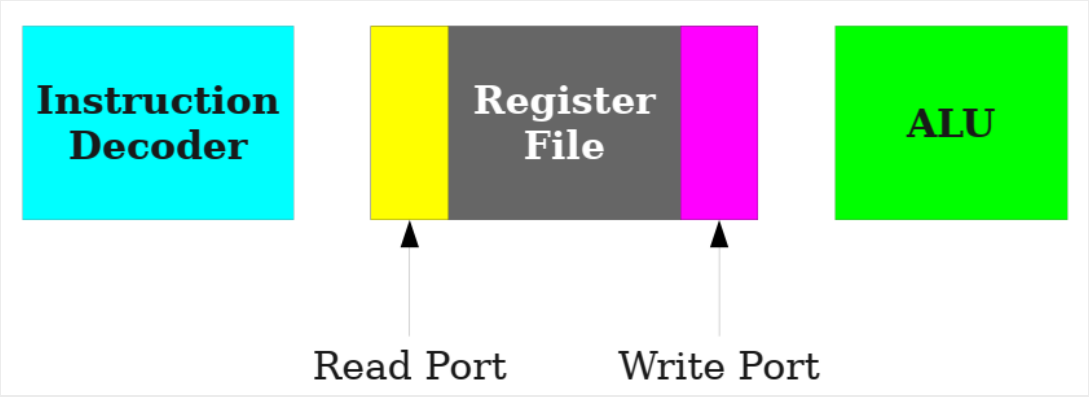
可能的优化




```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t7, $t5, $t6      # $t7 = $t5 + $t6
add $t4, $t3, $t2      # $t4 = $t3 + $t2
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```

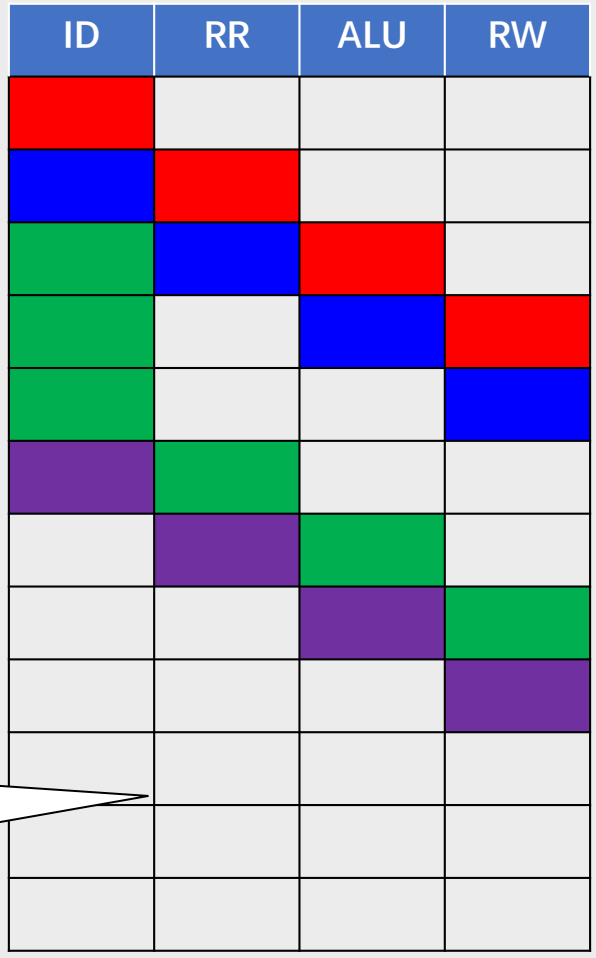


可能的优化

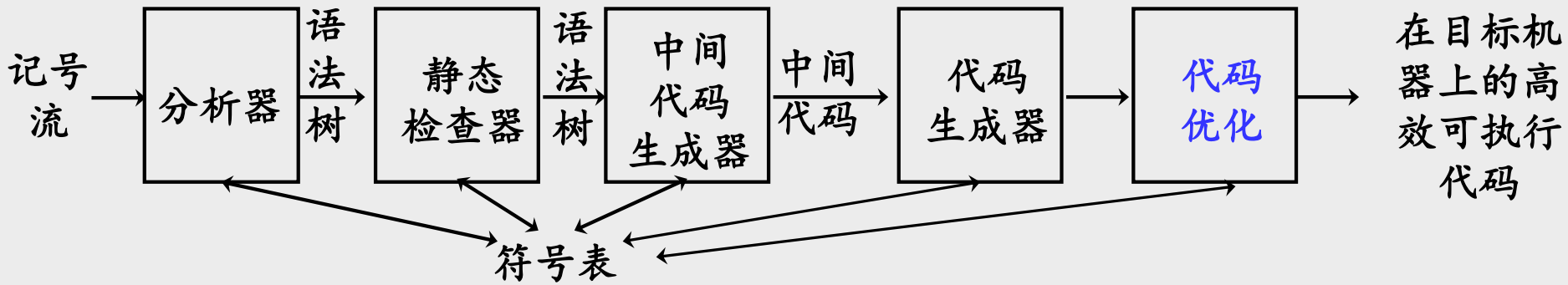



```
add $t2, $t0, $t1      # $t2 = $t0 + $t1
add $t7, $t5, $t       6
add $t4, $t3, $t
add $t0, $t0, $t7      # $t0 = $t0 + $t7
```

节省两个时钟周期



本节提纲



- 流水线并行的例子
- 指令调度与数据依赖分析
- 数据依赖指导下的指令调度

指令调度

- 由于处理器流水线并行机制，指令的执行顺序对性能有较大影响
- 指令调度：重新排列机器代码指令，旨在最小化执行特定指令序列所需的时钟周期数。
- All good optimizing compilers have some sort of instruction scheduling support.
- 然而，在处理器流水线上执行的顺序代码内含着一些指令之间的依赖关系，在指令调度期间执行的任何转换都必须保留这些依赖关系，以维护被调度代码的逻辑。

数据依赖关系

- A data dependency in machine code is a set of instructions whose behavior depends on one another.
- Intuitively, a set of instructions that cannot be reordered around each other.

三种数据依赖关系

- **read-after-write，RAW**
  - 当一条指令读取另一条指令写入的结果时，会产生写后读相关性，读指令必须在写指令一定时钟周期后再读取而不会产生阻塞。

```
x = ...
... = x
```

- **write-after-read，WAR**
  - 当一条指令写在另一条指令的操作数上时，会产生反向依赖或称读后写依赖。 读指令必须在写指令之前经过适当的周期数才能安全读取，而不阻塞写指令。

```
... = x
x = ...
```

- **write-after-write，WAW**
  - 如果两条指令写入同一个目标，就会产生单个输出或写后写依赖关系

```
x = ...
x = ...
```

分析数据依赖关系

t0 = t1 + t2

t1 = t0 + t1

t3 = t2 + t4

t0 = t1 + t2

t5 = t3 + t4

t6 = t2 + t7

分析数据依赖关系

t0 = t1 + t2

t1 = t0 + t1

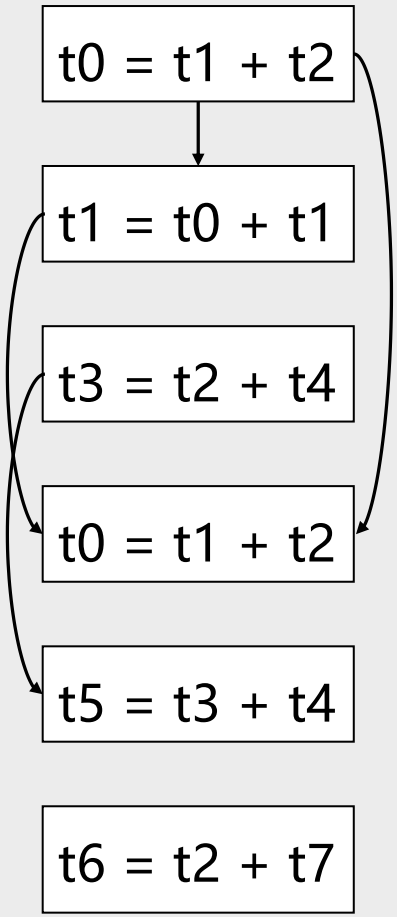
t3 = t2 + t4

t0 = t1 + t2

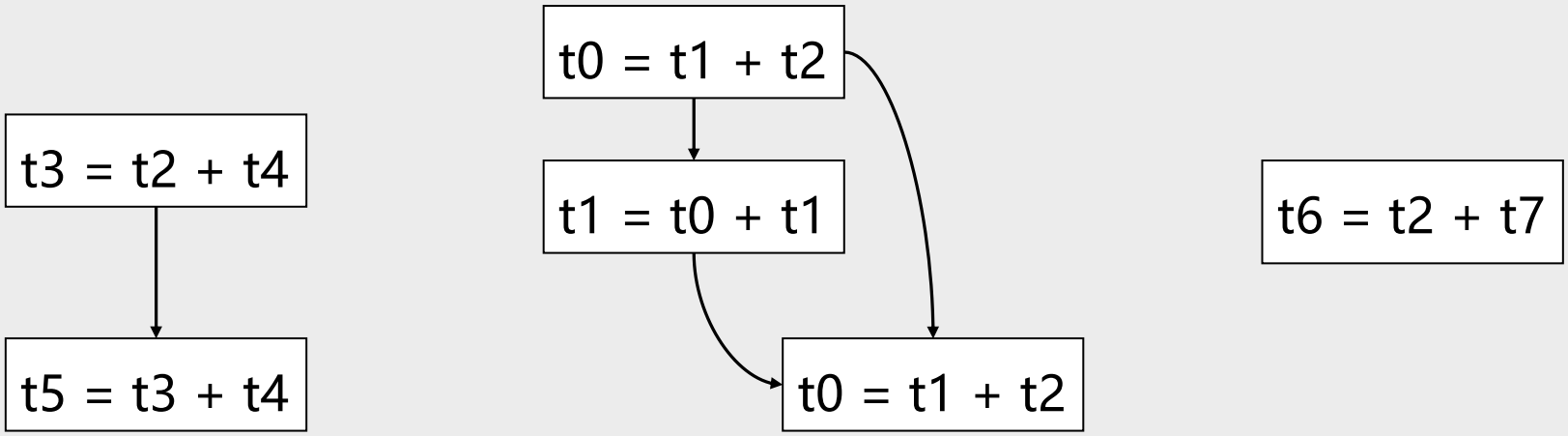
t5 = t3 + t4

t6 = t2 + t7

分析数据依赖关系



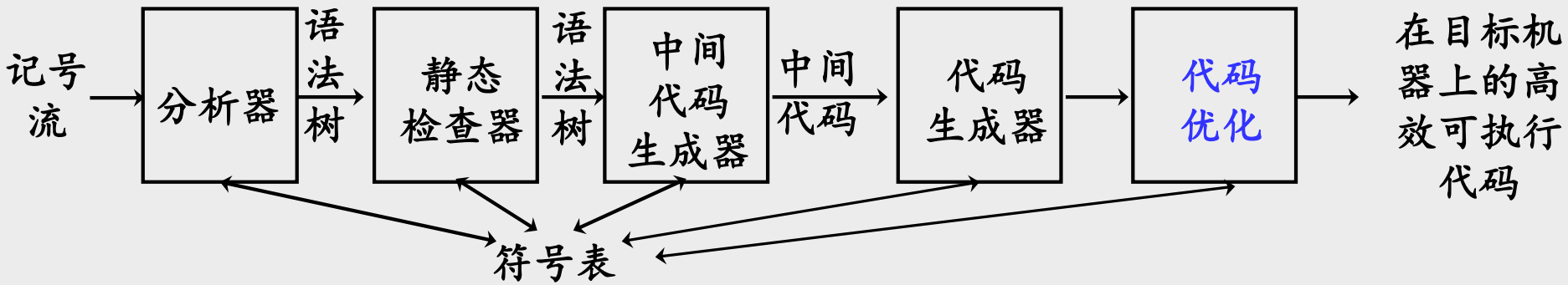
分析数据依赖关系



指令数据依赖图[HennessyGross，1983]

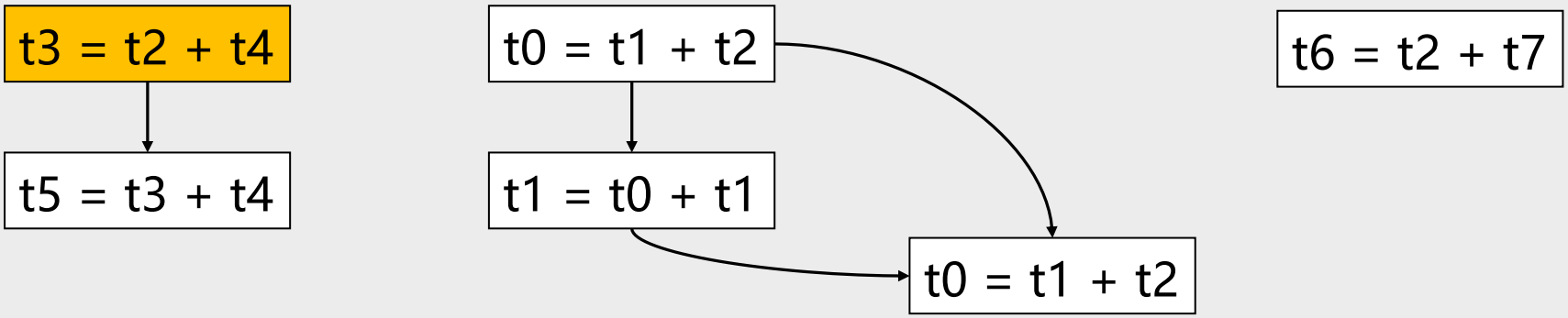
- **The graph of the data dependencies in a basic block is called the data dependency graph.**
- **Always a directed acyclic graph (DAG):**
  - Directed: One instruction depends on the other.
  - Acyclic: No circular dependencies allowed.
- **Can schedule instructions in a basic block in any order as long we never schedule a node before all its parents.**
- **Idea: Do a topological sort of the data dependency graph and output instructions in that order.**

本节提纲

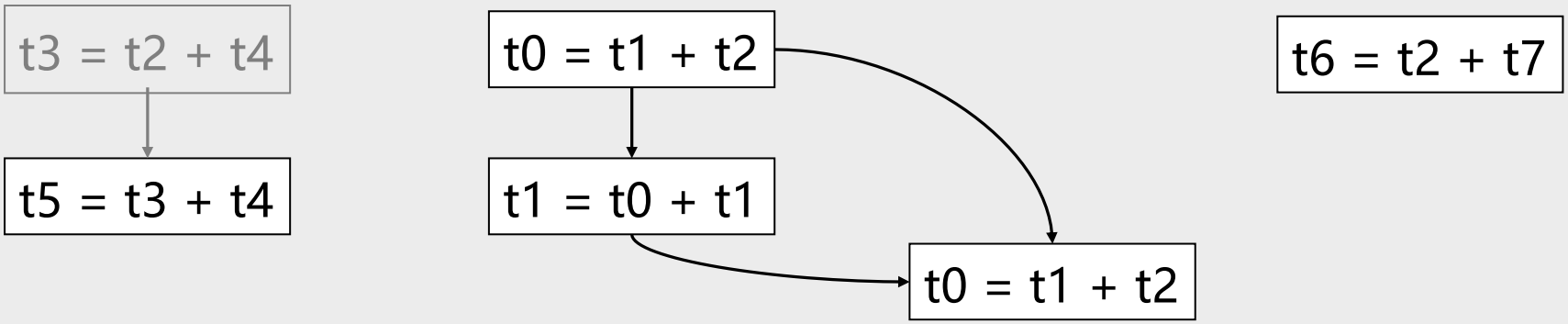


- 流水线并行的例子
- 指令调度与数据依赖分析
- **数据依赖指导下的指令调度**

数据依赖指导下的指令调度

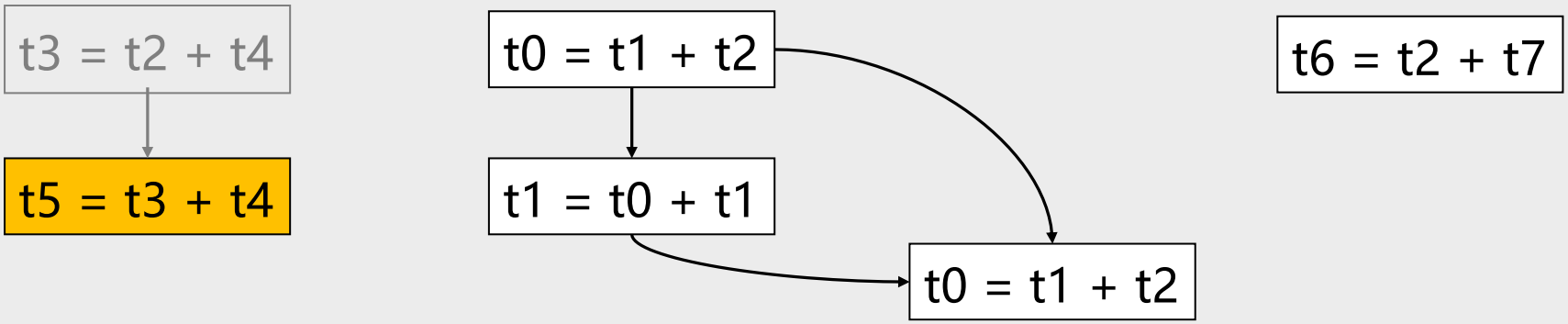


数据依赖指导下的指令调度



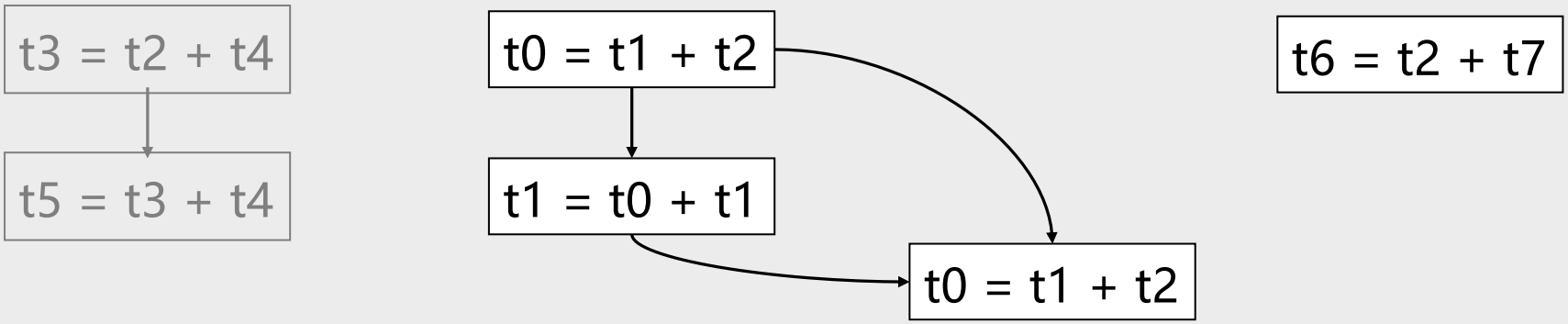
t3 = t2 + t4

数据依赖指导下的指令调度



t3 = t2 + t4

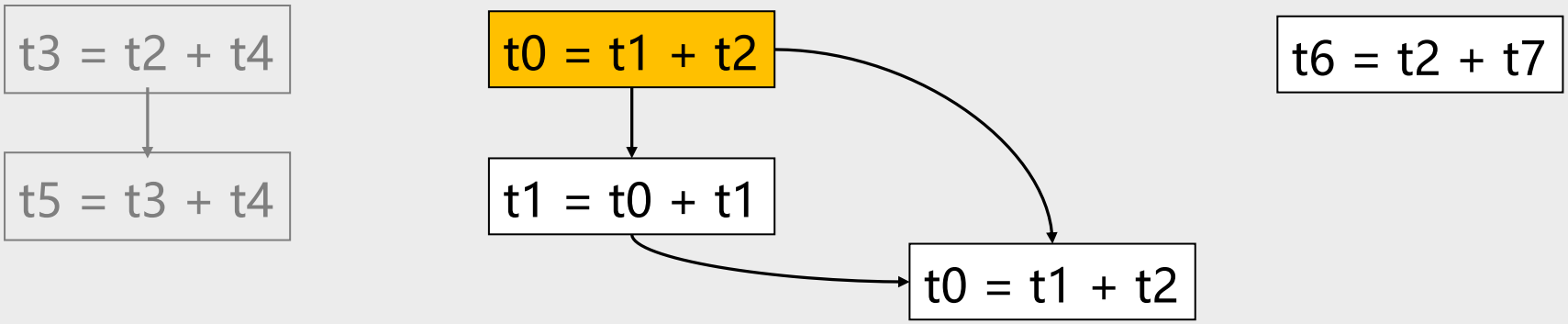
数据依赖指导下的指令调度



t3 = t2 + t4

t5 = t3 + t4

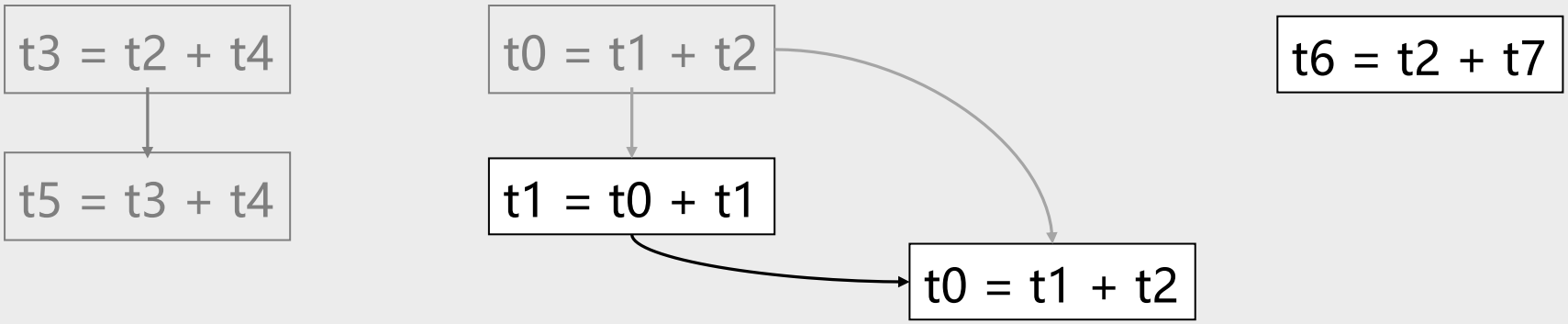
数据依赖指导下的指令调度



t3 = t2 + t4

t5 = t3 + t4

数据依赖指导下的指令调度

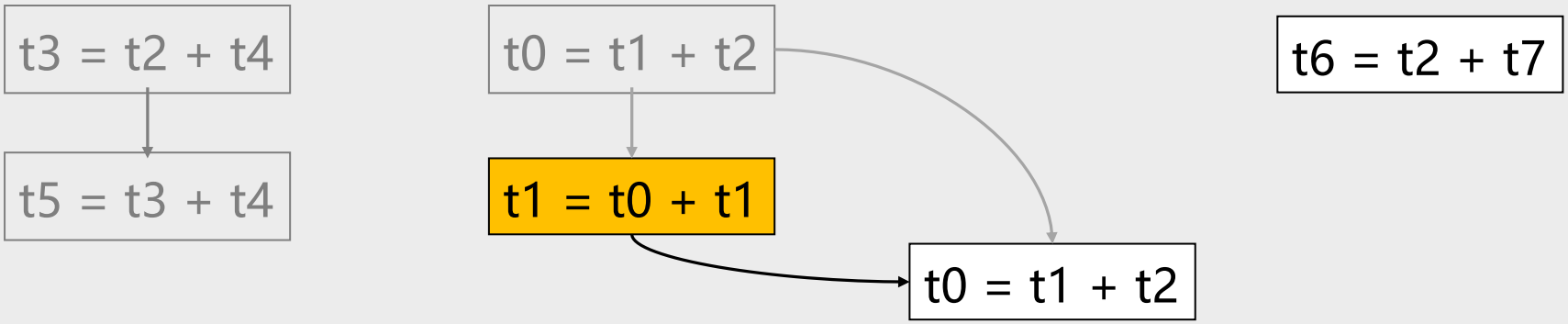


t3 = t2 + t4

t5 = t3 + t4

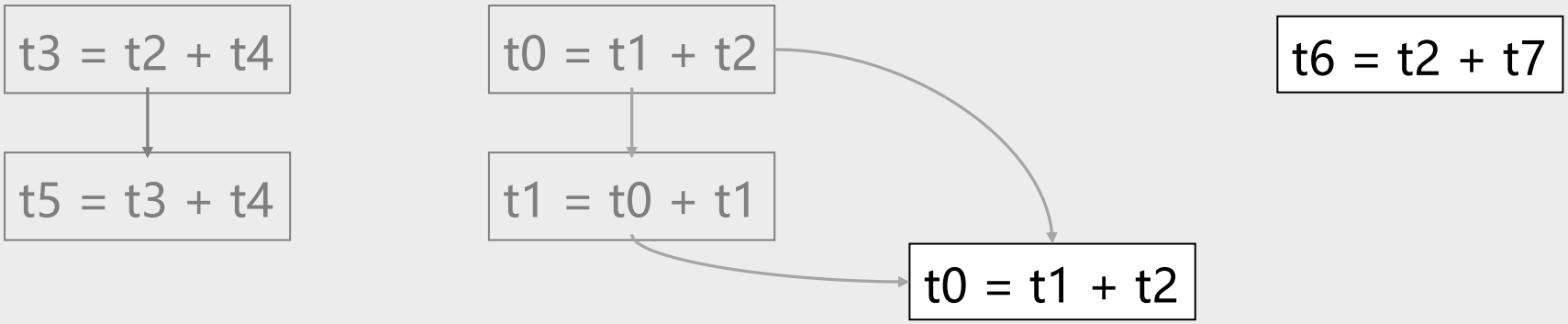
t0 = t1 + t2

数据依赖指导下的指令调度



t3 = t2 + t4
t5 = t3 + t4
t0 = t1 + t2

数据依赖指导下的指令调度



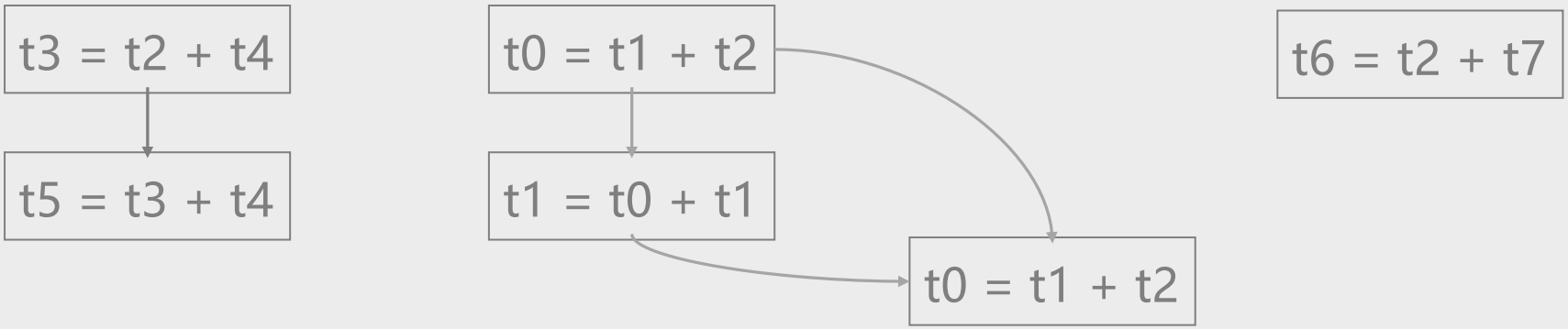
t3 = t2 + t4

t5 = t3 + t4

t0 = t1 + t2

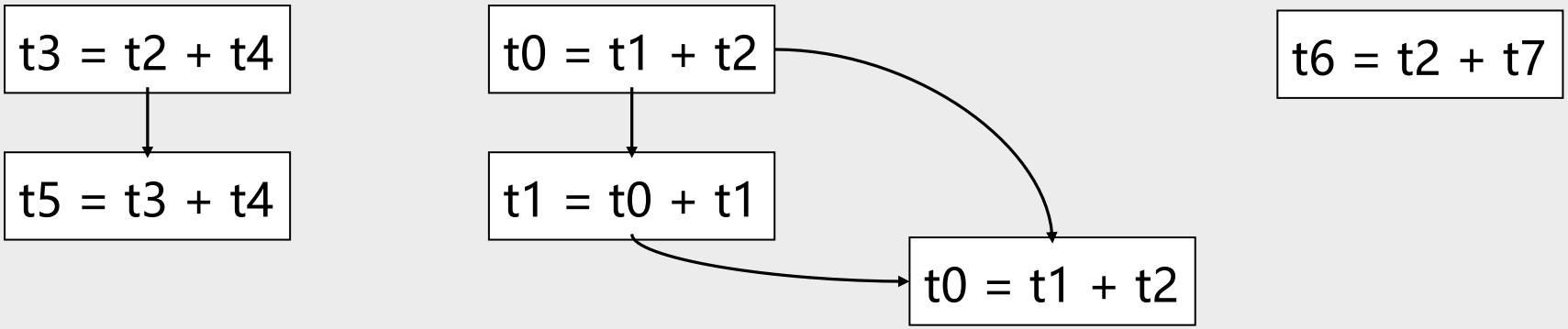
t1 = t0 + t1

数据依赖指导下的指令调度



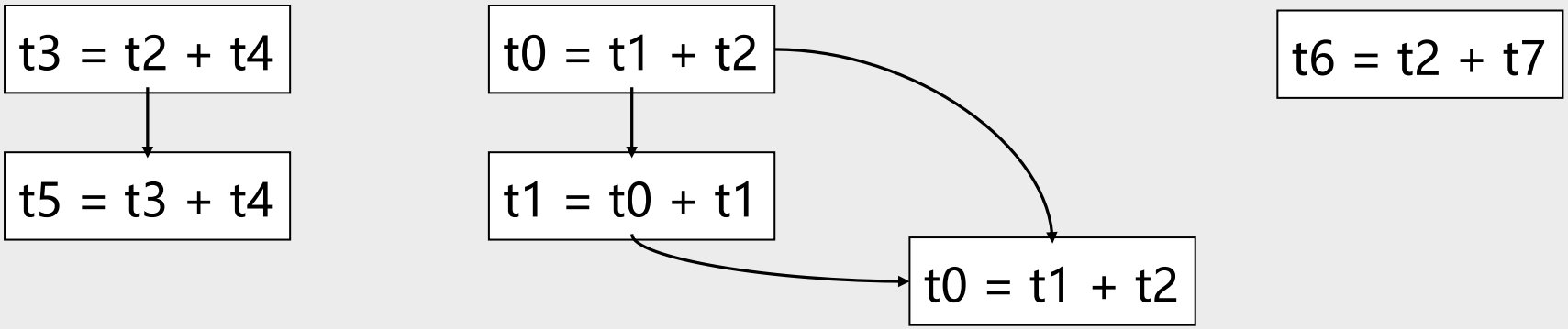
t3 = t2 + t4
t5 = t3 + t4
t0 = t1 + t2
t1 = t0 + t1
t0 = t1 + t2
t6 = t2 + t7

数据依赖指导下的指令调度



t3 = t2 + t4
t5 = t3 + t4
t0 = t1 + t2
t1 = t0 + t1
t0 = t1 + t2
t6 = t2 + t7

数据依赖指导下的指令调度

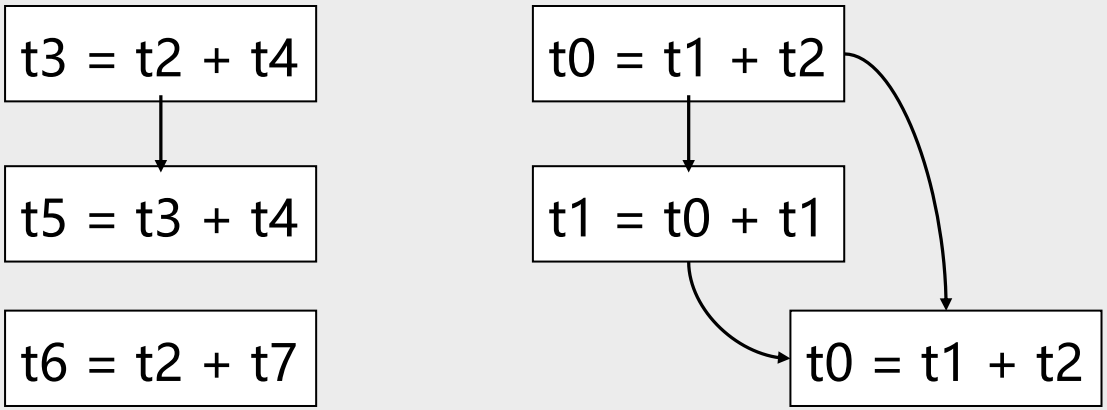


t3 = t2 + t4	t0 = t1 + t2
t5 = t3 + t4	t3 = t2 + t4
t0 = t1 + t2	t6 = t2 + t7
t1 = t0 + t1	t1 = t0 + t1
t0 = t1 + t2	t5 = t3 + t4
t6 = t2 + t7	t0 = t1 + t2

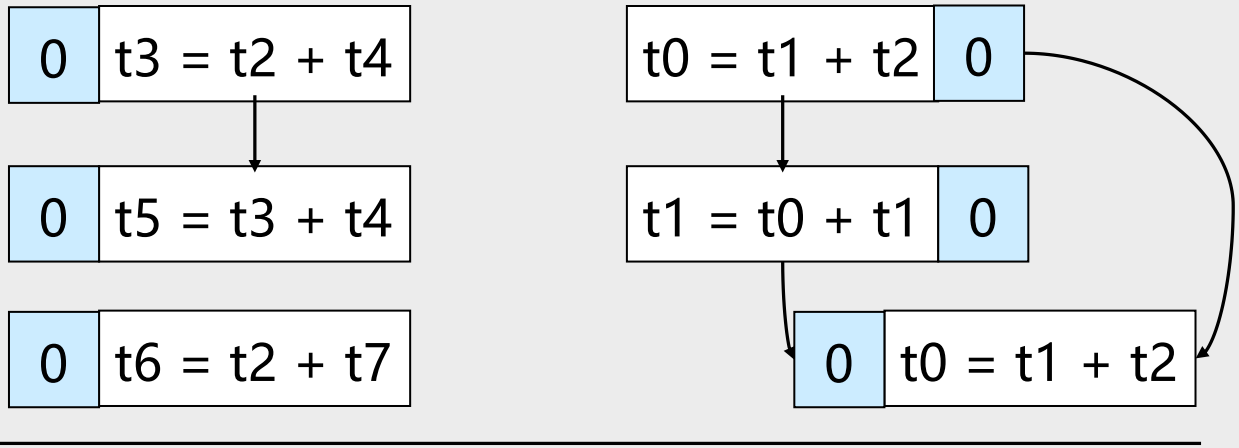
指令调度的空间

- **There can be many valid topological orderings of a data dependency graph. How do we pick one that works well with the pipeline?**
- **In general, finding the fastest instruction schedule is known to be NP-hard.**
  - Don't expect a polynomial-time algorithm anytime soon!
- **Heuristics are used in practice:**
  - Schedule instructions that can run to completion without interference before instructions that cause interference.
  - Schedule instructions with more dependents before instructions with fewer dependents.
  - Adapting DAG to be weighted! (边的权重为指令等待时间)

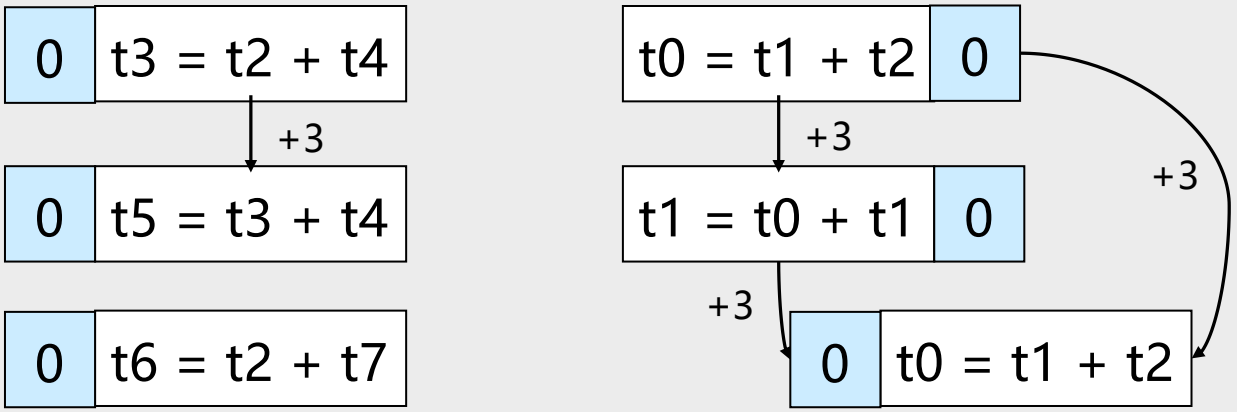
升级版的指令调度



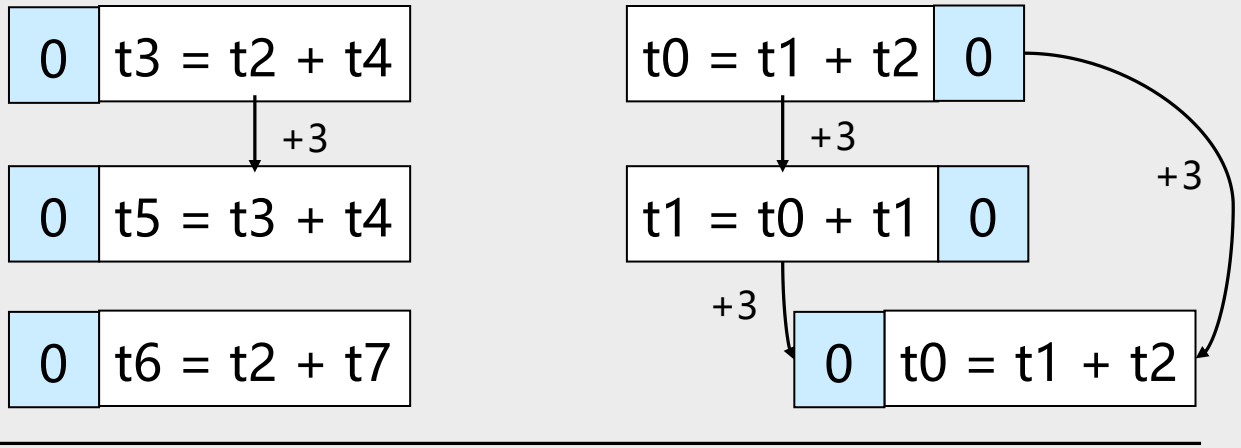
升级版的指令调度



升级版的指令调度

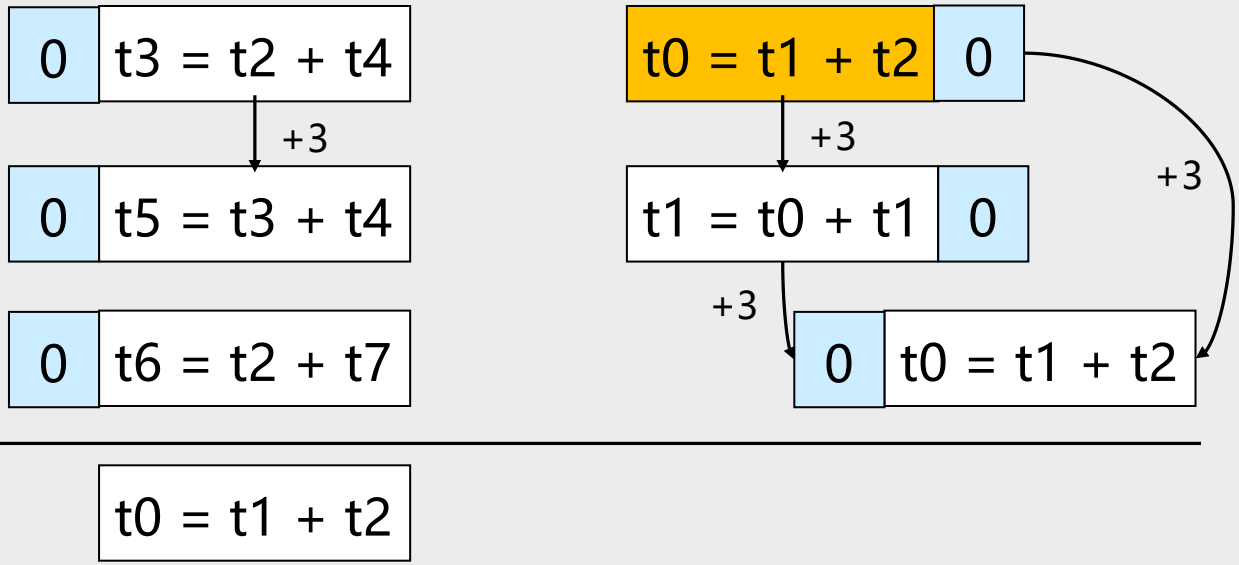


升级版的指令调度



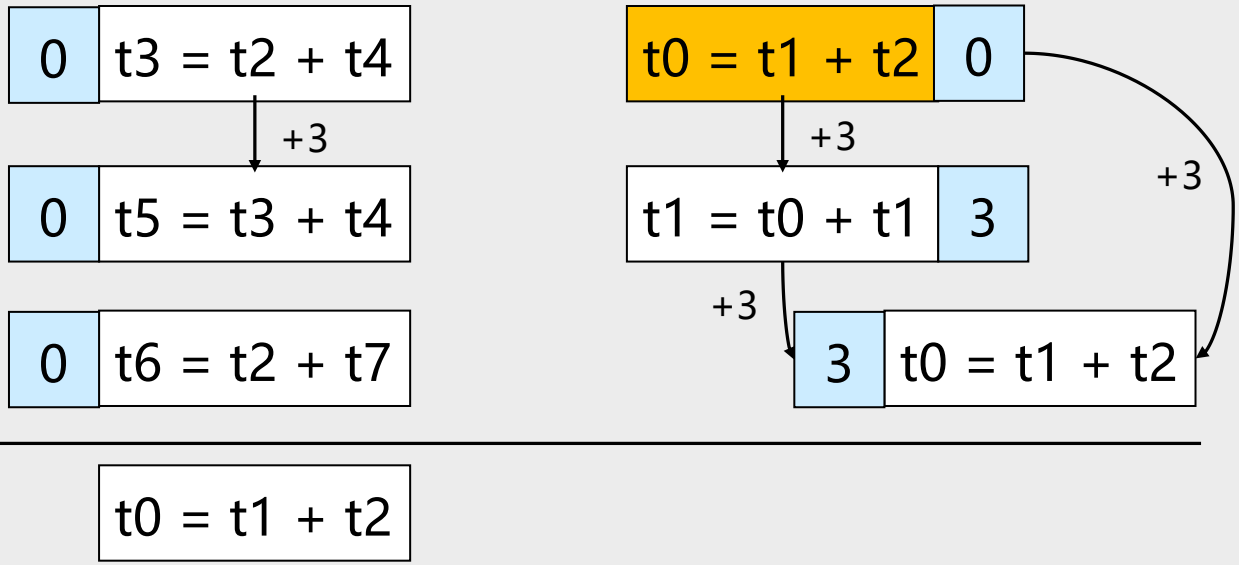
ID	RR	ALU	RW

升级版的指令调度



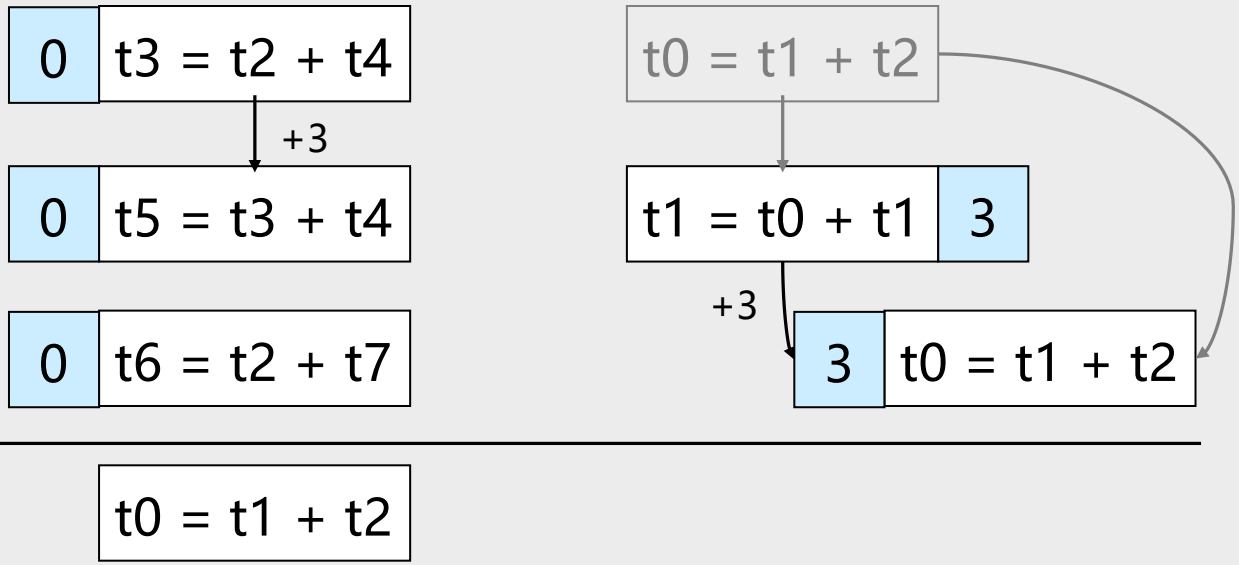
ID	RR	ALU	RW

升级版的指令调度



ID	RR	ALU	RW

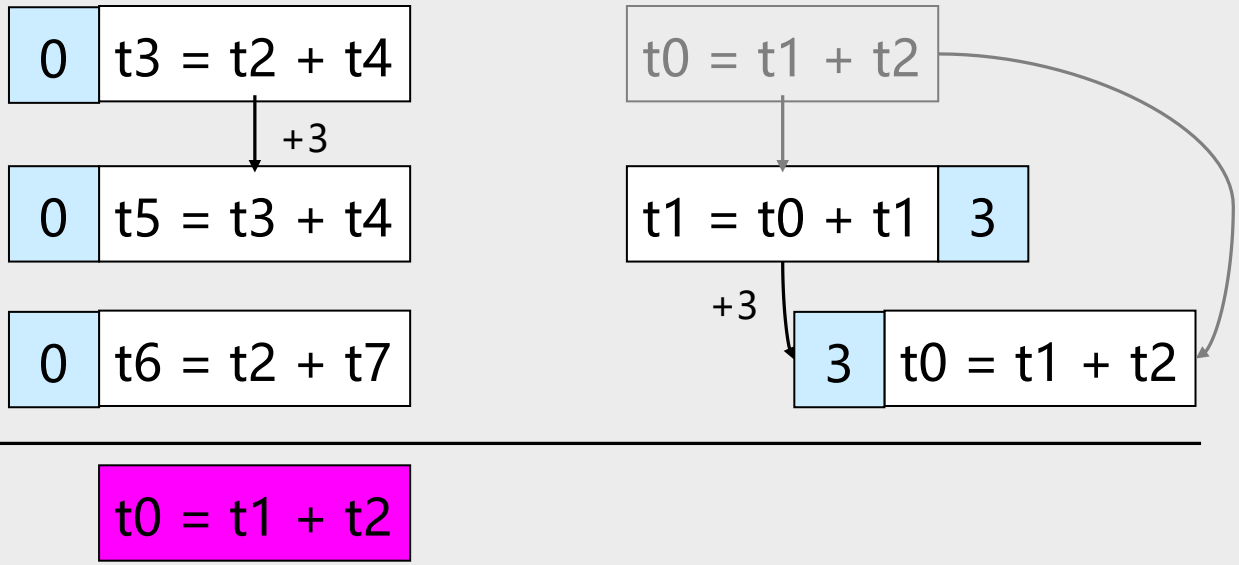
升级版的指令调度



t0 = t1 + t2

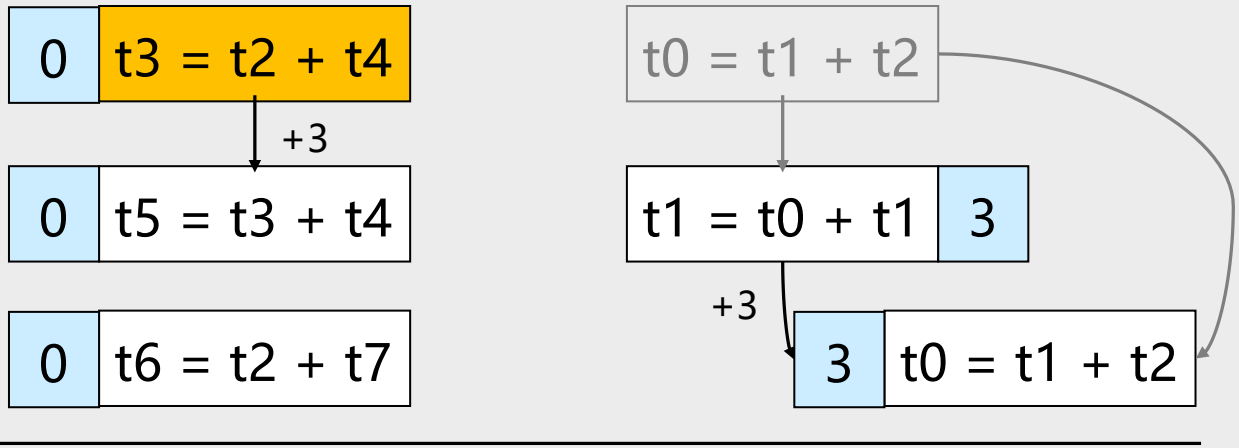
ID	RR	ALU	RW

升级版的指令调度



ID	RR	ALU	RW

升级版的指令调度

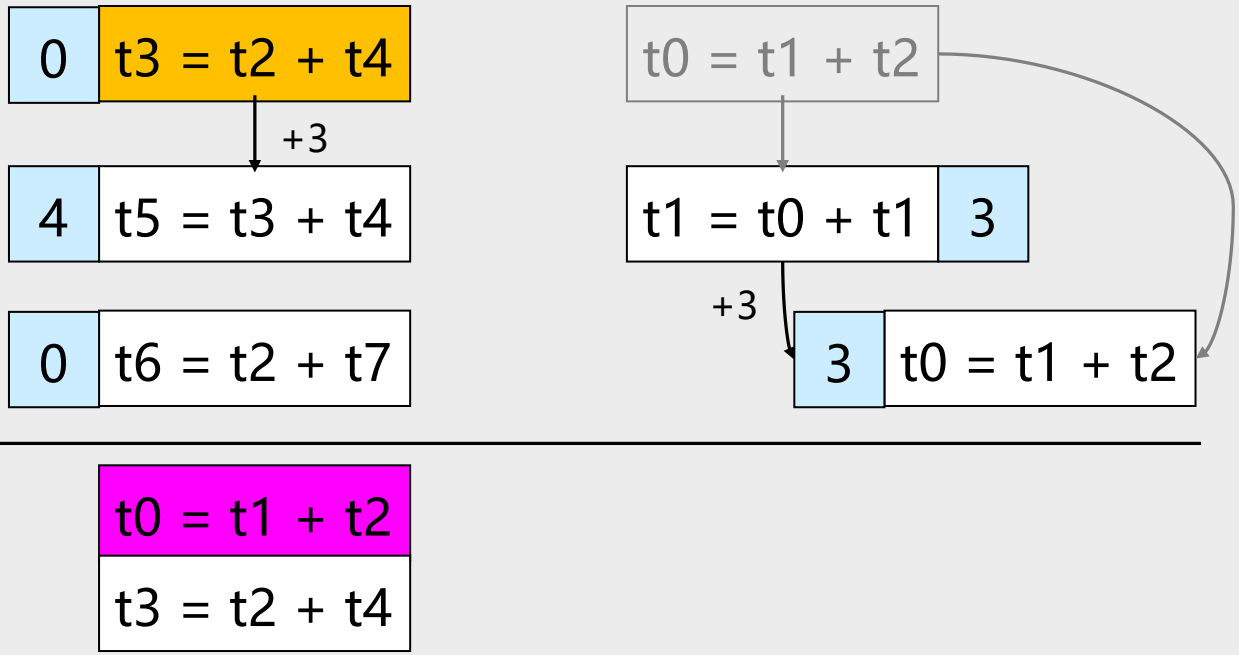


ID	RR	ALU	RW

t0 = t1 + t2

t3 = t2 + t4

升级版的指令调度

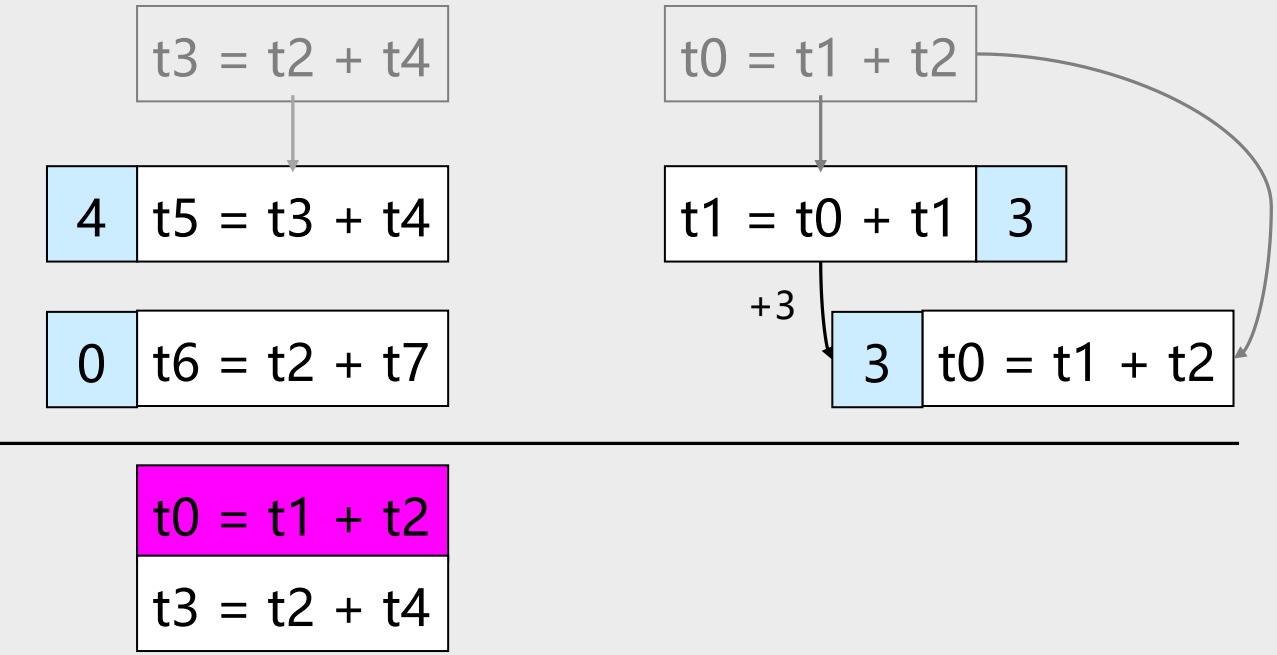


ID	RR	ALU	RW

t0 = t1 + t2

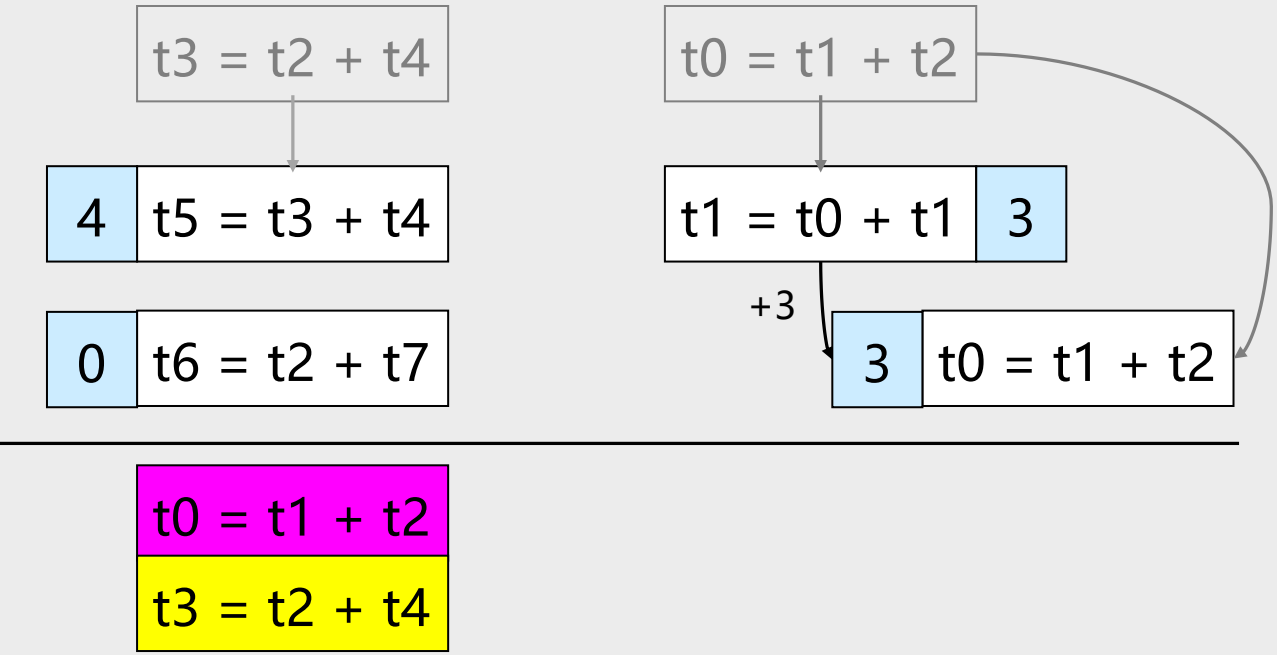
t3 = t2 + t4

升级版的指令调度



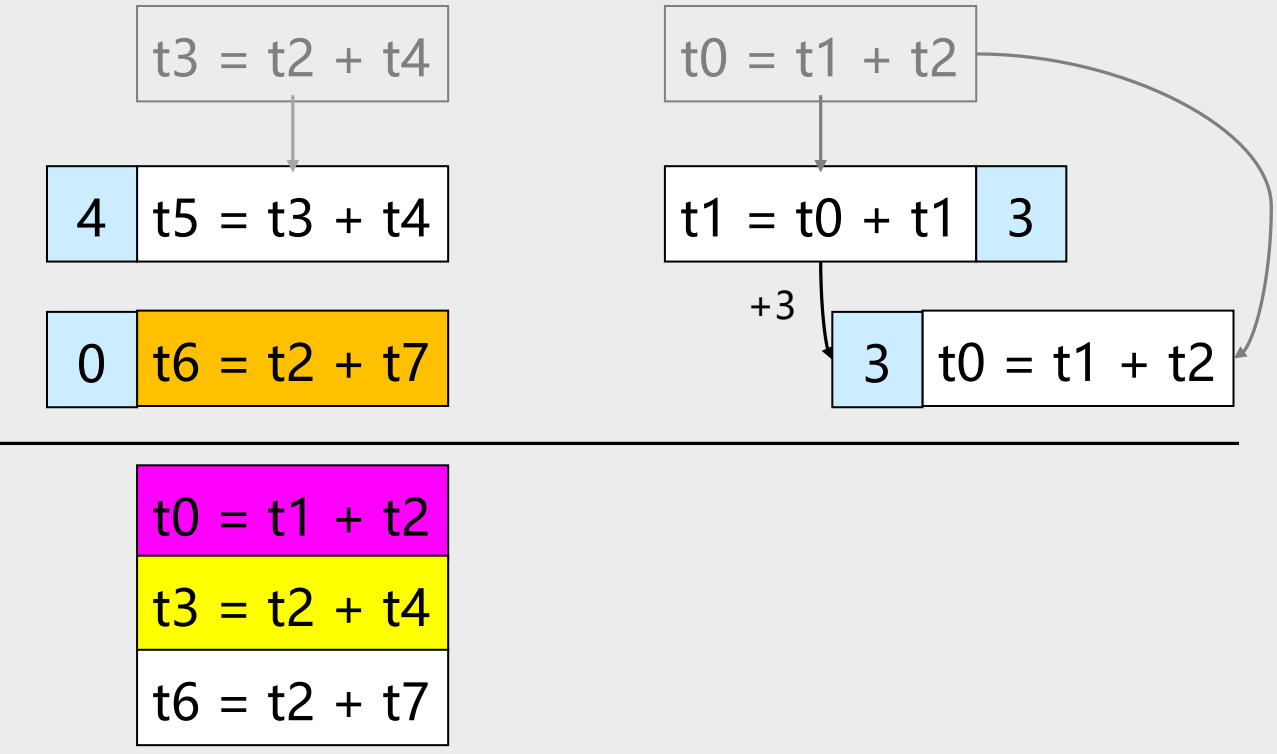
ID	RR	ALU	RW

升级版的指令调度



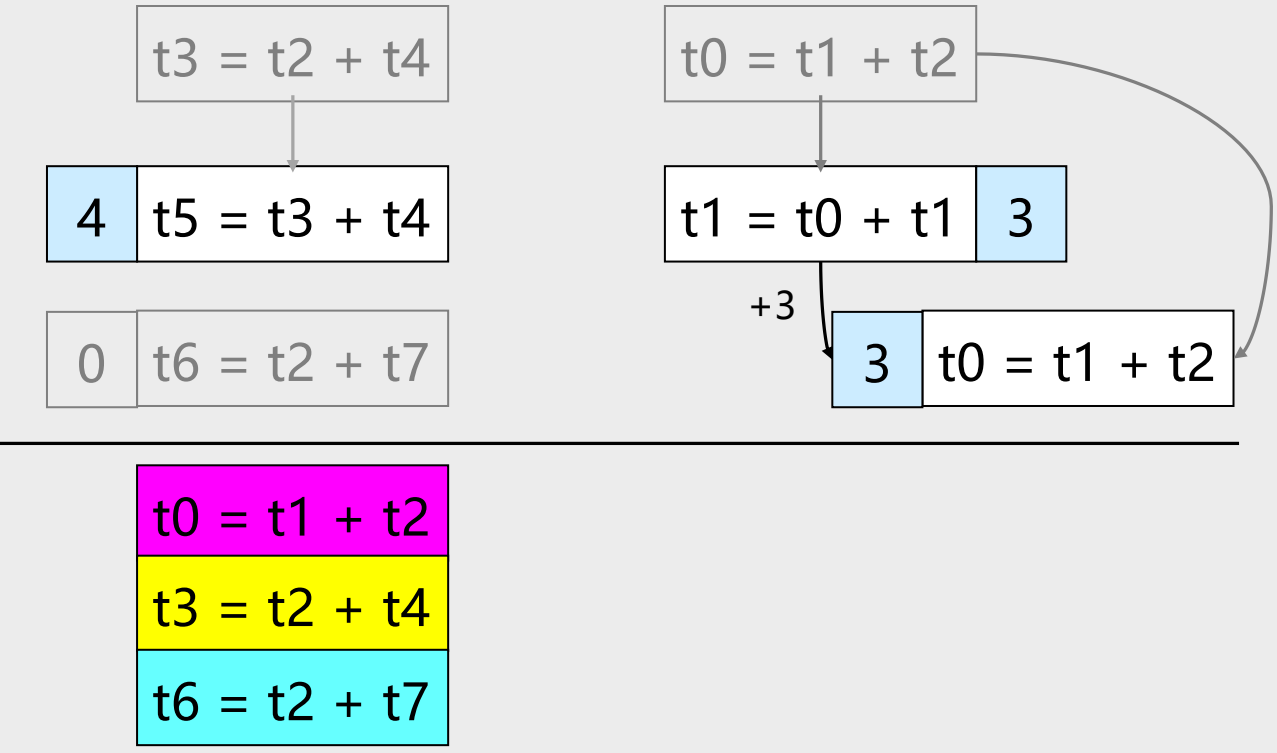
ID	RR	ALU	RW

升级版的指令调度



ID	RR	ALU	RW

升级版的指令调度



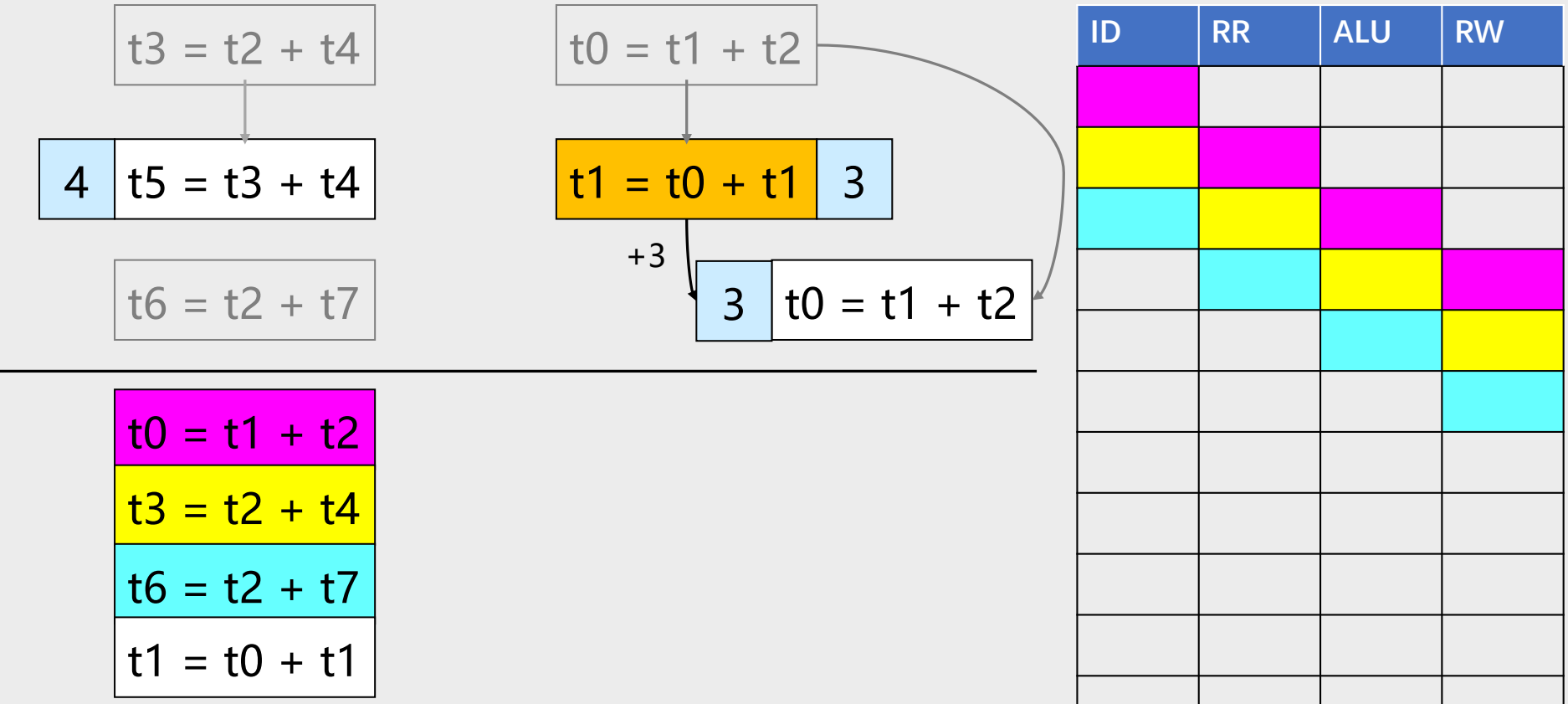
ID	RR	ALU	RW

t0 = t1 + t2

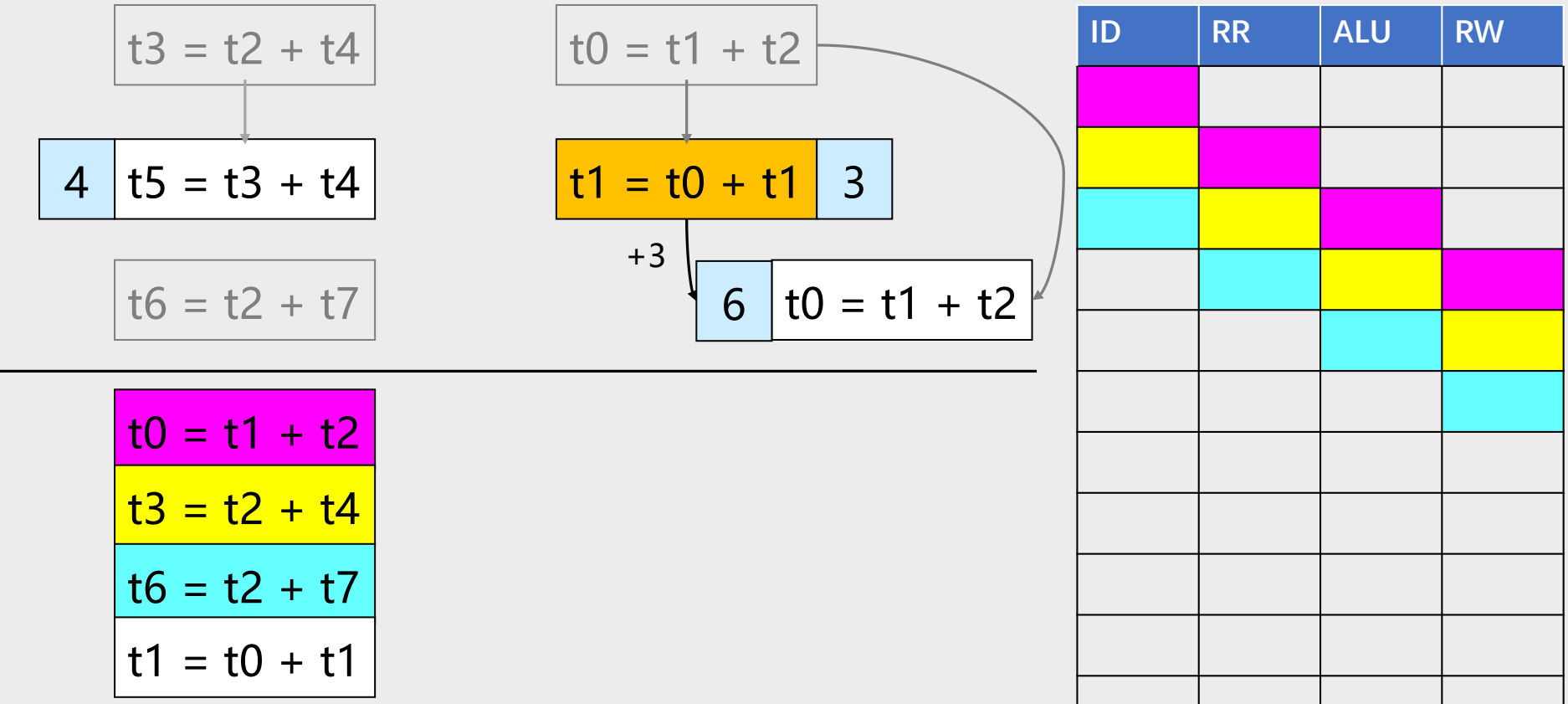
t3 = t2 + t4

t6 = t2 + t7

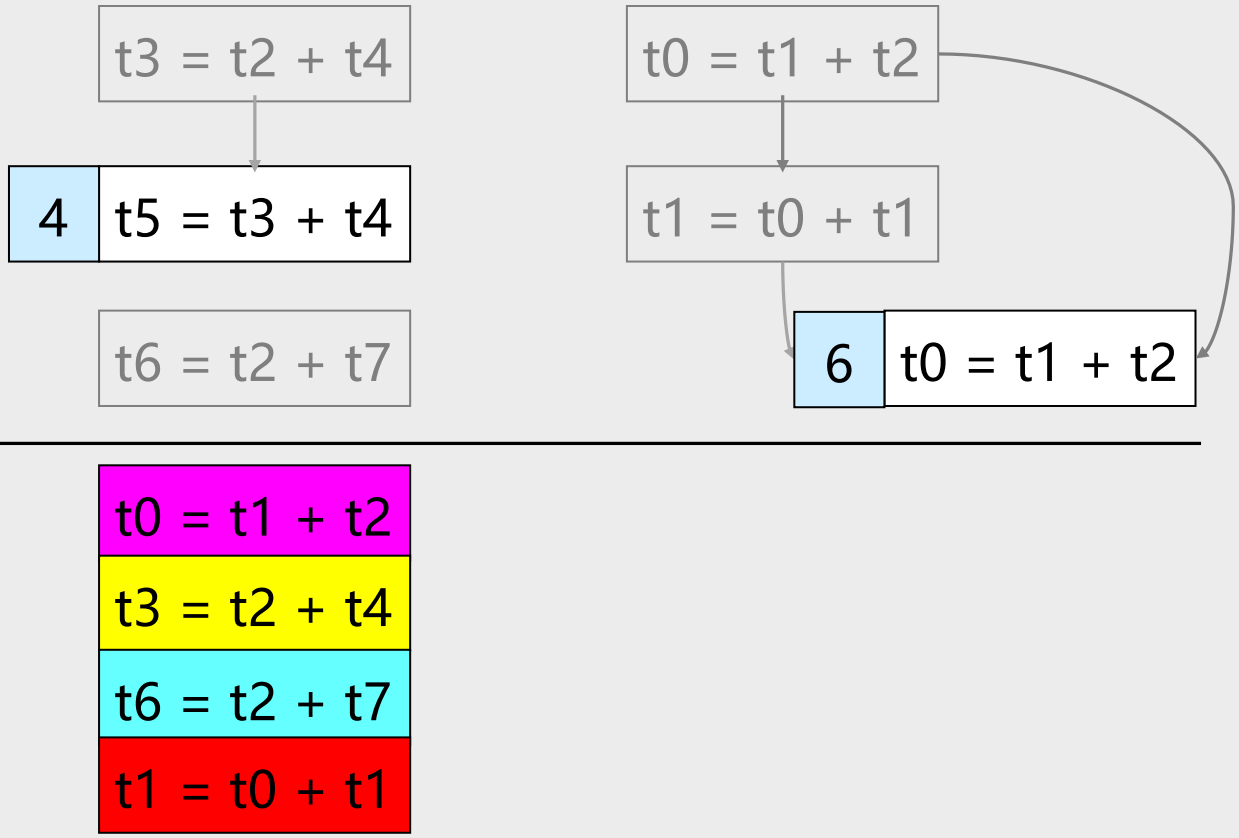
升级版的指令调度

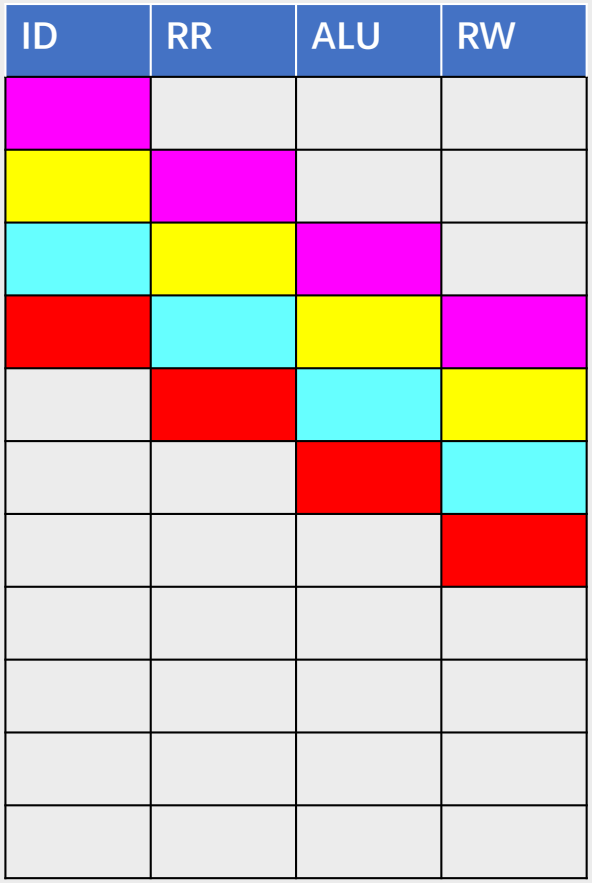

t3 = t2 + t4
t0 = t1 + t2
4
t5 = t3 + t4
t1 = t0 + t1
3
+3
3
t0 = t1 + t2
t6 = t2 + t7
t0 = t1 + t2
t3 = t2 + t4
t6 = t2 + t7
t1 = t0 + t1
ID
RR
ALU
RW

升级版的指令调度


t3 = t2 + t4
t0 = t1 + t2
4
t5 = t3 + t4
t1 = t0 + t1
3
+3
6
t0 = t1 + t2
t6 = t2 + t7
t0 = t1 + t2
t3 = t2 + t4
t6 = t2 + t7
t1 = t0 + t1
ID
RR
ALU
RW

升级版的指令调度


t3 = t2 + t4
t0 = t1 + t2
4
t5 = t3 + t4
t1 = t0 + t1
t6 = t2 + t7
6
t0 = t1 + t2
t0 = t1 + t2
t3 = t2 + t4
t6 = t2 + t7
t1 = t0 + t1


ID
RR
ALU
RW

升级版的指令调度

t3 = t2 + t4

4 t5 = t3 + t4

t6 = t2 + t7

t0 = t1 + t2

t1 = t0 + t1

6 t0 = t1 + t2

ID	RR	ALU	RW

t0 = t1 + t2

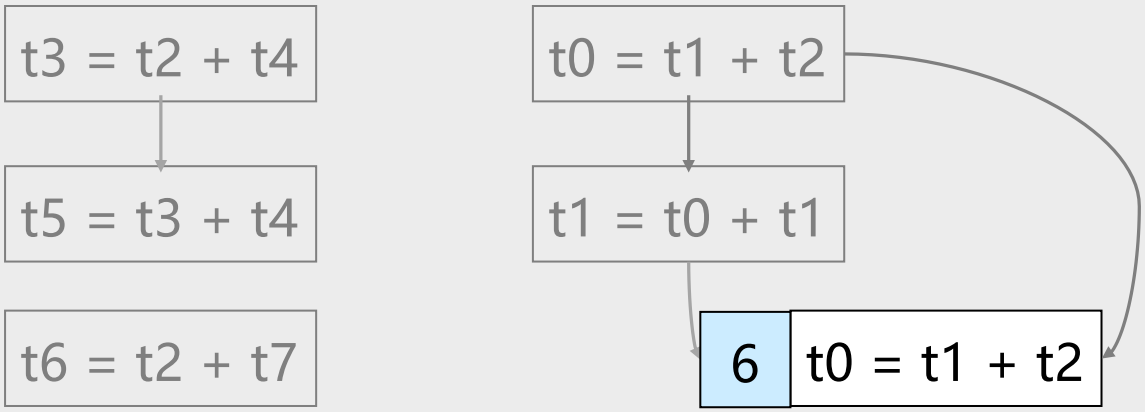
t3 = t2 + t4

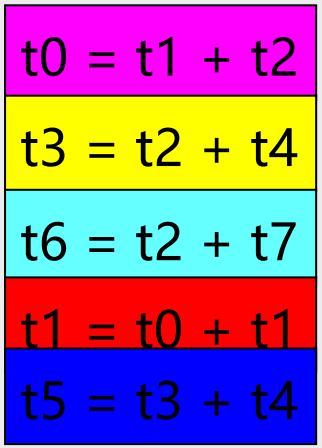
t6 = t2 + t7

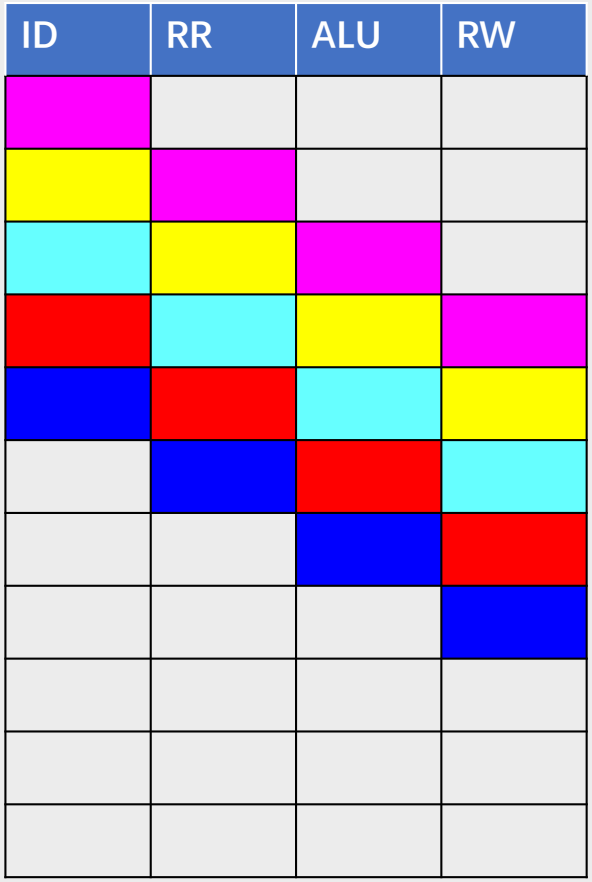
t1 = t0 + t1

t5 = t3 + t4

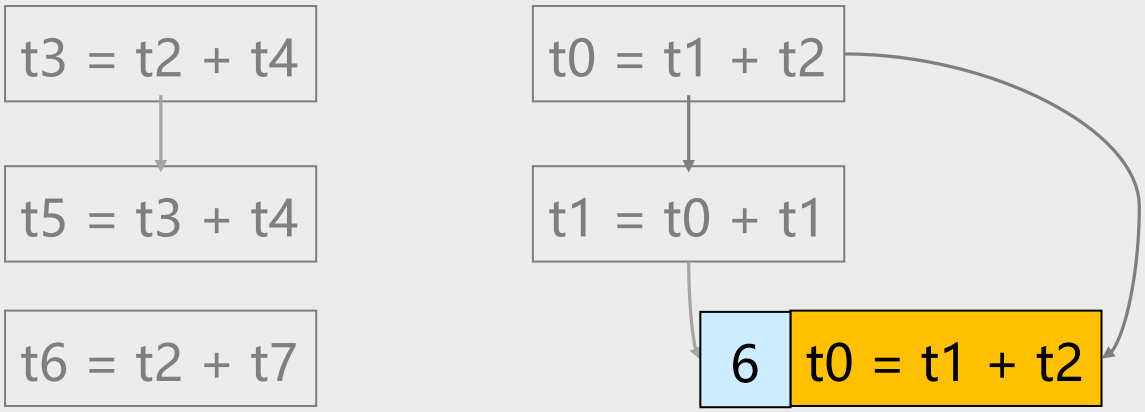
升级版的指令调度

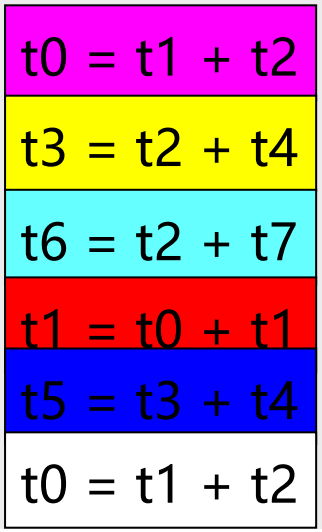

t3 = t2 + t4
t5 = t3 + t4
t6 = t2 + t7
t0 = t1 + t2
t1 = t0 + t1
6
t0 = t1 + t2

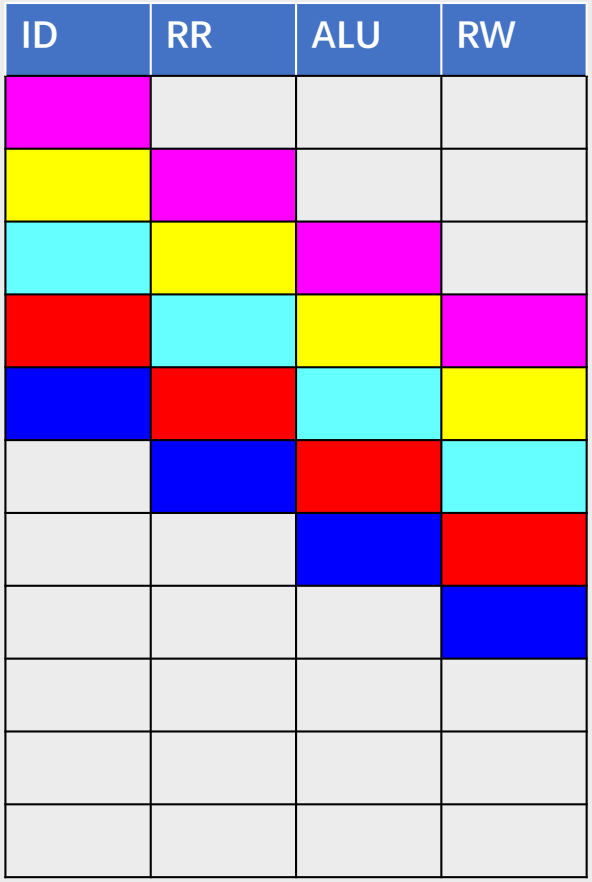

t0 = t1 + t2
t3 = t2 + t4
t6 = t2 + t7
t1 = t0 + t1
t5 = t3 + t4


ID
RR
ALU
RW

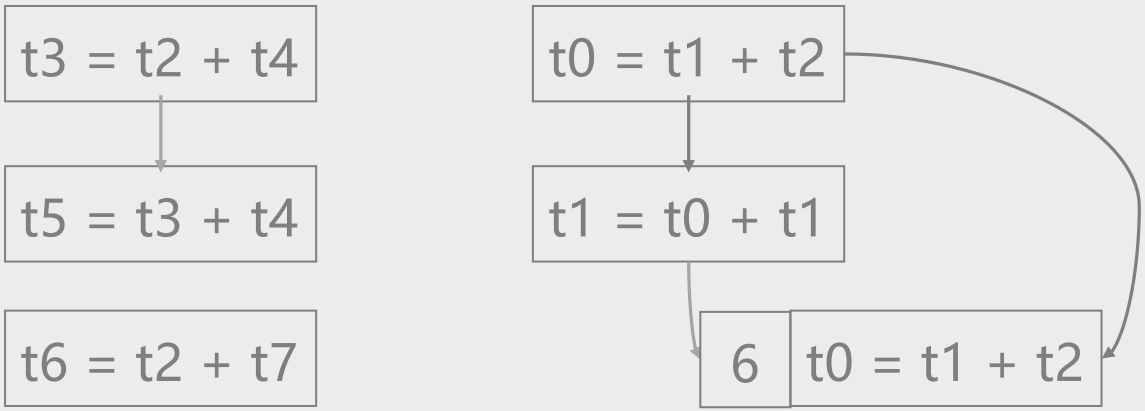
升级版的指令调度

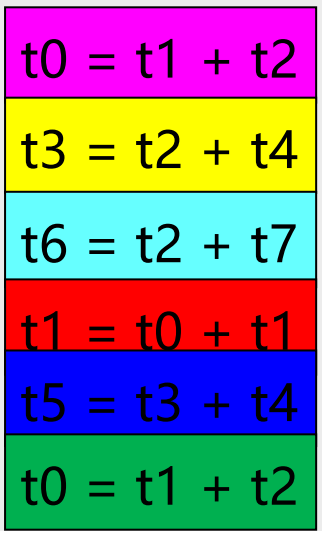

t3 = t2 + t4
t5 = t3 + t4
t6 = t2 + t7
t0 = t1 + t2
t1 = t0 + t1
6
t0 = t1 + t2

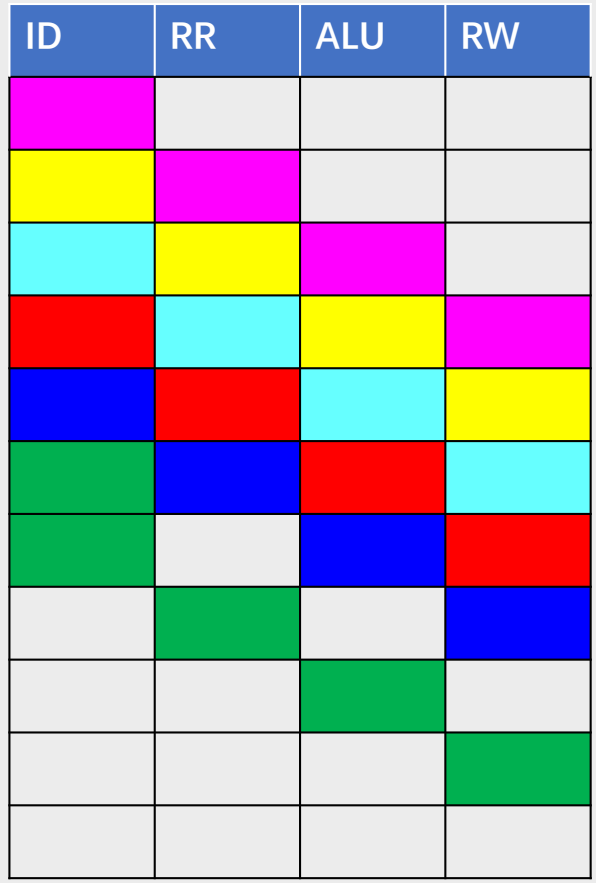

t0 = t1 + t2
t3 = t2 + t4
t6 = t2 + t7
t1 = t0 + t1
t5 = t3 + t4
t0 = t1 + t2


ID
RR
ALU
RW

升级版的指令调度


t3 = t2 + t4
t5 = t3 + t4
t6 = t2 + t7
t0 = t1 + t2
t1 = t0 + t1
6
t0 = t1 + t2


t0 = t1 + t2
t3 = t2 + t4
t6 = t2 + t7
t1 = t0 + t1
t5 = t3 + t4
t0 = t1 + t2


ID
RR
ALU
RW

不同调度之间的性能差距

t0 = t1 + t2

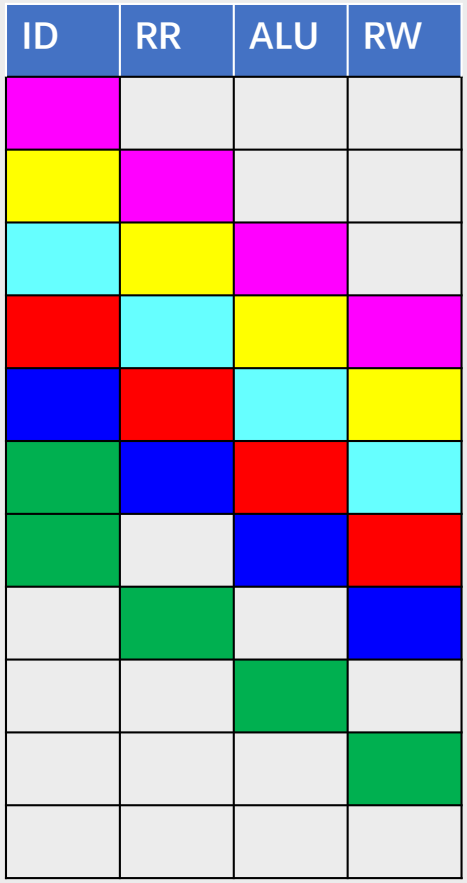
t3 = t2 + t4

t6 = t2 + t7

t1 = t0 + t1

t5 = t3 + t4

t0 = t1 + t2



t0 = t1 + t2

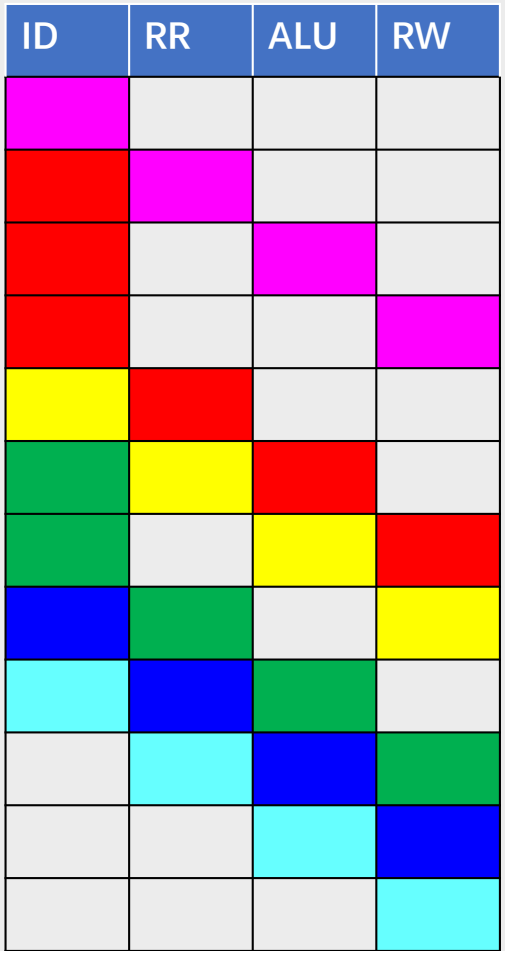
t1 = t0 + t1

t3 = t2 + t4

t0 = t1 + t2

t5 = t3 + t4

t6 = t2 + t7



《编译原理和技术》

机器相关的代码优化

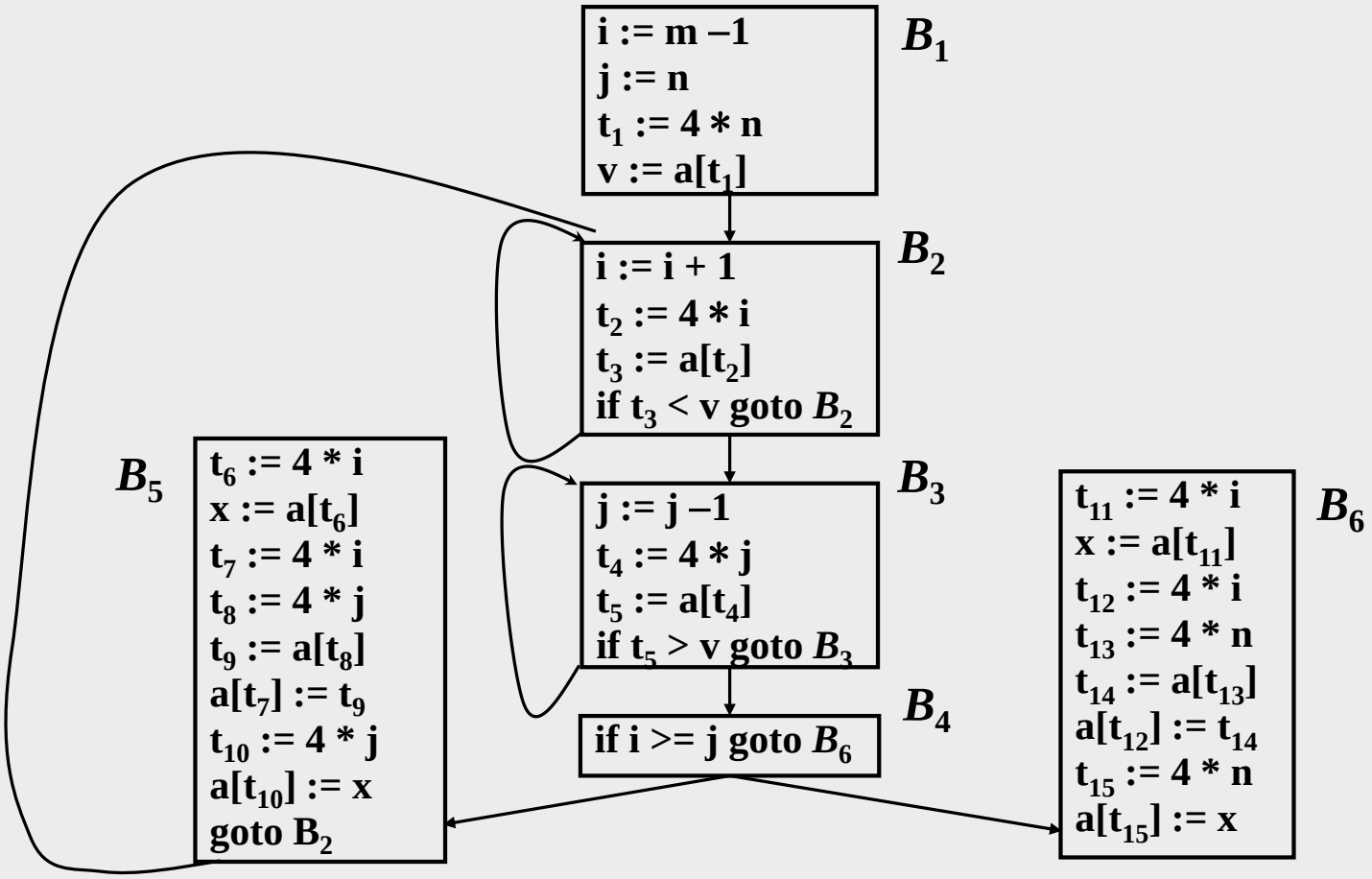
谢谢！

《编译原理和技术》

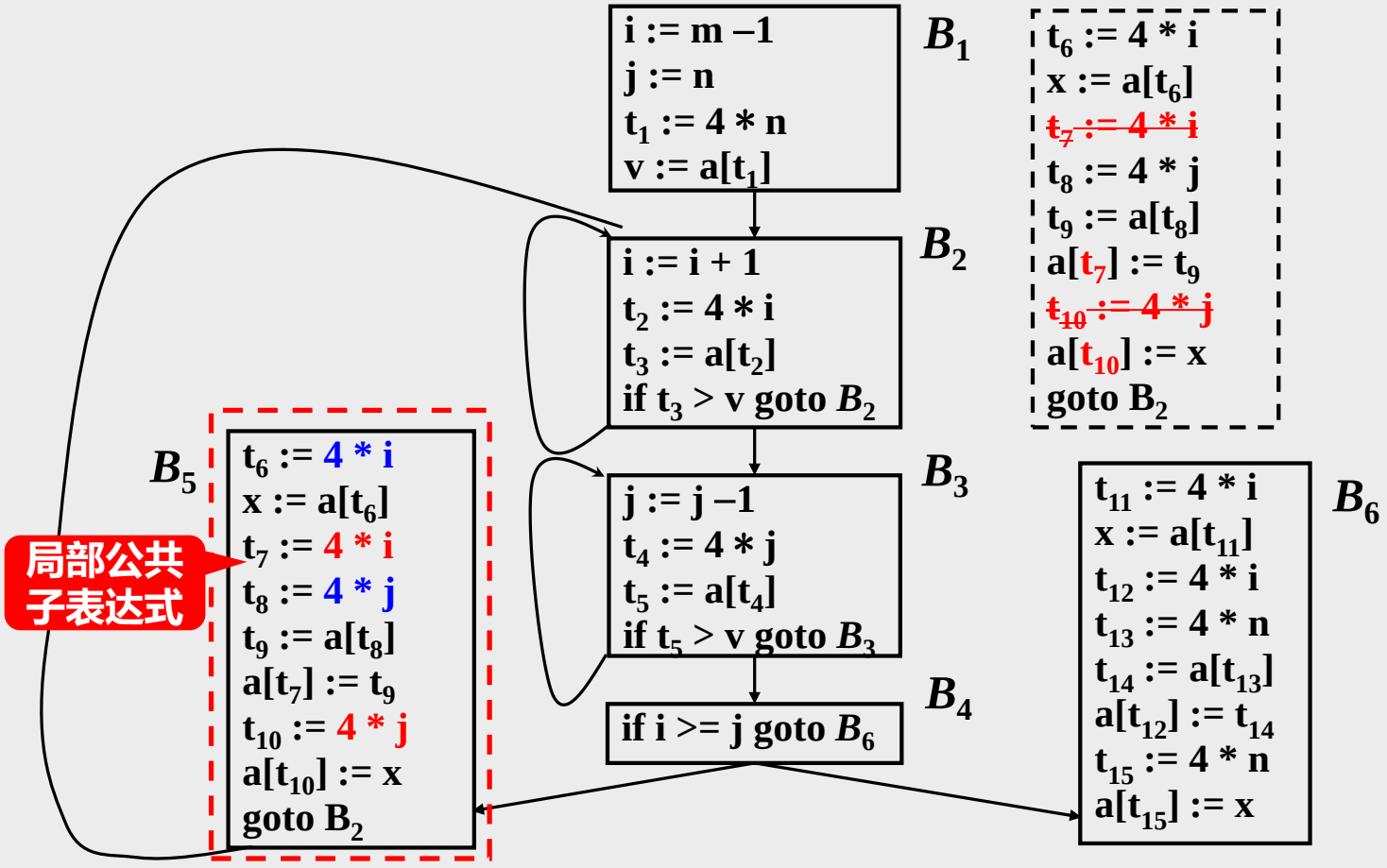
数据流优化之GVN

中科大计算机学院
李 诚
2022-11-28

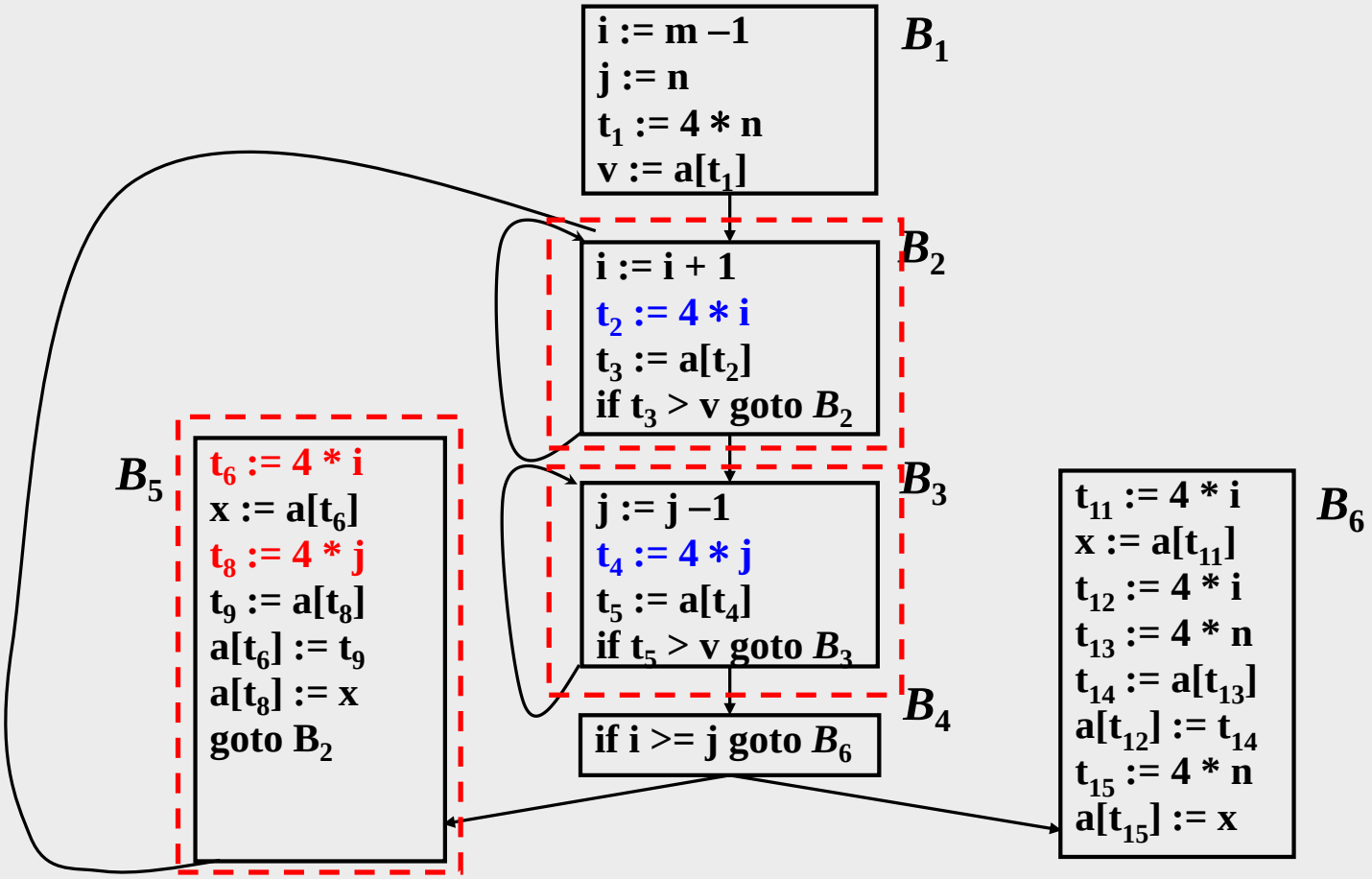
公共子表达式删除-回顾


i := m –1
j := n
t1 := 4 * n
v := a[t1]
B1
i := i + 1
t2 := 4 * i
t3 := a[t2]
if t3 < v goto B2
B2
j := j –1
t4 := 4 * j
t5 := a[t4]
if t5 > v goto B3
B3
if i >= j goto B6
B4
t6 := 4 * i
x := a[t6]
t7 := 4 * i
t8 := 4 * j
t9 := a[t8]
a[t7] := t9
t10 := 4 * j
a[t10] := x
goto B2
B5
t11 := 4 * i
x := a[t11]
t12 := 4 * i
t13 := 4 * n
t14 := a[t13]
a[t12] := t14
t15 := 4 * n
a[t15] := x
B6

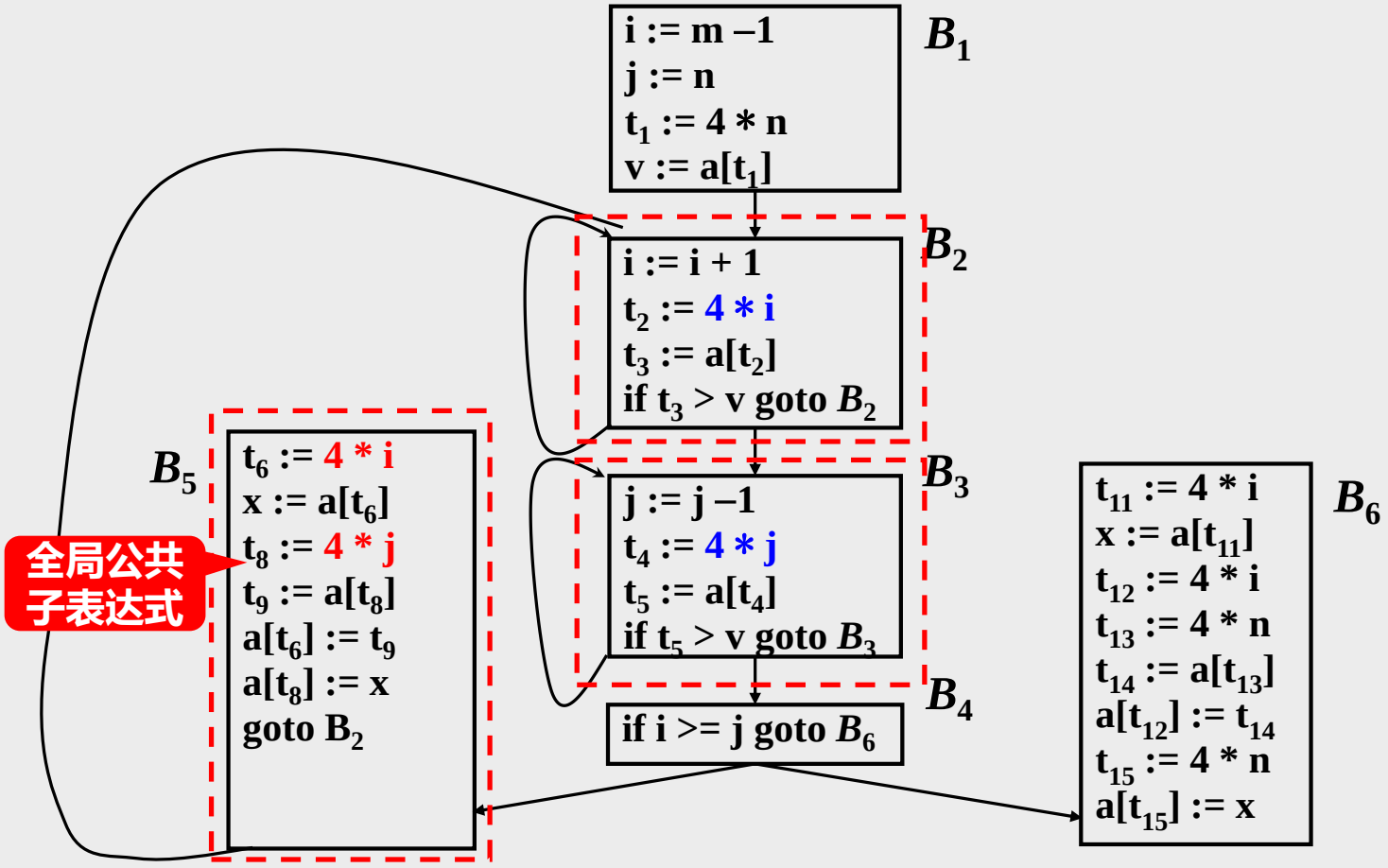
快排中的公共子表达式删除



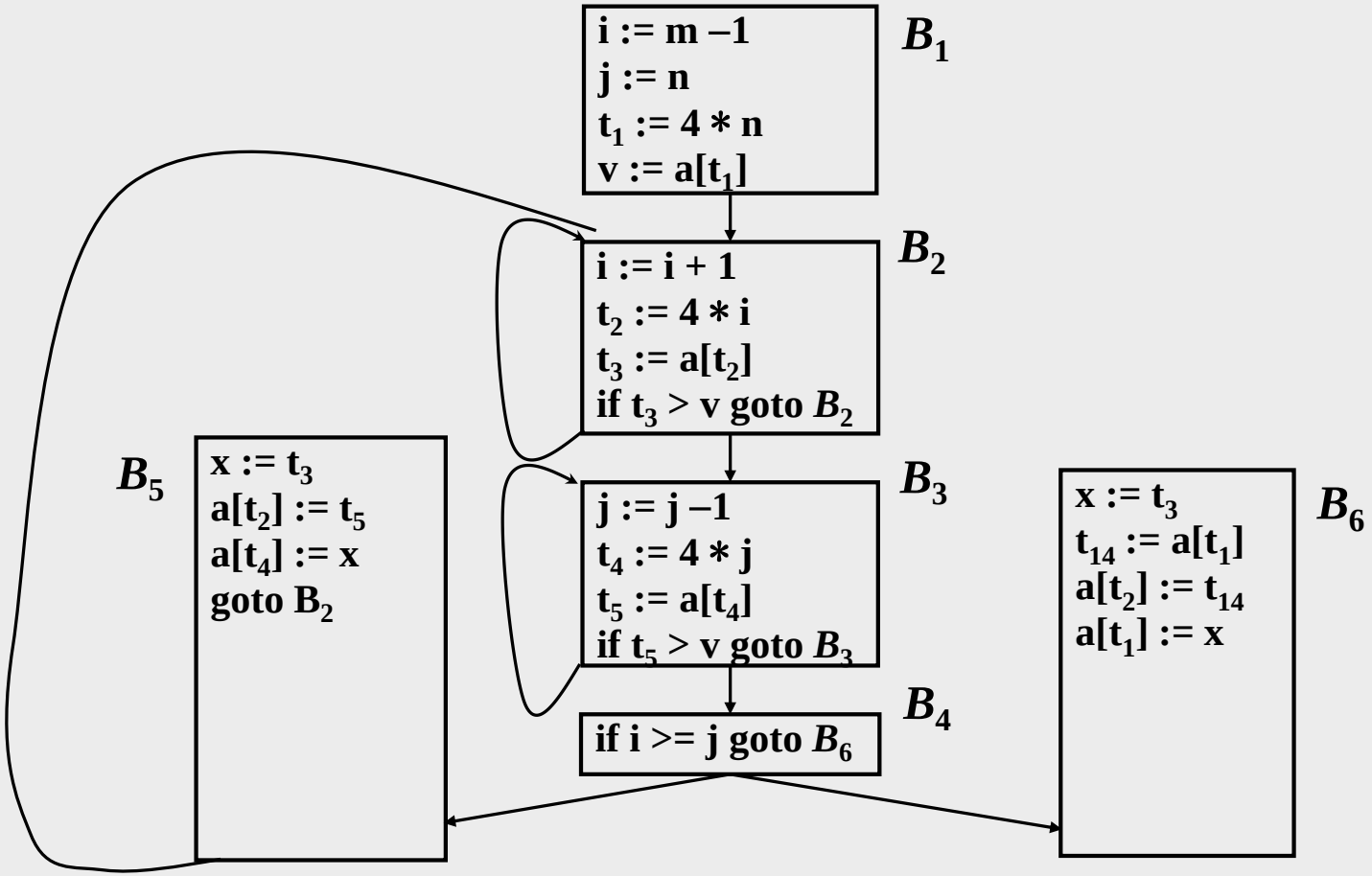
快排中的公共子表达式删除



快排中的公共子表达式删除



快排中的公共子表达式删除



如何实现冗余代码的删除?

- **关键在于冗余代码的识别**
  - 在程序点p和到达p的路径l，x = e，在l上p之前，是否存在一个与e计算结果相同的子表达式e’
- **通过值编号value numbering方法识别**
- **全局值编号global value numbering需要借助数据流分析方法**

值编号Value Numbering

- 考虑以下基本块代码

```
x = a + 37
y = x + 42
x = a + 37
y = x + 42
```

- 冗余计算是 a + 37 和 x + 42
- 最后两条指令的计算可以删除，直接用前面的x和y来代替

值编号Value Numbering

- 转变成静态单赋值格式SSA

```
x = a + 37          x1 = a + 37
y = x + 42          y1 = x1 + 42
x = a + 37          x2 = a + 37
y = x + 42          y2 = x2 + 42
```

- 每一次赋值产生一个新的变量
- 每一次引用使用唯一确定的变量
  - x2 = x1; y2 = y1
  - 如何识别？

值编号Value Numbering

- 在SSA中，为每一个变量或表达式的结果分配一个值编号v_i

```
x = a + 37
y = x + 42
x = a + 37
y = x + 42
```

```
x1 = a + 37

y1 = x1 + 42

x2 = a + 37

y2 = x2 + 42
```

```
a: v1
v1 + 37: v2 // a + 37
x1: v2
```

值编号Value Numbering

- 在SSA中，为每一个变量或表达式的结果分配一个值编号v_i

```
x = a + 37
y = x + 42
x = a + 37
y = x + 42
```

```
x1 = a + 37

y1 = x1 + 42

x2 = a + 37

y2 = x2 + 42
```

```
a:  v1
v1 + 37:  v2 // a + 37
x1:  v2

v2 + 42: v3 //x1 + 42
y1: v3
```

值编号Value Numbering

- 在SSA中，为每一个变量或表达式的结果分配一个值编号v_i

```
x = a + 37
y = x + 42
x = a + 37
y = x + 42
```

```
x1 = a + 37

y1 = x1 + 42

x2 = a + 37

y2 = x2 + 42
```

```
a:  v1
v1 + 37:  v2 // a + 37
x1:  v2

v2 + 42: v3 //x1 + 42
y1: v3

v1 + 37:  v2 // a + 37
x2:  v2
```

值编号Value Numbering

- 在SSA中，为每一个变量或表达式的结果分配一个值编号v_i

```
x = a + 37
y = x + 42
x = a + 37
y = x + 42
```

```
x1 = a + 37

y1 = x1 + 42

x2 = a + 37

y2 = x2 + 42
```

```
a:  v1
v1 + 37:  v2 // a + 37
x1:  v2

v2 + 42: v3 //x1 + 42
y1: v3

v1 + 37:  v2 // a + 37
x2:  v2

v2 + 42:  v3 // x2 + 42
y2: v3
```

值编号Value Numbering

- 在SSA中，为每一个变量或表达式的结果分配一个值编号v_i

```
x = a + 37
y = x + 42
x = a + 37
y = x + 42
```

```
x1 = a + 37

y1 = x1 + 42

x2 = a + 37

y2 = x2 + 42
```

```
a:  v1
v1 + 37:  v2 // a + 37
x1:  v2

v2 + 42: v3 //x1 + 42
y1: v3

v1 + 37:  v2 // a + 37
x2:  v2

v2 + 42:  v3 // x2 + 42
y2: v3
```

- 后面的值用前面的相同值编号的计算结果替代

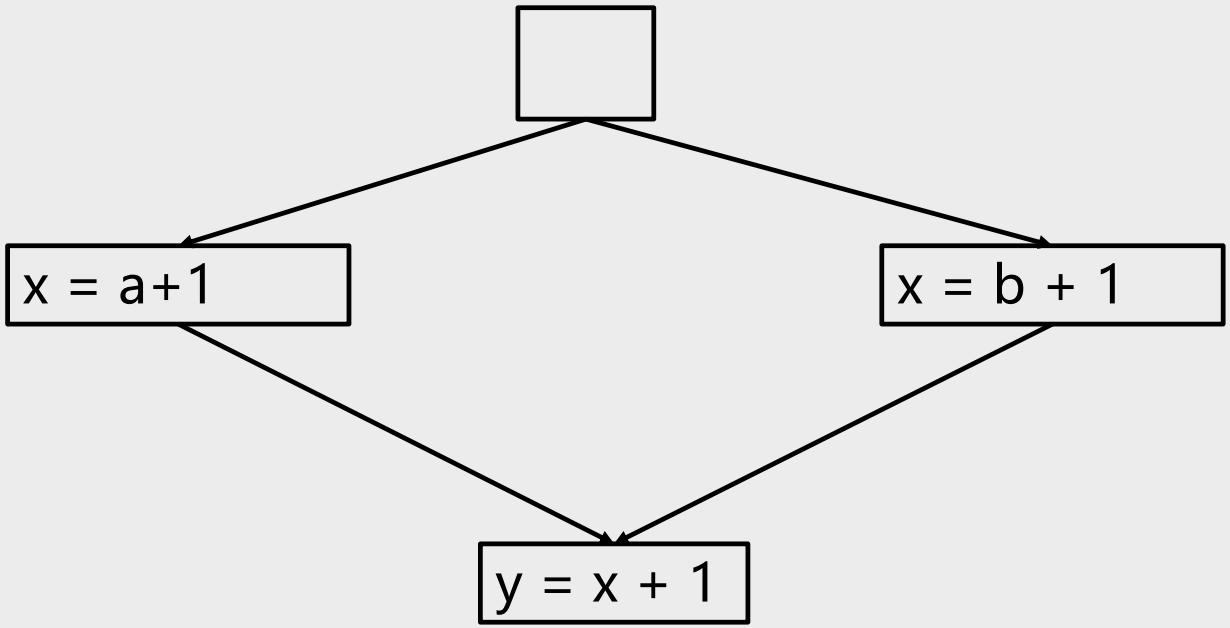
值编号Value Numbering

- 几个概念
  - 每一个a；37；a + 37；x1 + 42都是表达式
  - 每一个v1；v1 + 37；v2 + 42都是值表达式
  - 每一个集合{v1，a}；{v2，x1，v1+37}；都是一个等价类 equivalence class，记为C
  - 在程序点p，所有等价类的集合P={C1，C2，C3，…，Cn}叫做一个分区 partition

	a：v1 v1 + 37：v2 // a + 37
x1 = a + 37	x1：v2
y1 = x1 + 42	v2 + 42: v3 //x1 + 42 y1: v3
x2 = a + 37	v1 + 37：v2 // a + 37 x2：v2
y2 = x2 + 42	v2 + 42：v3 // x2 + 42 y2: v3

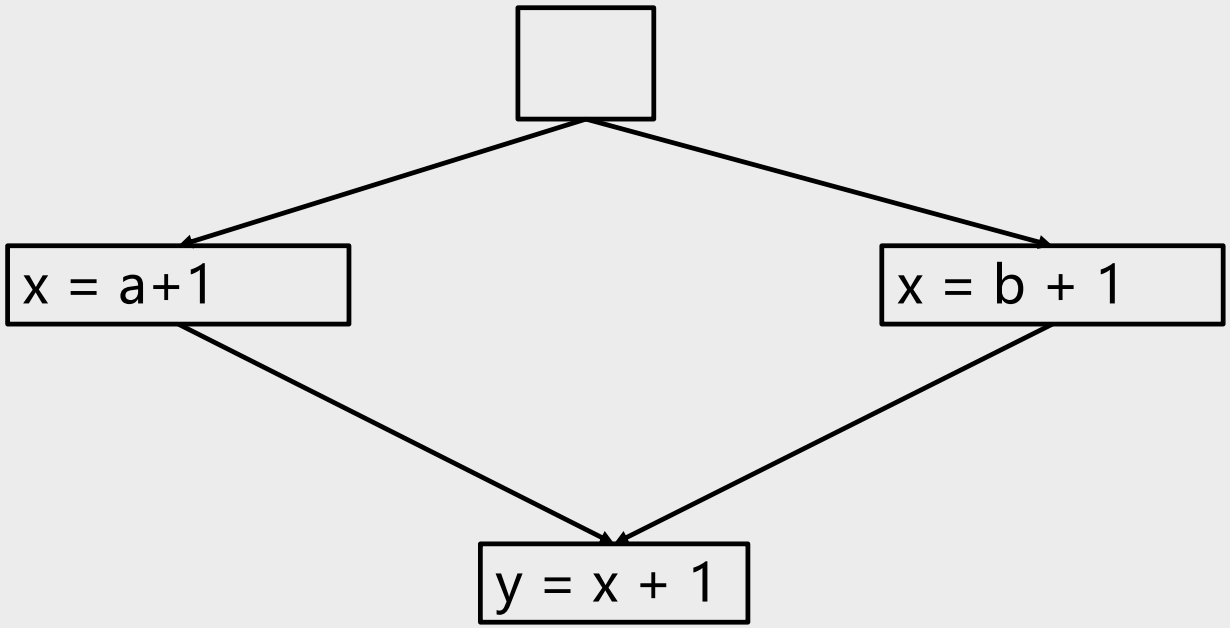
全局值编号Global Value Numbering

- 把上述在基本块内部的value numbering扩展到全程序，跨基本块



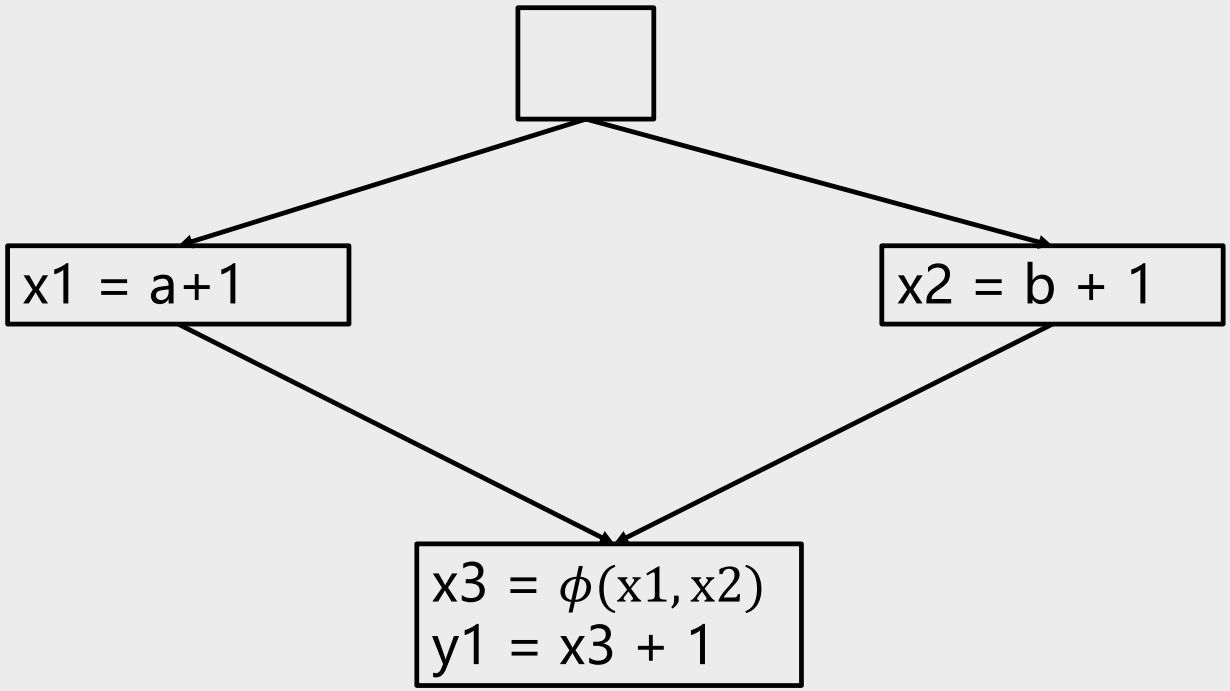
全局值编号Global Value Numbering

- 难点在于如何处理多条路径的汇聚？也就是说一个基本块有多个前驱节点的时候，如何值编码？



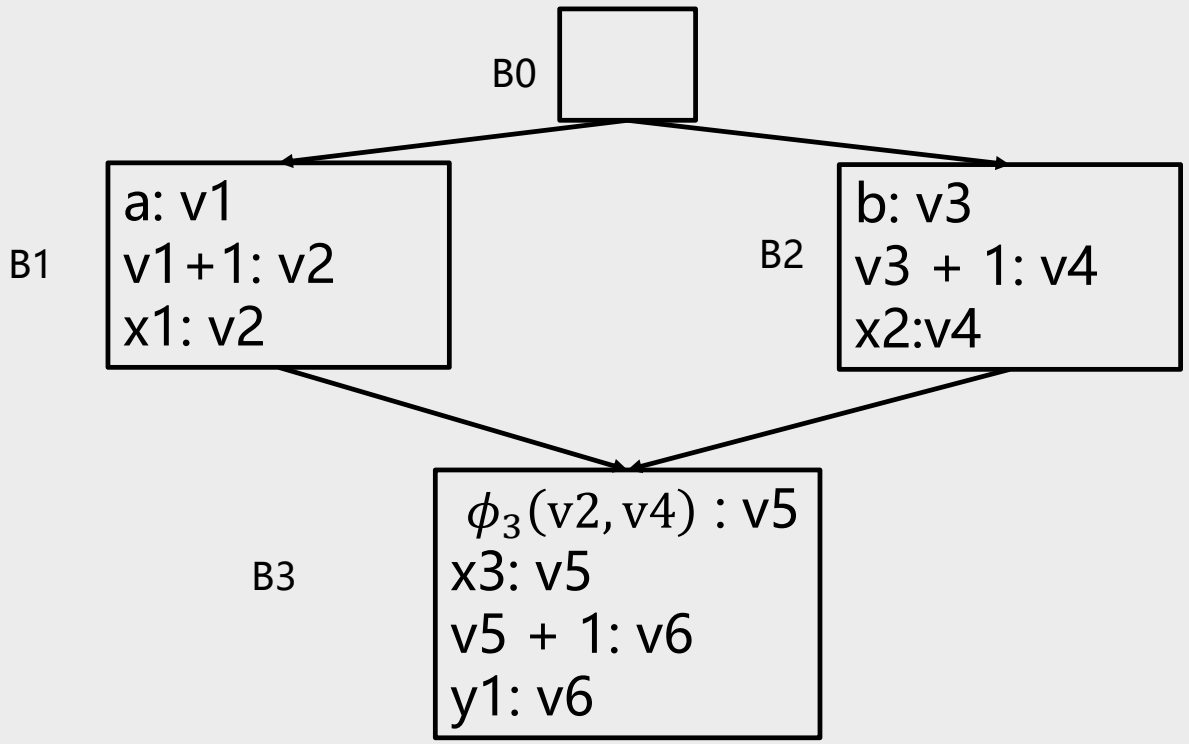
全局值编号Global Value Numbering

- 难点在于如何处理多条路径的汇聚？也就是说一个基本块有多个前驱节点的时候，如何值编码？



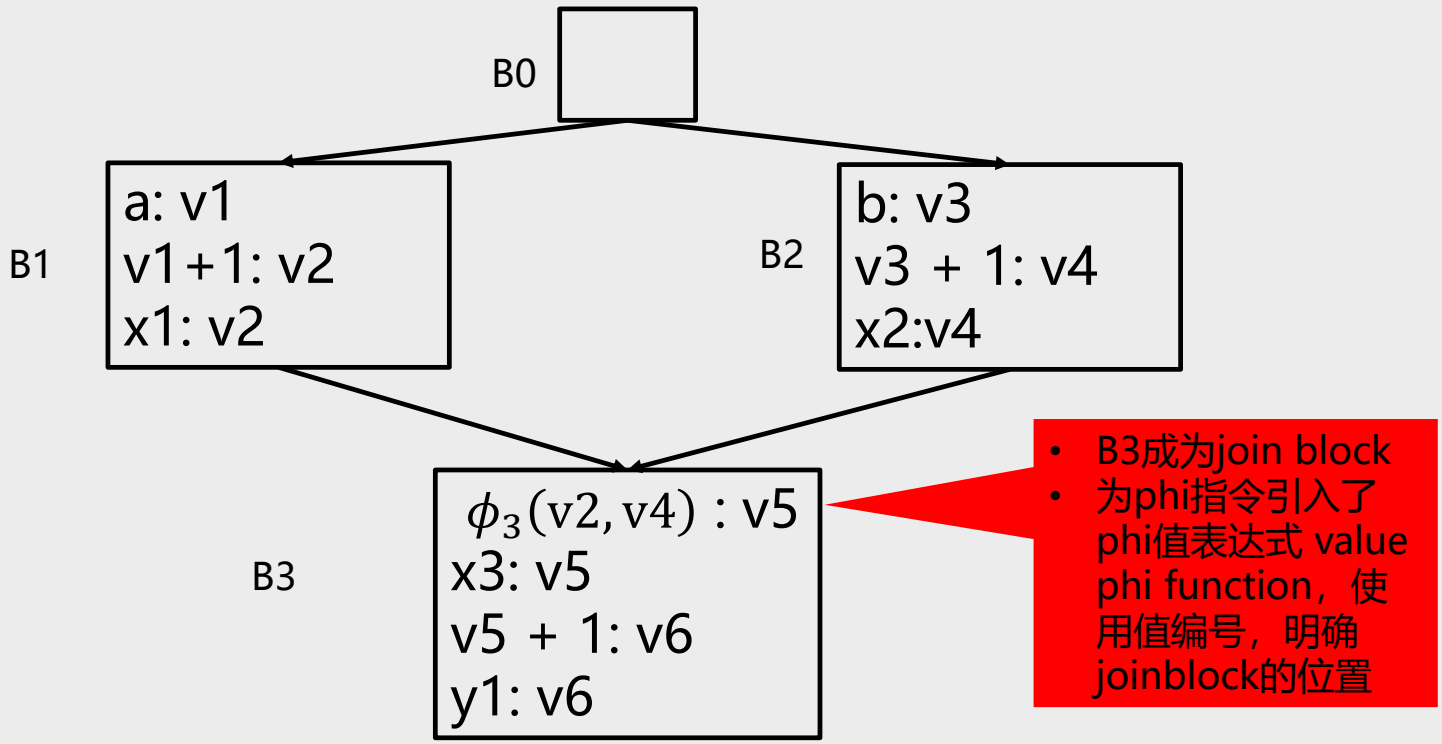
全局值编号Global Value Numbering

- 难点在于如何处理多条路径的汇聚？也就是说一个基本块有多个前驱节点的时候，如何值编码？



全局值编号Global Value Numbering

- 难点在于如何处理多条路径的汇聚？也就是说一个基本块有多个前驱节点的时候，如何值编码？



全局值编号Global Value Numbering

- 核心任务：沿着控制流图control flow graph，计算在每一个程序点（或者说，基本块的开始和结尾处）的表达式等价类分区
- 这些等价类既包含普通的值表达式，有包含复杂的phi值表达式
- 数据流分析是关键！

数据流分析模式

- 数据流值代表在任一程序点能观测到的所有可能程序状态集合的一个**抽象**
- 对于一个语句*s*
  - s之前的程序点对应的数据流值用IN[*s*]表示
  - s之后的程序点对应的数据流值用OUT[*s*]表示
- 对于一个基本块呢?

数据流分析模式

- **传递函数(transfer function) f**
  - 语句前后两点的数据流值受该语句的语义约束
  - 若沿执行路径正向传播，则$\mathrm{OUT}[s] = f_s(\mathrm{IN}[s])$
  - 若沿执行路径逆向传播，则$\mathrm{IN}[s] = f_s(\mathrm{OUT}[s])$

若基本块B由语句$s_1, s_2, \ldots, s_n$依次组成，则

- $\mathrm{IN}[s_{i+1}] = \mathrm{OUT}[s_i]$，$i = 1, 2, \ldots, n-1$

考虑的是在语句执行后输入输出之间的变化关系

基本块上的数据流模式

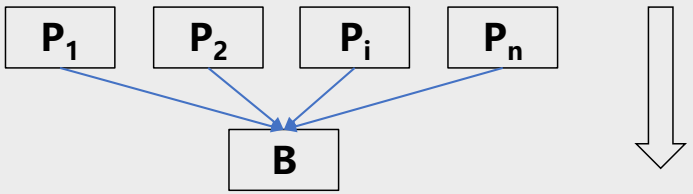
- IN[B]：紧靠基本块B之前的数据流值
  - IN[B] = IN[s_1]
- OUT[B]：紧靠基本块B之后的数据流值
  - OUT[B] = OUT[s_n]
- f_B :基本块B的传递函数
  - 前向数据流： OUT[B] = f_B (IN[B])
    - $f_B = f_n \circ \ldots \circ f_2 \circ f_1$
  - 逆向数据流： IN[B] = f_B (OUT[B])
    - $f_B = f_1 \circ \ldots \circ f_{n-1} \circ f_n$

基本块间的数据流分析模式

- 控制流约束
  - 正向传播

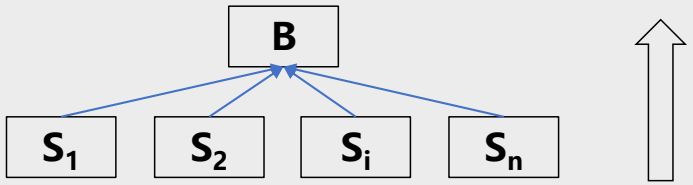
$$IN[B] = \cup_{P\text{是}B\text{的前驱}} OUT[P]$$

U 是汇合的意思，并不一定代表并集，也可能是交集等运算



  - 逆向传播

$$OUT[B] = \cup_{S\text{是}B\text{的后继}} IN[S]$$



- 约束方程组的解通常不是唯一的
  - 求解的目标是要找到满足这两组约束（控制流约束和迁移约束）的最“精确”解

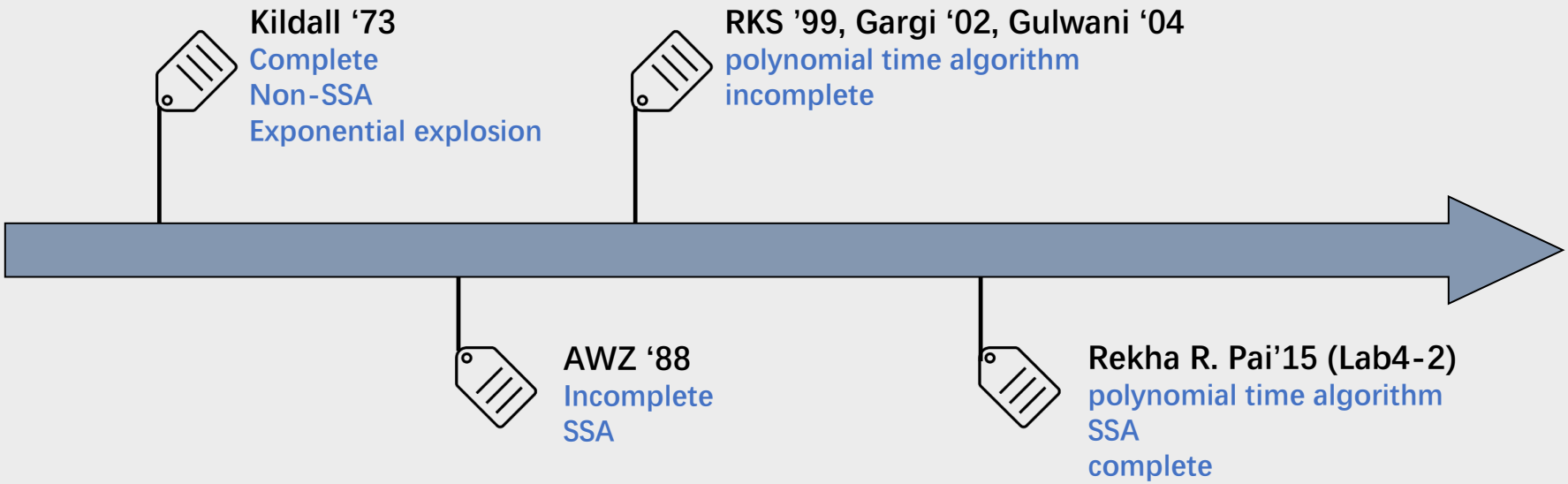
考虑的是在其他语句或块对于输入的影响和本次执行的输出对其他语句和块的影响

GVN的数据流分析方法

- **正向传播，从前驱向后继结点传播信息，先算IN，再算OUT**
- **数据流值，即状态**
  - **IN/OUT**：分别对应基本块入口处和出口处的等价类分区
- **传递函数f**
  - 对于每一条指令s，OUT[s] = f(IN[s])
- **约束方程组**
  - IN[B] = ∪ $_{P\text{是}B\text{的前驱}}$**OUT[*P*]** //这里需要考虑一个**join operator**
  - OUT[*B*] = f_B (IN[*B*]) //考虑到基本块内若干指令的f串起来

GVN算法及伪代码讲解

GVN 算法发展


Kildall ‘73
Complete
Non-SSA
Exponential explosion
RKS ’99, Gargi ‘02, Gulwani ‘04
polynomial time algorithm
incomplete
AWZ ‘88
Incomplete
SSA
Rekha R. Pai’15 (Lab4-2)
polynomial time algorithm
SSA
complete

参考论文

- **Detection of Redundant Expressions: A Complete and Polynomial-Time Algorithm in SSA**

基本概念

- **程序及IR假设**
  - 假设程序是SSA IR格式
  - 流图中有entry和exit基本块，均为空
  - 每个基本块至多有两个前驱结点
- **join block**
  - 有两个前驱的基本块

基本概念

- **表达式expression，有如下形态**
  - 常量
  - 变量
  - x⊕y，x, y是常量或变量， ⊕是二元运算符
  - $\phi(x1, x2)$

基本概念

- **表达式等价 Herbrand equivalence [1]**
  - 如果两个表达式的运算符相同，且操作数也是Herbrand等价的，那么他们是Herbrand等价的。
  - 仅考虑结构等价，递归判断
- **对phi表达式的等价判断需要考虑每条路径等价，因此，需要对phi指令进行额外处理**

[1] source: The Value Flow Graph - A Program Representation for Optimal Program Transformations

基本概念

- 对phi指令的特殊处理：数据流分析中，phi 指令视作为 join block 的前驱的copy指令进行处理

例如：

```
bb1:
        x1 = 1 + 1
        br bb3
bb2:
        x2 = 2 + 2
        br bb3
bb3:
        x3=phi(x1,x2)
        ...
```

```
bb1:
        x1 = 1 + 1
        x3 = x1
        br bb3
bb2:
        x2 = 2 + 2
        x3 = x2
        br bb3
bb3:
        ...
```

基本概念

- **值表达式value expression，有如下形态**
  - $v_i \oplus v_j = \{x \oplus y \mid x \in C_i,\ y \in C_j,$ Ci和Cj是两个等价类，分别对应值编号vi和vj$\}$ //简写为ve
  - $\phi_k(v_i, v_j)$ //value phi function，简写为vpf
- **一个值表达式对应具有相同值编码的一组表达式的集合**

基本概念

- **等价类equivalence class**
  - $vr, x1, y1$
  - $vs, z1, vr+1$
  - $vm, xn : \phi_k(vi, vj)$
- **分区partition**
  - $P = \{ \cdots ; \{vr, x1, y1\}, \{vs, z1, vr+1\}, \{vm, xn : \phi_k(vi, vj)\}, \cdots \}$

数据流分析伪代码

```
detectEquivalences(G){
        PIN[B1] = ∅ // “B1” is the first block
        POUT[B1] = transferFunction(PIN[B1] )
       for each block
            POUT[B] = ⊤ // 特殊符号top
       while any POUT changes
            for each block B
                 if B has two predecessors
                       then PIN[B] = Join(POUT[P1], POUT[P2])
                 else PIN[B] = POUT[P]
                 POUT[B] = transferFunction(PIN[B])
}
```

数据流分析伪代码

- **对于基本块中的每一条指令s，传递函数如右所示**

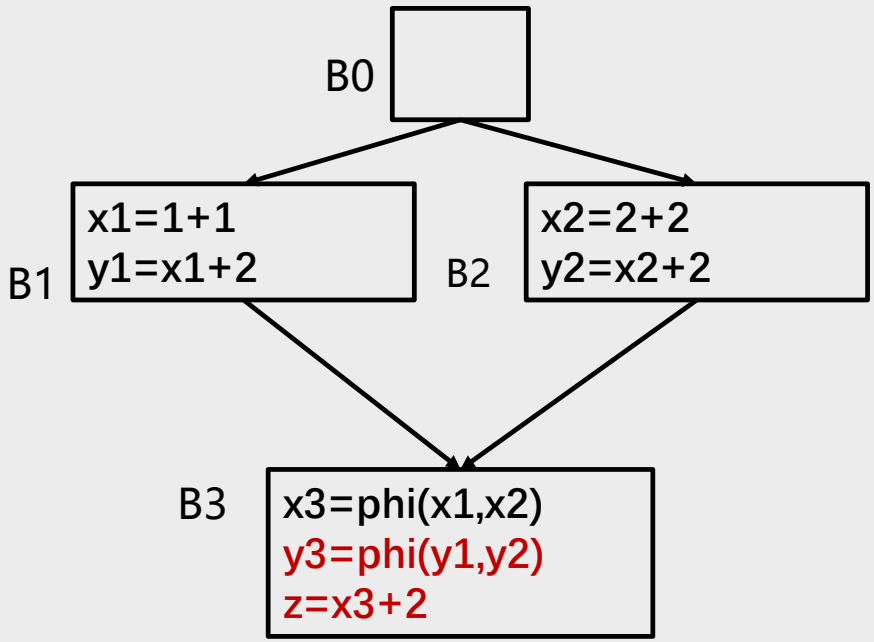
```
transferFunction(x = e, PIN[s]){
    POUT[s] = PIN[s]
    if x is in a class Ci in POUT[s]
        then Ci = Ci - {x}
    ve  = valueExpr(e) //找到值表达式
    vpf = valuePhiFunc(ve, PIN[s]) //找到值phi表达式

    if ve or vpf is in a class Ci in POUT[s]
       then Ci = Ci ∪ {x, ve} // 建立映射
    else POUT[s] = POUT[s] ∪ {vn, x, ve : vpf }
          // vn是新值编号
     return POUT[s]
}
```

数据流分析伪代码

valuePhiFunc(*ve*, P[B]){ //P是基本块B的分区
  if *ve* is of the form $\phi_k(v_{i1}, v_{j1}) \oplus \phi_k(v_{i2}, v_{j2})$
    then *vi* = getVN(POUT[kl], $v_{i1} \oplus v_{i2}$) //左前驱
      if (*vi* = = NULL)
        then *vi* = valuePhiFunc($v_{i1} \oplus v_{i2}$, POUT[kl])
      vj = getVN(POUT[kr], $v_{j1} \oplus v_{j2}$) //右前驱
      if (*vj* == NULL)
        then *vj* = valuePhiFunc($v_{j1} \oplus v_{j2}$, POUT[kr])
  return $\phi_k(vi, vj)$ // *vi*, *vj* are non-NULL
}

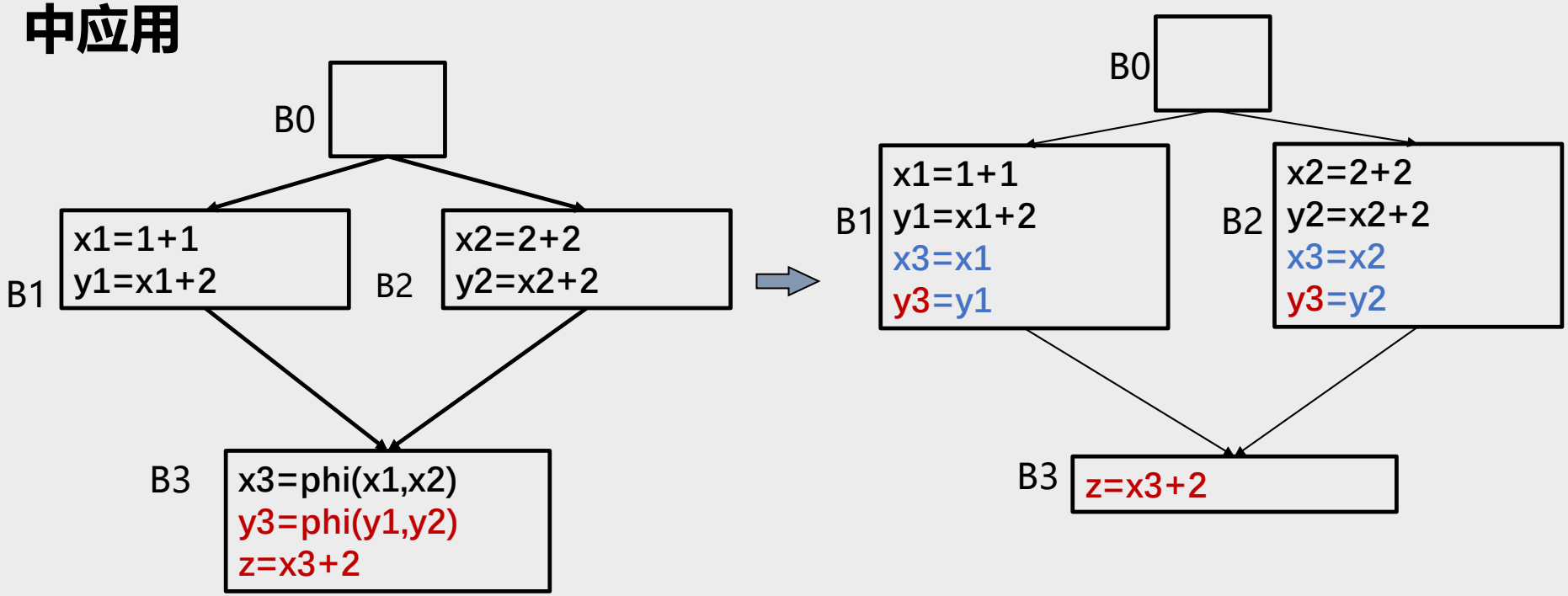
基本概念

- 借助phi指令向copy语句的转换，在数据流分析中，我们可以在下例中应用



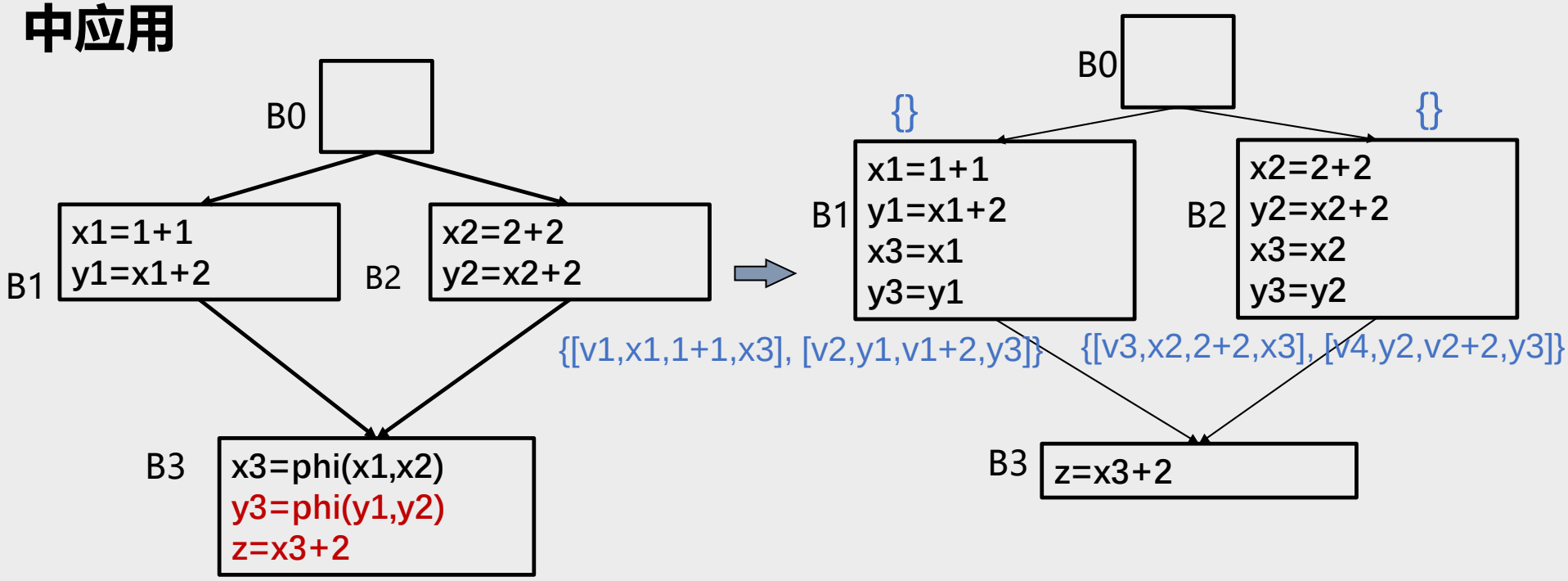
基本概念

- 借助phi指令向copy语句的转换，在数据流分析中，我们可以在下例中应用



基本概念

- 借助phi指令向copy语句的转换，在数据流分析中，我们可以在下例中应用



数据流分析伪代码

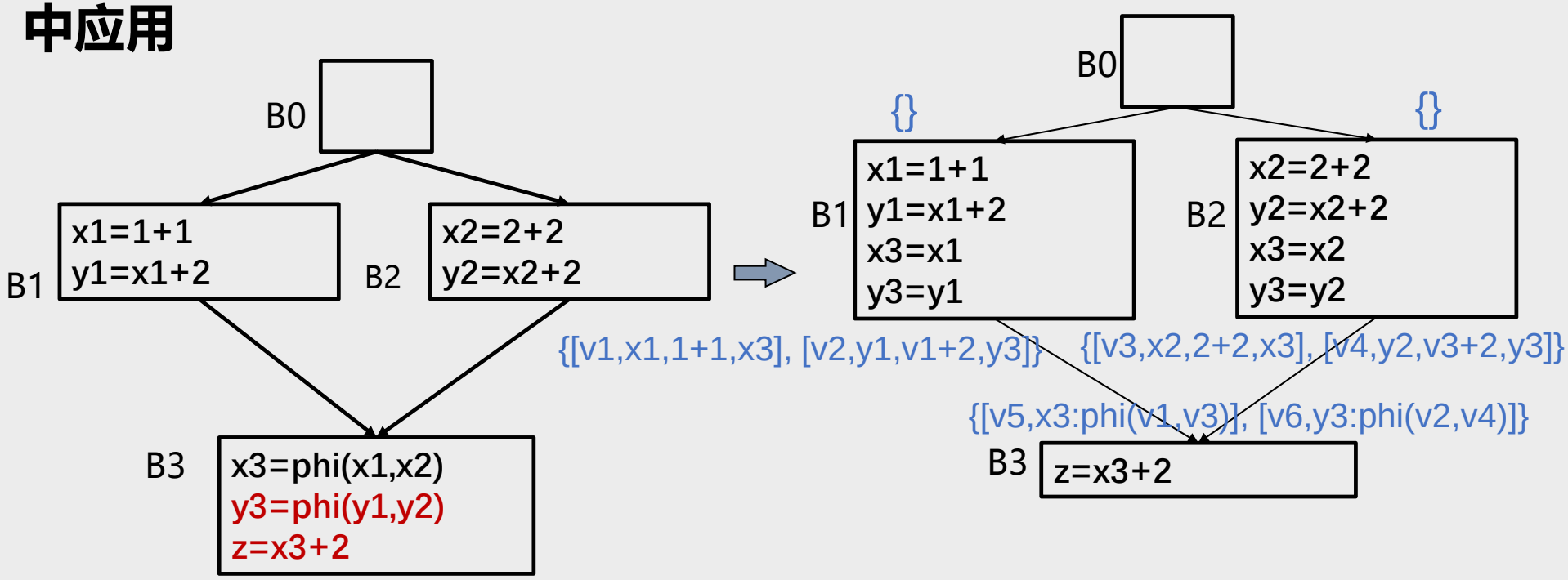
```
Join(P1, P2){
   P = {}
   for each pair of classes Ci ∈ P1 and Cj ∈ P2
        Ck = Intersect(Ci, Cj)
        P = P ∪ Ck
    return P  // Ignore when Ck is empty
}
```

数据流分析伪代码

Intersect(*Ci, Cj*){
    Ck = *Ci* ∩ *Cj* // set intersection
    if *Ck* ≠ ∅ and *Ck* does not have value number
        then *Ck* = *Ck* ∪ {*vk*} // *vk* is new value number
                Ck = (*Ck* - {*vpf*}) ∪ {ϕ_b(*vi, vj*)}
                // *vpf* is value ϕ-function in *Ck*, *vi* ∈ *Ci*, *vj* ∈ *Cj*
                // *b* is join block
    return *Ck*
}

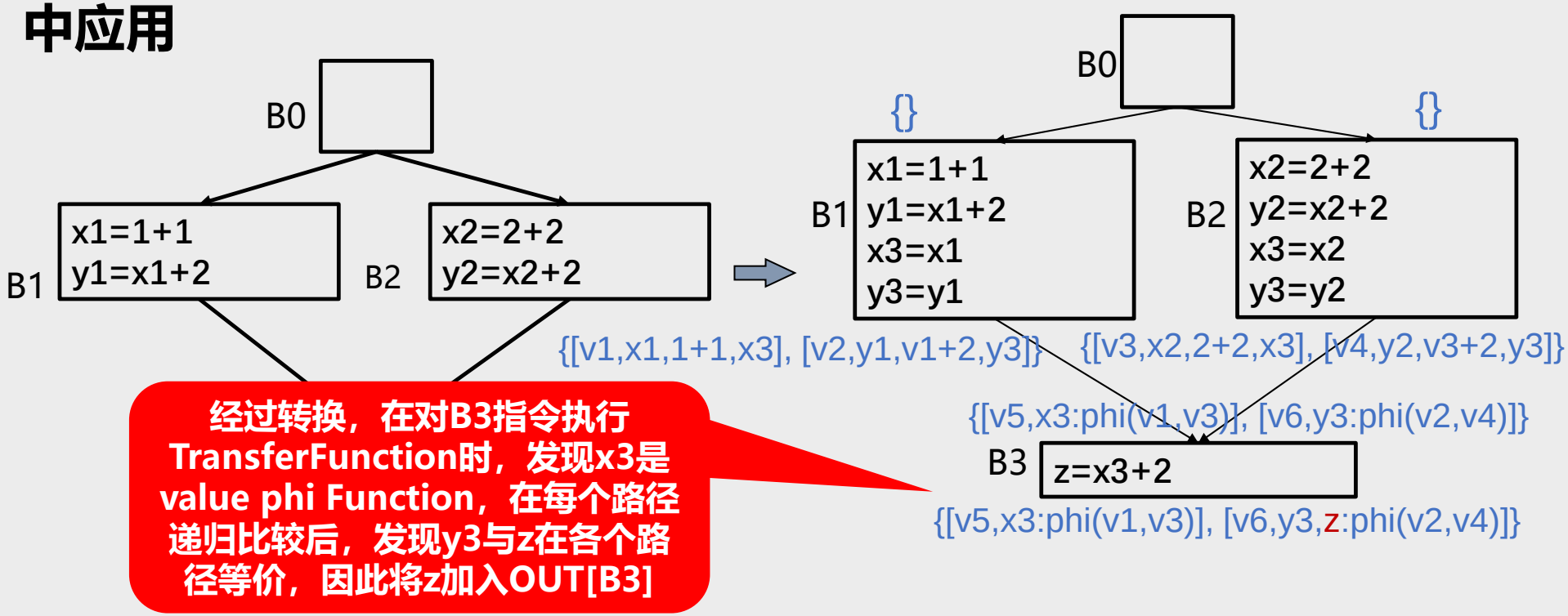
基本概念

- 借助phi指令向copy语句的转换，在数据流分析中，我们可以在下例中应用



基本概念

- 借助phi指令向copy语句的转换，在数据流分析中，我们可以在下例中应用



助教演示

代码展示及提示

- 腾讯会议接入

后续课程安排

- 11.30的复习课换到12.14
- 12.5 华为编程语言及编译实验室guest lecture
- 12.7 Lab5面向自主指令集的编译器设计开放实验tutorial
- 12.14 课程复习课

《编译原理和技术》

数据流优化之GVN

谢谢！