编译原理

第一章 编译原理简介 回顾

计算机科学与技术学院

第一章 编译原理简介 回顾

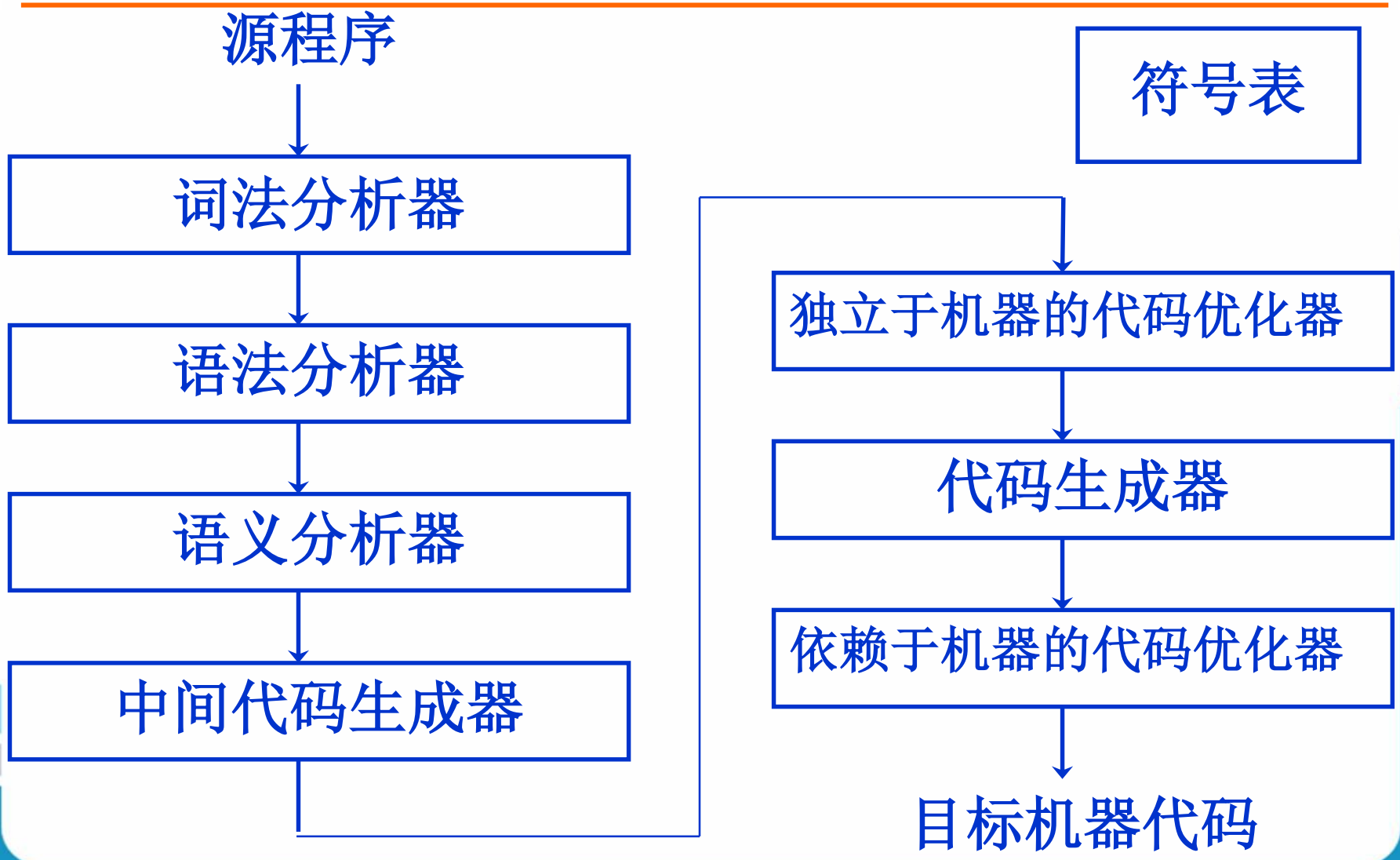
❖ 编译器：

- 编译器是一种翻译器，它的特点是目标语言比源语言低级


```
position := initial + rate * 60
```

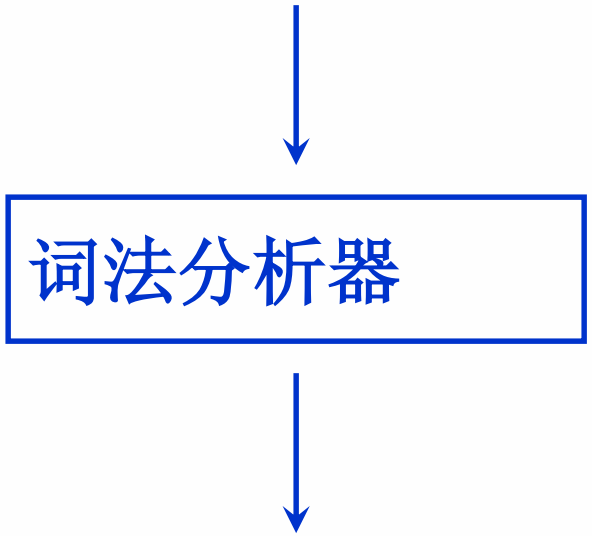
```
MOVF   id3,R2
MULF   #60.0,R2
MOVF   id2,R1
ADDF   R2,R1
MOVF   R1,id1
```

第一章 编译原理简介 回顾



第一章 编译原理简介 回顾

position = initial + rate * 60 ← 字符流



符 号 表

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

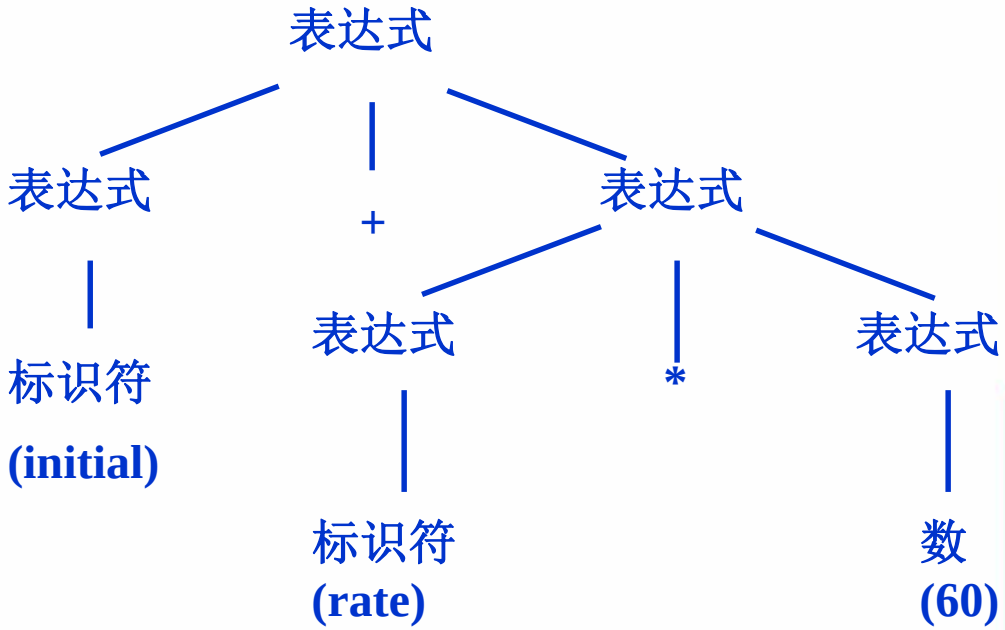
⟨id, 1⟩ ⟨=⟩ ⟨id, 2⟩ ⟨+⟩ ⟨id, 3⟩ ⟨*⟩ ⟨60⟩ ← 记号流

第一章 编译原理简介 回顾

表达式的语法特征

- 任何一个标识符都是表达式
- 任何一个数都是表达式
- 如果e_1和e_2都是表达式，那么
  - $e_1 + e_2$
  - $e_1 * e_2$
  - (e_1)

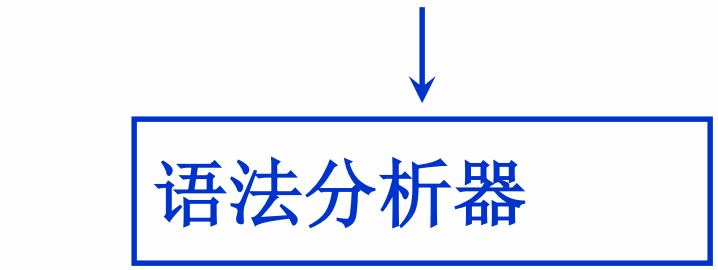
也都是表达式



initial + rate * 60的分析树

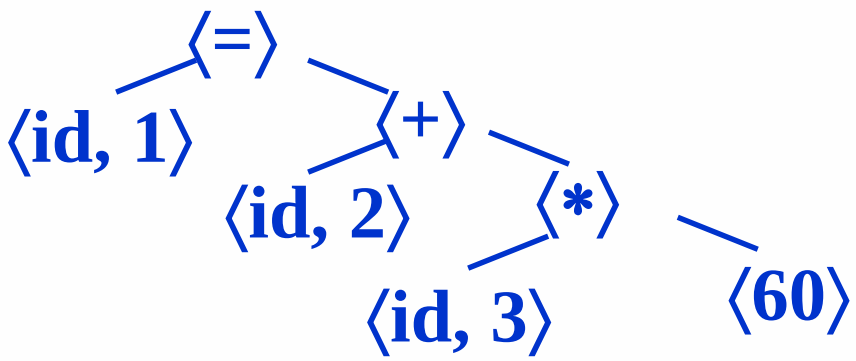
第一章 编译原理简介 回顾

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



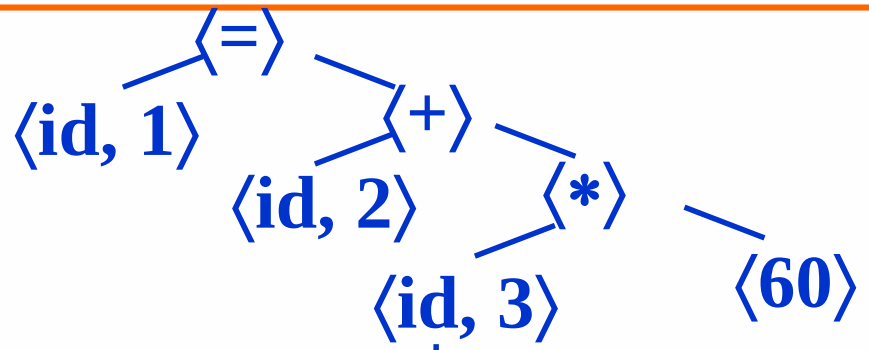
符 号 表

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



← 语法树

第一章 编译原理简介 回顾

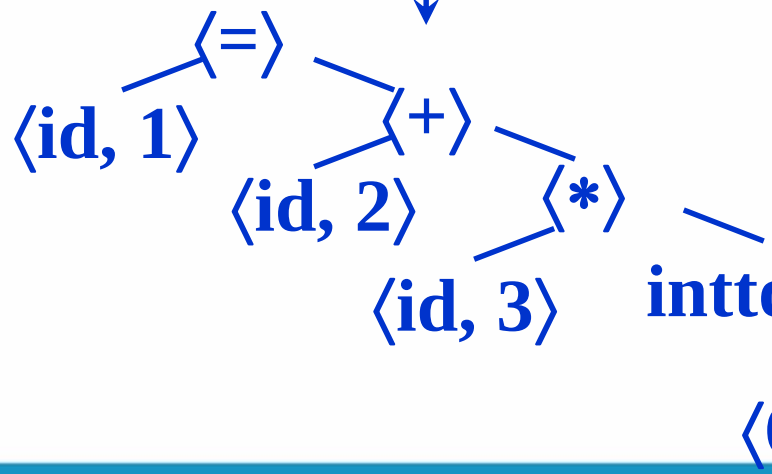


← 语法树

符 号 表

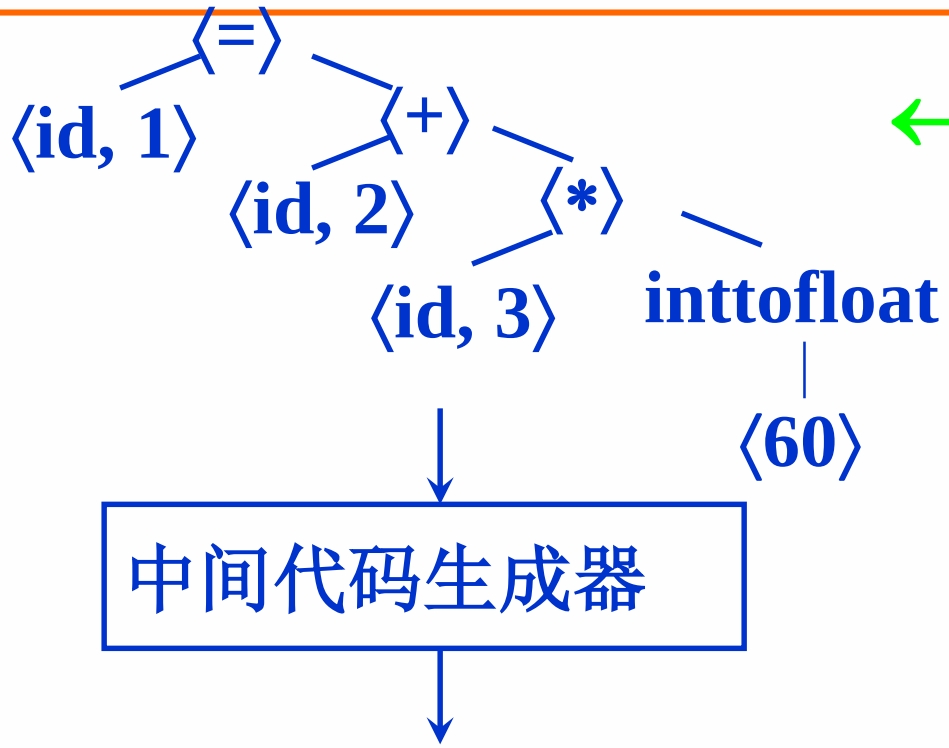
1	position	· · ·
2	initial	· · ·
3	rate	· · ·

语义分析器



← 语法树

第一章 编译原理简介 回顾



← 语法树

符 号 表

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

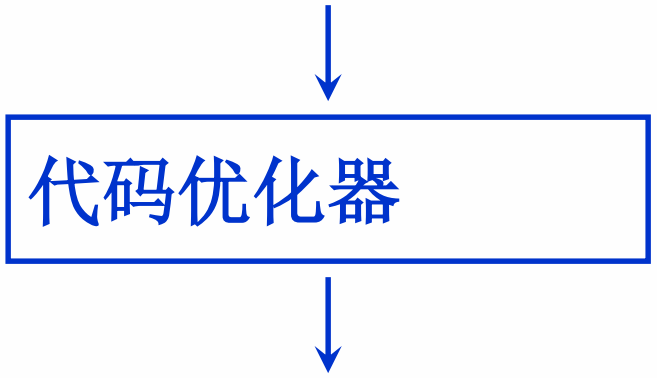
中间代码生成器

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

← 三地址中间代码

第一章 编译原理简介 回顾

t1 = inttofloat(60) ← 三地址中间代码
t2 = id3 * t1
t3 = id2 + t2
id1 = t3



t1 = id3 * 60.0 ← 三地址中间代码
id1 = id2 + t1

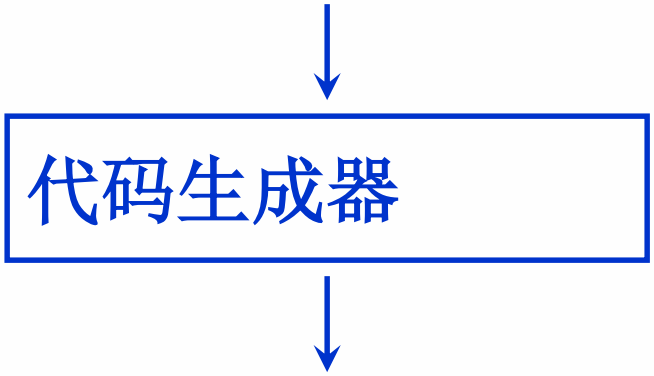
符 号 表

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

第一章 编译原理简介 回顾

t1 = id3 * 60.0
id1 = id2 + t1

← 三地址中间代码



符 号 表

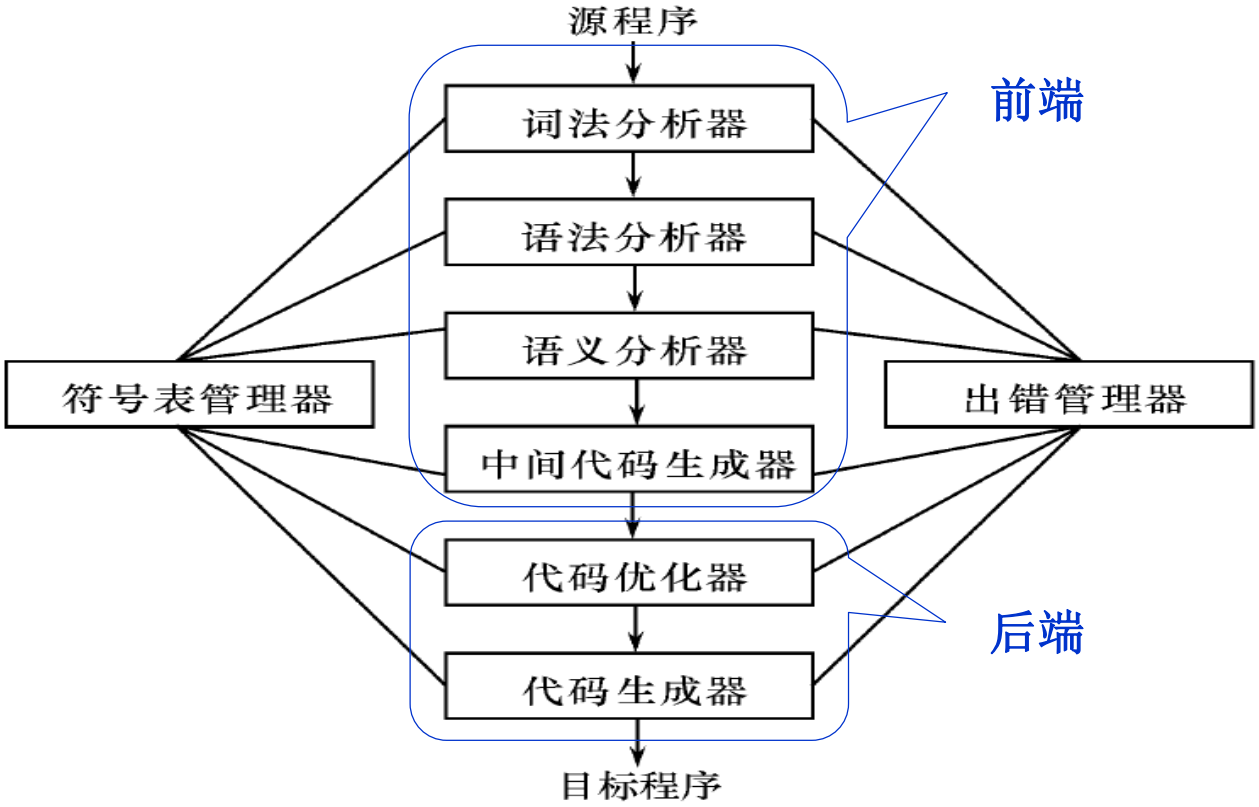
1	position	···
2	initial	···
3	rate	···

MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1

← 汇编代码

第一章 编译原理简介 回顾

- 编译器的工作可以分成若干阶段，每个阶段把源程序从一种表示变换成另一种表示。



第一章 编译原理简介 回顾

- 编译系统
  - 除了编译器外，还需要一些其他工具的帮助，才能得到可执行的目标程序，这些工具包括预处理器、汇编器和连接器等
    - C语言的编译系统
    - Java语言的编译系统

第一章 编译原理简介 回顾

- 编译性语言、解释性语言和脚本语言
  - 高级语言翻译成机器语言，计算机才能执行高级语言编写的程序。
  - 翻译有两种方式：
    - 编译：一次性编译成机器语言文件，不用重新编译，效率高
    - 解释：每个语句都是执行的时候才翻译，每执行一次就翻译一次，效率比较低
  - 脚本语言是一种解释性的语言
    - JavaScript，ASP,PHP,PERL
  - Java语言
    - 既要编译，又要解释；编译只有一次，程序执行时解释执行；通过编译器，把java程序翻译成一种中间代码——字节码，然后通过JVM解释成相应平台的语言

编译原理

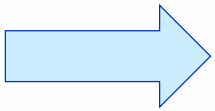
第二章 一个简单的语法制导翻译器

计算机科学与技术学院

第二章 一个简单的语法制导翻译器

- 以简单实现编译器前端示例：
  - 词法分析
  - 语法分析
  - 中间代码生成

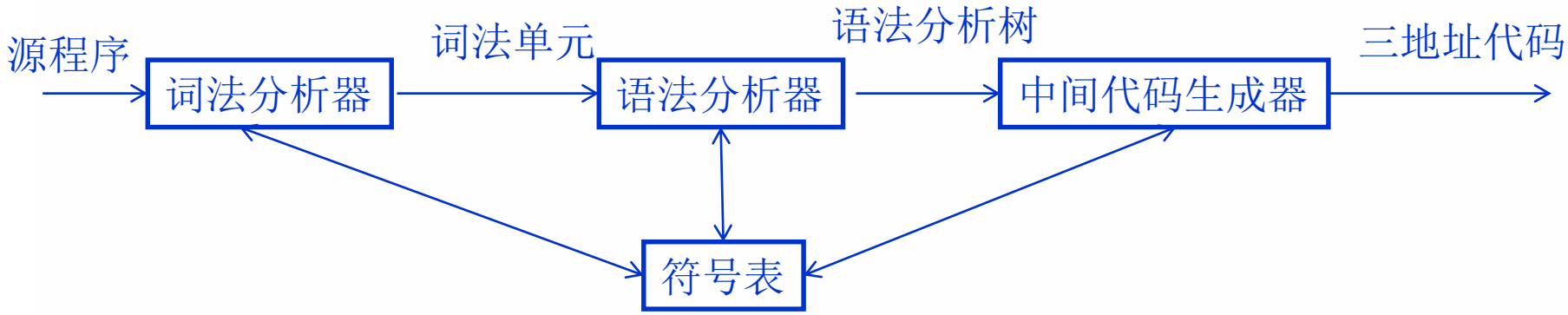
```
{
    int i; int j; float[100] a; float v; float x;
    while ( true ) {
        do i = i+1; while ( a[i] < v );
        do j = j-1; while ( a[j] > v );
        if ( i >= j ) break;
        x = a[i]; a[i] = a[j]; a[j] = x;
    }
}
```



```
 1:  i = i + 1
 2:  t1 = a [ i ]
 3:  if t1 < v goto 1
 4:  j = j - 1
 5:  t2 = a [ j ]
 6:  if t2 > v goto 4
 7:  ifFalse i >= j goto 9
 8:  goto 14
 9:  x = a [ i ]
10:  t3 = a [ j ]
11:  a [ i ] = t3
12:  a [ j ] = x
13:  goto 1
14:
```

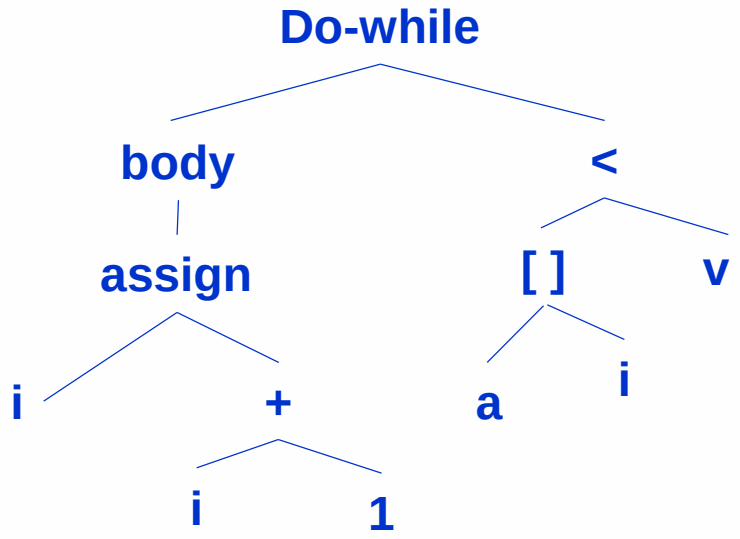
第二章 一个简单的语法制导翻译器

- 语法制导翻译器
  - 语法（syntax）：程序的正确形式
  - 语义（semantics）：程序的含义，即程序在运行时做什么事情



第二章 一个简单的语法制导翻译器

- 中间代码
  - 抽象语法树（abstract syntax tree），常简称为语法树（syntax tree）



do i = i + 1; while (a[i] < v)

第二章 一个简单的语法制导翻译器

- 中间代码
  - 三地址码：$x = y$ **op** z
    - 最多只执行一个运算，通常是计算、比较或者分支跳转

```
1: i = i + 1
2: t1 = a [ i ]
3: if t1 < v goto 1
```

```
do i = i + 1; while (a[i] < v)
```

第二章 一个简单的语法制导翻译器

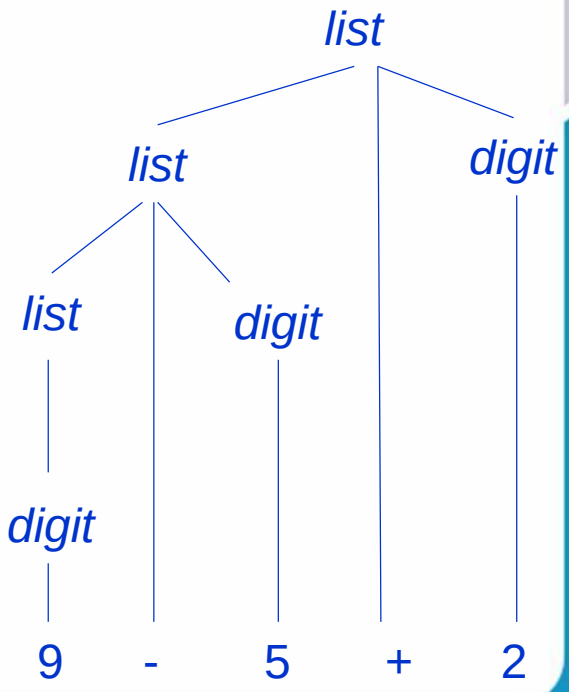
- 语法定义
  - **if** (expression) statement **else** statement
  - *stmt* → **if** (*expr*) *stmt* **else** *stmt*
    - 产生式
      - 终结符：**if** **else**
      - 非终结符：*expr* *stmt*

第二章 一个简单的语法制导翻译器

- 文法定义
  - 上下文无关文法（context-free grammar）
    - 终结符号集合
    - 非终结符号集合
    - 产生式集合
    - 开始符号 ∈ 非终结符号集合
    - 以计算表达式 9 – 5 + 2 为例

第二章 一个简单的语法制导翻译器

- 文法定义
  - 上下文无关文法（context-free grammar）
    - *list* → *list* + *digit*
    - *list* → *list* - *digit*
    - *list* → *digit*
    - *digit* → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  - 以计算表达式 9 – 5 + 2 为例



第二章 一个简单的语法制导翻译器

- 文法定义
  - 上下文无关文法（context-free grammar）
    - 终结符号集合：+ - 1 2 3 4 5 6 7 8 9
    - 非终结符号集合
    - 产生式集合
    - 开始符号 ∈ 非终结符号集合
    - 以计算表达式 9 – 5 + 2 为例

第二章 一个简单的语法制导翻译器

- 文法定义
  - 上下文无关文法（context-free grammar）
    - 终结符号集合：+ - 1 2 3 4 5 6 7 8 9
    - 非终结符号集合：*list digit*
    - 产生式集合
    - 开始符号 ∈ 非终结符号集合
    - 以计算表达式 9 – 5 + 2 为例

第二章 一个简单的语法制导翻译器

- 文法定义
  - 上下文无关文法（context-free grammar）
    - 终结符号集合：+ - 1 2 3 4 5 6 7 8 9
    - 非终结符号集合：*list digit*
    - 产生式集合：
      - *list* → *list* + *digit*
      - *list* → *list* - *digit*
      - *list* → *digit*
      - *digit* → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    - 开始符号 ∈ 非终结符号集合

第二章 一个简单的语法制导翻译器

- 文法定义
  - 上下文无关文法（context-free grammar）
    - 终结符号集合：+ - 1 2 3 4 5 6 7 8 9
    - 非终结符号集合：*list* *digit*
    - 产生式集合：
      - *list* → *list* + *digit*
      - *list* → *list* - *digit*
      - *list* → *digit*
      - *digit* → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    - 开始符号 ∈ 非终结符号集合: *list*

第二章 一个简单的语法制导翻译器

- 推导（derivation）
  - 从开始符号出发，不断替换产生式左端的非终结符
  - 可以从开始符号推导得到的所有终结符号串的集合称为该文法定义的语言（language）
    - 例：9 – 5 + 2
      - *list* → *list* + *digit*
      - *list* → *list* - *digit*
      - *list* → *digit*
      - *digit* → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    - {1, 2, 1 + 1, 2 – 1, ….}是该文法定义的语言

第二章 一个简单的语法制导翻译器

- 推导（derivation）
  - 从开始符号出发，不断替换产生式左端的非终结符
  - 可以从开始符号推导得到的所有终结符号串的集合称为该文法定义的语言（language）
    - 另例：Java函数调用
      - max(x, y)

第二章 一个简单的语法制导翻译器

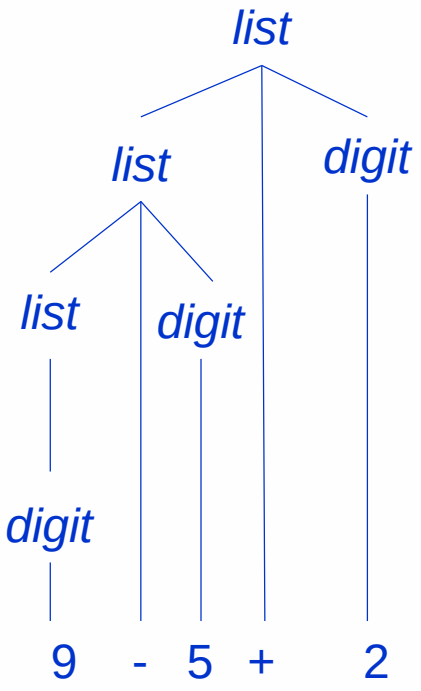
- 推导（derivation）
  - 从开始符号出发，不断替换产生式左端的非终结符
  - 可以从开始符号推导得到的所有终结符号串的集合称为该文法定义的语言（language）
    - 另例：Java函数调用
      - max(x, y)
      - 设计一个文法如下
        - *call* → **id** (*opt_params*)
        - *opt_params* → *params* | ε
        - *params* → *params*, *param* | *param*

第二章 一个简单的语法制导翻译器

- 语法分析：
  - 接受一个终结符号串作为输入，找出从文法的开始符号推导出这个串的方法
- 语法分析树：
  - 用图形（树）的方式展现语法分析的过程
  - 上下文无关文法的语法分析树：
    - 根节点的标号为文法的开始符号
    - 每个叶子节点的标号为一个终结符号或ε
    - 每个内部节点的标号为一个非终结符号
    - 内部节点 A，子节点$X_1, X_2, \ldots X_n$， 那么必然存在产生式： $A \rightarrow X_1 X_2 \ldots X_n$. 其中$X_1, X_2, \ldots X_n$可以是终结符，也可以是非终结符。

第二章 一个简单的语法制导翻译器

- 语法分析树
  - *list* → *list* + *digit*
  - *list* → *list* - *digit*
  - *list* → *digit*
  - *digit* → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



第二章 一个简单的语法制导翻译器

- 语法分析：
  - 为一个给定的终结符号串构建一棵句法树的过程称为对该符号串进行语法分析
  - 一些术语：
    - 语言（language）
    - 结果（yield）
    - 叶子节点/终结节点
    - 内部节点/非终结节点

第二章 一个简单的语法制导翻译器

- 文法的二义性
  - 一个文法可能有多棵语法分析树能够生成同一个给定的终结符号串
- 例：

  string → string + string

  string → string – string

  string → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

给定符号串9 – 5 + 2

其两棵语法树：图2-6

第二章 一个简单的语法制导翻译器

- 运算符的结合性
  - 运算符的左（右）结合性：
    - 当一个运算分量左右两侧都有该运算符时，该运算分量属于其左（右）边的运算符
  - 左结合文法
    - *list* → *list* + *digit*
    - *list* → *list* - *digit*
    - *list* → *digit*
    - *digit* → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    - 例：9 – 5 – 2 （图2-7左图）

第二章 一个简单的语法制导翻译器

- 运算符的结合性
  - 运算符的左（右）结合性：
    - 当一个运算分量左右两侧都有该运算符时，该运算分量属于其左（右）边的运算符
  - 右结合文法
    - *right* → *letter* = *right* | *letter*
    - *letter* → *a* | *b* | … | *z*
    - 例：a = b = c （图2-7右图）

第二章 一个简单的语法制导翻译器

- 运算符的优先级
- 例：+ - * /
  - 具有相同结合性与优先级的运算符放一类
    - + -
    - * /
  - 为表达式创建基本单元：数位和带括号的表达式
    - *factor* → **digit** | (*expr*)
  - 次高优先级，且考虑左结合性
    - *term* → *term* * *factor*
      | *term* / *factor*
      | *factor*

第二章 一个简单的语法制导翻译器

- 运算符的优先级
- 例：+ - * /
  - 次高优先级，且考虑左结合性
    - *expr* → *expr* + *term*
      | *expr* - *term*
      | *term*

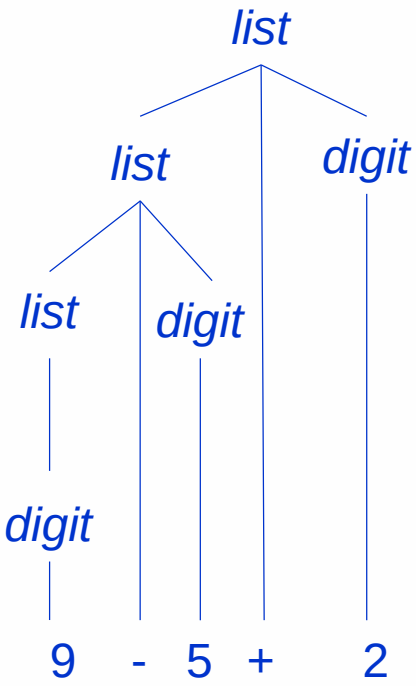
第二章 一个简单的语法制导翻译器

- 运算符的优先级
- 例：+ - * /

  - *factor* → **digit** | (*expr*)
  - *term* → *term* * *factor*
    | *term* / *factor*
    | *factor*
  - *expr* → *expr* + *term*
    | *expr* - *term*
    | *term*

第二章 一个简单的语法制导翻译器

- 语法制导翻译：对语法树进行**语义分析**
  - 例如：将中缀表达式转化成为后缀
    - 9 - 5 + 2 → 9 5 – 2 +



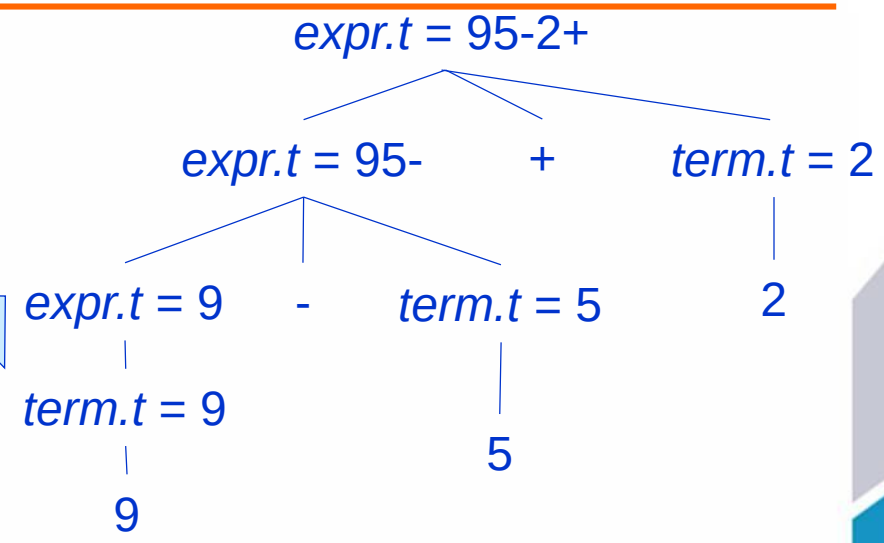
→ ?

第二章 一个简单的语法制导翻译器

- 语法制导翻译：对语法树进行**语义分析**
  - 语法制导定义
  - 语法制导翻译方案

第二章 一个简单的语法制导翻译器

- 语法制导翻译
  - 语法制导定义 (syntax-directed definition)
    - 每个文法符号和一个属性集合相关联
      - 例树中".t"是属性
    - 每个产生式和一组语义规则相关联



产生式	语义规则
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid$ '+'
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid$ '-'
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t =$ '0'
$term \rightarrow 1$	$term.t =$ '1'
…	…
$term \rightarrow 9$	$term.t =$ '9'

第二章 一个简单的语法制导翻译器

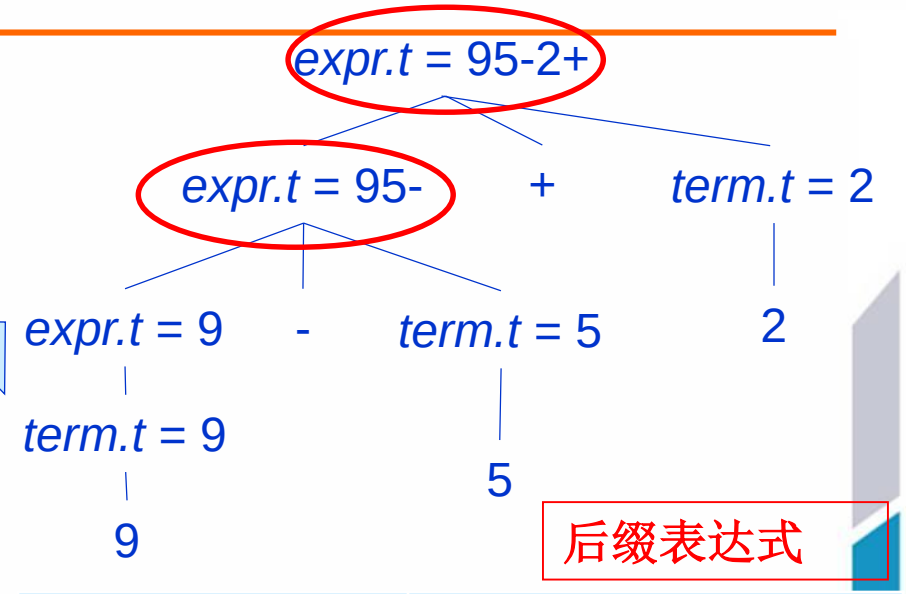
- 语法制导翻译
  - 属性
    - 综合属性
      - 如果某个属性在语法分析树节点N上的值由N的子节点和N本身的属性值确定，则该属性为综合属性
    - 继承属性
      - 如果某个属性由语法分析树中该节点本身、父节点以及兄弟节点上的属性值决定，则该属性为继承属性

第二章 一个简单的语法制导翻译器

- 后缀表达式（postfix notation）：*E*
  - 如何*E*是一个变量或常量，则*E*的后缀是本身
  - 如果*E*是一个形如E_1 **op** E_2的表达式，**op**是二目运算符，那么*E*的后缀表示是：E_1' E_2' **op**，这里E_1'和E_2'分别是E_1和E_2的后缀表示
  - 如果*E*是一个形如（E_1）的表达式，则E的后缀表示就是E_1的后缀表示

第二章 一个简单的语法制导翻译器

- 语法制导翻译
  - 语法制导定义 (syntax-directed definition)
    - 每个文法符号和一个属性集合相关联
      - 例树中".t"是属性
    - 每个产生式和一组语义规则相关联

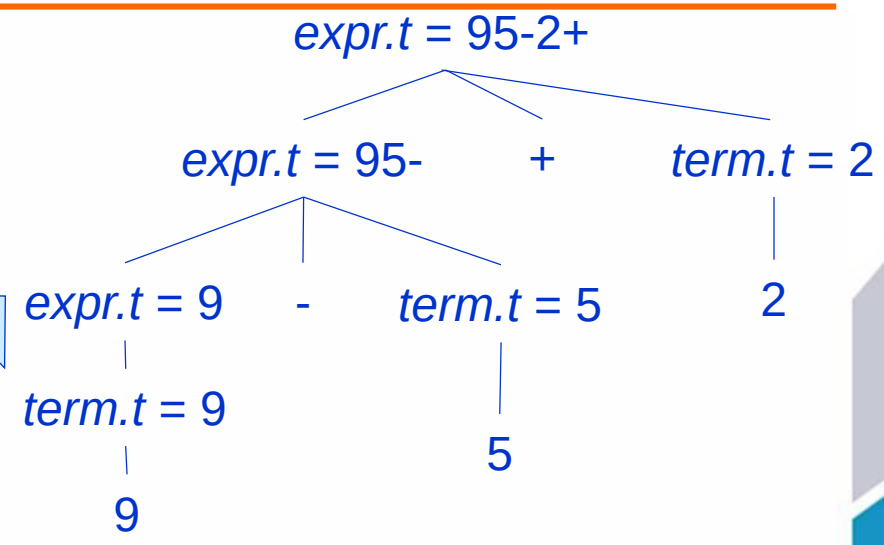


产生式	语义规则
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid$ '+'
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid$ '-'
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t =$ '0'
$term \rightarrow 1$	$term.t =$ '1'
…	…
$term \rightarrow 9$	$term.t =$ '9'

第二章 一个简单的语法制导翻译器

- 语法制导翻译
  - 语法制导定义 (syntax-directed definition)
    - 每个文法符号和一个属性集合相关联
      - 例树中".t"是属性
    - 每个产生式和一组语义规则相关联

简单语法制导定义

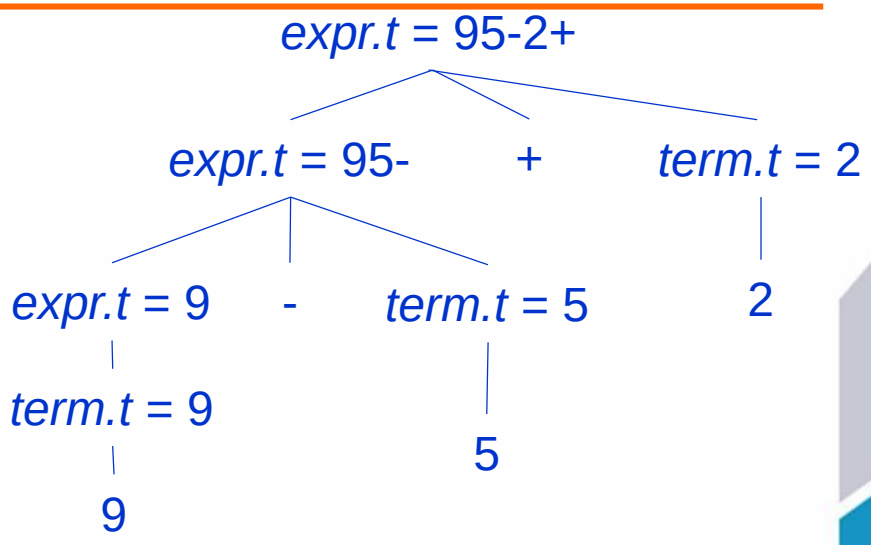


产生式	语义规则
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid$ '+'
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid$ '-'
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t =$ '0'
$term \rightarrow 1$	$term.t =$ '1'
…	…
$term \rightarrow 9$	$term.t =$ '9'

第二章 一个简单的语法制导翻译器

- 语法制导翻译
  - 语法制导定义 (syntax-directed definition)
  - 深度优先遍历

```
procedure visit ( node N) {
    for(从左到右遍历N的每个子节点C) {
        visit(C);
    }
    按照节点N上的语义规则求值;
}
```



产生式	语义规则
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid$ '+'
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid$ '-'
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t =$ '0'
$term \rightarrow 1$	$term.t =$ '1'
…	…
$term \rightarrow 9$	$term.t =$ '9'

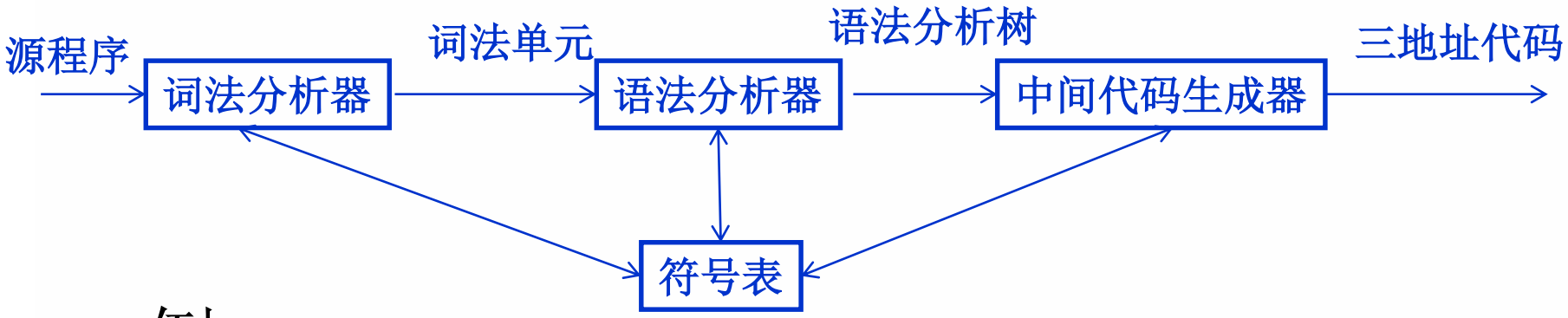
编译原理

第二章 一个简单的语法制导翻译器 上周课回顾

计算机科学与技术学院

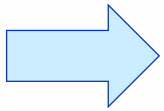
第二章 一个简单的语法制导翻译器

- 一个编译器的前端



- 例：

do i = i + 1; while (a[i] < v)



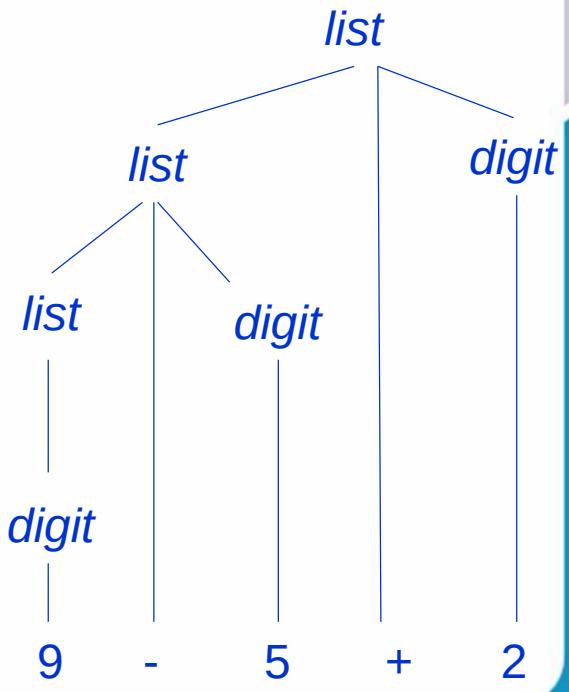
1: i = i + 1
2: t1 = a [i]
3: if t1 < v goto 1

第二章 一个简单的语法制导翻译器

- 文法定义
  - 上下文无关文法（context-free grammar）
    - 终结符号集合
    - 非终结符号集合
    - 产生式集合
    - 开始符号 ∈ 非终结符号集合
    - 以计算表达式 9 – 5 + 2 为例

第二章 一个简单的语法制导翻译器

- 文法定义
  - 上下文无关文法（context-free grammar）
    - *list* → *list* + *digit*
    - *list* → *list* - *digit*
    - *list* → *digit*
    - *digit* → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  - 以计算表达式 9 – 5 + 2 为例

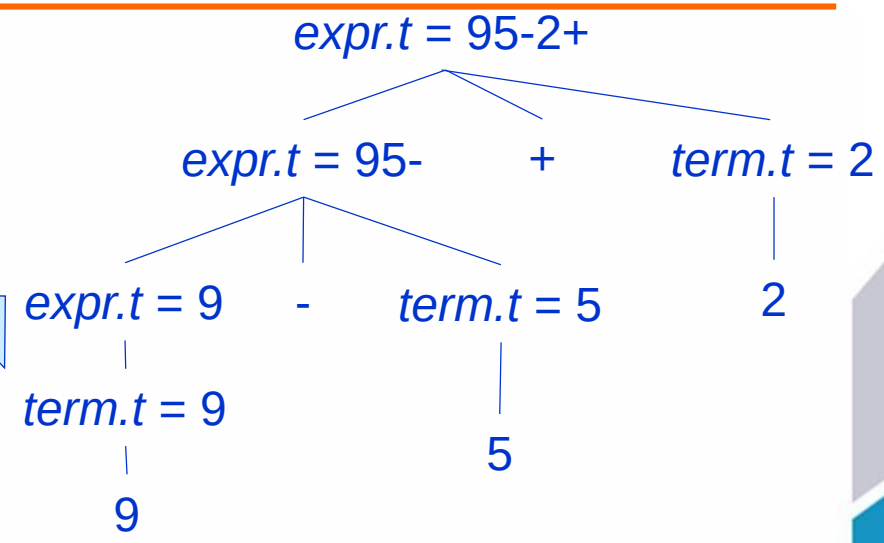


第二章 一个简单的语法制导翻译器

- 语法制导翻译：对语法树进行语义分析
  - 语法制导定义
  - 语法制导翻译方案

第二章 一个简单的语法制导翻译器

- 语法制导翻译
  - 语法制导定义 (syntax-directed definition)
    - 每个文法符号和一个属性集合相关联
      - 例树中".t"是属性
    - 每个产生式和一组语义规则相关联

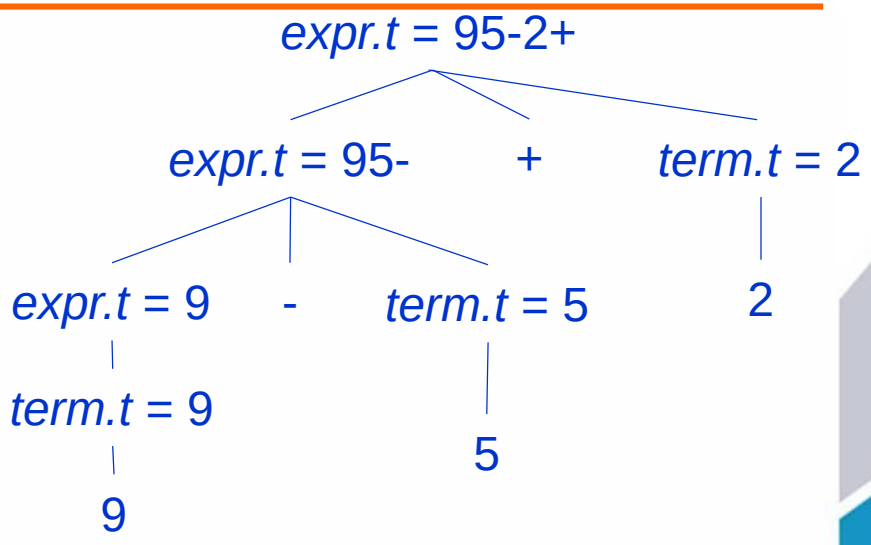


产生式	语义规则
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid$ '+'
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid$ '-'
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t =$ '0'
$term \rightarrow 1$	$term.t =$ '1'
…	…
$term \rightarrow 9$	$term.t =$ '9'

第二章 一个简单的语法制导翻译器

- 语法制导翻译
  - 语法制导定义 (syntax-directed definition)
  - 深度优先遍历

```
procedure visit ( node N) {
    for(从左到右遍历N的每个子节点C) {
        visit(C);
    }
    按照节点N上的语义规则求值;
}
```



产生式	语义规则
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid$ '+'
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \mid\mid term.t \mid\mid$ '-'
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t =$ '0'
$term \rightarrow 1$	$term.t =$ '1'
…	…
$term \rightarrow 9$	$term.t =$ '9'

编译原理

第二章 一个简单的语法制导翻译器 第二次课

计算机科学与技术学院

第二章 一个简单的语法制导翻译器

- 语法制导翻译：对语法树进行语义分析
  - 语法制导定义
  - 语法制导翻译方案
    - 将程序片段附加到一个文法的各个产生式上的表示法

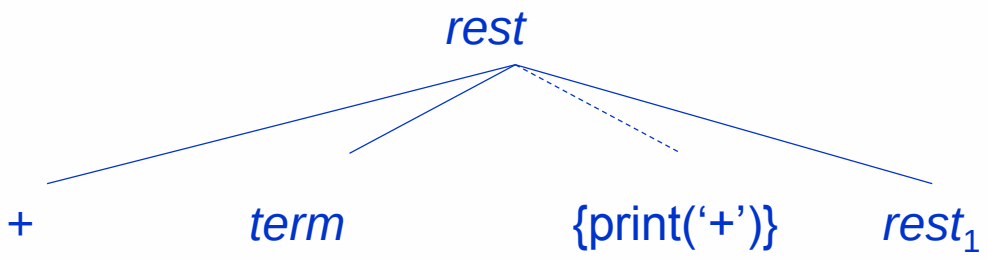
$$rest \rightarrow + \ term \ \{ \ print (\ ' + ' \ ) \ \} \ rest_1$$

第二章 一个简单的语法制导翻译器

- 语法制导翻译：对语法树进行**语义分析**
  - 语法制导定义
  - **语法制导翻译方案**
    - 将程序片段附加到一个文法的各个产生式上的表示法

$$rest \rightarrow + \ term \ \{ \text{print (‘ + ‘) } \} \ rest_1$$

    - 被嵌入到产生式体中的程序片段成为语义动作（semantic action）。语义动作用花括号括起来。



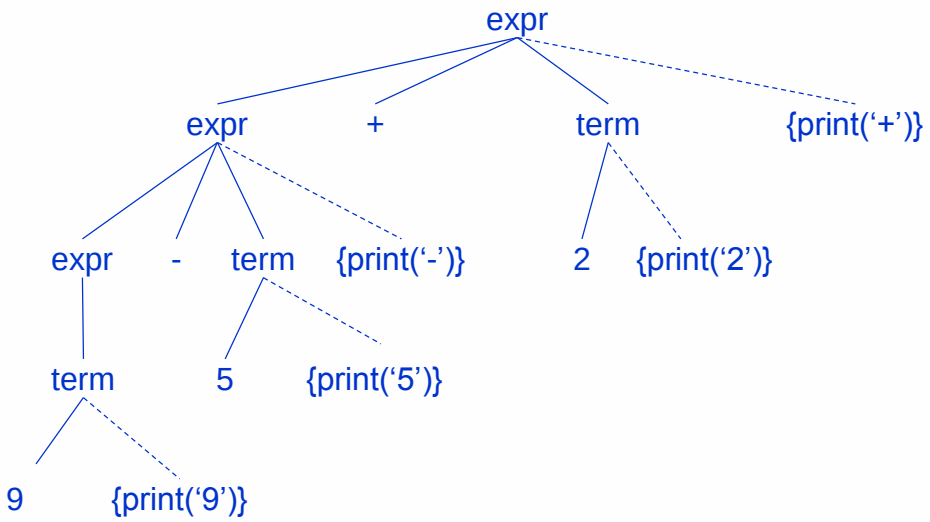
第二章 一个简单的语法制导翻译器

- 语法制导翻译：对语法树进行**语义分析**
  - 语法制导定义
  - **语法制导翻译方案**
    - 将程序片段附加到一个文法的各个产生式上的表示法
      - 如何E是一个变量或常量，则E的后缀是本身
      - 如果E是一个形如E_1 **op** E_2的表达式，**op**是二目运算符，那么E的后缀表示是：E_1' E_2' **op**，这里E_1'和E_2'分别是E_1和E_2的后缀表示
      - 如果E是一个形如（E_1）的表达式，则E的后缀表示就是E1的后缀表示

  - (9 – 5) + 2 → 9 5 – 2 +
  - ? ← 9 5 2 + - 3 *

第二章 一个简单的语法制导翻译器

- 语法制导翻译：对语法树进行**语义分析**
  - 语法制导定义
  - 语法制导翻译方案



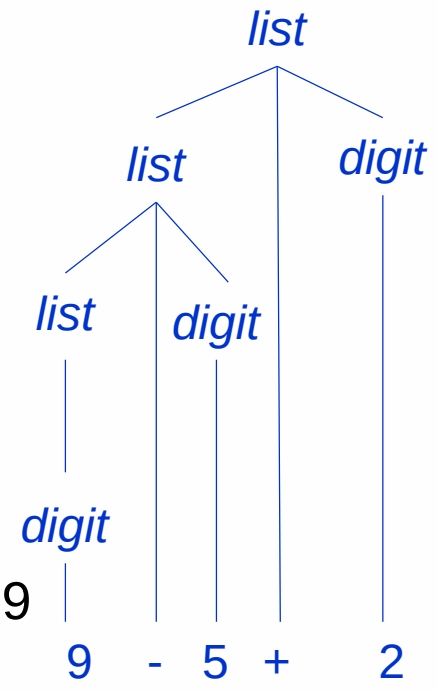
翻译方案	
$expr \rightarrow expr_1 + term$	{print('+')}
$expr \rightarrow expr_1 - term$	{print('-')}
$expr \rightarrow term$	
$term \rightarrow 0$	{print('0')}
$term \rightarrow 1$	{print('1')}
…	
$term \rightarrow 9$	{print('9')}

第二章 一个简单的语法制导翻译器

- 语法分析
  - 决定如何使用一个文法生成一个终结符号串的过程
    - 给定输入：
      - 终结符号串：9 – 5 + 2
      - 文法：
        - *list* → *list* + *digit*
        - *list* → *list* - *digit*
        - *list* → *digit*
        - *digit* → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    - 输出：？

第二章 一个简单的语法制导翻译器

- 语法分析
  - 决定如何使用一个文法生成一个终结符号串的过程
    - 给定输入：
      - 终结符号串：9 – 5 + 2
      - 文法：
        - *list* → *list* + *digit*
        - *list* → *list* - *digit*
        - *list* → *digit*
        - *digit* → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    - 输出：？



第二章 一个简单的语法制导翻译器

- 语法分析
  - 决定如何使用一个文法生成一个终结符号串的过程
    - 上下文无关文法
      - 通常Parsing的时间复杂度是$O(n^3)$
      - 对于程序设计语言，时间复杂度是$O(n)$
        - **自顶向下方法**
        - 自底向上方法

第二章 一个简单的语法制导翻译器

- 语法分析
  - 自顶向下分析方法
    - 例：给定文法：

stmt → **expr**
| **if (expr)** *stmt*
| **for (** *optexpr* **;** *optexpr* **;** *optexpr* **)** *stmt*
| **other**

optexpr → ε
| **expr**

第二章 一个简单的语法制导翻译器

- 语法分析
  - 自顶向下分析方法

stmt → **expr**
| **if (expr)** *stmt*
| **for (** *optexpr* **;** *optexpr* **;** *optexpr* **)** *stmt*
| **other**

optexpr → ε
| **expr**

输入是：**for (; expr ; expr) other**

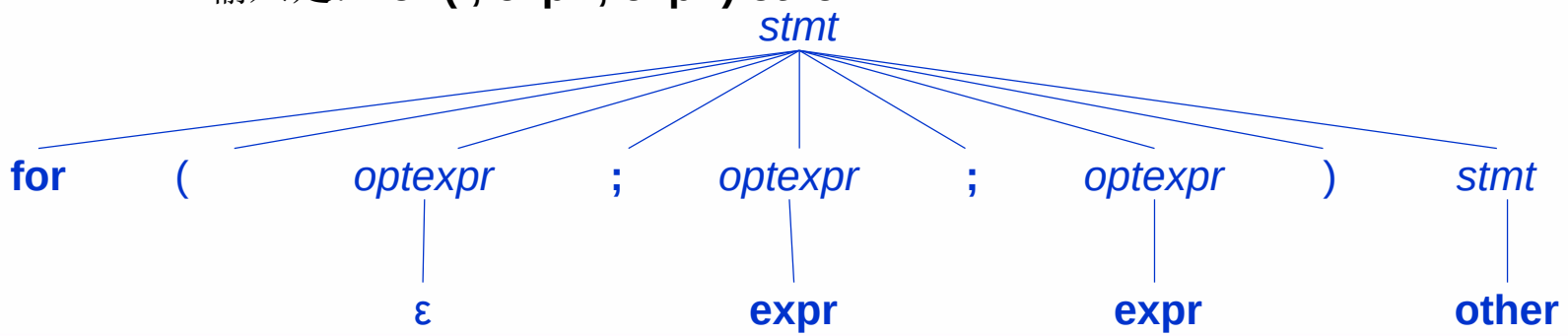
第二章 一个简单的语法制导翻译器

- 语法分析
  - 自顶向下分析方法

stmt → **expr ;**
| **if (expr)** *stmt*
| **for (** *optexpr* **;** *optexpr* **;** *optexpr* **)** *stmt*
| **other**

optexpr → ε
| **expr**

输入是：**for (; expr ; expr) other**



语法
分析树

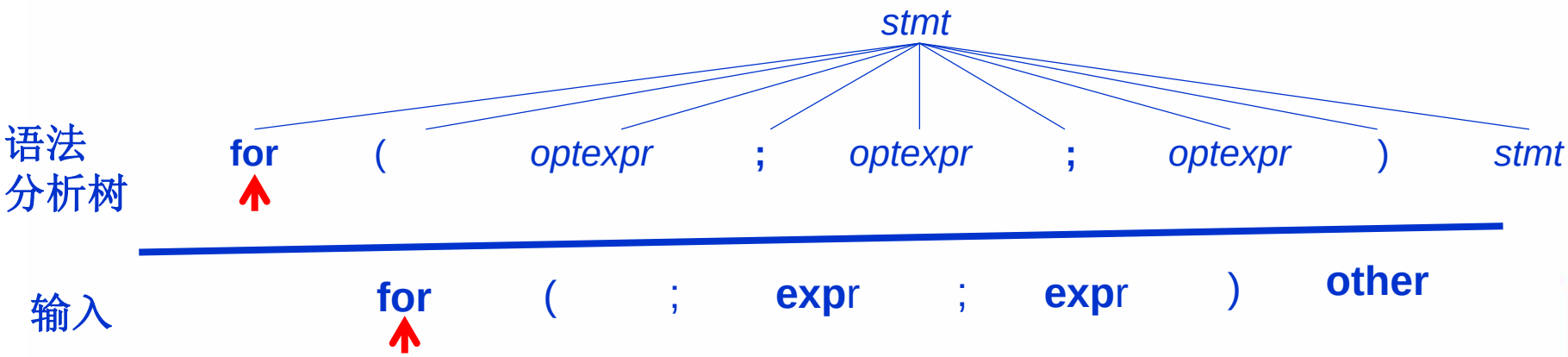
stmt

输入　　**for**　　(　　;　　**expr**　　;　　**expr**　　)　　**other**

语法
分析树

stmt

输入 **for** (; **expr** ; **expr**) **other**



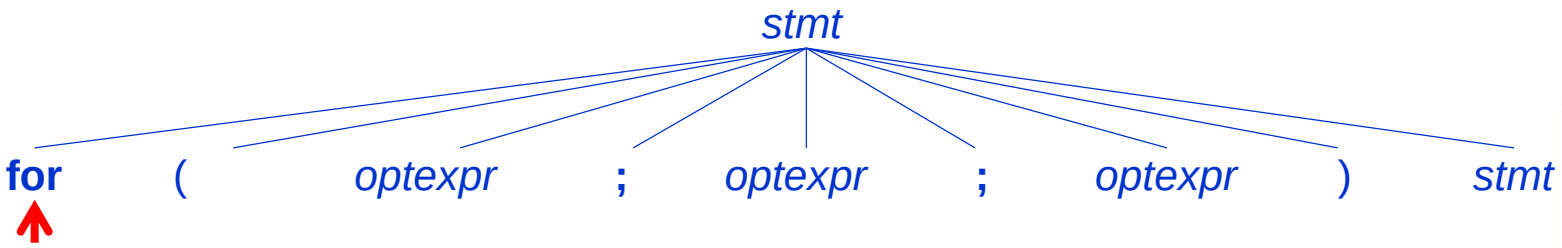
stmt

语法
分析树

stmt

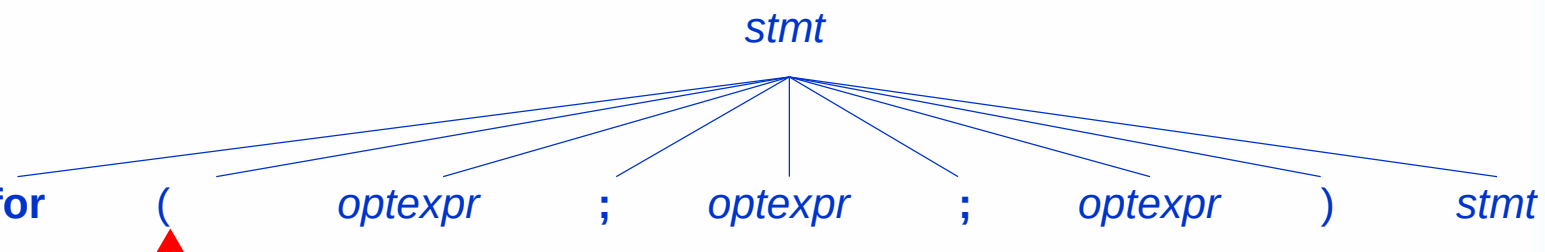
输入 **for** (; **expr** ; **expr**) **other**

语法
分析树



输入 **for** (; **expr** ; **expr**) **other**

语法
分析树



输入 **for** (; **expr** ; **expr**) **other**

第二章 一个简单的语法制导翻译器

- 语法分析
  - 自顶向下分析方法
    - 一般来说，为一个非终结符号选择产生式是一个“尝试并犯错”的过程 —— 回溯
    - 预测语法分析法：不需要回溯
      - 要求设计的文法满足：当考虑到一个输入符号（终结符）的时候，只有一种非终结符可以生成这个输入符号，是确定性的。
      - FIRST集合
        - 令α是一个文法符号（终结符号或非终结符号）串，FIRST(α)是由α生成的一个或多个终结符号串的第一个符号的集合。
        - 如果α是ε或者可以生成ε，那么ε也在FIRST(α)中

第二章 一个简单的语法制导翻译器

- 语法分析
  - 自顶向下分析方法
    - 一般来说，为一个非终结符号选择产生式是一个“尝试并犯错”的过程 —— 回溯
    - 预测语法分析法：不需要回溯
      - 要求设计的文法满足：当考虑到一个输入符号（终结符）的时候，只有一种非终结符可以生成这个输入符号，是确定性的。
        - 有两个产生式A → α和A→β，预测分析法要求FIRST(α)和FIRST(β)不相交
        - 如果输入符号在FIRST(α)中，就用A → α，如果输入符号在FIRST(β)中，就用A → β

第二章 一个简单的语法制导翻译器

- 语法分析
  - 自顶向下分析方法
    - 预测分析法

stmt → **expr ;**
| **if (expr)** *stmt*
| **for (** *optexpr* **;** *optexpr* **;** *optexpr* **)** *stmt*
| **other**

optexpr → ε
| **expr**

- FIRST(*stmt*) = {**expr**, **if**, **for**, **other**}
- FIRST(**expr ;**) = { **expr** }

```
void stmt() {
    switch(lookahead) {
    case expr:
        match(expr); match(';'); break;
    case if:
        match(if); match('('); match(expr); match(')'); stmt();
        break;
    case for:
        match(for); match('(');
        optexpr(); match(';'); optexpr(); match(';'); optexpr();
        match(')'); stmt(); break;
    case other:
        match(other); break;
    default:
        report("syntax error");
    }
}
```

stmt → **expr**
| **if (expr)** *stmt*
| **for (** *optexpr* **;** *optexpr* **;** *optexpr* **)** *stmt*
| **other**

第二章 一个简单的语法制导翻译器

```
void optexpr() {
      if(lookahead == expr) match(expr);
}
```

optexpr → ε
| **expr**

```
void match(terminal t) {
      if(lookahead == t) lookahead = nextTerminal;
      else report("syntax error");
}
```

第二章 一个简单的语法制导翻译器

```
void optexpr() {
    if(lookahead == expr) match(expr);
}
```

optexpr → ε
| **expr**

何时使用ε产生式

输入：**for (; expr ; expr) other**

```
void match(terminal t) {
    if(lookahead == t) lookahead = nextTerminal;
    else report("syntax error");
}
```

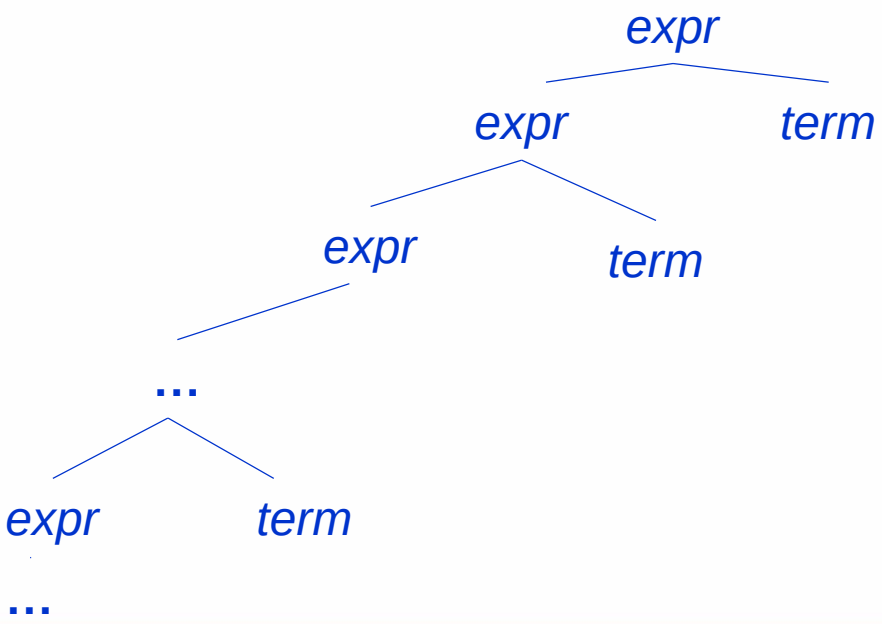
第二章 一个简单的语法制导翻译器

- 设计一个预测分析器
  - 对应于非终结符A：
    - 检查向前看符号，决定使用A的哪个产生式。如果一个产生式的体为α(α不是空串ε)，且向前看符号在FIRST(α)中，那么就选择这个产生式。
    - 然后，模拟被选中的产生式的体，也就是，从左向右逐个执行此产生式体的符号。“执行”就是调用相应非终结符的过程。
- 嵌入翻译方案
  - 翻译动作作为一个非终结符

第二章 一个简单的语法制导翻译器

- 语法分析
  - 左递归
    - 可能使递归下降语法分析器进入无限循环

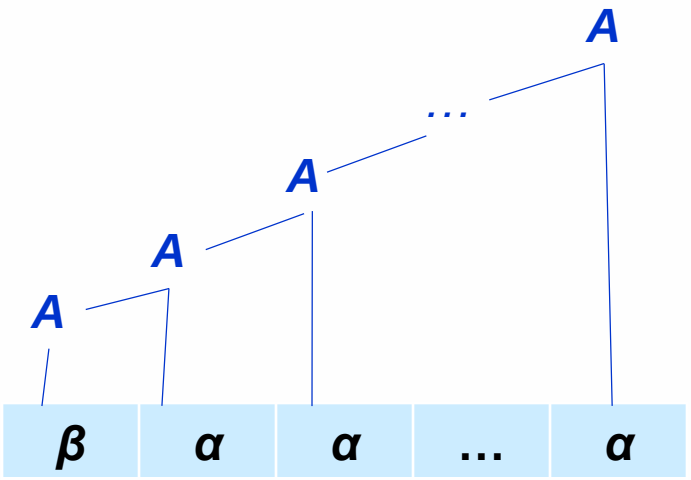
例： *expr* → *expr* + *term*



第二章 一个简单的语法制导翻译器

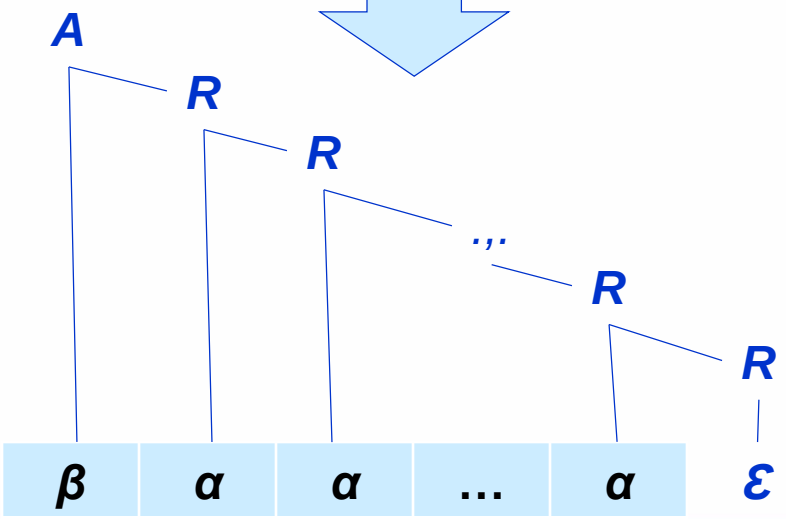
- 语法分析
  - 消除左递归

$A \rightarrow A\,\alpha \mid \beta$



$A \rightarrow \beta\,R$

$R \rightarrow \alpha\ R \mid \varepsilon$



第二章 一个简单的语法制导翻译器

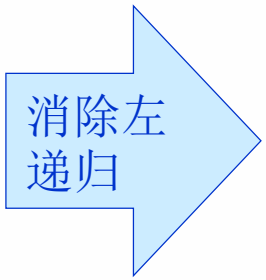
- 实现一个简单表达式的翻译器
  - 中缀表达式 翻译成 后缀表达式

翻译方案	
$expr \rightarrow expr_1 + term$	{print('+')}
$expr \rightarrow expr_1 - term$	{print('-')}
$expr \rightarrow term$	
$term \rightarrow 0$	{print('0')}
$term \rightarrow 1$	{print('1')}
…	
$term \rightarrow 9$	{print('9')}

第二章 一个简单的语法制导翻译器

- 实现一个简单表达式的翻译器
  - 中缀表达式 翻译成 后缀表达式
    - 消除左递归的翻译方案：

$A \rightarrow A\alpha \mid A\beta \mid \gamma$



$A \rightarrow \gamma R$

$R \rightarrow \alpha R \mid \beta R \mid \varepsilon$

第二章 一个简单的语法制导翻译器

- 实现一个简单表达式的翻译器
  - 中缀表达式 翻译成 后缀表达式
    - 消除左递归的翻译方案：

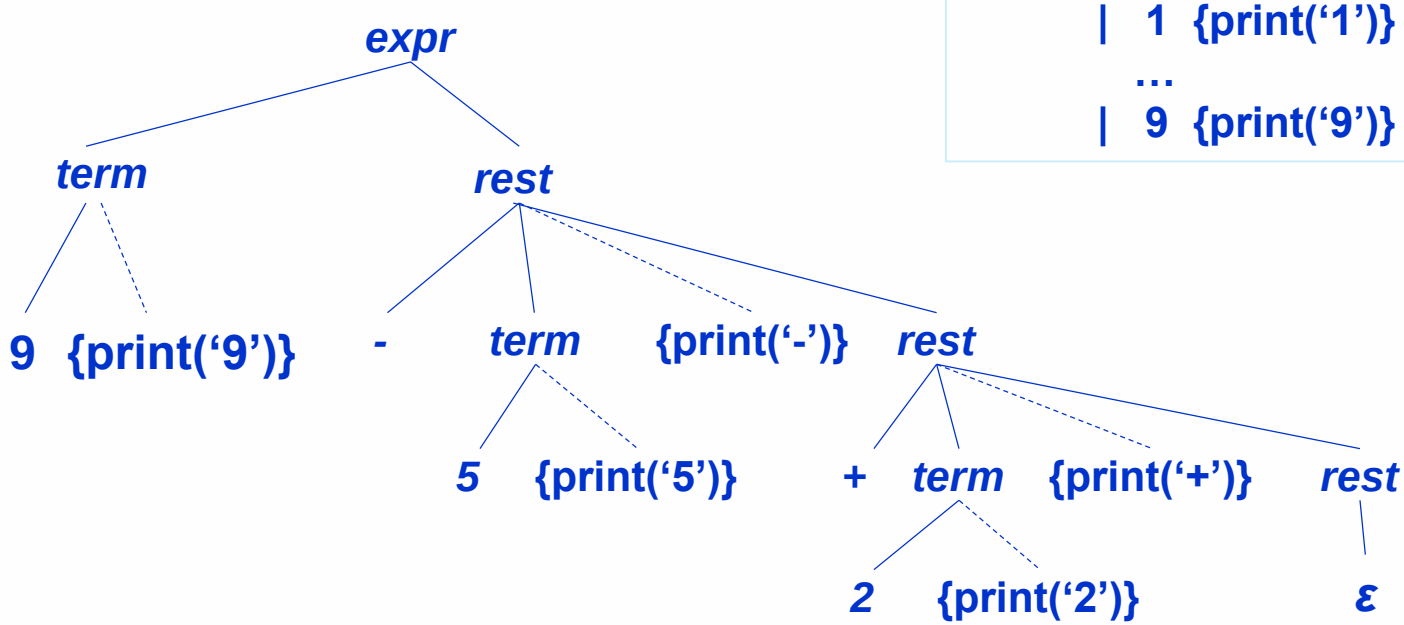
```
expr → expr + term    {print('+')}
     | expr - term    {print('-')}
     | term

term → 0              {print('0')}
     | 1              {print('1')}
     …
     | 9              {print('9')}
```

消除左递归

```
expr → term rest

rest → + term {print('+')} rest
     | - term {print('- ')} rest
     | ε

term → 0  {print('0')}
     | 1  {print('1')}
     …
     | 9  {print('9')}
```

第二章 一个简单的语法制导翻译器

- 实现一个简单表达式的翻译器
  - 中缀表达式 翻译成 后缀表达式
    - 翻译过程图解：



expr → *term rest*

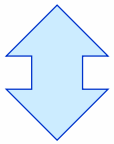
rest → + *term* {print('+')} *rest*
| - *term* {print('- ')} *rest*
| ε

term → 0 {print('0')}
| 1 {print('1')}
…
| 9 {print('9')}

第二章 一个简单的语法制导翻译器

- 实现一个简单表达式的翻译器
  - 中缀表达式 翻译成 后缀表达式
    - 非终结符的过程

expr → term rest

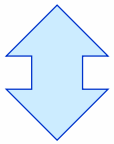


```
void expr() {
    term(); rest()
}
```

第二章 一个简单的语法制导翻译器

- 实现一个简单表达式的翻译器
  - 中缀表达式 翻译成 后缀表达式
    - 非终结符的过程

rest → + *term* {print('+')} *rest*
| - *term* {print('- ')} *rest*
| ε

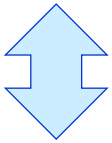


```
void rest () {
   if( lookahead == '+') {
      match('+'); term(); print('+'); rest();
   }
   else if( lookahead == '-') {
      match('-'); term(); print('-'); rest();
   }
   else { }  /*不对输入作任何处理*/
}
```

第二章 一个简单的语法制导翻译器

- 实现一个简单表达式的翻译器
  - 中缀表达式 翻译成 后缀表达式
    - 非终结符的过程

term → 0 {print('0')}
| 1 {print('1')}
…
| 9 {print('9')}



```
void term() {
    if( lookahead是一个数位) {
        t = lookahead; match(lookahead); print(t);
    }
    else {report("语法错误");}
```

第二章 一个简单的语法制导翻译器

- 实现一个简单表达式的翻译器
  - 中缀表达式 翻译成 后缀表达式
    - 翻译器的简化

```
void rest () {
    if( lookahead == '+') {
        match('+'); term(); print('+'); rest();
    }
    else if( lookahead == '-') {
        match('-'); term(); print('-'); rest();
    }
    else { }  /*不对输入作任何处理*/
}
```

第二章 一个简单的语法制导翻译器

- 实现一个简单表达式的翻译器
  - 中缀表达式 翻译成 后缀表达式
    - 翻译器的简化

```
void rest () {
    while(true) {
        if( lookahead == '+') {
            match('+'); term(); print('+'); continue;
        }
        else if( lookahead == '-') {
            match('-'); term(); print('-'); continue;
        }
        break;
    }
}
```

第二章 一个简单的语法制导翻译器

- 实现一个简单表达式的翻译器
  - 中缀表达式 翻译成 后缀表达式
    - 完整的程序：图2-27
      - 类Postfix
      - 类Parser

第二章 一个简单的语法制导翻译器

- 词法分析

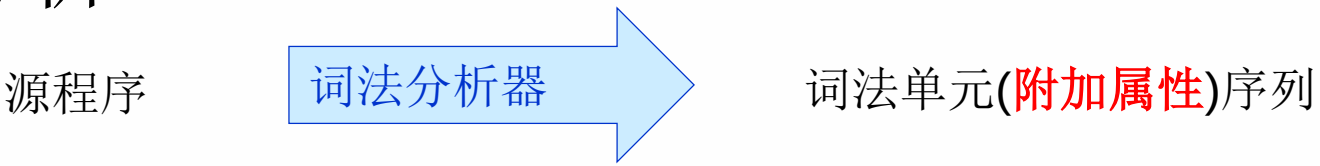
源程序 → 词法分析器 → 词法单元序列

$position := initial + rate * 60$

$id_1 := id_2 + id_3 * 60$

第二章 一个简单的语法制导翻译器

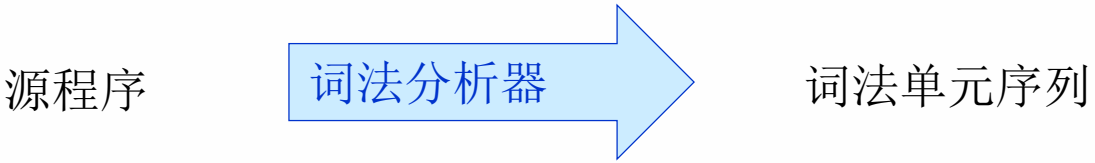
- 词法分析




```
expr → expr + term      {print('+')}
     |  expr -  term     {print('-')}
     |  term

term → term * factor    {print('*')}
     |  term / factor    {print('/')}
     |  factor

Factor → (expr )
       |   num          {print(num.value)}
       |   id           {print(id.lexeme)}
```

第二章 一个简单的语法制导翻译器

- 词法分析



  - 剔除空白和注释
  - 识别和计算常量
  - 识别关键字和标识符

第二章 一个简单的语法制导翻译器

- 词法分析
  - 剔除空白和注释

```
for ( ; ; peek = next input character) {
      if( peek is a blank or a tab ) do nothing;
      else if( peek is a newline)
            line = line + 1;
      else break;
}
```

第二章 一个简单的语法制导翻译器

- 词法分析
  - 识别和计算常量

```
if ( peek holds a digit) {
        v = 0;
        do {
              v = v * 10 + integer value of digit peek;
              peek = next  input  character
         }while ( peek holds a digit);
         return token (num, v);
}
```

第二章 一个简单的语法制导翻译器

- 词法分析
  - 识别关键字和标识符

```
if ( peek 存放了一个字母) {
    将字母或数位读入一个缓冲区b;
    s = b 中的字符形成的字符串;
    w = words.get(s) 返回的词法单元;
    if ( w 不是 null) return w;
    else {
        将键-值对(s, <id, s>)加入到words;
        return 词法单元<id, s>;
    }
}
```

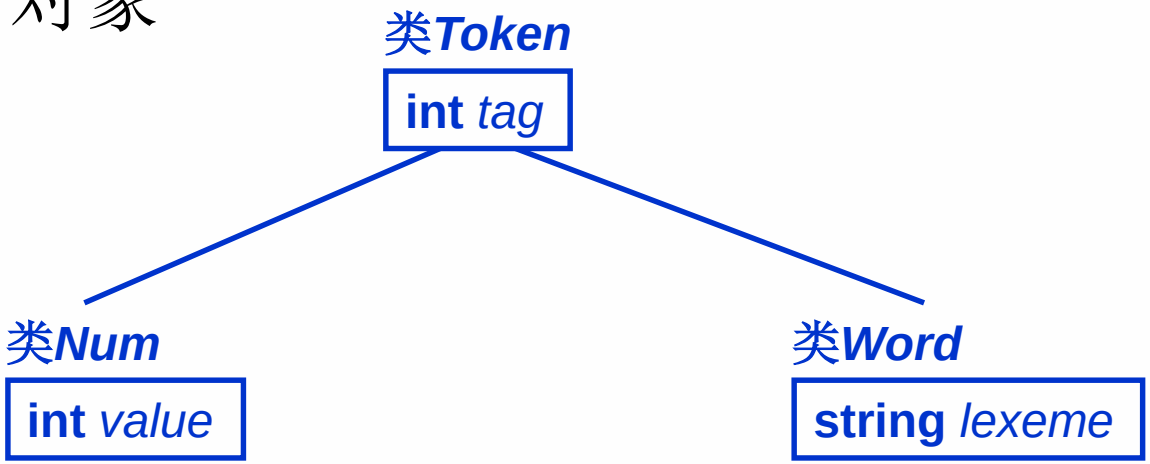
第二章 一个简单的语法制导翻译器

- 词法分析
  - 主流程

```
Token scan () {
    跳过空白符，见2.6.1节；
    处理数字，见2.6.3节；
    处理保留字和标识符，见2.6.4节；
    /*如果运行到这里，就将预读字符peek作为一个词法单元*/
    Token t = new Token (peek);
     peek = 空白符 /*按照2.6.2讨论的方法初始化*/;
     return t;
}
```

第二章 一个简单的语法制导翻译器

- 词法分析
  - 各个对象



第二章 一个简单的语法制导翻译器

- 词法分析
  - 类Token

```
//Token.java
Package lexer;
public class Token {
      public  final  int  tag;
      public  Token ( int  t ) { tag = t; }
}
```

```
//Tag.java
Package lexer;
public class Tag {
     public final  static int
     NUM = 256; ID = 257; TRUE = 258; FALSE = 259;
}
```

第二章 一个简单的语法制导翻译器

- 词法分析
  - 子类Num和子类Word

```
package lexer;              //文件Num.java
public  class  Num extends Token {
    public  final int value;
    public  Num ( int v )  {super(Tag.NUM); value = v; }
}

package lexer;              //文件Word.java
public  class  Word  extends  Token {
      public  final  String  lexeme;
      public  Word ( int t, String s)  {
            super(t); lexeme = new String (s);
      }
}
```

第二章 一个简单的语法制导翻译器

- 词法分析
  - 词法分析器：Lexer
    - 图2-34、图2-35

第二章 一个简单的语法制导翻译器

```
package lexer;                    //文件Lexer.java
import java.io.*; import java.util.*;
public class Lexer {
        public int line = 1;
        private char peek = ' ';
        private Hashtable words = new Hashtable();
        void reserve(Word t) {words.put(t.lexeme, t);}
        public Lexer () {
                reserve(new Word(Tag.TRUE, "true"));
                reserve(new Word(Tag.FALSE, "false"));
        }
        public Token scan() throws IOException {
                for( ; ; peek = (char)System.in.read() ) {
                        if(peek == ' ' || peek == '\t') continue;
                        else if(peek == '\n') line = line + 1;
                        else break;
                }
```

```
            if(Character.isDigit(peek)) {
                int v = 0;
                do {
                    v = 10 * v + Character.digit(peek, 10);
                    peek = (char)System.in.read();
                }while(Character.isDigit(peek));
                return new Num(v);
            }
            if(Character.isLetter(peek)) {
                StringBuffer b = new StringBuffer();
                do {
                    b.append(peek);
                    peek = (char) System.in.read();
                }while(Character.isLetterOrDigit(peek));
                String s = b.toString();
                Word w = (Word)words.get(s);
                if( w != null ) return w;
                w = new Word(Tag.ID, s);
                words.put(s, w);
                return w;
            }
            Token t = new Token(peek);
            peek = ' ';
            return t;
        }
}
```

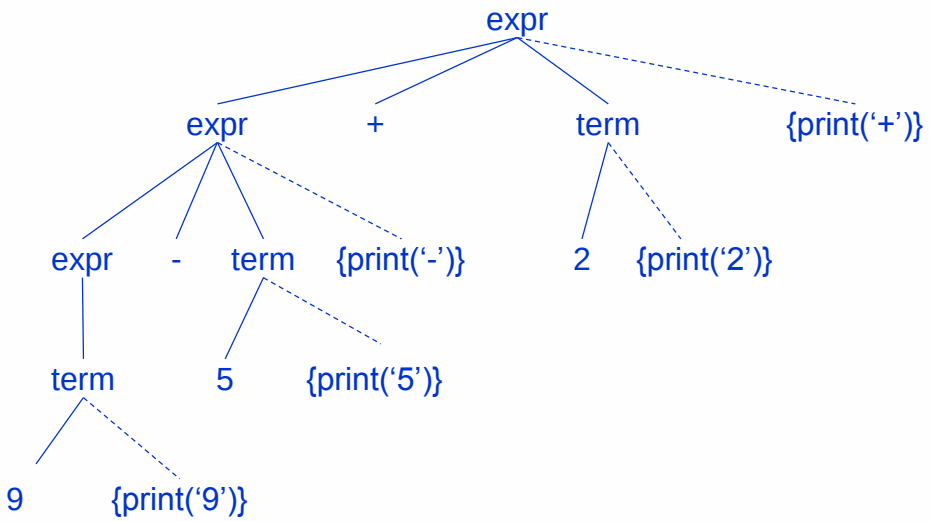
编译原理

第二章 一个简单的语法制导翻译器 上周课回顾

计算机科学与技术学院

第二章 一个简单的语法制导翻译器

- 语法制导翻译：对语法树进行**语义分析**
  - 语法制导定义
  - 语法制导翻译方案



翻译方案	
$expr \rightarrow expr_1 + term$	{print('+')}
$expr \rightarrow expr_1 - term$	{print('-')}
$expr \rightarrow term$	
$term \rightarrow 0$	{print('0')}
$term \rightarrow 1$	{print('1')}
…	
$term \rightarrow 9$	{print('9')}

自顶向下分析法:

stmt → **expr**
| **if (expr)** *stmt*
| **for (** *optexpr* ;
optexpr ;
optexpr **)**
stmt
| **other**

optexpr → **ε** | **expr**

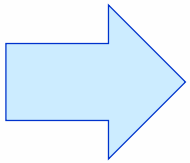
```
void stmt() {
    switch(lookahead) {
    case expr:
        match(expr); match(';'); break;
    case if:
        match(if); match('('); match(expr); match(')');
stmt();
        break;
    case for:
        match(for); match('(');
        optexpr(); match(';'); optexpr(); match(';'); optexpr();
        match(')'); stmt(); break;
    case other:
        match(other); break;
    default:
        report("syntax error");
    }
}
```

```
void optexpr() {if(lookahead == expr) match(expr);}

void match(terminal t) {
      if(lookahead == t) lookahead = nextTerminal;
      else report("syntax error");
}
```

第二章 一个简单的语法制导翻译器

- 基于自顶向下分析法实现翻译方案：

```
expr → expr + term    {print('+')}
     |  expr -  term   {print('-')}
     |  term

term  →  0            {print('0')}
      |  1            {print('1')}
         …
      |  9            {print('9')}
```

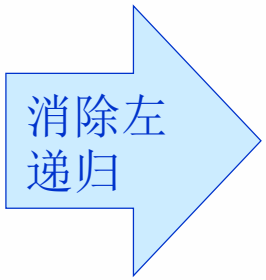


```
void expr(){
   case … {
        expr();
        match('+');
        term();
        print('+');
   }

   …….
}
```

第二章 一个简单的语法制导翻译器

- 实现一个简单表达式的翻译器
  - 中缀表达式 翻译成 后缀表达式
    - 消除左递归的翻译方案：

$A \rightarrow A\alpha \mid A\beta \mid \gamma$



$A \rightarrow \gamma R$

$R \rightarrow \alpha R \mid \beta R \mid \varepsilon$

第二章 一个简单的语法制导翻译器

- 消除左递归的翻译方案：

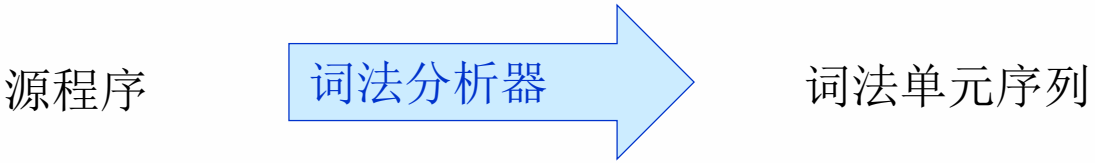
```
expr → expr + term    {print('+')}
     | expr - term    {print('-')}
     | term

term → 0              {print('0')}
     | 1              {print('1')}
     …
     | 9              {print('9')}
```

消除左递归

```
expr → term rest

rest → + term {print('+')} rest
     | - term {print('- ')} rest
     | ε

term → 0 {print('0')}
     | 1 {print('1')}
     …
     | 9 {print('9')}
```

第二章 一个简单的语法制导翻译器

- 词法分析



  - 剔除空白和注释
  - 识别和计算常量
  - 识别关键字和标识符

第二章 一个简单的语法制导翻译器

- 词法分析
  - 剔除空白和注释

```
for ( ; ; peek = next input character) {
      if( peek is a blank or a tab ) do nothing;
      else if( peek is a newline)
            line = line + 1;
      else break;
}
```

第二章 一个简单的语法制导翻译器

- 词法分析
  - 识别和计算常量

```
if ( peek holds a digit) {
        v = 0;
        do {
              v = v * 10 + integer value of digit peek;
              peek = next  input  character
         }while ( peek holds a digit);
         return token (num, v);
}
```

第二章 一个简单的语法制导翻译器

- 词法分析
  - 识别关键字和标识符

```
if ( peek 存放了一个字母) {
    将字母或数位读入一个缓冲区b;
    s = b 中的字符形成的字符串;
    w = words.get(s) 返回的词法单元;
    if ( w 不是 null) return w;
    else {
        将键-值对(s, <id, s>)加入到words;
        return 词法单元<id, s>;
    }
}
```

编译原理

第二章 一个简单的语法制导翻译器 第三次课

计算机科学与技术学院

第二章 一个简单的语法制导翻译器

- 符号表
  - 符号表的每个条目中包含与一个标识符相关的信息，比如它的字符串、类型、存储位置等。
  - 为每个作用域设置一个符号表
    - *block* → ‘{’ *decls stmts* ‘}’
    - *stmts* → *stmts stmt*
    - *stmt* → *block*

第二章 一个简单的语法制导翻译器

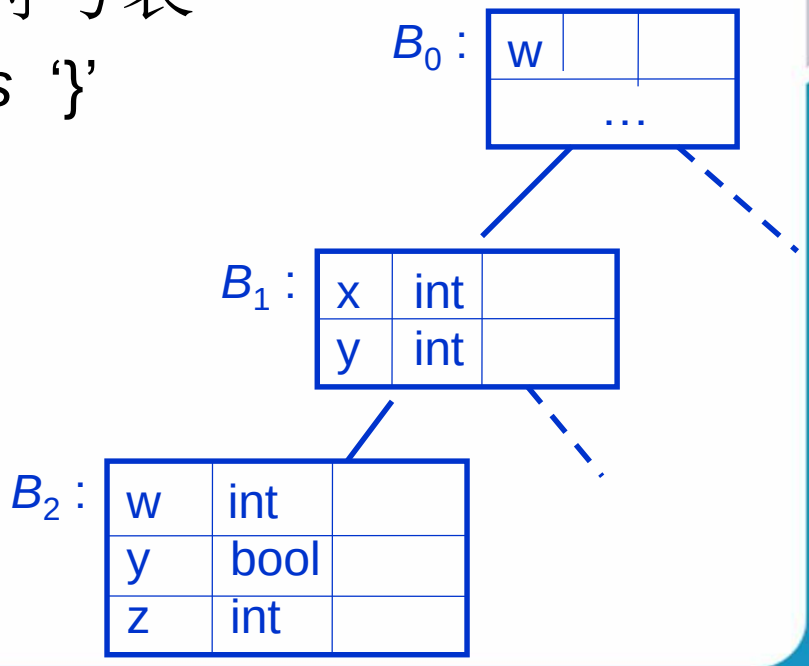
- 符号表
  - 符号表的每个条目中包含与一个标识符相关的信息，比如它的字符串、类型、存储位置等。
  - 为每个作用域设置一个符号表
    - *block* → ‘{’ *decls* *stmts* ‘}’

```
1)  {  int x1 ; int y1 ;
2)       { int w2 ; bool y2 ; int z2 ;
3)            …w2…; …x1…; …y2…; …z2…;
4)       }
5)       …w0…; …x1…; …y1…;
6)  }
```

第二章 一个简单的语法制导翻译器

- 符号表
  - 符号表的每个条目中包含与一个标识符相关的信息，比如它的字符串、类型、存储位置等。
  - 为每个作用域设置一个符号表
    - *block* → ‘{’ *decls* *stmts* ‘}’

```
1)  {  int x1 ; int y1 ;
2)     { int w2 ; bool y2 ; int z2 ;
3)           …w2…; …x1…; …y2…; …z2…;
4)     }
5)     …w0…; …x1…; …y1…;
6)  }
```



第二章 一个简单的语法制导翻译器

- 符号表
  - 符号表的创建
    - 类Env：图2-37


```
package symbols;
import java.util.*;
public class Env {
        private Hashtable table;
        protected Env prev;

        public Env (Env p) {
                table = new Hashtable(); prev = p;
        }

}
```

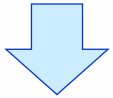
```
package symbols;
import java.util.*;
public class Env {
        private Hashtable table;
        protected Env prev;

        public Env (Env p) {
                table = new Hashtable(); prev = p;
        }

        public void put(String s, Symbol sym) {
                table.put(s, sym);
        }




}
```

```
package symbols;
import java.util.*;
public class Env {
        private Hashtable table;
        protected Env prev;

        public Env (Env p) {
                table = new Hashtable(); prev = p;
        }

        public void put(String s, Symbol sym) {
                table.put(s, sym);
        }

        public Symbol get(String s) {
                for(Env e = this; e != null; e = e.prev ) {
                        Symbol found = (Symbol)(e.table.get(s));
                        if( found != null ) return found;
                }
                return null;
        }
}
```

第二章 一个简单的语法制导翻译器

- 符号表
  - 符号表的创建
    - 类Env：图2-37
  - 符号表的使用

{ int x; char y; { bool y; x; y; } x; y; }



{ { x:int; y:bool } x:int; y:char; }

第二章 一个简单的语法制导翻译器

- 符号表
  - 符号表的使用：图2-38

第二章 一个简单的语法制导翻译器

program	→		{*top* = *null*;}
		block	
block	→	‘{’	{ *saved* = *top*;
			top = new *Env*(*top*);
			print(“{”); }
		decls stmts ‘;’	{ *top* = *saved*;
			print(“}”);}

第二章 一个简单的语法制导翻译器

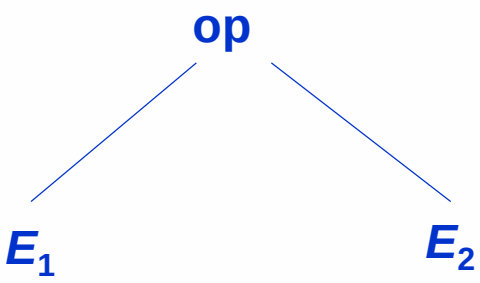
```
program  →                          {top = null;}
           block
  block  →  '{'                     { saved = top;
                                      top = new Env(top);
                                      print("{"); }
           decls stmts ';'          { top = saved;
                                      print("}");}
  decls  →  decls decl
         |  ε
  decl   →  type id ;               { s = new Symbol;
                                      s.type = type.lexeme;
                                      top.put(id.lexeme, s);}
  stmts  →  stmts  stmt
         |  ε
  stmt   →  block
         |  factor ;                { print(";"); }
```

第二章 一个简单的语法制导翻译器

```
program  →                          {top = null;}
           block
  block  →  '{'                     { saved = top;
                                      top = new Env(top);
                                      print("{"); }
           decls stmts ';'          { top = saved;
                                      print("}");}
  decls  →  decls decl
         |  ε
  decl   →  type id ;               { s = new Symbol;
                                      s.type = type.lexeme;
                                      top.put(id.lexeme, s);}
  stmts  →  stmts  stmt
         |  ε
  stmt   →  block
         |  factor ;                { print(";"); }
  factor →  id                      { s = top.get(id.lexeme);
                                      print(id.lexeme);
                                      print(":");
                                      print(s.type);}
```

第二章 一个简单的语法制导翻译器

- 生成中间代码
  - 两种中间表示形式
    - 抽象语法树



    - 三地址码

$$x = y \ \mathbf{op} \ z$$

第二章 一个简单的语法制导翻译器

- 生成中间代码
  - 两种中间表示形式
    - 抽象语法树的构造
      - 树节点：类*Node*
        - 子类：*expr*　代表各种表达式
        - 子类：*stmt*　代表各种语句
      - 图2-39：构造抽象语法树的翻译方案

第二章 一个简单的语法制导翻译器

- 生成中间代码
  - 语句的抽象语法树

program → *block*	{ return *block.n*; }
block → '{' *stmts* '}'	{ *block.n* = *stmts.n*; }
stmts → $stmts_1$ *stmt*	{ *stmts.n* = new *Seq*($stmts_1$.*n*, *stmt.n*);}
| ε	{ *stmts.n* = null; }
stmt → *expr* ;	{ *stmt.n* = new *Eval*(*expr.n*); }
| if(*expr*) $stmt_1$	{*stmt.n* = new *If*(*expr.n*, $stmt_1$.*n*);}
| while(*expr*) $stmt_1$	{*stmt.n* = new *While*(*expr.n*, $stmt_1$.*n*);}
| do $stmt_1$ while (*expr*)	{*stmt.n* = new *Do*($stmt_1$.*n*, *expr.n*);}
| block	{*stmt.n* = *block.n*;}

第二章 一个简单的语法制导翻译器

- 生成中间代码
  - 表达式的抽象语法树

$expr \rightarrow rel = expr_1$	{*expr.n* = new *Assign*('=', rel.n, expr.n);}
$\mid rel$	{*expr.n* = *rel.n*; }
$rel \rightarrow rel_1 < add$	{*rel.n* = new *Rel*('<', rel_1.*n*, *add.n*); }
$\mid rel_1 < add$	{*rel.n* = new *Rel*('≤', rel_1.*n*, *add.n*); }
$\mid add$	{*rel.n* = *add.n*; }
$add \rightarrow add_1 + term$	{*add.n* = new *Op*('+', add_1.*n*, *term.n*);}
$\mid term$	{*add.n* = *term.n*; }
$term \rightarrow term_1 * factor$	{*term.n* = new *Op*('*', $term_1$.*n*, *factor.n*); }
$\mid factor$	{*term.n* = *factor.n*; }
$factor \rightarrow (expr)$	{*factor.n* = *expr.n*; }
$\mid$ num	{*factor.n* = new *Num*(*num.value*); }

第二章 一个简单的语法制导翻译器

- 生成中间代码
  - 三地址码

$x = y \ \mathbf{op} \ z$

$x[\ y\ ] = z$

$x = y[\ z\ ]$

ifFlase x goto L

ifTrue x goto L

goto L

$x = y$

第二章 一个简单的语法制导翻译器

- 生成中间代码
  - 三地址码
    - 语句的翻译。例： if *expr* then $stmt_1$

```
class If extends Stmt {
    Expr E; Stmt S;
    public If(Expr x, Stmt y) {E = x; S = y; after = newlabel(); }
    public void gen() {
        Expr n = E.rvalue();
        emit("ifFalse " + n.toString() + "goto " + after);
        S.gen();
        emit(after + ":");
    }
}
```

第二章 一个简单的语法制导翻译器

- 生成中间代码
  - 三地址码
    - 表达式的翻译
      - i – j + k → t1 = i – j; t2 = t1 + k
      - 2 * a [i] → t1 = a [i]; t2 = 2 * t1
      - a [i]出现在复制表达式的左边时，需要使用左值函数
    - 图2-44：*lvalue*的伪代码
    - 图2-45：*rvalue*的伪代码

第二章 一个简单的语法制导翻译器

- 生成中间代码
  - 三地址码
    - 图2-44：*lvalue*的伪代码

```
Expr lvalue(x, Expr) {
    if ( x是一个Id结点 )  return x;
    else if( x 是一个Access(y,z)结点，且y是一个Id结点 ) {
        return new Acess(y, rvalue(z));
    }
}
```

  - 例：a [2 * k] → t = 2 * k; a [t];

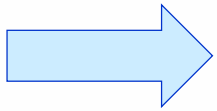
第二章 一个简单的语法制导翻译器

```
Expr rvalue(x : Expr) {
        if(x是一个Id或者Constant结点) return x;
        else if(x是一个Op(op, y, z)或者Rel(op, y, z) 结点) {
                t = 新的临时名字;
                生成对应于t = rvalue(y) op rvalue(z)的指令串;
                return 一个代表t的新结点;
        }
        else if(x是一个Access(y,z)结点) {
                t = 新的临时名字;
                调用lvalue(x)，它返回一个Access(y,z')的结点;
                生成对应于t= Access(y, z')的指令串;
                return 一个代表t的新结点;
        }
        else if(x是一个Assign(y,z)结点) {
                z'= rvalue(z);
                生成对应于lvalue(y) = z'的指令串;
                return z';
        }
}
```

第二章 一个简单的语法制导翻译器

- 生成中间代码
  - 三地址码
  - 例：

a[i] = 2*a[j-k]



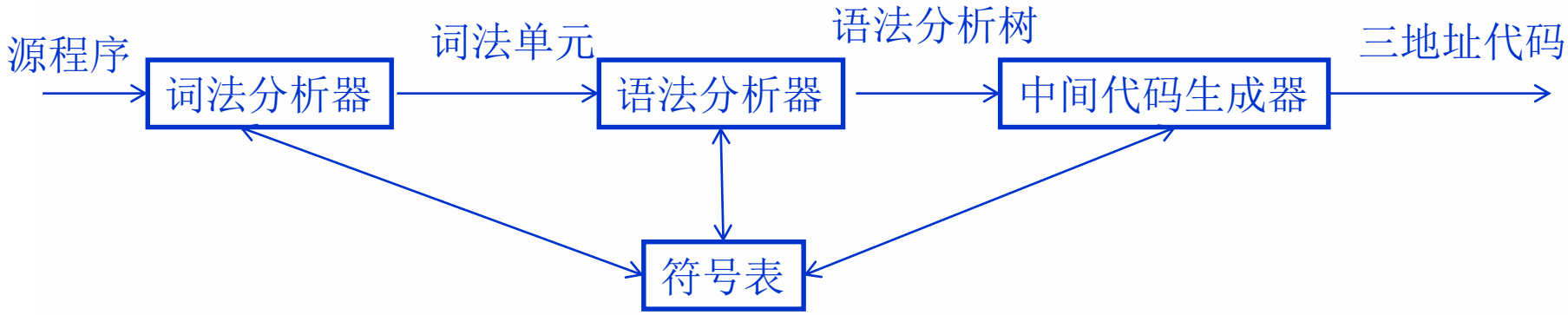
```
t3 = j – k
t2 = a [ t3 ]
t1 = 2 * t2
a [ i ] = t1
```

编译原理

第二章 一个简单的语法制导翻译器 总结

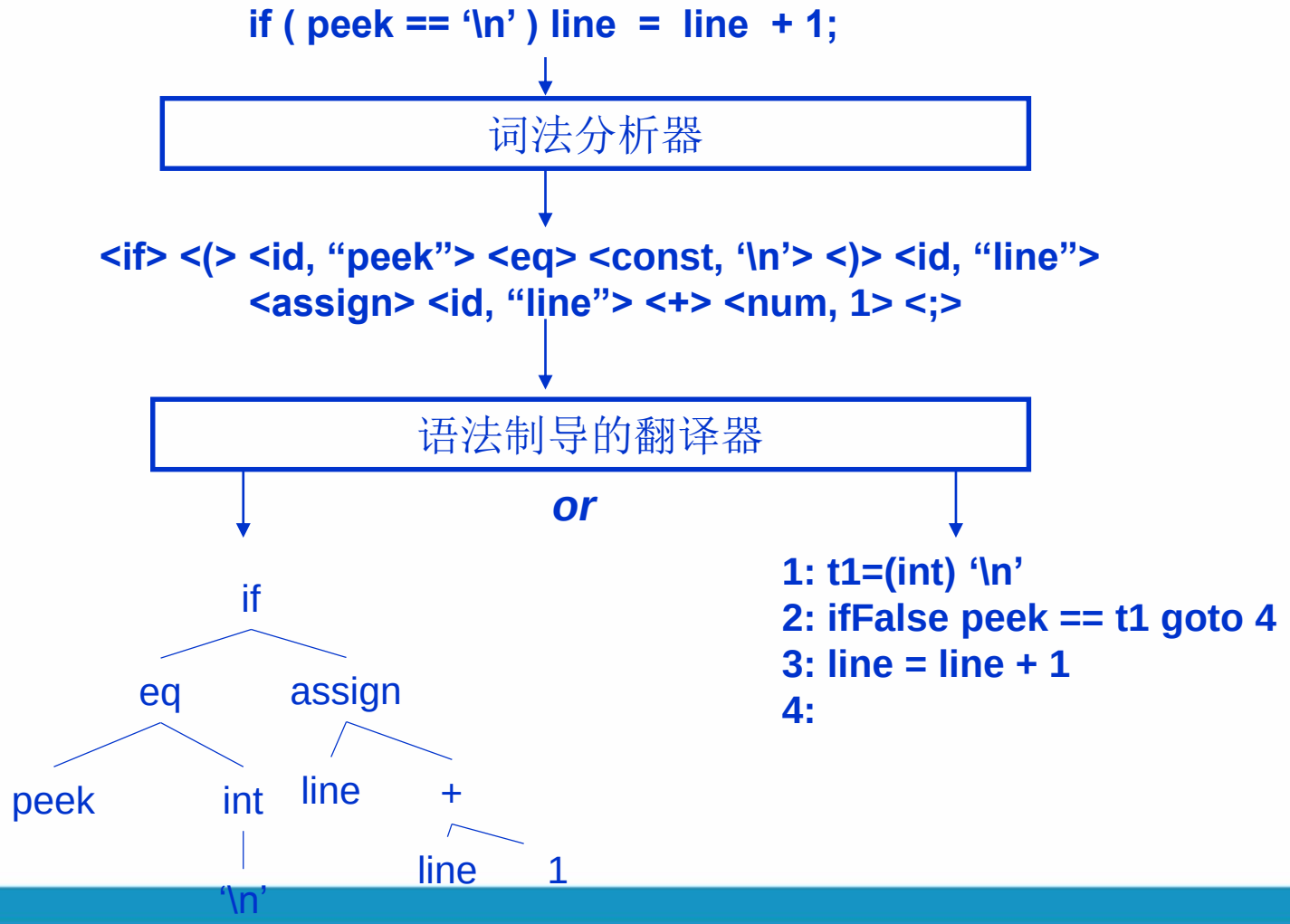
第二章 一个简单的语法制导翻译器

- 语法制导翻译器
  - 语法（syntax）：程序的正确形式
  - 语义（semantics）：程序的含义，即程序在运行时做什么事情



第二章 一个简单的语法制导翻译器

- 语法制导翻译器



第二章 一个简单的语法制导翻译器

- 语法制导翻译器
  - 词法：正则表达式（第三章）
    - 剔除空白和注释
    - 识别和计算常量
    - 识别关键字和标识符
  - 文法：上下文无关文法
    - 终结符集合
    - 非终结符集合
    - 产生式规则集合
    - 开始符∈非终结符集合

自顶向下分析法:

stmt → **expr**
| **if (expr)** *stmt*
| **other**

optexpr → **ε** | **expr**

```
void stmt() {
    switch(lookahead) {
    case expr:
        match(expr); match(';'); break;
    case if:
        match(if); match('('); match(expr); match(')');stmt();
        break;
case other:
        match(other); break;
    default:
        report("syntax error");
    }
}
void optexpr() {if(lookahead == expr) match(expr);}
void match(terminal t) {
      if(lookahead == t) lookahead = nextTerminal;
      else report("syntax error");
}
```

第二章 一个简单的语法制导翻译器

- 生成中间代码
  - 语句的抽象语法树

program → *block*	{ return *block.n*; }
block → '{' *stmts* '}'	{ *block.n* = *stmts.n*; }
stmts → $stmts_1$ *stmt*	{ *stmts.n* = new *Seq*($stmts_1$.*n*, *stmt.n*);}
| ε	{ *stmts.n* = null; }
stmt → *expr* ;	{ *stmt.n* = new *Eval*(*expr.n*); }
| if(*expr*) $stmt_1$	{*stmt.n* = new *If*(*expr.n*, $stmt_1$.*n*);}
| while(*expr*) $stmt_1$	{*stmt.n* = new *While*(*expr.n*, $stmt_1$.*n*);}
| do $stmt_1$ while (*expr*)	{*stmt.n* = new *Do*($stmt_1$.*n*, *expr.n*);}
| block	{*stmt.n* = *block.n*;}

第二章 一个简单的语法制导翻译器

- 生成中间代码
  - 表达式的抽象语法树

$expr \rightarrow rel = expr_1$	{*expr.n* = new *Assign*('=', rel.n, expr.n);}
$\mid rel$	{*expr.n* = *rel.n*; }
$rel \rightarrow rel_1 < add$	{*rel.n* = new *Rel*('<', rel_1.*n*, *add.n*); }
$\mid rel_1 < add$	{*rel.n* = new *Rel*('≤', rel_1.*n*, *add.n*); }
$\mid add$	{*rel.n* = *add.n*; }
$add \rightarrow add_1 + term$	{*add.n* = new *Op*('+', add_1.*n*, *term.n*);}
$\mid term$	{*add.n* = *term.n*; }
$term \rightarrow term_1 * factor$	{*term.n* = new *Op*('*', $term_1$.*n*, *factor.n*); }
$\mid factor$	{*term.n* = *factor.n*; }
$factor \rightarrow (expr)$	{*factor.n* = *expr.n*; }
$\mid$ num	{*factor.n* = new *Num*(*num.value*); }

第二章 一个简单的语法制导翻译器

- 生成中间代码
  - 三地址码
    - 语句的翻译。例： if *expr* then $stmt_1$

```
class If extends Stmt {
    Expr E; Stmt S;
    public If(Expr x, Stmt y) {E = x; S = y; after = newlabel(); }
    public void gen() {
        Expr n = E.rvalue();
        emit("ifFalse " + n.toString() + "goto " + after);
        S.gen();
        emit(after + ":");
    }
}
```

第二章 一个简单的语法制导翻译器

- 生成中间代码
  - 三地址码
    - 表达式的翻译
      - i – j + k → t1 = i – j; t2 = t1 + k
      - 2 * a [i] → t1 = a [i]; t2 = 2 * t1
      - a [i]出现在复制表达式的左边时，需要使用左值函数
    - *lvalue*函数实现左值
    - *rvalue*函数实现右值