



中国科学技术大学
University of Science and Technology of China



《编译原理与技术》

中间代码生成 I

计算机科学与技术学院

李 诚

12/11/2018



□目标：为PL0语言实现一个简单的编译器

❖Project 1: 词法分析

❖Project 2: 语法分析

❖Project 3: 语法错误处理+对前两个project的扩展,
11.15 release, 11.30提交

❖Project 4: 代码生成, 12.1 release, 12.15提交

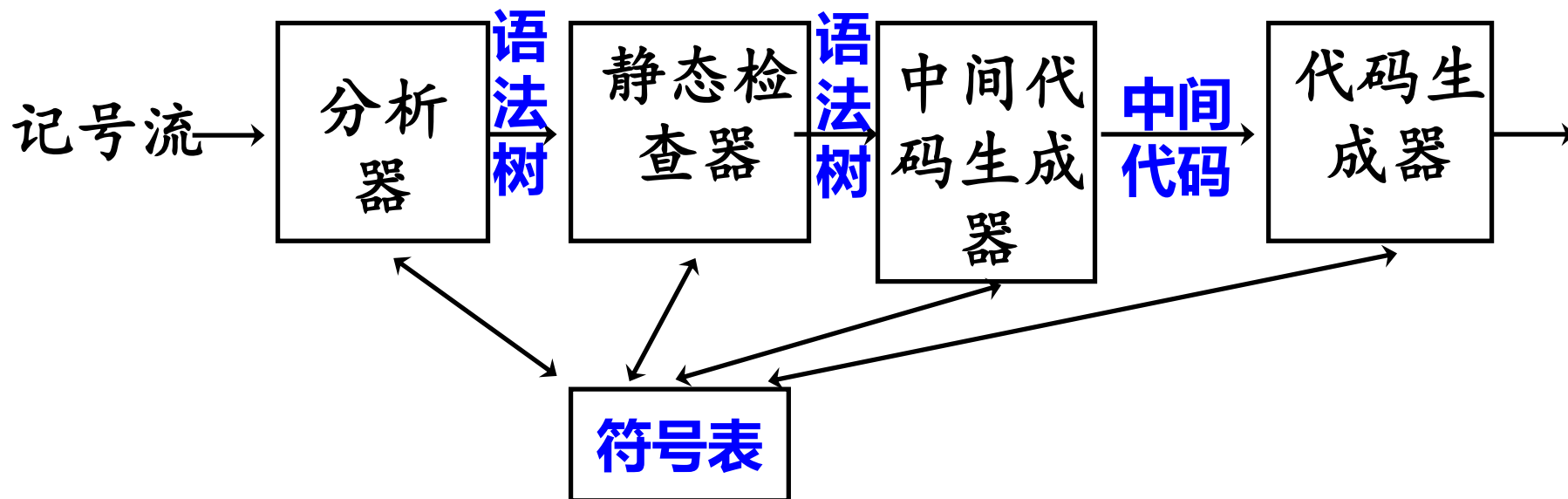
❖项目结束后, 需要进行**答辩**, 每一组准备ppt,
每一名同学都要汇报

➤自己做了什么

➤学到了什么

➤对课程和实验的意见与建议

- **期中考试的成绩在本周之内公布。**
- **由于老师出差，周四的课取消，后面补回来。**
- **周四的上机课正常进行，助教会发布新的项目内容，并对之前项目的完成情况进行说明，请大家尽量都去。**



□中间语言(Intermediate Representation)

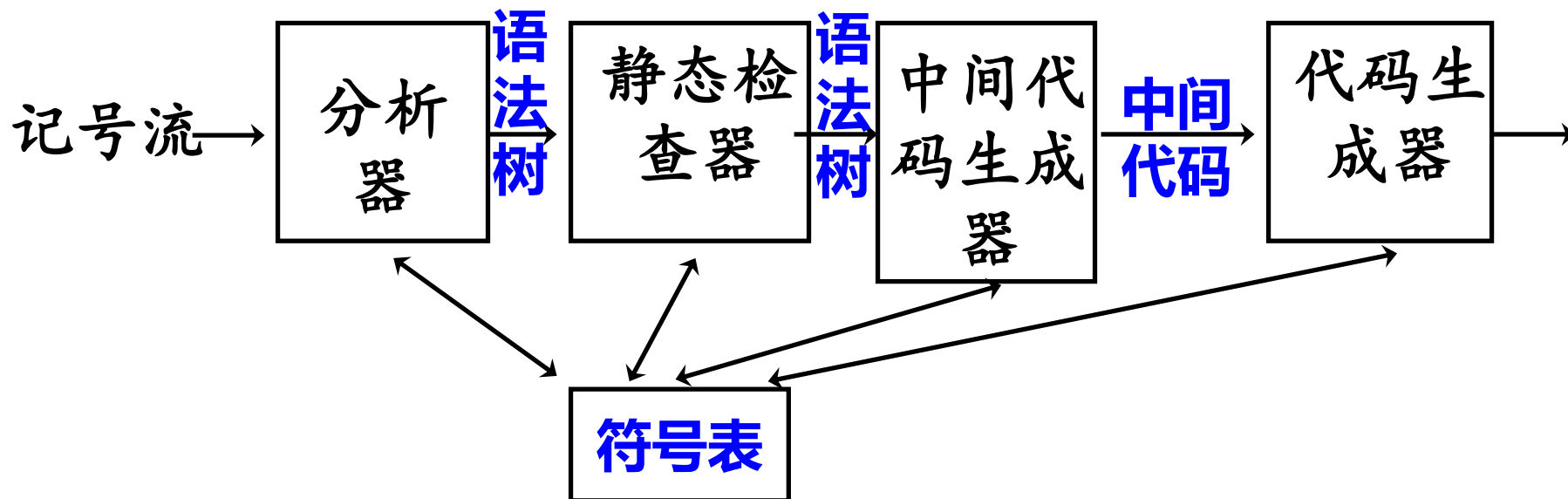
❖ 后缀表达式、图表示、三地址码、静态单赋值

□中间代码生成

❖ 声明语句(更新符号表)

❖ 表达式、赋值语句(产生临时变量、查询符号表)

❖ 布尔表达式、控制流语句(标号/回填、短路计算)



□中间语言(Intermediate Representation)

❖ 后缀表达式、图表示、三地址码、静态单赋值

□中间代码生成

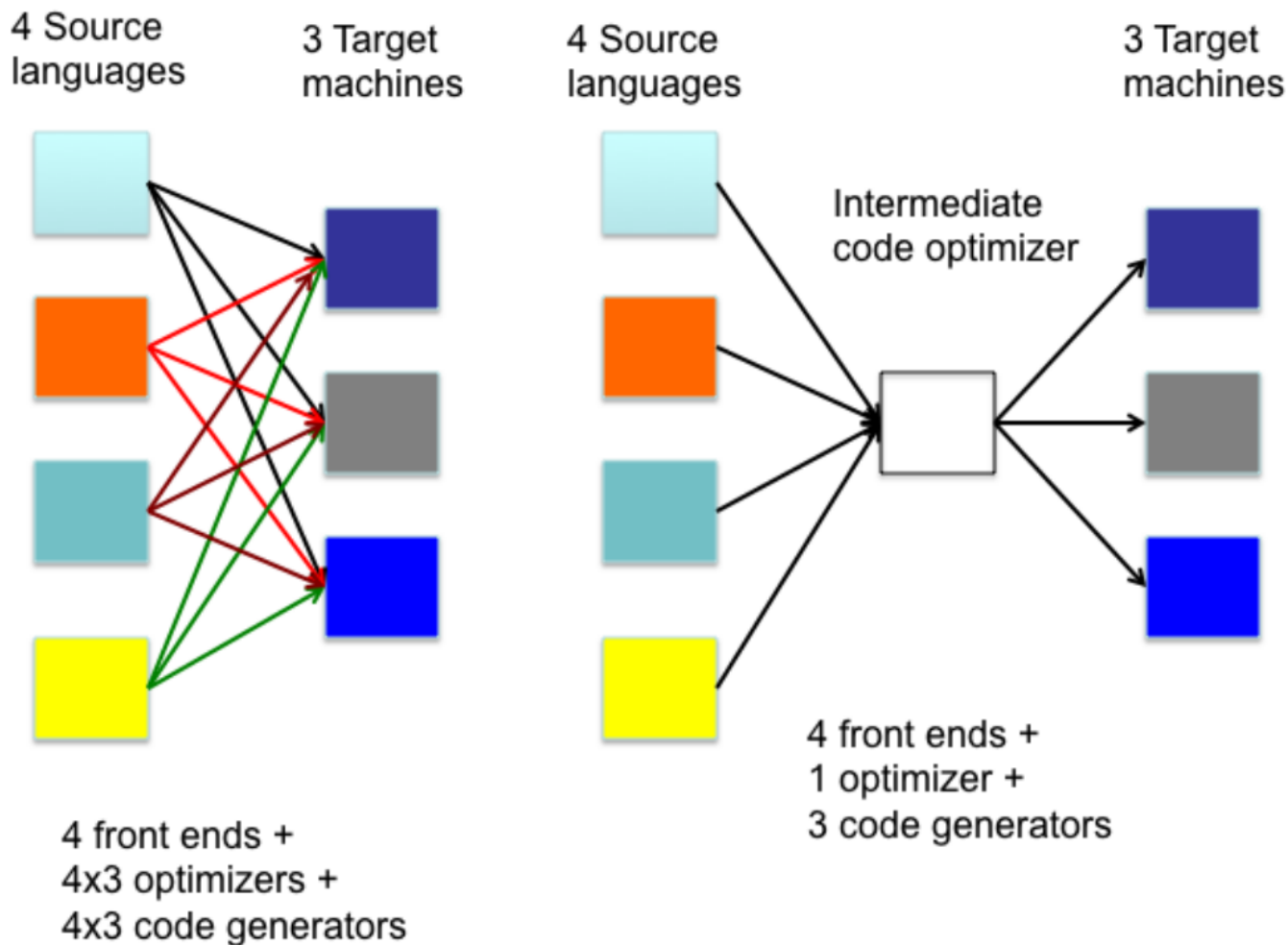
❖ 声明语句(更新符号表)

❖ 表达式、赋值语句(产生临时变量、查询符号表)

❖ 布尔表达式、控制流语句(标号/回填、短路计算)



为什么需要中间语言?



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



*uop*是一元运算符

$$E \rightarrow E \text{ op } E \mid uop E \mid (E) \mid \text{id} \mid \text{num}$$

表达式 E

id

num

$E_1 \text{ op } E_2$

$uop E$

(E)

后缀式 E'

id

num

$E_1' E_2' \text{ op}$

$E' uop$

E'



□ 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 | |



□ 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

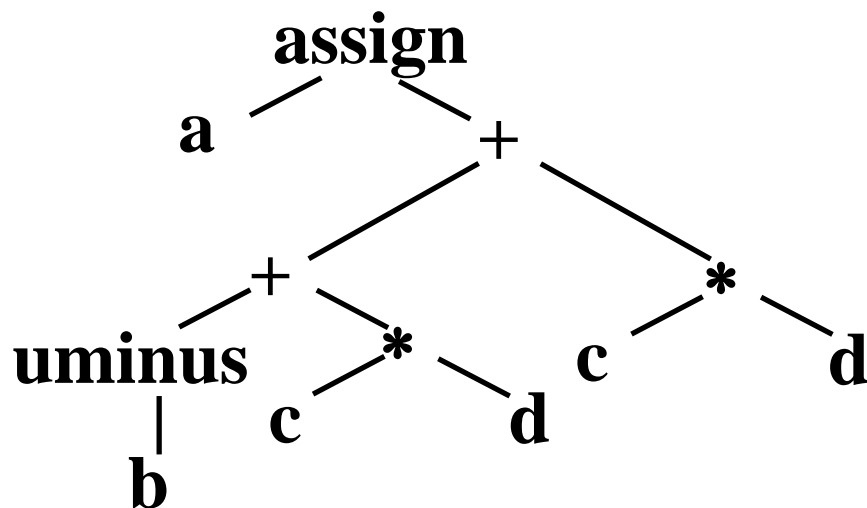
□ 后缀表示的表达能力

❖ 可以拓广到表示赋值语句和控制语句

❖ 但很难用栈来描述控制语句的计算



□语法树是一种图形化的中间表示

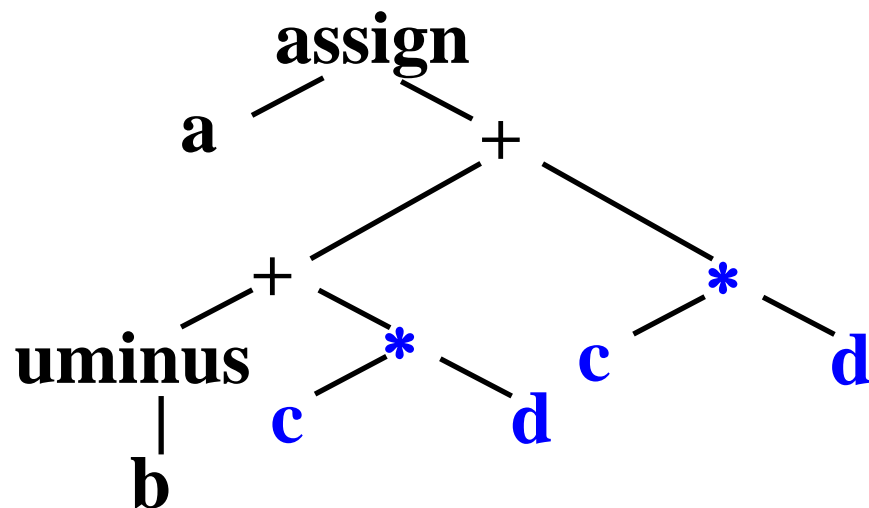


(a) 语法树

$a = (-b + c*d) + c*d$ 的图形表示



□语法树是一种图形化的中间表示



(a) 语法树

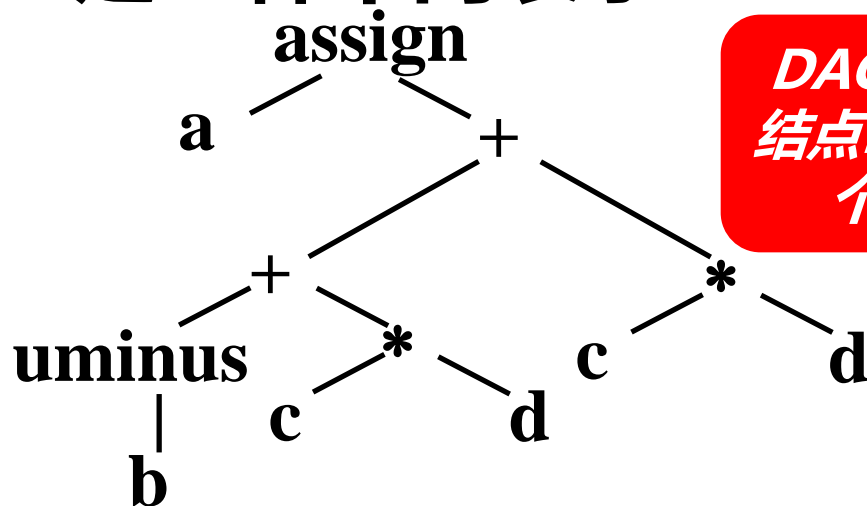
*c*d*是公共子
表达式

$a = (-b + c*d) + c*d$ 的图形表示



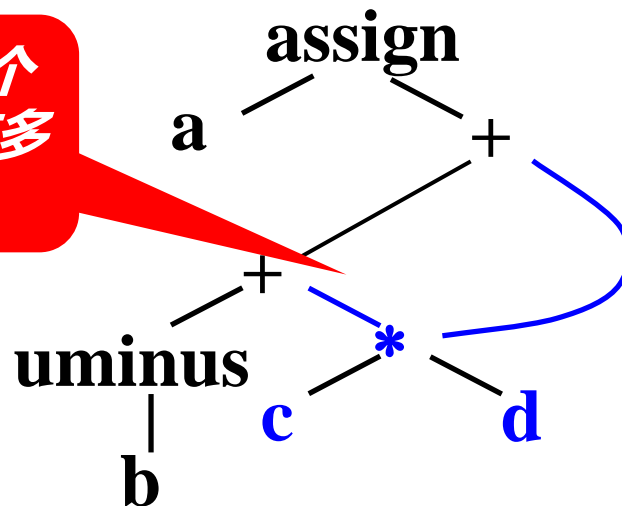
□语法树是一种图形化的中间表示

□有向无环图(Directed Acyclic Graph, DAG)也是一种中间表示



(a) 语法树

DAG中的一个
结点N可能有多
个父结点



(b) DAG

$a = (-b + c*d) + c*d$ 的图形表示



构造赋值语句语法树的语法制导定义(第四章内容)

修改构造结点的函数可生成有向无环图

| 产生式 | 语义规则 |
|---------------------------|---|
| $S \rightarrow id = E$ | $S.nptr = mkNode('assign', mkLeaf(id, id.entry), E.nptr)$ |
| $E \rightarrow E_1 + E_2$ | $E.nptr = mkNode('+', E_1.nptr, E_2.nptr)$ |
| $E \rightarrow E_1 * E_2$ | $E.nptr = mkNode('*', E_1.nptr, E_2.nptr)$ |
| $E \rightarrow -E_1$ | $E.nptr = mkUNode('uminus', E_1.nptr)$ |
| $E \rightarrow (E_1)$ | $E.nptr = E_1.nptr$ |
| $F \rightarrow id$ | $E.nptr = mkLeaf(id, id.entry)$ |



□ 三地址代码 (three-address code)

一般形式: $x = y \text{ op } z$

- 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

$$t_1 = -b$$

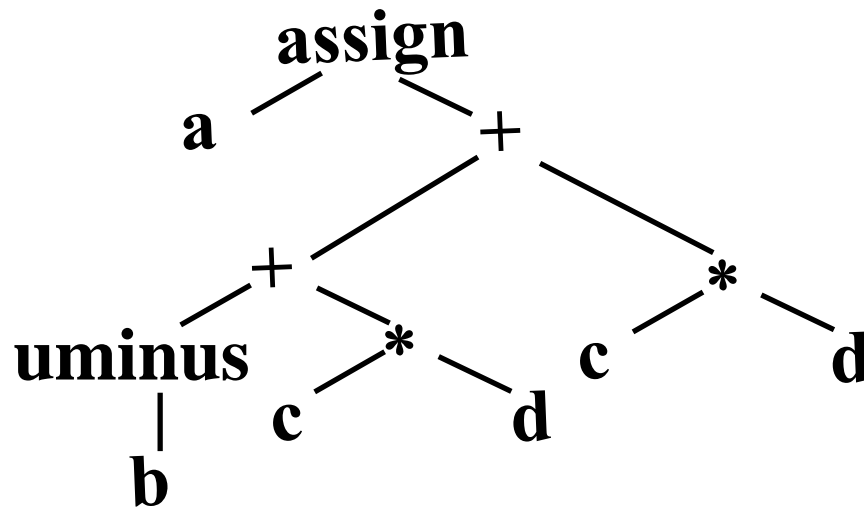
$$t_2 = c * d$$

$$t_3 = t_1 + t_2$$

$$t_4 = c * d$$

$$t_5 = t_3 + t_4$$

$$a = t_5$$





□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

$$t_1 = -b$$

$$t_2 = c * d$$

$$t_3 = t_1 + t_2$$

$$t_4 = c * d$$

$$t_5 = t_3 + t_4$$

$$a = t_5$$

DAG的代码

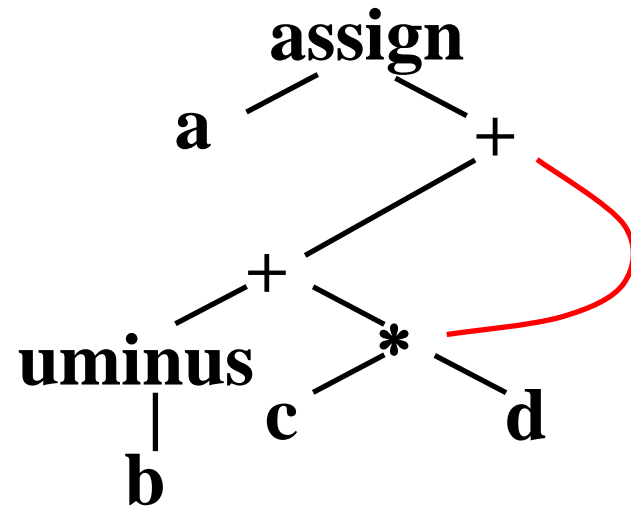
$$t_1 = -b$$

$$t_2 = c * d$$

$$t_3 = t_1 + t_2$$

$$t_4 = t_3 + t_2$$

$$a = t_4$$





□常用的三地址语句

❖ 赋值语句

$x = y \text{ op } z, \quad x = \text{op } y, \quad x = y$

❖ 无条件转移

`goto L`

❖ 条件转移

`if $x \text{ relop } y$ goto L`

❖ 过程调用

`param x 和 call p, n`

❖ 过程返回

`return y`

❖ 索引赋值

$x = y[i]$ 和 $x[i] = y$

❖ 地址和指针赋值

$x = \&y, \quad x = *y$ 和 $*x = y$



□三地址代码只说明了指令的组成部分，为描述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)



□四元式(Quadruple)

❖例： $a = b * -c + b * -c$

| # | Op | Arg1 | Arg2 | Result |
|-----|--------|------|------|--------|
| (0) | uminus | c | | t1 |
| (1) | * | t1 | b | t2 |
| (2) | uminus | c | | t3 |
| (3) | * | t3 | b | t5 |
| (4) | + | t2 | t4 | t5 |
| (5) | = | t5 | | a |

缺点：临时变量太多，增加时间和空间成本



□三元式(Triple)

❖例： $a = b * -c + b * -c$

| # | Op | Arg1 | Arg2 |
|-----|--------|------|------|
| (0) | uminus | c | |
| (1) | * | (0) | b |
| (2) | uminus | c | |
| (3) | * | (2) | b |
| (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。



□ 间接三元式 (Indirect triple)

❖ 例: $a = b * -c + b * -c$

指令序列可以任意调整顺序

| # | Op | Arg1 | Arg2 |
|------|----|------|------|
| (14) | + | x | y |
| (15) | + | y | z |
| (16) | * | (14) | (15) |
| (17) | + | (14) | z |
| (18) | + | (16) | (17) |

List of pointers to table

| # | Statement |
|-----|-----------|
| (1) | (14) |
| (2) | (15) |
| (3) | (16) |
| (4) | (17) |
| (5) | (18) |

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



中国科学技术大学
University of Science and Technology of China

| | | | | |
|-------|---------|-------------------------|-------------------|-----------------|
| 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 | 按编号次序计算 | | 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 |



□ 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

$$p = a + b$$

$$q = p - c$$

$$p = q * d$$

$$p = e - p$$

$$q = p + q$$

静态单赋值形式

$$p_1 = a + b$$

$$q_1 = p_1 - c$$

$$p_2 = q_1 * d$$

$$p_3 = e - p_2$$

$$q_2 = p_3 + q_1$$



□ 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

❖ 一个变量在不同路径上都定值的解决办法

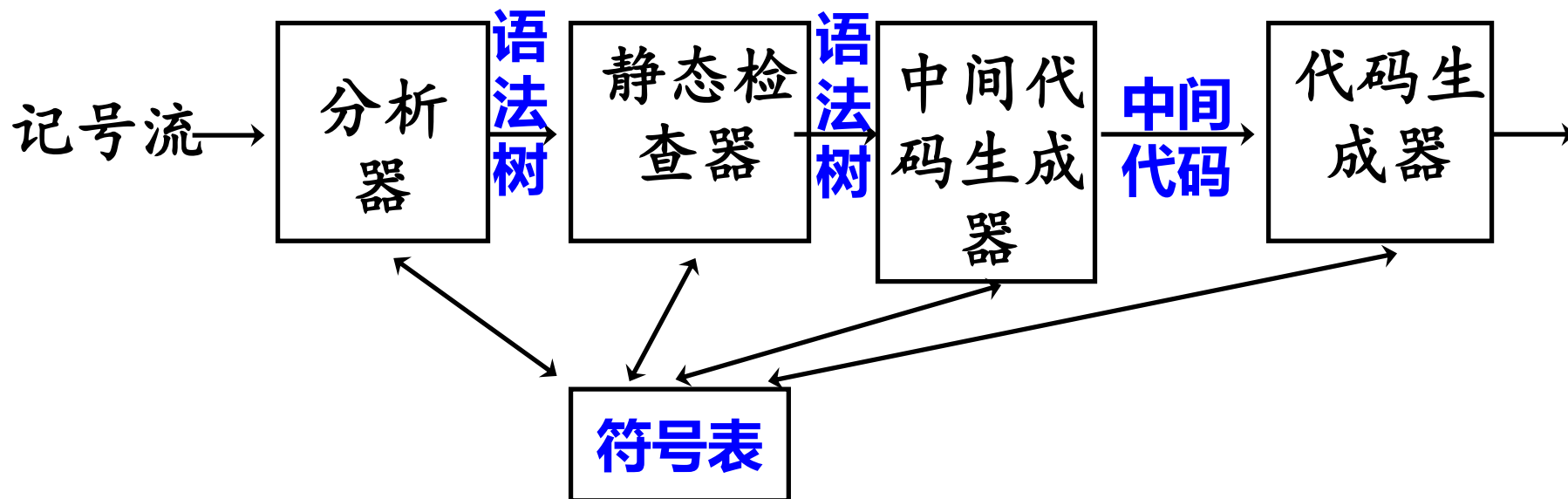
```
if (flag) x = -1; else x = 1;
```

```
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
```

□中间语言(Intermediate Representation)

❖ 后缀表达式、图表示、三地址码、静态单赋值

□中间代码生成

❖ 声明语句(更新符号表)

❖ 表达式、赋值语句(产生临时变量、查询符号表)

❖ 布尔表达式、控制流语句(标号/回填、短路计算)



□ 类型与符号表的变化

- ❖ 多样化类型 \Rightarrow 整型(字节、字)、浮点型、类型符号表
- ❖ 1个某类型的数据 \Rightarrow m 个字节(m 为类型对应的字宽)

□ 语句的翻译

- ❖ 声明语句：不生成指令，但会更新符号表(作用域，字宽及存放的相对地址)
- ❖ 赋值语句：引入临时变量、数组/记录元素的地址计算、类型转换
- ❖ 控制流语句：跳转目标的确定(引入标号或使用回填技术)、短路计算



□ 类型检查后的符号表

- ❖ 符号表条目：(标识符、存储类别、类型信息)
- ❖ 存储类别：extern, static, register, ...
- ❖ 类型信息：(类别标识, 该类别关联的其他信息)
 - 如数组(array(len, elemtype))

□ 本章符号表的变化

- ❖ 作用域 => 多个符号表
- ❖ 变量：字宽、存储的相对地址(以字节为单位)
- ❖ 记录类型：用符号表管理各个成员的字宽、相对地址



□边解析边生成中间代码

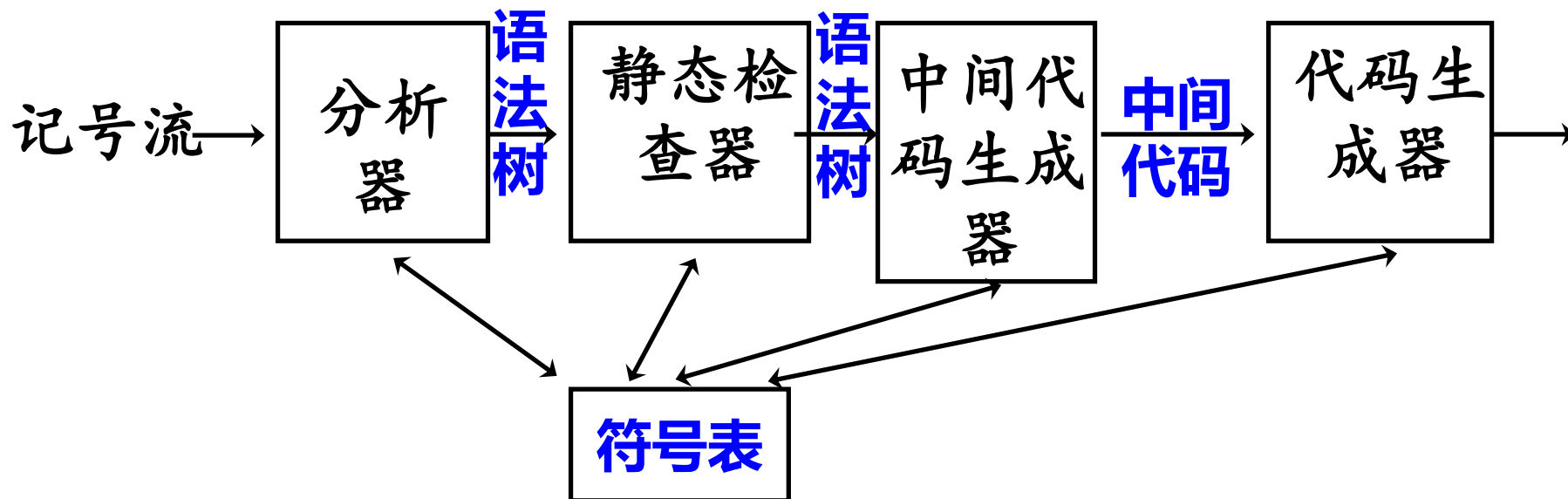
- ❖ 语法制导的翻译方案

- ❖ 难点：理解分析器的运转机制、继承属性的处理

□基于树访问的中间代码生成

- ❖ 重点：树结构的设计、访问者模式、enter/exit接口及实现

本节将以基于树访问的中间代码生成方法为主来讲解，这是现代编译器使用的主流方法。



□中间语言(Intermediate Representation)

❖ 后缀表达式、图表示、三地址码、静态单赋值

□中间代码生成

❖ 声明语句(更新符号表)

❖ 表达式、赋值语句(产生临时变量、查询符号表)

❖ 布尔表达式、控制流语句(标号/回填、短路计算)



□知识要点

- ❖分配存储单元
- ❖更新符号表
- ❖作用域的管理
- ❖记录类型的管理



□ 主要任务

❖ 为局部名字分配存储单元 符号表条目：

➤ 名字、类型、字宽、偏移

❖ 作用域信息的保存

➤ 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



□例：文法 G_1 如下：

$P \rightarrow D ; S$

$D \rightarrow D ; D$

$D \rightarrow \text{id} : T$

$T \rightarrow \text{integer} \mid \text{real} \mid \text{array} [\text{num}] \text{ of } T_1 \mid \uparrow T_1$



□ 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



| 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。 | | |



计算被声明名字的类型和相对地址

$P \rightarrow \{offset = 0\} D ; S$

相对地址初始化为0

$D \rightarrow D ; D$

$D \rightarrow id : T \{ \text{enter}(id.lexeme, T.type, offset);$
 $offset = offset + T.width \}$

更新符号表信息

$T \rightarrow \text{integer} \{ T.type = \text{integer}; T.width = 4 \}$

$T \rightarrow \text{real} \{ T.type = \text{real}; T.width = 8 \}$

类型=>字宽

$T \rightarrow \text{array} [\text{number}] \text{ of } T_1$

$\{ T.type = \text{array}(\text{num.val}, T_1.type);$
 $T.width = \text{num.val} * T_1.width \}$

$T \rightarrow \uparrow T_1 \{ T.type = \text{pointer}(T_1.type); T.width = 4 \}$



□ 所讨论语言的文法

$P \rightarrow D; S$

$D \rightarrow D ; D / \text{id} : T /$

$\text{proc id} ; D ; S$

□ 管理作用域(过程嵌套)

- ❖ 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

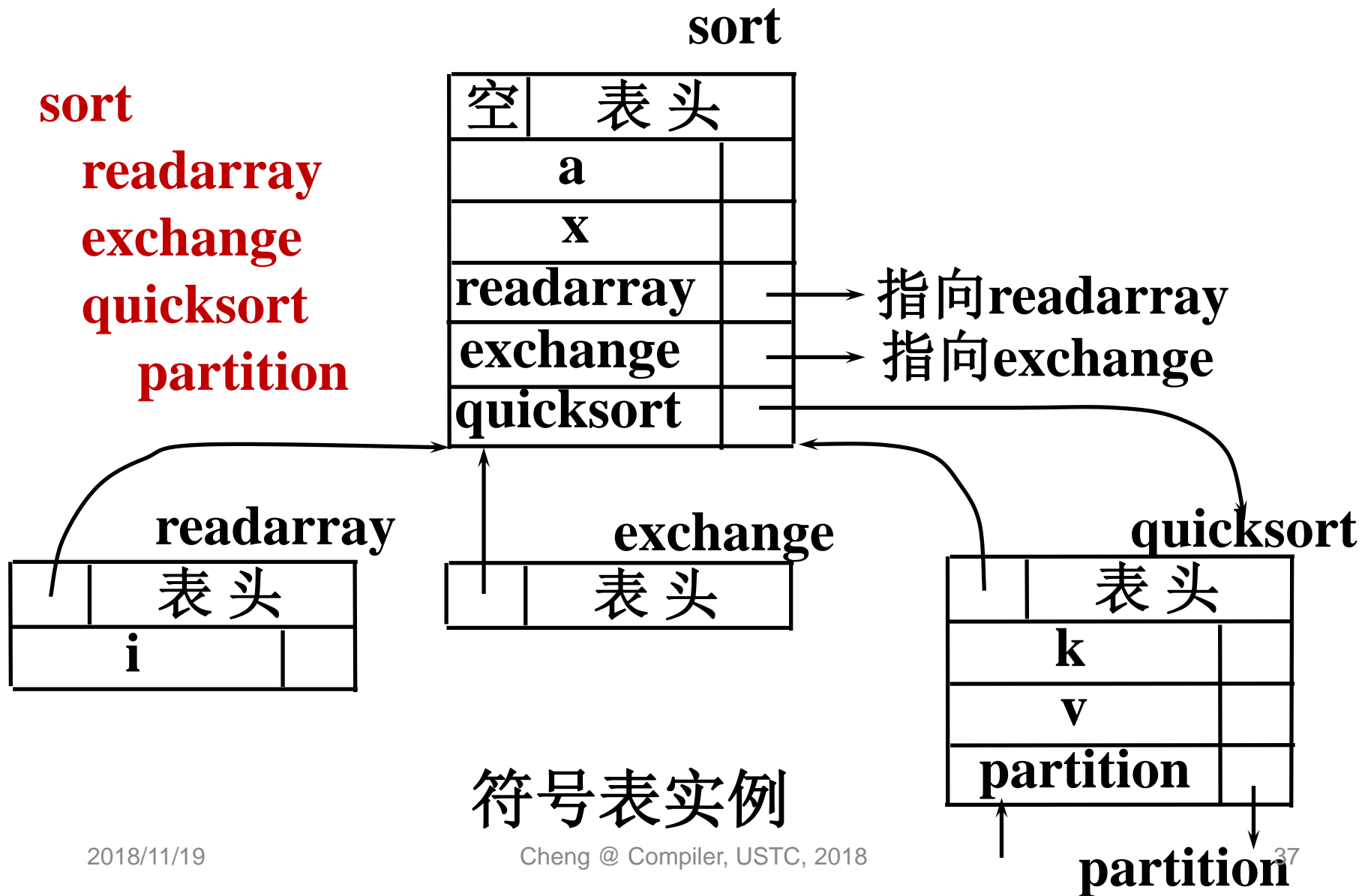
var k, v:....;

partition

var i, j:....;

教科书186页图6.14

过程参数被略去





□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**

□语义动作作用到的函数

mkTable(previous)：为新符号表建立连接

enter(table, name, type, offset)：为变量建立条目

addWidth(table, width)：为符号表添加变量累加宽度

enterProc(table, name, newtable)：为过程名创建条目



$P \rightarrow \mathbf{M} D; S$

$\mathbf{M} \rightarrow \varepsilon$

$D \rightarrow D_1; D_2$

$D \rightarrow \text{proc id} ; \mathbf{N} D_1; S$

$D \rightarrow \text{id} : T$

$\mathbf{N} \rightarrow \varepsilon$



$P \rightarrow \mathbf{M} D; S$

***tblptr*: 符号表栈**
***offset*: 偏移量栈**

$\mathbf{M} \rightarrow \varepsilon \quad \{ t = mkTable (nil);$
 $\quad \quad \quad \textit{push}(t, \textit{tblptr}); \textit{push} (0, \textit{offset}) \}$

$D \rightarrow D_1; D_2$

$D \rightarrow \textit{proc id}; \mathbf{N} D_1; S$

$D \rightarrow \textit{id} : T$

$\mathbf{N} \rightarrow \varepsilon$



$P \rightarrow \mathbf{M} D; S$

$\mathbf{M} \rightarrow \varepsilon \quad \{t = mkTable (nil);$
 $\quad \quad \quad push(t, tblptr); push (0, offset) \}$

$D \rightarrow D_1; D_2$

$D \rightarrow \text{proc id} ; \mathbf{N} D_1; S$

$D \rightarrow \text{id} : T \quad \{enter(top(tblptr), id.lexeme, T.type, top(offset));$
 $\quad \quad \quad top(offset) = top(offset) + T.width \}$

$\mathbf{N} \rightarrow \varepsilon$



$P \rightarrow M D; S$

$M \rightarrow \varepsilon$ $\{t = mkTable (nil);$
 $push(t, tblptr); push (0, offset) \}$

$D \rightarrow D_1; D_2$

$D \rightarrow \text{proc id} ; N D_1; S$ $\{t = top(tblptr);$
 $addWidth(t, top(offset)); pop(tblptr); pop(offset);$
 $enterProc(top(tblptr), id.lexeme, t) \}$

$D \rightarrow \text{id} : T$ $\{enter(top(tblptr), id.lexeme, T.type, top(offset));$
 $top(offset) = top(offset) + T.width \}$

$N \rightarrow \varepsilon$

访问D前: 新建该过程的符号表, 进入该过程的作用域



$P \rightarrow \mathbf{M} D; S$

$\mathbf{M} \rightarrow \varepsilon$ $\{t = mkTable (nil);$
 $push(t, tblptr); push (0, offset) \}$

$D \rightarrow D_1; D_2$

$D \rightarrow \text{proc id} ; \mathbf{N} D_1; S$ $\{t = top(tblptr);$
 $addWidth(t, top(offset)); pop(tblptr); pop(offset);$
 $enterProc(top(tblptr), id.lexeme, t) \}$

$D \rightarrow \text{id} : T$ $\{enter(top(tblptr), id.lexeme, T.type, top(offset));$
 $top(offset) = top(offset) + T.width \}$

$\mathbf{N} \rightarrow \varepsilon$ $\{t = mkTable(top(tblptr));$
 $push(t, tblptr); push(0, offset) \}$



$P \rightarrow M D; S \{ \text{addWidth}(\text{top}(\text{tblptr}), \text{top}(\text{offset}));$
 $\text{pop}(\text{tblptr}); \text{pop}(\text{offset}) \}$

$M \rightarrow \varepsilon \quad \{ t = \text{mkTable}(\text{nil});$
 $\text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}) \}$

$D \rightarrow D_1; D_2$

$D \rightarrow \text{proc id}; N D_1; S \{ t = \text{top}(\text{tblptr});$
 $\text{addWidth}(t, \text{top}(\text{offset})); \text{pop}(\text{tblptr}); \text{pop}(\text{offset});$
 $\text{enterProc}(\text{top}(\text{tblptr}), \text{id.lexeme}, t) \}$

$D \rightarrow \text{id} : T \{ \text{enter}(\text{top}(\text{tblptr}), \text{id.lexeme}, T.\text{type}, \text{top}(\text{offset}));$
 $\text{top}(\text{offset}) = \text{top}(\text{offset}) + T.\text{width} \}$

$N \rightarrow \varepsilon \quad \{ t = \text{mkTable}(\text{top}(\text{tblptr}));$
 $\text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}) \}$



□描述记录的文法

$T \rightarrow \text{record } D \text{ end}$

记录类型单独建符号表，域的相对地址从0开始

$T \rightarrow \text{record } L D \text{ end}$

$L \rightarrow \varepsilon$

```
record
  a : ...;
  r : record
    i : ...;
    . . .
  end;
  k : ...;
end
```



□描述记录的文法

$T \rightarrow \text{record } D \text{ end}$

记录类型单独建符号表，域的相对地址从0开始

$T \rightarrow \text{record } \textcolor{blue}{L} D \text{ end}$

$L \rightarrow \varepsilon \text{ } \textcolor{red}{t = mkTable(nil);}$

$\textcolor{red}{push(t, tblptr); push(0, offset) \}$

record

a : ...;

r : record

i : ...;

...

end;

k : ...;

end

建立符号表，进入作用域



□描述记录的文法

$T \rightarrow \text{record } D \text{ end}$

记录类型单独建符号表，域的相对地址从0开始

$T \rightarrow \text{record } \textcolor{blue}{L} D \text{ end}$

$\{T.type = \text{record } (top(tblptr)) \};$

$T.width = top(offset);$

$pop(tblptr); pop(offset) \}$

$L \rightarrow \varepsilon \ t = mkTable(nil);$

$push(t, tblptr); push(0, offset) \}$

设置记录的类型表达式和宽度，退出作用域

record

a : ...;

r : record

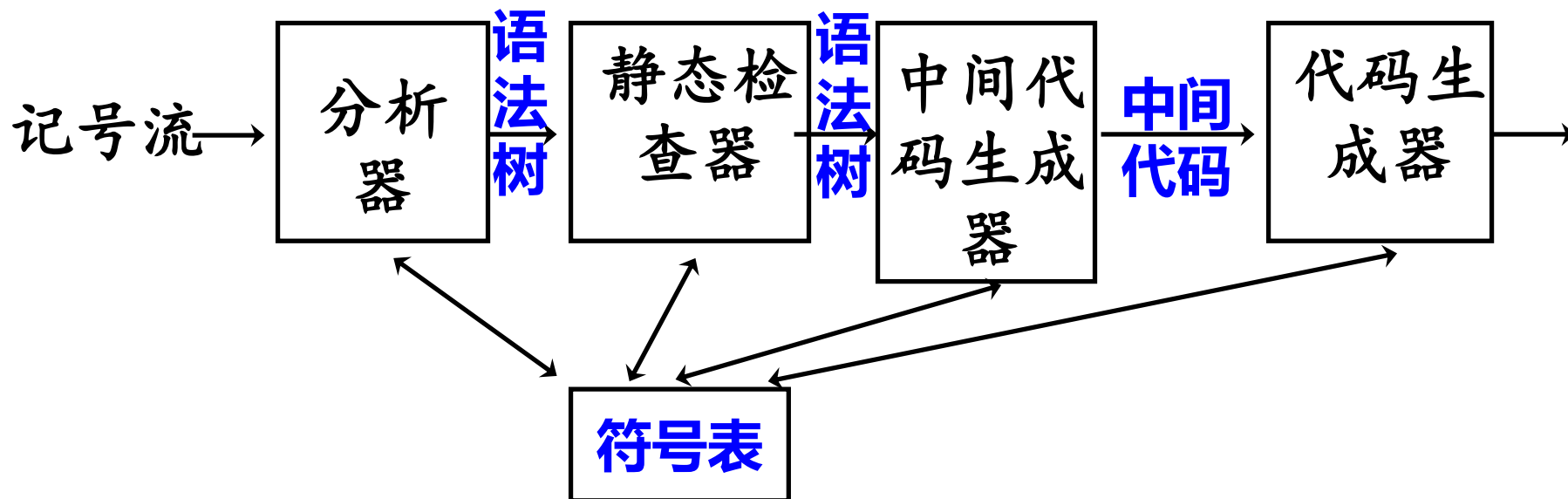
i : ...;

...

end;

k : ...;

end



□中间语言(Intermediate Representation)

❖ 后缀表达式、图表示、三地址码、静态单赋值

□中间代码生成

❖ 声明语句(更新符号表)

❖ 表达式、赋值语句(产生临时变量、查询符号表)

❖ 布尔表达式、控制流语句(标号/回填、短路计算)



□知识要点

- ❖ 分配临时变量, 存储表达式计算的中间结果
- ❖ 数组元素的地址计算
- ❖ 类型转换



□主要任务

- ❖ 复杂的表达式 \Rightarrow 多条计算指令组成的序列
- ❖ 分配临时变量保存中间结果
- ❖ id: 查符号表获得其存储的场所
- ❖ 数组元素: 元素地址计算
 - 符号表中保存数组的基址和用于地址计算的常量表达式的值
 - 数组元素在中间代码指令中表示为“基址[偏移]”
- ❖ 可以进行一些语义检查
 - 类型检查、变量未定义/重复定义/未初始化
- ❖ 类型转换: 因为目标机器的运算指令是区分类型的



□赋值语句文法

$$S \rightarrow \text{id} := E \quad E \rightarrow E_1 + E_2 \mid -E_1 \mid (E_1) \mid \text{id}$$

□语义动作作用到的函数

❖ 获取id的地址和存放E结果的场所

➤ *lookup(id.lexeme)*

❖ 产生临时变量

➤ *newTemp();*

❖ 输出翻译后的指令

➤ *Emit(addr, op, arg1, arg2)* : 三地址码

□属性: *E.place* 符号表条目的地址



□符号表中的名字

$S \rightarrow id := E$ $\{p = lookup(id.lexeme);$
 if $p \neq nil$ then
 $emit(p, '=', E.place)$
 else $error$ }

$E \rightarrow E_1 + E_2$
 $\{E.place = newTemp();$
 $emit(E.place, '=', E_1.place, '+', E_2.place) \}$



□符号表中的名字

$$E \rightarrow -E_1 \{ E.place = newTemp();$$
$$emit(E.place, '=', 'uminus', E_1.place) \}$$
$$E \rightarrow (E_1) \{ E.place = E_1.place \}$$
$$E \rightarrow id \quad \{ p = lookup(id.lexeme);$$
$$if p \neq nil \text{ then } E.place = p \text{ else error } \}$$



中国科学技术大学
University of Science and Technology of China



《编译原理与技术》

中间代码生成 I

TBA