2013-2014 第二学期

Sun Yat-Sen University
2011 级电子政务
JollinZhu (11331448)

JGer-Jolin

[编译原理]

2013-2014 学年度第二学期----中山大学软件学院----软件工程 2011 级电子政务方向----编译原理课程讲义

授课老师：娄定俊　　编辑：朱琳 Jolin Zhu

第一章 引论

*问题的提出:没有软件的计算机能执行的语言(指令)是非常低级的语言,即机器语言,例如:

0001 01 00 00100000

0011 01 10 00000010

而面向人的高级语言直观,书写方便,不易出错,例如:

IF i < j THEN i := j+1

我们的问题是: 如何使高级语言写的程序能在计算机上执行?

编译(compiling): 代表从面向人的源语言表示的算法到面向硬件的目标语言表示的算法的一个等价变换.

§1.1 翻译和解释

一. 解释器(Interpreter)

对于一种程序设计语言 PL, 可以定义一种抽象机, PL 的运算, 数据结构和控制结构是这种机器的存储元素和指令. 实现这种抽象机的软件称为解释器.

解释器对源程序的输入直接执行源程序中说明的操作.

(见书 P2)

二. 编译器(compiler)

编译器是将源语言程序通过翻译序列$(SL,L_1),(L_1,L_2),\cdots,(L_k,TL)$变成和源程序等价的目标语言程序的软件. 其中 SL 称为源语言, TL 称为目标语言, L_i (i=1, 2, …, k)称为中间语言.

*把高级语言程序翻译成低级目标(机器)语言程序的软件即是编译程序. (见书 P2)

能够完成一种语言到另一种语言变换的软件称为翻译器.

§1.2 编译的阶段(Phases)

一. 编译阶段的划分(见书 P5):

二. 各阶段的功能

1.词法分析(Lexical Analysis)

词法分析读源程序的字符序列,识别一个个单词(lexeme),并把它们组成记号(token)流. 记号流的每个记号代表逻辑上有内聚力的字符序列; 例如: 标识符,关键字,标点符号或算符. 形成记号的字符序列叫做该记号的单词.

例如: (见书 P7)

*注意: 记号是单词在编译程序内部的表示.

2. 语法分析(Syntax analysis)

语法分析把源程序的记号序列分组成语法短语,它识别记号序列的层次结构. 源程序的语法短语常用分析树来表示.

例如: (见上例)

3. 语义分析(Semantic analysis)

语义分析阶段检查源程序的语义错误,并为以后的代码生成阶段收集类型信息.它使用语法分析阶段确定的层次结构来标识表达式的算符和运算对象及语句的语义.

语义分析所作的检查包括: 运算对象类型的相容性,名字是否定义, 形参与实参的一致性等.

例如: (见上例)

4. 中间代码生成(Intermediate code generation)

将源语言的记号流翻译成某种中间语言形式,这种中间语言程序是某种抽象机的程序.

中间语言有两个性质:它容易产生和容易翻译成目标程序.

例如: (见上例)

5. 代码优化(Code optimization)

代码优化阶段试图改进中间代码,以便产生较快的机器代码.

代码优化分为: 1.独立于机器的代码优化; 和

2.依赖于机器的代码优化;

例如: (见上例)

6. 代码生成(Code generation)

代码生成阶段生成可以重定位的机器代码或汇编码.

*在此阶段,为源程序所用的每个变量选择存储单元和分配寄存器,把中间代码翻译成等价的机器指令序列.

例如: (见上例)

7. 符号表管理(Symbol-table management)

记录源程序中使用的标识符和收集每个标识符(identifier)的各种属性(attribute).

*这些属性提供标识符的存储分配,类型和作用域信息,如果是过程名标识符,还有参数个数和类型,参数传递方式和返回值类型.

符号表管理阶段与上述各阶段交互作用.

8. 错误诊断和报告(Error detection and report)

每个阶段都可能碰到错误.在发现错误后,该阶段必须处理此错误,使得以后的编译可以继续进行,以便进一步察觉源程序的其它错误.

§1.3 阶段的分组

一. 前端和后端

1. 前端(Front-end):

是只依赖于源语言,几乎独立于目标机器的阶段或阶段的一部分组成的.

通常是:词法分析,语法分析,符号表建立,语义分析和中间代码生成.

2. 后端(back-end):

是指编译器中依赖于目标机器的部分.

它们一般独立于源语言而与中间代码有关.通常包括:代码优化,代码生成和出错处理,符号表操作.

二. 遍(pass)

编译的几个阶段常用一遍来实现.一遍包括读一个输入文件和写一个输出文件. 即扫描一遍被编译的程序.

编译的分遍常常因实现的方便和需要而定.例如:把词法分析,语法分析,语义分析和中间代码生成组成一遍.

三. 减少遍数

在实现编译程序时,我们希望尽量减少遍数,因为每遍都要花时间来读写中间文件. 但有时由于某种需要必须分多遍扫描.

§1.4 编译程序的伙伴

一. 预处理器(preprocessor)

预处理器完成以下任务:

1. 将同一个程序的保存在不同文件中的几个模块安装在一起.
2. 扩展宏定义
3. 语言功能扩充

二. 汇编器(Assembler)

编译产生的目标代码可能是汇编码(assembly code),需要汇编器将其翻译成可重定位的(relocatable)机器代码.

三. 连接编辑器(Linker)

大的源程序常常分段编译, 需要连接编辑器将分段编译后产生的代码连接在一起, 成为可在机器内运行的程序.连接编辑器解决一个文件中的代码访问其它外部文件中存储地址的问题.

四. 装配器(Loader)

装配器把所有可执行的目标文件中的代码一起装入内存执行.

*阅读: 教材 1.1, 1.2 节

第二章 词法分析

§2.1 词法分析器(lexical analyzer)的作用

一、 作用

词法分析阶段的主要任务是读输入字符流,产生用于语法分析的记号序列.

*词法分析器作为语法分析器(parser)的一个子程序或联立程序来实现,当收到来自分析器的“取下一个记号”的命令时,词法分析器读输入字符,直到它能确认下一个记号为止.然后,词法分析器将该记号返回给语法分析器.

词法分析器的几个其它任务:

1. 删去注释和无用空白;
2. 把错误信息和源程序联系起来;
3. 作预处理.

二、 分离词法分析的理由

我们将词法分析阶段从语法分析阶段中分离出来的理由是:

1. 简化设计是重要的原因;
2. 编译器的效率会改进;
3. 编译的可移植性加强.

三、 记号、模式、单词

1. 记号(token)

单词在编译程序内部的表示. 很多字符串有同样的记号输出.

2. 模式(pattern)

具有同一记号输出的字符串集合由叫做模式的规则来描述.

*模式能匹配该集合的任一字符串.

3. 单词(lexeme)

是源程序中具有逻辑含义的极小字符串.

4. 例子 (见书 P112)

*我们把记号作为源语言文法的终结符.

5. 记号包括: 关键字、算符、标识符、常数、文字串和标点符号, 如括号、逗号和分号.

五. 记号的属性(attribute)

由于一个记号代表不只一个单词,必须提供附加信息,区别对应同一记号的不同单词. 另外,编译中后面的阶段需要用到这些单词的属性.

1. 记号的内部形式

(类码, 属性)

通常类码标识不同的记号, 属性为指向该标识符在符号表中条目的入口的指针.对常数来说,则是常数的值.

2. 例子: FORTRAN 中

E=M*C**2

输出: (见书 P113)

§2.2 输入缓冲区(Input buffering)

*由于词法分析器是编译器中逐个读源程序字符的唯一阶段,编译的相当可观的时间消耗在词法分析阶段.所以加快词法分析是编译器设计需要关心的问题.

一. 缓冲区对

*设两半缓冲区,每一半是 4096 字节,即磁盘的一块

*两个指针

*工作原理

*向前扫描的情况

例: FORTRAN 语言中

DO 5 I=1.25

DO 5 I=1,25

二. 标记

*缓冲区结束标记

*文件尾标记

*工作原理

§2.3 记号的说明

一. 串和语言

1. 字母表(alphabet): Σ 非空、有穷集
2. 串(string): 是字母表中符号的有穷序列. 也叫句子或字.
   例如: Σ={a, b},
   串: s=abb, 串长 |s| = 3
   空串: ε , |ε| = 0
3. 语言(language): 字母表上串的集合
   例如: Σ={a, b}, L= {a, b, aab, abb, ba}
   空语言: ϕ, 只含空串的语言: { ε }
4. 连接运算(concatenation):
   设 x = ab, y = bb, 则 xy = abbb
   ∀ x, ε x = x ε = x
   指数(exponentiation): $s^0 = \varepsilon$, $s^{i+1} = s^i s$, 其中 s 是任意串.

二. 语言的运算(Operations on languages)

设 L、M 为字母表Σ上的语言:

1. $L \cup M = \{x \mid x \in L \text{ 或 } x \in M\}$ (union)
   $LM = \{xy \mid x \in L \text{ 且 } y \in M\}$ (concatenation)
   指数: $L^0 = \{\varepsilon\}$, $L^{i+1} = L^i L$ (exponentiation)

   $L^* = \bigcup_{i=0}^{\infty} L^i$ ((Kleene) closure)

   $L^+ = \bigcup_{i=1}^{\infty} L^i$ (positive closure)

2. 例子:
   设 L={A, B, …, Z, a, b, …, z}, D={0, 1, 2, …, 9}
   例 1: L∪D
   例 2: LD
   例 3: L^4
   例 4: L^*
   例 5: $L(L \cup D)^*$
   例 6: D^+

*阅读: 教材 3.1, 3.2, 3.3.1, 3.3.2 节

作业 1:

1. 将下列程序的字符序列转换为记号序列, 并给每个记号以合理的属性值.

PASCAL

```
Function max(i, j:integer):integer;
/* return maximum of integers i and j */
begin
  if i > j then max := i
  else max := j
end;
```

三. 正规式(Regular expression)

正规式(正则式)代表一个语言.

1. 定义: 给定字母表Σ,定义Σ上的正规式如下:
   (1) ε 是正规式,表示{ ε };
   (2) 如果 a∈Σ,那么 a 是正规式, 表示{a};
   (3) 假定 r 和 s 都是正规式,分别表示语言 L(r)和 L(s), 那么
      a) (r)|(s)是正规式,表示 L(r)∪L(s);
      b) (r)(s)是正规式,表示 L(r)L(s);
      c) $(r)^*$是正规式,表示$(L(r))^*$;
      d) (r)是正规式, 表示 L(r).

   除此以外,其它的不是正规式. 正规式表示的语言叫正规集(regular set).

2. 约定
   *优先级最高, 左结合(left associative);
   连接(concatenation)优先级其次, 左结合;
   |优先级最低, 左结合;
   可以删除不必要的括号.
   例如: (a) | ((b)*(c)) 可写成 a | b*c

3. 例子: 令Σ={a, b}
   a | b 表示 {a, b}
   (a | b)(a | b) 表示 {aa, ab, ba, bb},
   也可用 aa | ab | ba | bb 表示.
   *同一语言的正规式表示不唯一.
   a^* 表示 { ε , a, aa, aaa, … }
   $(a | b)^*$ 表示 { ε , a, b, aa, ab, ba, bb, aaa, aab, …}
   $a | a^*b$ 表示 {a, b, ab, aab, aaab, … }

4. 正规式的运算规则
   *两个正规式相等,当且仅当它们代表的语言相同.
   设 r, s 和 t 是任意正规式.
   1) r | s = s | r
   2) r | (s | t) = (r | s) | t
   3) (rs)t = r(st)
   4) r(s | t) = rs | rt
      (s | t)r = sr | tr

5) $\varepsilon r = r, \quad r\varepsilon = r$

6) $r^* = (r \mid \varepsilon)^*$

7) $r^{**} = r^*$

证明举例:

证明(4): 要证 L(r(s|t))=L(rs|rt).

L(r(s|t))=L(r)(L(s)∪L(t))

={xy|x∈L(r)∧(y∈L(s)∨y∈L(t))}

={xy|(x∈L(r)∧y∈L(s))∨(x∈L(r)∧y∈L(t))}

={xy| x∈L(r)∧y∈L(s)}∪{xy| x∈L(r)∧y∈L(t)}

=L(r)L(s)∪L(r)L(t)

=L(rs|rt)

证明(6): 要证 $L(r^*)= L((r|\varepsilon)^*)$

∀ x, 设 $x\in L(r^*)= \bigcup_{i=0}^{\infty} L(r)^i$.故 $x\in L(r)^k$

由于 $L(r)\subseteq L(r|\varepsilon)=L(r)\cup L(\varepsilon)$

$L(r)^k \subseteq L(r|\varepsilon)^k$,故 $x\in L(r|\varepsilon)^k$,

从而 $x\in L((r|\varepsilon)^*)=\bigcup_{i=0}^{\infty} L(r\mid\varepsilon)^i$. 因而 $L(r^*)\subseteq L((r|\varepsilon)^*)$.

∀ x,设 $x\in L((r|\varepsilon)^*)= \bigcup_{i=0}^{\infty} L(r\mid\varepsilon)^i = \bigcup_{i=0}^{\infty} (L(r)\cup L(\varepsilon))^i$.

故 $x\in(L(r)\cup L(\varepsilon))^k$,从而

$x = \varepsilon^{\beta 1} w_1 \varepsilon^{\beta 2} w_2 \varepsilon^{\beta 3} \cdots w_t \varepsilon^{\beta(t+1)} = w_1 w_2 \cdots w_t$ ，其中

$\beta_1+\beta_2+\cdots+\beta_{(t+1)}+ t = k$, $w_j \in L(r)$, $j=1, 2, \cdots, t$

因而有 $x\in L(r)^t$,从而 $x\in L(r)^*=L(r^*)$.也就有

$L((r|\varepsilon)^*)\subseteq L(r^*)$. 因此 $L(r^*)=L((r|\varepsilon)^*)$.

四. 正规定义(Regular definition)

$d_1 \rightarrow r_1$

$d_2 \rightarrow r_2$

…

$d_n \rightarrow r_n$

各个 d_i 的名字不同且 $d_i \notin \Sigma$ (i=1, 2, …, n). 每个 r_i 是 $\Sigma \cup \{d_1, d_2, \cdots d_{i-1}\}$ 上的正规式.

例: Pascal 语言标识符(identifier)的定义

letter→A|B|…|Z|a|b|…|z

digit→0|1|…|9

id→letter(letter|digit)*

PASCAL 中无符号数

digit→0|1|…|9

digits→digit(digit)*

optional-fraction→.digits | ε

optional-exponent→(E(+|-|ε)digits)|ε

num→digits optional-fraction optional-exponent

五. 表示的缩写

1. 一个或多个实例(正闭包)

$r^+ = r\,r^*$, $r^* = r^+ \mid \varepsilon$

2. 零个或一个实例(instance)

$r? = r \mid \varepsilon$

3. 假设 $\Sigma = \{\cdots, a_1, a_2, \cdots, a_n, \cdots\}$, 那么 $a_1|a_2|\cdots|a_n$ 可表示为

$[a_1a_2\cdots a_n]$或$[a_1-a_n]$.

例如: [abc]表示 a|b|c, [a－z]表示 a|b|…|z

六. 非正规集

例: { wcw | w∈{a,b}*}.

*正规式表示语言的能力是有限的,有些语言不能由正规式表示.

§2.4 记号的识别(Recognition of tokens)

例:给定正规定义

if→if

then→then

else→else

relop→<|<=|=|<>|>|>=

id→letter(letter|digit)*

num→digit+(.digit+)?(E(+|-)?digit+)?

假定单词由空格(或制表符、换行符)分开

delim→blank|tab|newline

ws→delim+

* 词法分析器(lexical analyzer)将剥去空格. 如果输入与 ws(white space)匹配,词法分析器不返回记号给分析器,它继续去寻找空格后面的记号,然后返回到分析器(parser).

* 上述正规定义描述的是 Pascal 语言中的关键字、关系符、标识符和数字.C 语言的标识符定义如下:

letter_→A|B|…|Z|a|b|…|z|_

digit→0|1|…|9

id→letter_(letter_|digit)*

以下以 Pascal 语言中的定义为例.

* 以下表中给出对应每一种正规式,词法分析器返回的记号和属性.

(见书 P130)

一、 转换图(Transition diagram)

*我们首先产生转换图作为构造词法分析器的第一步.

1. 状态(state)
2. 转换边(edge)
3. 标记(label)
4. 接受状态(accepting states or final states)
5. 开始状态(start state or initial state)
6. 工作原理

例子: (见黑板)
例 1: 关系符的转换图 (见书 P131)

*关于指针的回退(retract)
例 2: 关键字和标识符的识别.(见书 P132)

*保留字(reserved words)的识别:

设置一张保存标识符信息的符号表(symbol table),把所有保留字预先存入其中, 符号表中的一个域说明当前标识符是保留字还是普通标识符.

当词法分析器识别一个标识符后,调用过程 install_id( ), 如果该标识符不在符号表中, 说明它不是保留字, 则 install_id( )在符号表中为它建立条目(entry), 并返回指向该条目的指针. 若该标识符已在符号表中, 但不是保留字, 则该过程返回该标识符已在符号表中条目的指针. 若该标识符已在符号表中, 并且是保留字, 则 install_id( )返回 0. 过程 gettoken 访问该标识符在符号表中的条目, 根据符号表中的信息, 如果该标识符是保留字,则返回它的记号, 如果是普通标识符, 则返回记号 id.

例 3: 无符号数的识别.(见书 P133)

*最长匹配原则
例 4: 空白的处理.(见书 P134)

二、 实现转换图

例子: 处理关系符的转换图转化为程序.

```
PROCEDURE GetRelop (VAR Token: Record name, att End);
VAR state, again : integer; c: char;
BEGIN
  Token.name := relop;
  state := 0;
  again := 1;
  WHILE again = 1 DO
      CASE state OF
      0: BEGIN
         c := nextchar( );
         IF c = '<' THEN state := 1
         ELSE IF c = '=' THEN state := 5
         ELSE IF c = '>' THEN state := 6
         ELSE BEGIN fail( ); again := 0 END
         END
      1: …
      …
      8: BEGIN
         retract( );
         Token.att := GT;
         again := 0
         END
  END;
```

§2.5 有穷自动机(Finite automata)

一、 语言的识别器(language recognizer)

语言的识别器是一个程序,它取串 x 作为输入,当 x 是该语言的句子时,回答“是”,否则回答“不是”.

*有穷自动机是更一般化的转换图,是转换图的数学模型.

*确定与不确定有穷自动机

*识别正规集.

二、 不确定有穷自动机 NFA(Nondeterministic Finite Automata)

1. 定义: 一个不确定有穷自动机 NFA M 是一个五元组

   $M=(S, \Sigma, \delta, s_0, F)$

   其中:

   S: 状态集(非空,有穷)

   Σ: 输入字母表(非空,有穷)

   s_0: 初始状态, $s_0 \in S$

   F: 终止状态集, $F \subseteq S$

   状态转换函数(transition function): $\delta : S \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^S$

   *解释 $\delta(s,a)=\{t_1,t_2,\cdots,t_k\}$.

2. 例子: 识别语言$(a|b)^*abb$ 的 NFA (见书 P148)

3. 用状态转换表表示(上例表示如下) (见书 P149)

4. 识别一个串

   NFA 接受一个输入串 x 当且仅当转换图中存在从开始状态到某个接受状态的路径,该路径上的标记拼成 x.

   例如: 上例中的 NFA 接受 aaabb

   注意:可能有多于一条路径接受同一个串

5. NFA 识别的语言

   一个 NFA 识别的语言是它接受的输入串的集合.

   例 1: 上例中 NFA 接受的语言为: $(a|b)^*abb$

   例 2: 接受 $aa^*|bb^*$的 NFA (见书 P150)

三、 确定有穷自动机 DFA(Deterministic Finite Automata)

1. 定义:一个确定有穷自动机 DFA M 是一个五元组

   $M=(S, \Sigma, \delta, s_0, F)$

   其中:

   S, Σ, s_0, F 同 NFA,

   状态转换函数: $\delta : S \times \Sigma \rightarrow S$

   *解释 $\delta(s, a) = t$

2. DFA 与 NFA 不同之处

   a) 任一状态都没有 ε 转换

   b) 对每一个状态 s 和输入符号 a, 最多只有一条标记为 a 的边离开 s .

3. 例子: 接受$(a|b)^*abb$ 的 DFA (见书 P151)

4. DFA 识别的串和语言

与 NFA 相同

*不同之处: 对于一个输入串 x, DFA 至多有一条从初始状态到终止状态的路径识别 x.

*阅读: 教材 3.3.3－3.3.5, 3.4, 3.6 节.

作业 2:

1.叙述由下列正规式描述的语言

(a) $0(0 \mid 1)^*0$

(b) $((\varepsilon \mid 0)1^*)^*$

2. 为下列语言写正规定义

(a) 所有包含 5 个元音的字母串,每个元音只出现一次且按序.

(b) 由偶数个 0 和奇数个 1 构成的所有 0 和 1 的串.

(c) 所有不含子串 011 的 0 和 1 的串.

3. 设 r 和 s 是正规式,试证:

(1) $(r \mid s)^* = (r^*s^*)^* = (r^* \mid s^*)^*$

(2) $r = b \mid ar$ 当且仅当 $r = a^*b$

四、 NFA 到 DFA 的变换(Conversion of an NFA to a DFA)

1. 等价性: NFA 与 DFA 识别语言的能力是一样的.DFA 是一个特殊的 NFA;对任一 NFA,可以构造一个与之等价的(识别同一语言的)DFA.

2. 基本思想: NFA 读入一个串后,进入一个状态子集.从 NFA 变换到 DFA 的一般思想是:让 DFA 的每个状态代表 NFA 的一个状态子集,这个 DFA 用它的状态去保持 NFA 在读输入符号后能到达的所有状态的踪迹.

*举例解释(见黑板)

3. 几个函数:

设 D 和 T 是 NFA 的状态子集.

(1) ε －闭包(D)={t|从状态 s∈D 出发,通过标记为 ε 的路径到达状态 t}(ε －closure(D))

(2) move(T,a)={t|从状态 s∈T 出发,经过 a 转换到达状态 t}

4. 算法(子集构造法)(The subset construction)

输入:一个 NFA N

输出:一个接受同样语言的 DFA D

设 s_0 是 NFA 的初始状态.D 的开始状态为 ε －闭包($\{s_0\}$).

初始, ε －闭包($\{s_0\}$)是 Dstates 中仅有的状态(state),并且尚未标记(unmarked).

```
while Dstates 中有尚未标记的状态 T do
    begin 标记 T(mark T);
        for 每个输入符号 a do
        begin U:= ε －闭包(move(T,a));
            if U 不在 Dstates 中 then
            把 U 作为尚未标记的状态加入 Dstates;
            δD[T,a]:= U
        end
    end
```

如果 D 的某个状态是至少含一个接受状态的 N 的状态子集, 那么它是 D 的一个接受状态.

5. 例子: 接受(a|b)*abb 的 NFA N 如下(见书 P155).
*将此 NFA 确定化.
*确定化后的转换表(见书 P155).
*确定化后的 DFA(见书 P156).

6. 直接模拟 NFA(Simulation of an NFA)
*由于一个 NFA 所对应的等价的 DFA 的状态数可能达到 NFA 状态数的指数数量级, 因而在某些应用中需要用程序直接模拟 NFA.
*对于词法分析器而言, 通常语言的词法分析器的 DFA 状态数与 NFA 状态数是同一数量级的.
算法(模拟 NFA N):
输入: 每个输入串 x 以 eof 结束. NFA N 的开始状态 s_0, 接受状态集 F, 转换函数 move.
输出: 如果 N 接受 x, 回答"Yes"; 否则, 回答"No".
1) S := ε－闭包({s_0});
2) c := nextChar( );
3) while c <> eof do
begin
4) S := ε－闭包(move(S,c));
5) c := nextChar( );
end
6) if (S∩F <> Φ) then return("Yes")
7) else return("No");
*该算法可以在 O(k(n+m))时间内实现, 其中 k 是输入串 x 的长度, n 是 NFA 的状态数, m 是 NFA 的转换边数.

§2.6 从正规式到 NFA

一、 从正规式构造 NFA

*正规式描述语言的能力与 NFA、DFA 识别语言的能力是一样的.
*从任一正规式 r 可以构造识别 L(r)的 NFA.

1. 算法: 从正规式构造 NFA(Thompson 构造)
输入: 字母表 Σ 上的正规式 r
输出: 接受 L(r)的 NFA N
根据正规式 r 的结构,归纳地构造
(1) 对 ε 构造 NFA(见书 P159)
*正确性证明
(2) 对任意 a∈Σ, 构造 NFA(见书 P159)
(3) 如果 N(s)和 N(t)是正规式 s 和 t 的 NFA,
a) 对 s|t, 构造 N(s|t) (见书 P160)
*正确性证明
b) 对 st, 构造 NFA(st) (见书 P160)

c) 对正规式 s^*, 构造 NFA(s^*) (见书 P161)

d) 对正规式(s), N(s)即是它的 NFA

2. 性质:

(1) N(r)的状态数最多是 r 中符号(operands)和算符(operators)数的两倍;

(2) N(r)只有一个开始状态和一个接受状态,接受状态没有向外的转换边, 开始状态没有进入的转换边.

(3) N(r)的每个状态,除接受状态外,或者有一个Σ中符号标识的向外转换边,或者最多有两个向外的ε转换边.

3. 例子: 正规式 r = (a | b)*abb 的 NFA

*r 的分析树(parse tree)(见书 P162)

*由 r 构造的 NFA(见书 P155), 构造过程见书 P162－163.

§2.7 DFA 的化简

一、 定理:

每个正规集都可由一个状态数最小的 DFA 识别,并且这个 DFA 唯一.

假设当前的 DFA M 的每一个状态对每个输入符号都有转换(transition). 必要时,引入死状态(dead state).

二、 极小化的原理

1. 我们说串 w 区别(distinguish)状态 s 和 t,如果 DFA M 从状态 s 出发,输入 w, 它停在某个接受状态,但从 t 出发,同样的输入, 它停在非接受状态.

例子: (见书 P156)

*状态 A 和 B 由 bb 区别;

*但状态 A 和 C 不能由任何串区别.

2. 算法思想:

极小化 DFA 状态数的算法就是把 DFA 的状态分成一些不相交的子集,每一子集中的状态都是不可区别的,不同子集的状态都是可区别的.每个子集合并成一个状态.

三、 算法: 极小化 DFA 的状态数

输入:一个 DFA M, 它的状态集合是 S,输入符号集Σ,转换函数 δ : S×Σ→S, 开始状态 s_0, 接受状态集 F .

输出: 一个 DFA M', 它和 M 接受同样的语言, 且状态数最少.

算法:

1. 构造状态集合的初始划分(initial partition)Π, 分成两组,接受状态组 F 和非接受状态组 S－F.

2. 应用下面的过程对Π构造新的划分 Π_{new}

```
for Π中的每个组(group)G do
begin
    把 G 划分成小组,G 的两个状态 s 和 t 在同一个小组中,当且仅当对所有的输入符号 a, s 和 t 的 a 转换是到Π的同一组中;
    在 Π_new 中,用 G 的划分代替 G;
end;
```

3. 如果$\Pi_{new}=\Pi$,令$\Pi_{final}:=\Pi$，再执行步骤(4),
否则令$\Pi:=\Pi_{new}$,转(2)

4. 在Π_{final}中的每个状态组中选一个状态代表它,这些代表(representative)是简化的 DFA M'的状态.如果 s 是这样一个代表,在 DFA M 中,s 的 a 转换到 t,并且 t 所在的组的代表是 r,那么在 M'中,s 的 a 转换到 r.包含 s_0 的状态组的代表是 M'的开始状态,M'的接受状态是那些在 F 中的代表.

5. 如果 M'有死状态,则去掉它.从开始状态不可及的状态也删除.从任何其它状态到死状态的转换都成为无定义(undefined).

四、 例子:上例中 DFA 最小化. (见书 P183)

*阅读: 教材 3.7.1, 3.7.2, 3.7.4, 3.9.6 节

作业 3:

1.给定以下 NFA N(见黑板),给出 N 处理输入串 ababbab 的移动序列, 并将该 NFA 化为等价的 DFA.

2. 构造下列正规式的 NFA, DFA 和最小 DFA
$(a \mid b)^*a(a \mid b)$.

3. 假设 L 和 M 都是正规集. 证明: L∪M, L∩M 和 $\overline{M}$ (补集)也是正规集.(提示: 根据 L 和 M 的 DFA,构造 L∩M 和 $\overline{M}$ 的 DFA).

第三章 语法分析(Syntax analysis)

*本章我们介绍用上下文无关文法描述程序的语法结构,并介绍自上而下和自下而上的语法分析技术.

§3.1 分析器(parser)的作用

一. 作用

1. 主要任务: 语法分析器读取词法分析器提供的记号串,检查它是否能由源语言的文法(grammar)产生.我们希望该分析器能以易于理解的形式报告任何语法错误,并从错误中恢复,使后面的分析能继续下去

2. 分析器完成的其它任务
把各种记号的信息收入符号表, 完成类型检查和其它的语义检查,产生中间代码等.

3. 语法分析的途径
   1) 通用方法(因代价太大,不具有实用性,本书不介绍)
   2) 自上而下(top-down)的方法
   3) 自下而上(bottom-up)的方法

二. 语法错误的处理

1. 程序中的错误

*词法(lexical)错误: 如标识符、关键字或算符的拼写错
*语法(syntactic)错误: 如算术表达式括号不配对等
*语义(semantic)错误: 如算符作用于不相容的运算对象
*逻辑(logical)错误: 如无穷的递归调用,死循环等.

2. 分析器出错处理的基本目标

*清楚而准确地报告错误的出现
*迅速地从每个错误中恢复过来,以便诊断后面的错误
*它不应该使正确程序的处理速度降低太多.

三. 错误恢复策略(error-recovery strategy)

1. 紧急方式(panic-mode)恢复: 发现错误时,分析器每次抛弃若干个输入记号,直到输入记号属于某个指定的同步记号集合为止.同步记号一般是一个定界符,如分号、end, 它们在源程序中的作用是清楚的.
这是最简单的方法, 适用于大多数分析方法.
例: (见黑板)

2. 短语级(phrase-level)恢复: 发现错误时,分析器对剩余输入作局部纠正,它用可以使分析器继续工作的记号串来代替剩余输入的前缀.
其主要缺点是很难应付实际错误出现在诊断点以前的情况.
例: (见黑板)

3. 出错产生式(error productions): 如果对常遇到的错误了解得很清楚, 我们可以扩充语言的文法, 增加产生错误结构的产生式,用此扩充的文法来构造分析器.如果分析器用了出错产生式,可以产生适当的错误诊断信息,以表示输入中的这个错误结构已被识别.
例: (见黑板)

4. 全局纠正(global correction): 我们希望一个理想的编译器,它在处理不正确的输入串时,作尽可能少的修改.选择最小的修改序列,以获得最小代价的全局纠正.
*这种技术代价太大,只有理论上的兴趣.最小的修改序列,不一定是程序员想要的.

§3.2 上下文无关文法(context-free grammar)

例: 一个条件语句可描述成

if E then S_1 else S_2

用文法表示

Stmt → if expr then Stmt else Stmt

一. 上下文无关文法

1. 定义: 一个上下文无关文法 G 是一个四元式

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

其中:

(1) V_T: 终结符(terminal)集, 非空、有限
*记号与终结符是同一东西.

(2) V_N: 非终结符(nonterminal)集, 非空、有限
$V_N \cap V_T = \phi$

(3) S: 开始符号(start symbol), $S \in V_N$

(4) P: 产生式(production)的有限集合. 每个产生式形如 $A \rightarrow \alpha$,其中 $A \in V_N$, $\alpha \in (V_N \cup V_T)^*$. 开始符号至少在某个产生式的左部出现一次.

2. 例子:

文法 G= ({id, +, －, *, /, ↑, (,)}, {expr, op}, expr, P)

P: expr→expr op expr

expr→(expr)

expr→－expr

expr→id

op→+

op→－

op→*

op→/

op→↑

二. 符号使用的约定

1. 下列符号是终结符

a) 字母表前面的小写字母, 如 a, b, c

b) 运算符号, 如+, －等

c) 标点符号,如括号,逗号等

d) 数字 0, 1, …, 9

e) 黑体串, 如 id, if

2. 下列符号是非终结符

a) 字母表前面的大写字母, 如 A, B, C

b) 字母 S, 常代表开始符号

c) 小写斜体的名字, 如 *expr* 或 *stmt*

3. 字母表后面的大写字母, 如 X, Y, Z, 代表文法符号(grammar symbol),也就是非终结符或终结符.

4. 字母表后面的小写字母, 主要是 u, v, …, z, 代表终结符串.

5. 小写希腊字母,例如: α, β, γ , 代表文法符号串,即 $\alpha, \beta, \gamma \in (V_T \cup V_N)^*$

6. 如果 $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \cdots, A \rightarrow \alpha_k$ 是所有以 A 为左部的产生式, 可以写成 $A \rightarrow \alpha_1 | \alpha_2 | \cdots | \alpha_k$.

7. 用产生式的集合来描述文法, 一般不再用四元式, 第一个产生式左部的符号是开始符号.

例: 表达式的文法

E→EAE | (E) | －E | id

A→+ | － | * | / | ↑

三. 推导

1. 推导(derivation)

如果 A→γ 是产生式， α 和 β 是文法符号的任意串， 则
αAβ $\Rightarrow$ αγβ 称为一步推导.
例: 给定文法
E→E+E | E*E | (E) | −E | id
E$\Rightarrow$E+E$\Rightarrow$E+(E)$\Rightarrow$E+(E*E)$\Rightarrow$id+(E*E)$\Rightarrow$id+(id*E)
$\Rightarrow$id+(id*id)

2. 多步推导

$\alpha \overset{*}{\Rightarrow} \beta$ 零步或多步推导

例如 E$\overset{*}{\Rightarrow}$id+(id*E)

$\alpha \overset{+}{\Rightarrow} \beta$ 一步或多步推导.

例如: E$\overset{+}{\Rightarrow}$id+(id*E)

3. 句型(sentential form)、句子(sentence)

如果 S$\overset{*}{\Rightarrow}$ α , α 是非终结符和终结符的串,则把 α 叫作 G 的句型.

若 α 是终结符串,则 α 称为 G 的句子.

例: E$\overset{*}{\Rightarrow}$id+(E*E), id+(E*E)是上述文法的句型.

E$\overset{*}{\Rightarrow}$id+(id*id), id+(id*id)是上述文法的句子.

4. 文法推导的语言

L(G) = { w | S$\overset{+}{\Rightarrow}$w 且 w∈V_T^* }

两个文法 G 和 G’ 称为等价的当且仅当 L(G)=L(G’).

5. 最左(leftmost)推导和最右(rightmost)推导

最左推导: wAγ $\underset{lm}{\Rightarrow}$ wδγ， 其中 w∈V_T^*, 且 A→δ 是产生式.

例如: E$\underset{lm}{\Rightarrow}$−E$\underset{lm}{\Rightarrow}$−(E)$\underset{lm}{\Rightarrow}$−(E+E)$\underset{lm}{\Rightarrow}$−(id+E)$\underset{lm}{\Rightarrow}$−(id+id)

若 S$\underset{lm}{\overset{*}{\Rightarrow}}$ α , 则 α 称为左句型(left-sentential form).

类似可以定义最右推导,最右推导也叫规范推导(canonical derivation).
最右推导的例子:

E$\underset{rm}{\Rightarrow}$−E$\underset{rm}{\Rightarrow}$−(E)$\underset{rm}{\Rightarrow}$−(E+E)$\underset{rm}{\Rightarrow}$−(E+id)$\underset{rm}{\Rightarrow}$−(id+id)

四. 分析树(parse tree)和推导

1. 分析树举例 (见书 P203)

2. 分析树与推导的关系

每个分析树对应一个最左(最右)推导. 它忽略了推导过程.

五. 二义性(ambiguity)

*如果一个文法的某个句子有两棵不同的分析树,则该文法是二义的(ambiguous).

例: 以上例中的文法为例 (见书 P204).

句子 id+id*id 有两棵不同的分析树.

*一个文法是二义的当且仅当它的一个句子有两个不同的最左(最右)推导.

例:

$E \underset{lm}{\Rightarrow} E+E \underset{lm}{\Rightarrow} id+E \underset{lm}{\Rightarrow} id+E*E \underset{lm}{\Rightarrow} id+id*E \underset{lm}{\Rightarrow} id+id*id$

$E \underset{lm}{\Rightarrow} E*E \underset{lm}{\Rightarrow} E+E*E \underset{lm}{\Rightarrow} id+E*E \underset{lm}{\Rightarrow} id+id*E \underset{lm}{\Rightarrow} id+id*id$

§3.3 语言和文法

一. 正规式与上下文无关文法的比较

1. 正规语言(集)与上下文无关语言

正规语言(集)一定是上下文无关语言,反之则不一定成立.

2. 从 NFA 构造上下文无关文法

首先确定终结符集 $V_T=\Sigma$,再为 NFA 的每个状态 i 引入非终结符 A_i,其中 A_0 是开始符号,因为 0 是开始状态.

如果状态 i 有一个 a 转换到状态 j,则引入产生式 $A_i \rightarrow aA_j$, 如果是 ε 转换,则引入 $A_i \rightarrow A_j$.如果 i 是接受状态,引入 $A_i \rightarrow \varepsilon$.

例子: 正规式$(a|b)^*abb$ 和以下文法等价.

NFA N: (见书 P148)

文法 G: $A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$

$A_1 \rightarrow bA_2$

$A_2 \rightarrow bA_3$

$A_3 \rightarrow \varepsilon$

*文法 G 由 NFA N 构造

*文法 G 对一个串的推导模拟了 NFA 识别该串的过程.

例如: ababb ∈ L(G).

$A_0 \Rightarrow aA_0 \Rightarrow abA_0 \Rightarrow abaA_1 \Rightarrow ababA_2 \Rightarrow ababbA_3 \Rightarrow ababb$

*文法 G 称为正规文法(与正规文法的定义稍有差别).

3. 用正规式描述词法(lexical syntax)的理由

(1) 语言的词法规则非常简单,不必用功能更强的上下文无关文法描述.

(2) 对于记号,正规式比上下文无关文法提供了更简洁和易于理解的符号表示.

(3) 从正规式可以自动地构造出比其它文法更有效的词法分析器.

(4) 把语言的语法结构分成词法和非词法两部分,为编译器前端的模块划分提供了方便的途径.

二. 适当地表示文法

文法: E→E+E|E－E|E*E|E/E|(E)|id 有二义性

结合算符的优先级和结合律,构造文法

E→E+T|E－T|T

T→T*F|T/F|F

F→(E)|id

可以消除二义性.

例: id+id+id, id+id*id ∈ L(G). 最左推导:

$$E \underset{lm}{\Rightarrow} E+T \underset{lm}{\Rightarrow} E+T+T \underset{lm}{\Rightarrow} T+T+T \underset{lm}{\Rightarrow} F+T+T \underset{lm}{\Rightarrow} id+T+T \underset{lm}{\Rightarrow} id+F+T$$

$$\underset{lm}{\Rightarrow} id+id+T \underset{lm}{\Rightarrow} id+id+F \underset{lm}{\Rightarrow} id+id+id$$

$$E \underset{lm}{\Rightarrow} E+T \underset{lm}{\Rightarrow} T+T \underset{lm}{\Rightarrow} F+T \underset{lm}{\Rightarrow} id+T \underset{lm}{\Rightarrow} id+T*F \underset{lm}{\Rightarrow} id+F*F \underset{lm}{\Rightarrow} id+id*F$$

$$\underset{lm}{\Rightarrow} id+id*id$$

*检查分析树,说明原因.(见黑板)

三. 消除二义性(Eliminating ambiguity)

语法: *stmt*→if *expr* then *stmt*

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

| other

有二义性.

例: if E_1 then if E_2 then S_1 else S_2

有两棵不同的分析树.(见书 P211)

*消除二义性实行最近匹配原则.

改写文法:

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*

四. 消除左递归(left recursion)

1. 左递归: 文法是左递归的,如果它有非终结符 A,对某个串 α,存在着推导 $A \overset{+}{\Rightarrow} A\alpha$.

*自上而下的分析方法不能处理左递归文法, 因此需要消除左递归.

2. 消除一步左递归(immediate left recursion)

A→Aα｜β

改为

A→βA’

A’ →αA’ ｜ε

*解释原理

3. 例子

E→E+T | T

T→T*F | F

F→(E) | id

改写成

E→TE’

E’→+TE’ | ε

T→FT’

T’→*FT’| ε

F→(E) | id

4. 一般情形

$A\rightarrow A\alpha_1|A\alpha_2|\cdots|A\alpha_m|\beta_1|\beta_2|\cdots|\beta_n$

改为

$A\rightarrow \beta_1 A'|\beta_2 A'|\cdots|\beta_n A'$

$A'\rightarrow \alpha_1 A'|\alpha_2 A'|\cdots|\alpha_m A'|\varepsilon$

5. 多步左递归

例:

S→Aa | b

A→Ac | Sd | e

消除左递归的算法:

输入: 无环, 无 ε 产生式的文法 G

输出: 无左递归的等价文法

方法:

a) 以任意次序排列非终结符 $A_1,A_2,\cdots,A_n$

b) for i := 1 to n do

begin

for j := 1 to i－1 do

用产生式 $A_i \rightarrow \delta_1\gamma|\delta_2\gamma|\cdots|\delta_k\gamma$ 代替

每个形式为 $A_i\rightarrow A_j\gamma$ 的产生式, 其中

$A_j\rightarrow \delta_1|\delta_2|\cdots|\delta_k$ 是所有当前 A_j 的产生式;

消除 A_i 产生式中的直接左递归;

end;

6. 例子:

S→Aa | b

A→Ac | Sd | e

化为

S→Aa | b
A→Ac | Aad | bd | e
再消除直接左递归, 得
S→Aa | b
A→bdA’| eA’
A’→cA’| adA’| ε

五. 提左因子(left factoring)

*在自上而下分析中, 如果两个产生式有左因子, 则无法选择用哪个产生式进行分析, 因此需要提左因子.

1. 例子: 设有文法: A→ α β1 | α β2
化为
A→ α A’
A’→ β1 | β2

2. 一般情形
设有文法
A→ α β1 | α β2| ... | α βn | γ
改写为
A→ α A’| γ
A’→ β1 | β2 |…| βn

3. 例子:
S→iEtS | iEtSeS | a , E→b
改写为
S→iEtSS’| a
S’→eS | ε
E→b

作业 4:

1.考虑文法
S→(L) | a
L→L,S | S
为句子(a,((a,a),(a,a)))构造最左推导和最右推导.

2.考虑文法
S→aSbS | bSaS | ε
(a)为句子 abab 构造两个不同的最左推导,以此说明该文法是二义的;(b) 为 abab 构造两个不同的最右推导;(c)为 abab 构造两个不同的分析树;（d）这个文法产生的语言是什么？

3.考虑文法
R→R‘|’R | R●R | R* | (R) | a | b
(注意:单引号中间的竖线是终结符)
a)证明该文法是二义的
b)构造一个等价的无二义文法.其中算符的优先级从高到低依次为: *, ● ,|, 且均满足左结合.

4. 消除以下文法的左递归
S→Sa | Ab | c
A→Ad | Se |

六. 非上下文无关语言结构

例 1: $L_1=\{wcw \mid w\in L((a \mid b)^*)\}$ 非上下文无关语言

例 2: $L_2=\{a^nb^mc^nd^m \mid n\geqslant 0$ 和 $m\geqslant 0\}$ 非上下文无关语言

例 3: $L_3=\{a^nb^nc^n \mid n\geqslant 0\}$ 非上下文无关语言

*解释以上语言的实际含义

以下是上下文无关语言

$L_1'=\{wcw^R \mid w\in L((a \mid b)^*)$, w^R 表示逆序的 w}

文法: S→aSa | bSb | c

$L_2'=\{a^nb^mc^md^n \mid n\geqslant 1$ 和 $m\geqslant 1\}$

文法: S→aSd | aAd

A→bAc | bc

$L_3'=\{a^nb^n \mid n\geqslant 1\}$

文法: S→aSb | ab

七. 形式语言鸟瞰

1. 0 型文法:

设 $G=(V_T,V_N, S, P)$是文法,并且它的每个产生式形如:

$\alpha \rightarrow \beta$,其中 $\alpha \in (V_N\cup V_T)^*$且至少含有一个非终结符,而

$\beta \in (V_N\cup V_T)^*$.则 G 称为 0 型文法,也叫短语文法.

2. 1、2、3 型文法:

如果 0 型文法加上以下第 i 条限制,就可以得到 i 型文法:

1) G 的任何产生式 $\alpha \rightarrow \beta$ 都满足$|\alpha| \leqslant |\beta|$, 仅仅 S→ ε 例外.如果有 S→ ε 产生式,则 S 不得出现在任何产生式右部.

2) G 的任何产生式形如: $A\rightarrow \beta$, $A\in V_N$, $\beta \in (V_N\cup V_T)^*$.

3) G 的任何产生式形如: $A\rightarrow aB$ 或 $A\rightarrow a$, $A,B\in V_N, a\in V_T$.

1 型文法叫上下文有关文法,2 型文法叫上下文无关文法,3 型文法叫正规文法.

例: $\{a^nb^nc^n \mid n\geqslant 1\}$的上下文有关文法.

S→aSBC bB→bb

S→aBC bC→bc

CB→BC cC→cc

aB→ab

*推导 aaabbbccc (见黑板)

§3.4 自上而下分析(Top-Down Parsing)

一. 自上而下分析的一般方法

1. 思想:

自上而下分析对任何输入串, 试图用一切可能的办法, 从文法的开始符号(根结点)出发, 自上而下, 从左到右地为输入串建立分析树. 或者说, 为输入串寻找最左推导.

2. 例子: 设有文法 S→cAd

A→ab | a

试图推出 cad (见书 P220).

3. 自上而下分析的困难和缺点
   1) 文法不能有左递归(left recursion)
   2) 需要回溯(backtracking)
   3) 回溯带来浪费,代价高

二. 预测分析器(predictive parser)

1. 无回溯的分析

要保证每次选择产生式去匹配,该产生式是唯一可选取的产生式.

2. 无回溯分析的条件

$FIRST(\alpha)=\{a \mid \alpha \overset{*}{\Rightarrow} a\ldots, a\in V_T\}$.

若 $\alpha \overset{*}{\Rightarrow} \varepsilon$,则 $\varepsilon \in FIRST(\alpha)$.

如果对于非终结符 A 的任意两个产生式 α_i 和 α_j，有

$FIRST(\alpha_i) \cap FIRST(\alpha_j)=\phi$,那么当要求用 A 匹配输入串时,A 就能根据它所面临的第一个输入符号 a,准确地指派一个产生式去匹配输入串.

3. 递归下降分析程序(Recursive-descent parser)

文法:

```
type→simple
       |↑id
       |array[simple] of type
simple→integer
      |char
      |num dotdot num
```

```
procedure match (t: token);
begin
    if lookahead = t then
    lookahead := nexttoken
    else error
end;

procedure  type;
begin
    if lookahead is in {integer,char,num} then
        simple
    else if lookahead = '↑' then
        begin
            match('↑'); match(id)
        end
    else if lookahead = array then
        begin
```

```
            match(array); match( '[' );
            simple; match( ')' );
            match(of); type;
        end
    else error
end;

procedure simple;
begin
    if lookahead = integer then
        match(integer)
    else if lookahead = char then
        match(char)
    else if lookahead = num then
        begin
            match(num);
            match(dotdot); match(num);
        end
    else error
end;
```

三. 非递归的预测分析器

(Nonrecursive Predictive Parser)

1. 组成: (见书 P227)

2. 工作原理:

   程序根据当前栈顶的符号 X 和输入符号 a 决定分析器的动作. 有三种可能:

   1) 如果 X=a=$,分析器宣告分析完全成功而停机;
   2) 如果 X=a≠$,分析器托出栈顶符号 X,推进输入指针,指向下一个符号.
   3) 如果 X 是非终结符,程序访问分析表 M.若 M[X,a]是 X 产生式,例如:M[X,a]={X→$Y_1Y_2\cdots Y_k$}.那么分析器用

   $Y_kY_{k-1}\cdots Y_1$ 代替栈顶 X,令 Y_1 在栈顶.如果 M[X,a]指示出错,分析器调用错误恢复例程.

3. 例子: 分析表(parsing table)(见书 P225)
   分析过程(见书 P228)

四. 开始符号和后继符号

*预测分析程序关键是构造分析表. 构造分析表需要两个集合.

1. FIRST 集和 FOLLOW 集

   $FIRST(\alpha)=\{a \mid \alpha \overset{*}{\Rightarrow} a\cdots, a\in V_T\}$;

   若 $\alpha \overset{*}{\Rightarrow} \varepsilon$,则 $\varepsilon \in FIRST(\alpha)$

FOLLOW(A)={a| $S \overset{*}{\Rightarrow} \cdots Aa \cdots, a \in V_T$}

如果 A 可以是某个句型的最右符号,那么$∈FOLLOW(A).

2. FIRST 集的构造:

计算所有文法符号 X 的 FIRST(X),直到每个 FIRST 集合不再增大为止.

1) 若 $X \in V_T$,那么 FIRST(X)={X}.
2) 若 X→ε 是产生式,那么把 ε 加入 FIRST(X).
3) 如果 X 是非终结符,且 $X \to Y_1Y_2 \cdots Y_k$ 是产生式,若对某个 i, a 属于 $FIRST(Y_i)$,并且 ε 属于所有的 $FIRST(Y_1),FIRST(Y_2),\cdots,FIRST(Y_{i-1})$,即 $Y_1Y_2 \cdots Y_{i-1} \overset{*}{\Rightarrow} \varepsilon$,则把 a 加入 FIRST(X).如果对所有的 j=1,2,…,k, 有 ε 属于 $FIRST(Y_j)$,那么把 ε 加入 FIRST(X).

   计算串 $X_1X_2 \cdots X_n$ 的 FIRST 集.若对某个 i, $a \in FIRST(X_i)$, 并且 ε 属于所有的 $FIRST(X_1),FIRST(X_2),\cdots,FIRST(X_{i-1})$,即 $X_1X_2 \cdots X_{i-1} \overset{*}{\Rightarrow} \varepsilon$,则把 a 加入 $FIRST(X_1X_2 \cdots X_n)$.如果对所有的 j=1,2,…,n,有 $\varepsilon \in FIRST(X_j)$,那么把 ε 加入 $FIRST(X_1 \cdots X_n)$.

3. FOLLOW 集的构造

对所有的非终结符 A 计算 FOLLOW(A).应用下面规则,直到每个 FOLLOW 集不再增大为止.

1) 把$加入 FOLLOW(S),其中 S 是开始符号,$是输入结束标记.
2) 如果有产生式 A→αBβ,那么除 ε 外,把 FIRST(β)中的所有元素加入 FOLLOW(B).
3) 如果有产生式 A→αB,或有产生式 A→αBβ 且 FIRST(β)含 ε,那么,把 FOLLOW(A)中的一切元素加入 FOLLOW(B).

*解释原理

4. 例子: 文法

E→TE'
E'→+TE'| ε
T→FT'
T'→*FT'| ε
F→(E)| id

计算 FIRST 集和 FOLLOW 集.

(见书 P222)

五. 构造预测分析表(predictive parsing table)

1. 算法: 构造预测分析表

输入:文法 G;
输出:分析表 M
方法:

1) 对文法的每个产生式 A→α 执行步骤(2)和(3);
2) 对 FIRST(α)中的每个终结符 a,把 A→α 加入 M[A,a];
3) 如果 ε 在 FIRST(α)中,对 FOLLOW(A)的每个终结符 b (包括$),把 A→α 加入 M[A,b].
4) M 中其它没有定义的条目是 error.

2. 例子：文法

E→TE'

E'→+TE'|ε

T→FT'

T'→*FT'|ε

F→(E)|id

FIRST,FOLLOW 集见前例(见书 P222),

构造分析表(见书 P225)

六. LL(1)文法

1. LL(1)文法

预测分析程序或递归下降分析程序可以不需要回溯地分析的文法类称为 LL(1)文法.

*其中第一个 L 表示从左到右扫描输入，第二个 L 表示产生最左推导，“1”表示每一步向前看一个输入符号来作出分析的决定.

换句话说，预测分析表没有多重定义条目的文法叫做 LL(1)文法.

2. 充要条件

文法 G 是 LL(1)的当且仅当 G 的任何两个产生式 A→α|β 满足下面的条件：

1) 没有终结符 a 能使 α 和 β 推出的串都以 a 开始，即

FIRST(α)∩FIRST(β)=ϕ;

2) α 和 β 中最多有一个可以推出空串;

3) 如果 $\beta \overset{*}{\Rightarrow} \varepsilon$,那么 α 推出的任何串不会以 FOLLOW(A)中的终结符开始，即 FIRST(α)∩FOLLOW(A)=ϕ.

*解释充要条件的原理

3. 二义文法

S→iEtSS'|a

S'→eS | ε

E→b

分析表(见书 P226)

*解释分析表中的多重条目,处理方法.

阅读：教材 4.3.5, 4.4 节

作业 5:

1. 为下面每个语言设计一个文法,其中哪些语言是正规的?

(a)每个 0 后面至少有一个 1 跟随的所有 0 和 1 的串;

(b) 0 和 1 的个数相等的所有 0 和 1 的串.

2. 给定文法

S→(L)|a

L→L,S|S

(a)消除该文法的左递归;

(b)为(a)题所得到的文法构造预测分析表.

3. 证明:没有 ε 产生式的文法,只要每个非终结符的各个选择(产生式)以不同的终结符开始,那么它是 LL(1)的.

§3.5 自下而上分析(Bottom-Up Parsing)

*本节介绍叫做移进－归约分析的自下而上分析的一般风格.

一. 移进－归约分析(shift-reduce parsing)

1. 思想:

   移进－归约分析为输入串构造分析树是从叶结点开始,朝根结点逆序前进,它实际上跟踪的是最右推导的逆过程.

2. 归约(reduce):

   做每一步归约时,一个子串和某个产生式右部匹配,用它的左部符号代替这个子串.

3. 例子: 文法

   E→E+T | T , T→T*F | F , F→(E) | id

   (构造 id*id 的分析树的过程, 见书 P234)

   最右推导: $id*id \underset{rm}{\Leftarrow} F*id \underset{rm}{\Leftarrow} T*id \underset{rm}{\Leftarrow} T*F \underset{rm}{\Leftarrow} T \underset{rm}{\Leftarrow} E$

二. 句柄(handle)

*上述过程第 3 步不能将 T 归约成 E,我们要找的是最右推导的逆过程.

1. 定义: 右句型(right sentential form) γ 的句柄是一个产生式 A→β 和 γ 中的一个位置,在这个位置可以找到串 β,用 A 代替 β 得到最右推导的前一个右句型.

   例: 如果 $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha Aw \underset{rm}{\Rightarrow} \alpha\beta w$, 那么 α 后的 β 称为 α β w 的句柄.

2. 例子: 文法见上例, 以下下划线的子串为句柄

   $E \underset{rm}{\Rightarrow} \underline{T} \underset{rm}{\Rightarrow} \underline{T*F} \underset{rm}{\Rightarrow} T*\underline{id} \underset{rm}{\Rightarrow} \underline{F}*id \underset{rm}{\Rightarrow} \underline{id}*id$

   对于二义文法, 同一个句型中可能有多个句柄.

   例: 文法

   1) E→E+E
   2) E→E*E
   3) E→(E)
   4) E→id

   最右推导和句柄的例子

   $E \underset{rm}{\Rightarrow} \underline{E+E} \underset{rm}{\Rightarrow} E+\underline{E*E} \underset{rm}{\Rightarrow} E+E*\underline{id_3} \underset{rm}{\Rightarrow} E+\underline{id_2}*id_3 \underset{rm}{\Rightarrow} \underline{id_1}+id_2*id_3$

   $E \underset{rm}{\Rightarrow} \underline{E*E} \underset{rm}{\Rightarrow} E*\underline{id_3} \underset{rm}{\Rightarrow} \underline{E+E}*id_3 \underset{rm}{\Rightarrow} E+\underline{id_2}*id_3 \underset{rm}{\Rightarrow} \underline{id_1}+id_2*id_3$

   *因为本文法是二义的,$E+E*id_3$ 有两个句柄.如果文法无二义性,那么最右推导唯一,句柄也唯一.

三. 剪句柄(Handle Pruning)

1. 最左归约: 最右推导的逆叫最左归约.
2. 剪句柄: 即找出句柄,构造最左归约.

四. 用栈实现移进－归约

1. 要解决的问题:
   1) 确定右句型中将要归约的子串
   2) 确定句柄的右端何时出现在栈顶.

2. 用栈实现移进－归约分析

   STACK INPUT

   $ w$

   分析器移进零个或多个输入记号入栈,直到句柄β在栈顶为止.再把β归约成某个恰当的产生式的左部. 反复执行以上步骤,直到最后栈中只剩$和文法开始符号,而输入串中只剩$, 则分析成功.

3. 分析器的动作:
   1) 移进动作: 下一个输入符号移到栈顶
   2) 归约动作: 分析器发现句柄的右端已在栈顶,它必须在栈中确定句柄的左端,再决定用什么样的非终结符代替句柄.
   3) 接受动作:分析器宣告分析成功.
   4) 报错动作:分析器发现语法错误,调用错误恢复例程.

4. 例子: (见书 P237)

§3.6 算符优先分析法

*这节我们介绍一种简单直观,广为使用的自下而上分析法,叫做算符优先分析法.这种方法特别有利于表达式分析. 算符优先分析法的归约不一定是严格的最左归约.

一. 引言

*通常的算术规则是乘除优先级高于加减,且左结合, 先括号里,后括号外.

例 1: 9+8－6/2*3 运算

例 2: 给定文法

E→E+E | E－E | E*E | E/E | (E) | i

归约 i+i*i

按算符优先级的高低归约.

二. 直观算符优先分析法

1. 优先关系:

   a <· b a 的优先级低于 b

   a $\dot{=}$ b a 的优先级等于 b

   a ·> b a 的优先级高于 b

   注意: 这不同于算术中的<, =, 或>.

   a <· b 不一定有 b ·> a, a $\dot{=}$ b 也不一定有 b $\dot{=}$ a

   a <· b 且 b <· c 也不一定有 a <· c

2. 优先关系表举例

   设有文法: E→E+E | E*E | E↑E | (E) | i

优先关系表

	+	*	↑	i	(	)	$
+	·>	<·	<·	<·	<·	·>	·>
*	·>	·>	<·	<·	<·	·>	·>
↑	·>	·>	<·	<·	<·	·>	·>
i	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	$\doteq$	
)	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·		

*解释:

+ <· * * ·> +
* <· ↑ ↑ ·> *
+ <· ↑ ↑ ·> +
+ ·> + * ·> * ↑ <· ↑
(<· θ) ·> θ
i ·> θ θ <· i
$ <· θ θ ·> $

三. 算符优先文法和优先表的构造

1. 算符文法: 如果一个文法的任何产生式的右部都不含两个相继的非终结符, 那么这种文法叫做算符文法.

2. 确定文法中的优先关系

假定 G 是一个不含 ε 产生式的算符文法.对任何一对终结符 a, b, 我们说:

1) a $\doteq$ b 当且仅当文法 G 中含有形如 P→…ab…或 P→…aQb…的产生式;

2) a <· b 当且仅当 G 中含有形如 P→…aR…的产生式,而 $R \overset{+}{\Rightarrow} b\cdots$ 或 $R \overset{+}{\Rightarrow} Qb\cdots$;

3) a ·> b 当且仅当 G 中含有形如 P→…Rb… 的产生式,而 $R \overset{+}{\Rightarrow} \cdots a$ 或 $R \overset{+}{\Rightarrow} \cdots aQ$;其中, P, Q, R∈V_N, a, b ∈V_T .

3. 算符优先文法

如果一个算符文法 G 中的任何终结符对(a, b)至多满足下述三个关系之一:

a $\doteq$ b, a <· b, a ·> b

则称 G 是算符优先文法.

4. 例子: 文法

E→E+T | T

T→T*F | F

F→P↑F | P

P→(E) | i

*根据文法, 有($\dot{=}$), + ＜· *, * ＜· ↑ , ↑ ＜· ↑ 等.

5. FIRSTVT 集与 LASTVT 集

FIRSTVT(P)={a | $P \overset{+}{\Rightarrow} a\cdots$ 或 $P \overset{+}{\Rightarrow} Qa\cdots$, $a\in V_T, Q\in V_N$ }

LASTVT(P)={a | $P \overset{+}{\Rightarrow} \cdots a$ 或 $P \overset{+}{\Rightarrow} \cdots aQ$, $a\in V_T, Q\in V_N$ }

6. 构造 FIRSTVT 集和 LASTVT 集的算法

构造 FIRSTVT(P). 对所有的非终结符 P 构造 FIRSTVT 集,直到它们的 FIRSTVT 集不再增大为止.

1) 若有产生式 P→a… 或 P→Qa…, 则 a∈FIRSTVT(P);
2) 若 a∈FIRSTVT(Q),且有产生式 P→Q…,则 a∈FIRSTVT(P);

构造 LASTVT(P).对所有的非终结符 P 构造 LASTVT 集,直到它们的 LASTVT 集不再增大为止.

3) 若有产生式 P→…a 或 P→…aQ,则 a∈LASTVT(P);
4) 若 a∈LASTVT(Q),且有产生式 P→…Q ,则 a∈LASTVT(P).

7. 例子: 文法

E→E+T | T

T→T*F | F

F→P↑F | P

P→(E) | i

构造 FIRSTVT 集和 LASTVT 集(见黑板)

8. 构造优先分析表的算法

FOR 每个产生式 $P\rightarrow X_1X_2\cdots X_n$ DO

FOR i:= 1 TO n－1 DO

BEGIN

IF X_i 和 X_{i+1} 均为终结符 THEN 置 $X_i \dot{=} X_{i+1}$;

IF i ≤n－2 且 X_i 和 X_{i+2} 均为终结符,但 X_{i+1} 为非终结符 THEN 置 $X_i \dot{=} X_{i+2}$;

IF X_i 为终结符而 X_{i+1} 为非终结符号 THEN

FOR FIRSTVT(X_{i+1})中每个 a DO

置 X_i ＜· a ;

IF X_i 为非终结符而 X_{i+1} 为终结符 THEN

FOR LASTVT(X_i)中的每个 a DO

置 a ·＞ X_{i+1}

END

9. 例子: 用上例中的 FIRSTVT 集和 LASTVT 集构造优先关系表. (该表见二. 2.条中的优先关系表)

四. 算符优先分析算法的设计

1. 最左素短语

我们把句型(括在$和$之间)的一般形式写成

$\$N_1a_1N_2a_2\cdots N_na_nN_{n+1}\$$

其中 a_i 为终结符, N_j 为非终结符.

一个算符优先文法 G 的任何句型的最左素短语是满足如下条件的最左子串 $N_ja_jN_{j+1}\cdots N_ia_iN_{i+1}$,其中:

$a_{j-1} \lessdot a_j$

$a_j \doteq a_{j+1}$, …, $a_{i-1} \doteq a_i$

$a_i \gtrdot a_{i+1}$

算符优先分析算法就是找出每个句型的最左素短语进行归约.

2. 算符优先分析算法

```
k:=1;  S[k] :='$';
REPEAT
    把下一个输入符号读进 a 中;
    IF S[k] ∈ VT THEN j:=k ELSE j:= k－1;
    WHILE S[j] ·> a  DO
    BEGIN
        REPEAT
            Q := S[j];
            IF S[j－1] ∈ VT THEN j:=j－1 ELSE j:= j－2
            UNTIL S[j] <· Q;
            把 S[j+1], …, S[k]归约为某个非终结符 N;
            k:=j+1;
            S[k] := N
    END;
    IF S[j] <· a OR S[j] ≐ a THEN
    BEGIN k:= k+1; S[k]:=a END
    ELSE ERROR
UNTIL a = '$'
```

五. 优先函数

1. 优先函数: 设两个函数 f 和 g, 它们把每个终结符映射到整数, 使得 $\forall a, b \in V_T$,

1) f(a) < g(b), 若 $a \lessdot b$

2) f(a) = g(b), 若 $a \doteq b$

3) f(a) > g(b), 若 $a \gtrdot b$

2. 优先函数的优缺点:

优点:

1) 节省存储空间
2) 便于执行比较运算

缺点:

1) 使用函数值, 使原来没有关系的终结符对变得有关系了.
2) 有些优先关系表不存在对应的优先函数

3. 从优先关系表构造优先函数

方法:

1) 为每个 a(终结符或$)建立符号 f_a 和 g_a;
2) 用下面的方法把所建立的符号分成若干组:如果 $a \doteq b$, 那么 f_a 和 g_b 在同一组中.
3) 建立有向图, 它的结点是(2)中的符号组.对任何 a 和 b,如果 $a < \cdot b$,从 g_b 所在组引一条有向边到 f_a 所在组;如果 $a \cdot > b$, 从 f_a 所在组引一条有向边到 g_b 所在组.
4) 如果(3)构造的图有圈,那么优先函数不存在.如果无圈,令 f(a)等于 f_a 所在组开始的最长有向路径上的顶点数;令 g(b)等于从 g_b 开始的最长有向路径上的顶点数.

4. 例子: 优先关系表见二. 2. 条中的表, 优先函数如下:

	+	*	↑	(	)	i	$
f	3	5	5	1	7	7	1
g	2	4	6	6	1	6	1

阅读: 教材 4.5 节和讲义

作业 6:

1. 给出文法

S→(L) | a
L→L, S | S

a)构造句子(a ,(a ,a))的最右推导, 指出每个右句型的句柄.
b)给出对应 a)题中的最右推导的移进—归约分析步骤.

2. 给出下列文法

E→E '|' T | T
T→T●F | F
F→F* | P
P→(E) | a | b

(1) 计算 FIRSTVT 集和 LASTVT 集, 并构造算符优先关系表;
(2) 根据优先关系表构造优先函数.

§3.7 LR 分析器(LR Parser)

*本节提出一种有效的、自下而上的语法分析技术, LR(k) 分析技术, 它能适用于一大类上下文无关文法的分析.

*其中:L 表示从左到右扫描输入符号串,R 表示构造最右推导的逆过程, k 表示在作分析决定时要向前看 k 个输入符号. 在实际的编译器中,我们只考虑 k=0 或 k=1 的情形. 当(k)省略时,表示 k=1.

一. LR 分析方法的优缺点

优点:

1. 能够构造 LR 分析器来识别所有能用上下文无关文法写的程序设计语言的结构.
2. LR 分析方法是已知的最一般的无回溯移进－归约方法. 它能够和其它移进－归约方法一样有效地实现.
3. LR 分析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(proper superset).
   *LR 分析方法向前看 k 个符号只需看右句型中某个产生式右边的 k 个符号,而预测分析方法向前看 k 个符号要看某个产生式右边的文法符号能推出的前 k 个符号. 因此,LR 分析法比 LL 分析法简单,适用的文法类更广.
4. LR 分析器能及时察觉语法错误,在自左向右扫描输入时,尽可能快地发现错误.

缺点:

对典型的程序设计语言文法,手工构造 LR 分析器工作量太大.
*目前已有许多自动生成 LR 分析器的生成器, 例如: Yacc.

二. LR 分析算法

1. 结构: (见书 P248)
   栈中: $s_0X_1s_1X_2s_2\cdots X_ms_m$
   其中: s_i : 状态(state) , X_i : 文法符号(grammar symbol)

   分析表: 动作函数 action 和转移函数 goto

2. 动作: 根据当前栈顶状态 s_m 和当前输入符号 a_i, action[s_m,a_i]有四种可能动作:
   1) 移进(shift)s, s 是一个状态
   2) 按文法产生式 A→β 归约(reduce)
   3) 接受(accept)
   4) 出错

3. 活前缀(viable prefix)
   文法 G 的活前缀是它的右句型(right sentential form)的前缀, 它不超过该句型中最左句柄(handle)的右端.
   例: 文法 E→E+T | T , T→T*F | F , F→(E) | id , 存在最右推导: $E \underset{rm}{\Rightarrow} T \underset{rm}{\Rightarrow} T*F \underset{rm}{\Rightarrow} T*id \underset{rm}{\Rightarrow} F*id \underset{rm}{\Rightarrow} (E)*id$

   (, (E, (E) 都是右句型(E)*id 的活前缀.

4. 格局(configuration):
   栈中内容: $s_0X_1s_1X_2s_2\cdots X_ms_m$, 剩余输入串: $a_ia_{i+1}\cdots a_n\$$
   组成当前格局: ($s_0X_1s_1X_2s_2\cdots X_ms_m$, $a_ia_{i+1}\cdots a_n\$$)
   这个格局代表右句型: $X_1X_2\cdots X_ma_ia_{i+1}\cdots a_n$

5. 分析器动作:
   设当前格局是($s_0X_1s_1X_2s_2\cdots X_ms_m$, $a_ia_{i+1}\cdots a_n\$$)
   1) 如果 action[s_m,a_i] =移进 s, 分析器进入格局: ($s_0X_1s_1X_2s_2\cdots X_ms_ma_is$, $a_{i+1}a_{i+2}\cdots a_n\$$)
   2) 如果 action[s_m,a_i] =归约 A→β, 分析器进入格局: ($s_0X_1s_1X_2s_2\cdots X_{m-r}s_{m-r}As$, $a_ia_{i+1}\cdots a_n\$$)

其中: s = goto[s_{m-r} , A], r 是 β 的长度. 这里分析器首先从栈中弹出 2r 个符号, 包括 r 个状态和 r 个文法符号, 这些文法符号刚好匹配产生式的右部 β ,即 β =$X_{m-r+1}X_{m-r+2}\cdots X_m$.

3) 如果 action[s_m, a_i] =接受, 分析完成.

4) 如果 action[s_m,a_i] =出错, 分析器发现错误,调用错误恢复例程.

6. 例子: 文法

1) E→E+T
2) E→T
3) T→T*F
4) T→F
5) F→(E)
6) F→id

分析表(parsing table) (见书 P252)

给出串 id*id+id 的移进－归约过程.(见书 P253)

*注意: 在实际的 LR 分析器中, 栈中不保存文法符号, 栈顶的状态代表栈中的内容.

三. 构造 SLR 分析表

1. 项目(item): 是右部的某个地方加点的 G 的产生式.

例如: 产生式 A→XYZ

有项目: A→.XYZ
A→X.YZ
A→XY.Z
A→XYZ.

产生式 A→ ε 只有一个项目 A→.

*解释项目中加点的意义.

2. 拓广文法(augmented grammar):

在文法 G 中引入产生式: S’→ S, 得拓广文法 G’, S’是 G’的开始符号, S 是 G 原来的开始符号.

*引入这个新的产生式的目的是指出什么时候分析结束, 宣布接受输入串.

3. 闭包函数

设 I 是项目集, 那么 closure(I)是由下面两条规则从 I 构造的项目集(set of items):

1) 初始, I 的每个项目都加入 closure(I);
2) 如果 A→ α . B β 在 closure(I)中, 且 B→ γ 是产生式, 那么, 如果 B→. γ 不在 closure(I)中, 则把它加入. 反复运用这条规则, 直到没有更多的项目可加入 closure(I)为止.

*A→ α . B β 的直观意义和 B→. γ 加入 closure(I)的意义.

例: 文法: E’→E
E→E+T | T
T→T*F | F
F→(E) | id

设 I = { E’→.E }, 那么 closure(I)为
{ E’→.E ,
E→.E+T ,
E→.T ,

T→.T*F ,
T→.F ,
F→.(E) ,
F→.id }

4. 核心项目与非核心项目(kernel items and nonkernel items)
   1) 核心项目: 它包括初始项目 S'→.S 和所有点不在左端的项.
   2) 非核心项目: 它们的点在左端.
   *一个项目集可以只保留核心项目, 并通过求闭包求得该项目集.

5. goto 函数:
   设 I 是项目集, X 是文法符号, 并设 I 中包含项目
   A→α.Xβ . 那么 goto(I,X) = closure(I'), 其中 I'是包含
   A→αX.β 的项目集.
   例: I = { E'→E. , E→E.+T }
   goto(I, +)包含: E→E+.T T→.T*F T→. F F→.(E) F→. id

6. 求拓广文法 G'的 LR(0)项目集规范族(canonical collection of sets of LR(0) items) 的算法
   Procedure item(G');
   BEGIN
   C := { closure({[S'→.S]})};
   REPEAT
   FOR C 的每个项目集 I 和每个文法符号 X DO
   IF goto(I, X)非空且不在 C 中 THEN
   把 goto(I, X)加入 C;
   UNTIL 本次循环没有项目集可以加入 C
   END
   例子: 文法: E'→E , E→E+T | T , T→T*F | F , F→(E) | id
   LR(0)项目集规范族(见书 P244)
   该项目集规范族可构成一个 DFA (见书 P244)

7. 构造 SLR 分析表算法
   输入: 拓广文法 G'
   输出: G'的 SLR 分析表函数 action 和 goto
   方法:
   1) 构造 C = {$I_0, I_1, \cdots, I_n$}, 即 G'的 LR(0)项目集规范族.
   2) 状态 i 从 I_i 构造, 它的分析动作如下确定:
      a) 如果[A→α.aβ]在 I_i 中, 并且 goto(I_i , a) = I_j , 那么置 action[i, a]为 sj .
      b) 如果[A→α.]在 I_i 中, 那么对所有 FOLLOW(A)中的 a,置 action[i, a]为 rj, j 是产生式 A→α 的编号.
      c) 如果[S'→S.]在 I_i 中, 那么置 action[i, $]为接受 acc.
   3) 对所有的非终结符 A, 使用下面规则构造状态 i 的转移: 如果 goto(I_i , A) = I_j , 那么 goto[i, A] = j .
   4) 不能由规则(2)和(3)定义的条目都置为出错.
   5) 分析器的初始状态是从包含[S'→.S]的项目集构造的状态.

8. 例子: 从第 6 条的文法和 LR(0)项目集规范族构造分析表(见书 P252)

9. 非 SLR 文法

*若对某个文法 G，按上述方法构造的 SLR 分析表的每一个条目都没有多重定义，那么该文法 G 称为 SLR 文法.

*存在非 SLR 的上下文无关文法.

例: 文法: S→L=R | R

L→*R | id

R→L

LR(0)项目集规范族(见书 P255).

*项目集 I_2 面临输入“=”时，产生移进/归约冲突.

*该文法虽然不是二义文法，但也不是 SLR 文法.

阅读: 教材 4.6 节

作业 7:

1. 考虑下面文法

E→E+T | T

T→TF | F

F→F* | a | b

为此文法构造 SLR 分析表.

四. 构造规范 LR 分析表(canonical LR parsing table)

1. SLR 分析中的问题:

*在 SLR 方法中，如果 I_i 包含项目[A→α.]且 a∈FOLLOW(A)，那么状态 i 要求 A→α 归约.

在有些情况下,当状态 i 出现在栈顶，活前缀 βα 在栈中，a 是当前输入符号，用 A→α 来归约却是非法的，因为在右句型中，βA 不能由 a 跟随.

例: (见书 P255)

状态 2 有项目 R→L.，=∈FOLLOW(R)，这时 S→L.=R 也在状态 2 中. 但用 R→L.归约不合法，因为在文法中，不存在以 R=…开始的右句型.

*为此，我们引进 LR(1)项目.

2. LR(1)项目(LR(1) items)

[A→α.β, a]，其中 A→α β 是产生式, a 是终结符.

*[A→α.β, a]当 β 不能推出空串时, a 无用. [A→α., a]仅当当前输入符号是 a 时才归约.

3. LR(1)项目对活前缀有效(valid for a viable prefix)

我们说 LR(1)项目[A→α.β, a]对活前缀 γ 有效，如果存在着推导 $S \overset{*}{\underset{rm}{\Rightarrow}} \delta A w \underset{rm}{\Rightarrow} \delta \alpha \beta w$，其中

1. $\gamma = \delta\alpha$, 和
2. a 是 w 的第一个符号或者 w 是 ε 且 a 是$.

例如: 文法: S→BB, B→aB | b

存在最右推导: $S \overset{*}{\underset{rm}{\Rightarrow}} aaBab \underset{rm}{\Rightarrow} aaaBab$

[B→a.B , a]对活前缀 γ =aaa 有效, 其中 δ =aa, A=B, w=ab,
α =a 且 β =B.

又例如: $S \overset{*}{\underset{rm}{\Rightarrow}} BaB \underset{rm}{\Rightarrow} BaaB$, 因此, 项目[B→a.B, $]对活前缀 Baa 有效.

Closure 运算的新定义. 考虑对活前缀 γ 有效的项目[A→α .B β ,a], 必有一个最右推导 $S \overset{*}{\underset{rm}{\Rightarrow}} \delta Aax \underset{rm}{\Rightarrow} \delta \alpha B \beta a x$, 其中 $\gamma = \delta \alpha$,假定 β a x 推出串 by, 那么对每个形式为 B→ η 的产生式, 有推导 $S \overset{*}{\underset{rm}{\Rightarrow}} \gamma Bby \underset{rm}{\Rightarrow} \gamma \eta by$. 于是[B→. η ,b]对 γ 有效. 这里 b∈FIRST(β a).

4. 算法: 构造 LR(1)项目集

输入:拓广文法 G'
输出:LR(1)项目集. 它们是对 G' 的一个或多个活前缀有效的项目集.
方法:

```
function   closure(I);
begin
   repeat
      for I 的每个项目[A→α .B β ,a] do
          for G'中的每个产生式 B→γ  do
              for  FIRST( β a)的每个终结符 b do
                  If  [B→. γ , b]不在 I 中  then
                      把[B→. γ , b]加入 I 中
   until  再没有新的项目可以加入 I 中;
   return (I);
end;
function   goto(I, X);
begin
     初始化 J 为空集;
     for I 中每一个项目[A→α .X β ,a] do
          将项目[A→α X. β ,a]加入集合 J;
     return (closure(J));
end;
procedure   items(G');
begin
     C := { closure({[S'→.S , $]}) }
     repeat
          for C 的每个项目集 I do
              for  每个文法符号 X do
                  If goto(I, X)非空并且不在 C 中  then
                      把 goto(I, X)加入 C 中
     until  再没有新的项目集可以加入 C;
end;
```

5. 例子: 文法: S'→S
S→CC

C→cC | d

构造 LR(1)项目集. (见书 P262)

6. 构造规范 LR 分析表算法

输入: 拓广文法 G'

输出: 文法 G'的规范 LR 分析表函数 action 和 goto

方法:

1) 构造 G'的 LR(1)项目集规范族 C={I_0, I_1, …, I_n}
2) 从 I_i 构造分析器状态 i, 状态 i 的分析动作确定如下:
   a) 如果[A→α.aβ,b]在 I_i 中且 goto(I_i, a) = I_j, 那么置 action[i, a]为 sj.
   b) 如果[A→α., a]在 I_i 中且 A≠S', 那么 action[i, a]置为 rj, j 是产生式 A→α 的序号.
   c) 如果[S'→S., $]在 I_i 中, 那么 action[i, $] = acc.
3) 从状态 i 的转移如下确定: 如果 goto(I_i, A) = I_j, 那么 goto[i, A] = j .
4) 用规则(2)和(3)未能定义的所有条目都置为出错.
5) 分析器的初始状态是从包含项目[S'→.S, $]的集合构造的那个状态.

7. 例子: 用书中 P262 的项目集构造分析表(见书 P266)

五. 构造 LALR 分析表

*LALR 表示 LookAhead LR.

1. LR 分析法的问题

状态太多. LALR 和 SLR 分析表对同一文法有同样多的状态, 而大多数普通的程序设计语言的结构又都能方便地由 LALR 文法表示.

例: (见书 P262)考虑该项目集族, I_4 和 I_7 作用不同, 但它们的项目除第二个成份不同外, 第一个成分相同. 我们称它们为同心(common core)的 LR(1)项目集.

2. LALR 分析的思想:

合并同心的 LR(1)项目集.

*合并后的 LALR 项目集不会产生移进/归约冲突, 但是可能产生归约/归约冲突.

3. 构造 LALR 项目集族的算法

输入: 拓广文法 G'

输出: G'的 LALR 分析表 action 和 goto 函数

方法:

1) 构造 LR(1)项目集族 C={I_0, I_1, …, I_n}
2) 对出现在 LR(1)项目集中的每个心, 找出所有与之同心的项目集, 用它们的并代替它们.
3) 令 C'={J_0, J_1, …, J_m}是合并后的 LR(1)项目集族.状态 i 的动作从 J_i 构造, 方法同 LR(1)分析法.
4) goto 表的构造如下: 如果 J 是一个或多个 LR(1)项目集的并, 即 J = $I_1 \cup I_2 \cup \cdots \cup I_k$, 那么 goto($I_1$, X), goto($I_2$, X),…, goto($I_k$, X)也同心, 因为 I_1, I_2, …, I_k 都同心.记 K 为所有和 goto(I_1, X)同心的项目集的并, 那么 goto(J, X) = K.

4. 构造分析表的算法同 LR(1)分析方法.
5. 例子: (见书 P262)

合并 I_3 和 I_6, I_4 和 I_7, I_8 和 I_9, 构造分析表(见书 P269)

6. 构造 LALR 分析表的有效方法.

阅读: 教材 4.7 节

作业 8:

1.考虑文法

(1) S→L=R (2) S→R

(3) L→*R (4) L→id

(1) R→L

a)构造规范 LR 分析表;

b)构造 LALR 分析表.

第四章 语法制导翻译和中间代码产生

§4.1 语法制导翻译概说

一. 语法制导翻译(Syntax-Directed Translation (SDT))

语法制导翻译的意思是: 让每一个产生式对应一个语义子程序(或称为语义规则或语义动作).在语法分析过程中,当一个产生式获得匹配(对于自上而下分析)或用于归约(对于自下而上分析)时,此产生式相应的语义子程序就进入工作, 完成既定的翻译任务.

二. 语义值

1. 语义值: 在描述语义动作时,我们需要赋予每个文法符号 X(终结符或非终结符)以种种不同方面的“值”, 如“类型”, “种属”,“地址”或“代码”等等. 记为 X.TYPE, X.CAT, X.VAL 等.

2. 语法制导定义(syntax-directed definition (SDD)):

一个语法制导定义是一个上下文无关文法以及相关联的属性(attributes)(语义值)和规则(semantic rules)(语义子程序).

3. 例子:

1) $E \rightarrow E^{(1)}+E^{(2)}$ {E.VAL := $E^{(1)}$.VAL + $E^{(2)}$.VAL}

2) E→0 {E.VAL := 0}

3) E→1 {E.VAL := 1}

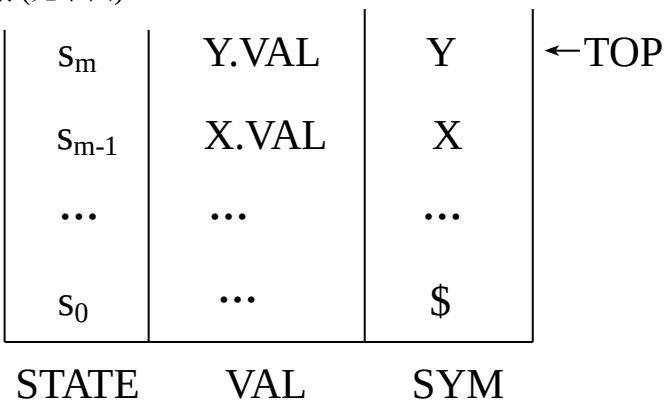
4. 语义值的传递

1) X→… {动作 1}

2) Y→… {动作 2}

3) A→XY {动作 3}

用栈保存(见下图)



三. 语义动作

(0) S'→E {print E.VAL}
(1) $E \rightarrow E^{(1)}+E^{(2)}$ {E.VAL := $E^{(1)}$.VAL + $E^{(2)}$.VAL}
(2) $E \rightarrow E^{(1)}*E^{(2)}$ {E.VAL := $E^{(1)}$.VAL * $E^{(2)}$.VAL}
(3) $E \rightarrow (E^{(1)})$ {E.VAL := $E^{(1)}$.VAL}
(4) E→n {E.VAL := LEXVAL}

用栈来实现, 语义动作化为

(0) S'→E print VAL[TOP]
(1) $E \rightarrow E^{(1)}+E^{(2)}$ VAL[TOP] :=VAL[TOP]+VAL[TOP+2]
(2) $E \rightarrow E^{(1)}*E^{(2)}$ VAL[TOP] :=VAL[TOP]*VAL[TOP+2]
(3) $E \rightarrow (E^{(1)})$ VAL[TOP] :=VAL[TOP+1]
(4) E→n VAL[TOP] := LEXVAL

*这是计算器的语义动作

§4.2 逆波兰表示法(后缀表示法)

一. 逆波兰表示法

即将表达式用后缀式形式表示.

例: a+b 表示为 ab+

(a+b)*c 表示为 ab+c*

*这种表示法不需要括号.

abc+* 表示 a*(b+c)

ab+cd+* 表示 (a+b)*(c+d)

*解释如何计算逆波兰表达式

二. 后缀式的计算

用栈来计算.自左至右扫描后缀式,每碰到运算量就把它推进栈,每碰到k目算符就把它作用于栈顶的k个项,并用运算结果来代替这 k 个项.

例: ab+c*

三. 后缀式的推广

考虑条件表达式: if e then x else y

*含义

可用后缀算符: exy¥

但这样计算有问题.

引入条件转移和无条件转移算符:

p jump, e_1e_2 p jlt , e p jez

*解释含义

假定后缀式放在一维数组 POST[1..n]中, 每个数组元素放一个运算量或一个算符.

if e then x else y 表示为

e p_1 jez x p_2 jump p_1: y p_2:

*后跟冒号的标号并不真正出现在后缀式中.

在 POST 数组中为: (见黑板)

四. 语法制导生成后缀式:

$E \rightarrow E^{(1)}$ op $E^{(2)}$ {E.CODE :=$E^{(1)}$.CODE ‖ $E^{(2)}$.CODE ‖ op}

$E \rightarrow (E^{(1)})$ {E.CODE := $E^{(1)}$.CODE }

E→i {E.CODE := i }

*捻接操作“‖”实现起来代价很高. 如果事先安排一个数组 POST 存放后缀式, k 为下标,初值为 1, 则语义动作为:

$E \rightarrow E^{(1)}$ op $E^{(2)}$ {POST[k] := op; k := k+1}

$E \rightarrow (E^{(1)})$ { }

E→i {POST[k] := i; k := k+1}

§4.3 三元式和树

一. 三元式(Triple)

例 1: A+B*C 表示成三元式:(见下图)

	op	ARG1	ARG2
(1)	*	B	C
(2)	+	A	(1)

例 2: X := A+B*C 表示成三元式:(见下图)

	op	ARG1	ARG2
(1)	*	B	C
(2)	+	A	(1)
(3)	:=	X	(2)

*三元式出现的先后顺序是和表达式各部分的归约顺序相一致的.

二. 翻译成三元式

产生式	语义动作
(1) $E \rightarrow E^{(1)}$ op $E^{(2)}$	{E.VAL:=TRIP(op, $E^{(1)}$.VAL, $E^{(2)}$.VAL)}
(2) $E \rightarrow (E^{(1)})$	{E.VAL:=$E^{(1)}$.VAL }
(3) $E \rightarrow -E^{(1)}$	{E.VAL:=TRIP(@, $E^{(1)}$.VAL, −)}
(4) E→i	{E.VAL:=ENTRY(i) }

*语义过程 ENTRY 是一个对 i 所代表的标识符查找符号表,以获得它在表中的入口的函数过程.

*语义过程 TRIP(op, arg1, arg2): 产生一个三元式(op, arg1, arg2), 该三元式的序号是保存该三元式运算结果的地址, 称为 value number.

三. 间接三元式(indirect triples)

例: 语句 X := (A+B)*C

Y := D ↑ (A+B)

间接三元式(见下图)

间接码表

(1)	(1)
(2)	(2)
(3)	(3)
(4)	(4)
(5)	(5)
(6)	(6)

三元式表

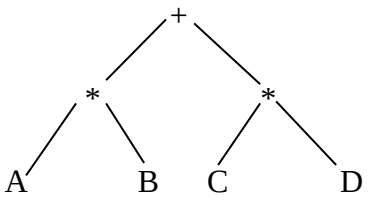
	op	ARG1	ARG2
(1)	+	A	B
(2)	*	(1)	C
(3)	:=	X	(2)
(4)	+	A	B
(5)	↑	D	(4)
(6)	:=	Y	(5)

*解释为什么设立间接码表.

四. 树

A*B+C*D

树代码(见下图)



*上述树型表示与以下三元式表示等价

	op	ARG1	ARG2
(1)	*	A	B
(2)	*	C	D
(3)	+	(1)	(2)

五. 翻译成树形表示

产生式	语义动作
(1) $E \rightarrow E^{(1)}$ op $E^{(2)}$	{E.VAL:=NODE(op, $E^{(1)}$.VAL, $E^{(2)}$.VAL)}
(2) $E \rightarrow (E^{(1)})$	{E.VAL := $E^{(1)}$.VAL }
(3) $E \rightarrow -E^{(1)}$	{E.VAL := UNARY(@, $E^{(1)}$.VAL) }
(4) $E \rightarrow i$	{E.VAL := LEAF(i) }

*解释语义动作

§4.4 四元式(Quadruples)

*四元式等价于三地址代码(three-address code)

一. 四元式表示

例: A :=－B*(C+D), 四元式表:(见下图)

	op	ARG1	ARG2	RESULT
(1)	@	B	__	T_1
(2)	+	C	D	T_2
(3)	*	T_1	T_2	T_3
(4)	:=	T_3	__	A

*产生四元式的顺序与归约顺序一致.临时变量可以任意引入.在实际中用一个数来表示.
*解释四元式比三元式优越的地方

§4.5 简单算术表达式和赋值语句到四元式的翻译

*这节我们讨论含简单变量的表达式和赋值语句到四元式的翻译.以后各节我们都假定中间语言是四元式.
为使问题简化, 我们暂时忽略语义检查.

一. 语义变量和语义过程

1. NEWTEMP: 它是一个函数过程, 每次调用时, 它都回送一个代表新临时变量名的整数码作为函数值. 临时变量名按产生顺序可想像为 T_1, T_2, … 等等.
2. ENTRY(i): 它是一个函数过程. 它对 i 所代表的标识符查找符号表,并返回它在表中的入口.
3. E.PLACE: 它是和非终结符 E 相联系的语义变量,表示存放 E 值的变量名在符号表的入口或整数码(若此变量是一个临时变量).
4. GEN(op, ARG1, ARG2, RESULT): 它是一个语义过程, 该过程把四元式(op, ARG1, ARG2, RESULT)填进四元式表中.

二. 语义动作:

	产生式	语义动作
(1)	A→i:=E	{GEN(:=, E.PLACE, －, ENTRY(i)) }
(2)	$E \rightarrow E^{(1)}+E^{(2)}$	{E.PLACE:=NEWTEMP; GEN(+, $E^{(1)}$.PLACE, $E^{(2)}$.PLACE, E.PLACE) }
(3)	$E \rightarrow E^{(1)}*E^{(2)}$	{E.PLACE:=NEWTEMP; GEN(*, $E^{(1)}$.PLACE, $E^{(2)}$.PLACE, E.PLACE) }
(4)	$E \rightarrow -E^{(1)}$	{E.PLACE:=NEWTEMP; GEN(@, $E^{(1)}$.PLACE, － , E.PLACE) }
(5)	$E \rightarrow (E^{(1)})$	{E.PLACE := $E^{(1)}$.PLACE }
(6)	E→i	{E.PLACE := ENTRY(i) }

三. 自下而上分析产生中间代码的过程

例: 分析 A:=－B*(C+D) 的过程 (见下图)

输入	栈	PLACE	四元式
A:=－B*(C+D)			
:=－B*(C+D)	i	A	
－B*(C+D)	i:=	A_	
B*(C+D)	i:=－	A_ _	
*(C+D)	i:=－i	A_ _B	
*(C+D)	i:=－E	A_ _*B*	(@,*B*,_,T_1)
*(C+D)	i:=E	A_T_1	
(C+D)	i:=E*	A_T_1_	
C+D)	i:=E*(	A_T_1_ _	
+D)	i:=E*(i	A_T_1_ _C	
+D)	i:=E*(E	A_T_1_ _C	
D)	i:=E*(E+	A_T_1_ _C_	
)	i:=E*(E+i	A_T_1_ _*C*_D	
)	i:=E*(E+E	A_T_1_ _*C*_D	(+,C,D,T_2)
)	i:=E*(E	A_T_1_ _T_2	
	i:=E*(E)	A_T_1_ _*T_2*_	
	i:=E*E	A_T_1_T_2	(*,T_1,T_2,T_3)
	i:=E	A_T_3	(:=,T_3,_,A)
	A		

四. 类型转换

*假设表达式中有整型和实型数, 整型与实型变量进行运算必须先转换为实型.

引入四元式: (itr, A, —, T)

例: 翻译 X := Y + I * J, 假设 I,J 是整型变量,X,Y 是实型变量. 生成四元式:

(*^{i}, I, J, T_1)

(itr, T_1, —, T_2)

(+^{r}, Y, T_2, T_3)

(:=, T_3, —, X)

产生式 $E \rightarrow E^{(1)}$ op $E^{(2)}$的语义子程序

```
{T := NEWTEMP;

    IF E(1).MODE = int AND E(2).MODE = int THEN
    BEGIN
        GEN(op^i, E(1).PLACE, E(2).PLACE, T);
        E.MODE := int
    END

    ELSE IF E(1).MODE = r AND E(2).MODE = r THEN
    BEGIN
        GEN(op^r, E(1).PLACE, E(2).PLACE, T);
        E.MODE := r
    END

    ELSE IF E(1).MODE = int /*and E(2).MODE = r*/ THEN
    BEGIN
        U := NEWTEMP;
        GEN(itr, E(1).PLACE, —, U);
        GEN(op^r, U, E(2).PLACE, T)
        E.MODE := r
    END

    ELSE /*E(1).MODE = r and E(2).MODE = int */
    BEGIN
        U := NEWTEMP;
        GEN(itr, E(2).PLACE, —, U);
        GEN(op^r, E(1).PLACE, U, T)
        E.MODE := r
    END;

    E.PLACE := T }
```

阅读: 讲义和教材 6.1, 6.2, 6.4.1, 6.4.2 节

作业 9::

1. 给出下面表达式的逆波兰表示
   (1) a*(－b+c)
   (2) if (x+y)*z then (a+b) ↑ c else a ↑ b ↑ c
2. 请将－(a+b)*(c+d)－(a+b+c)分别表示成三元式和四元式序列.
3. 写出赋值句 A:=B*(－C+D)的自下而上语法制导翻译过程,给出所产生的四元式.

§4.6 布尔表达式到四元式的翻译

一. 布尔表达式(Boolean expression)的作用

1. 用作控制语句(flow-of-control statement)的条件式
2. 用于逻辑演算,计算逻辑值

*布尔表达式算符优先顺序(从高到低):

rop, ┐, ∧, ∨

设∨和∧满足左结合.

二. 布尔表达式按算术表达式方法翻译

例: A∨B∧C=D 翻译成

(=, C, D, T_1)

(∧, B, T_1, T_2)

(∨, A, T_2, T_3)

三. 优化方法:

A∨B 解释成: if A then true else B

A∧B 解释成: if A then B else false

┐A 解释成: if A then false else true

*在计算布尔表达式不产生副作用时,可以这样计算.

四. 作为条件控制的布尔表达式的翻译

1. 几个转移语句四元式

   (jnz, A_1, －, p)

   (j θ , A_1, A_2, p)

   (j , －, －, p)

2. 例子:

   if A∨B < D then S_1 else S_2

   翻译成

(1) (jnz, A, －, 5)

(2) (j , －,－, 3)

(3) (j< , B, D, 5)

(4) (j ,－,－,p+1)

(5) (关于 S_1 的四元式序列)

(p) (j, －,－,q)

(p+1) (关于 S_2 的四元式序列)

(q)

3. “真”出口(true exits)和“假”出口(false exits)

对每个布尔子表达式,给它赋予一个“真”出口和一个“假”出口, “真”出口是该子式为“真”时的出口, “假”出口是该子式为“假”时的出口.

4. 真链(true chain)、假链(false chain)与回填(backpatch)

五. 语义变量与语义过程

1. E.TC 指向 E 的真链的链首
2. E.FC 指向 E 的假链的链首
3. NXQ 指向下一个将要形成但尚未形成的四元式的地址(编号). NXQ 的初值为 1, 每执行一次 GEN 之后, NXQ 将自动累增 1.
4. MERG(p_1, p_2): 函数过程. 将 p_1 为首的链连接到以 p_2 为首的链的链尾. p_2 作为合并后的链的链首. 若 p_2=NIL, 合并后, p_2 指向 p_1 的链首.
5. BACKPATCH(p, t): 把 p 为首的链所链接的四元式的第四区段都填为 t.

六. 语义子程序

(1) E→true
{E.TC := NXQ; E.FC := NIL;
GEN(j, －, －, 0) }

(2) E→false
{E.TC := NIL; E.FC := NXQ;
GEN(j, －, －, 0) }

(3) E→i
{E.TC := NXQ; E.FC := NXQ+1;
GEN(jnz, ENTRY(i), －, 0);
GEN(j, －, －, 0) }

(4) $E \rightarrow i^{(1)}$ rop $i^{(2)}$
{ E.TC := NXQ; E.FC := NXQ+1;
GEN(jrop, ENTRY($i^{(1)}$), ENTRY($i^{(2)}$), 0);
GEN(j, －, －, 0) }

(5) $E \rightarrow (E^{(1)})$
{E.TC := $E^{(1)}$.TC; E.FC := $E^{(1)}$.FC }

(6) $E \rightarrow \neg E^{(1)}$
{E.TC := $E^{(1)}$.FC; E.FC := $E^{(1)}$.TC }

(7) $E^A \rightarrow E^{(1)} \wedge$
{BACKPATCH($E^{(1)}$.TC, NXQ);
E^A.FC := $E^{(1)}$.FC }

(8) $E \rightarrow E^A E^{(2)}$
{E.TC := $E^{(2)}$.TC;
E.FC := MERG(E^A.FC, $E^{(2)}$.FC) }

(9) $E^O \rightarrow E^{(1)} \vee$
{BACKPATCH($E^{(1)}$.FC, NXQ);
E^O.TC := $E^{(1)}$.TC }

(10) $E \rightarrow E^O E^{(2)}$
{E.FC := $E^{(2)}$.FC;

E.TC := MERG(E^{O}.TC, $E^{(2)}$.TC) }

*如果最终需要把布尔式的值保留在某个临时单元 T 中, 则我们拓广文法:

S'→E

{T := NEWTEMP;

BACKPATCH(E.TC, NXQ);

BACKPATCH(E.FC, NXQ+2);

GEN(:=, '1' ,－, T);

L := NXQ+2;

GEN(j, －, －, L);

GEN(:=, '0' ,－, T);

S'.PLACE := T }

七. 例子

D∧(B∨┐(C < K ∨ A))

翻译成四元式序列如下:

(1) (jnz, D, －, 3)
(2) (j, －, －, 0)
(3) (jnz, B, －, 0)
(4) (j, －, －, 5)
(5) (j<, C, K, 2)
(6) (j, －, －, 7)
(7) (jnz, A, －, 5)←E.FC
(8) (j, －, －, 3)←E.TC

*计值的例子

§4.7 控制语句(flow-of-control statements)的翻译

一. 标号(label)和转移语句(goto statement)

标号定义的形式:

L : S

*当这种语句被处理之后, 标号 L 称为"定义了". 在符号表中, 标号 L 的"地址"栏将登记上语句 S 的第一个四元式的地址(编号).

如果 goto L; 是向后转移的处理:

如果 goto L; 是向前转移的处理:

(见黑板)

1. 建链
2. 符号表建立入口.

翻译到 S→label S

label→i:

当用 label→i:进行归约时,应做如下的语义动作:

1) 若 i 代表的标识符(假定为 L)不在符号表中, 则把它填入,置"类型"为"标号", "定义否"为"已", "地址"为 NXQ.
2) 若 L 已在符号表中, 但"类型"不是"标号"或"定义否"为"已",则报告出错.

3) 若 L 已在符号表中,则把标志“未”改为“已”,然后把地址栏中的链头(记为 q)取出,同时把 NXQ 填入其中, 最后执行 BACKPATCH(q, NXQ).

二. 条件语句(控制语句)

(1) S→if E then S
(2) | if E then S else S
(3) | while E do S
(4) | begin L end
(5) | A
(6) L→L; S
(7) | S

三. 条件语句的代码结构

1. if 语句: if E then S_1 和 if E then S_1 else S_2
   代码结构(见书 P401)

2. while 语句: while E do S_1
   代码结构(见书 P401)

3. 语句嵌套
   设立语义变量 S.CHAIN, L.CHAIN
   例如: if E_1 then if E_2 then S_1 else S_2 else S_3
   代码结构(见黑板)

四. 条件语句的语义动作

S→$CS^{(1)}$
{S.CHAIN := MERG(C.CHAIN, $S^{(1)}$.CHAIN) }
S→$T^PS^{(2)}$
{S.CHAIN := MERG(T^P.CHAIN, $S^{(2)}$.CHAIN) }
S→$W^dS^{(1)}$
{BACKPATCH($S^{(1)}$.CHAIN, W^d.QUAD);
GEN(j, －, －, W^d.QUAD);
S.CHAIN := W^d.CHAIN }
S→begin L end
{S.CHAIN := L.CHAIN }
S→A
{S.CHAIN := 0 /*空链*/ }
L→$L^SS^{(1)}$
{L.CHAIN := $S^{(1)}$.CHAIN }
L→S
{ L.CHAIN := S.CHAIN }
C→if E then
{BACKPATCH(E.TC, NXQ);
C.CHAIN := E.FC }
T^P→C $S^{(1)}$ else

{q := NXQ;
GEN(j, －,－,0);
BACKPATCH(C.CHAIN, NXQ);
T^P.CHAIN := MERG($S^{(1)}$.CHAIN, q) }
W→while
{W.QUAD := NXQ }
W^d→W E do
{BACKPATCH(E.TC, NXQ);
W^d.CHAIN := E.FC;
W^d.QUAD := W.QUAD }
L^S→L;
{BACKPATCH(L.CHAIN, NXQ) }

五. 例子

while A < B do
if C < D then X := Y + Z
的四元式代码如下:
(1) (j<, A, B, 3)
(2) (j, －, －, 0)←S.CHAIN
(3) (j<, C, D, 5)
(4) (j, －, －, 1)
(5) (+, Y, Z, T_1)
(6) (:=, T_1, －, X)
(7) (j, －, －, 1)

六. 循环语句

1. 语法
S→for i := $E^{(1)}$ to $E^{(2)}$ do $S^{(1)}$

2. PASCAL 的解释
i := $E^{(1)}$;
LIMIT := $E^{(2)}$;
goto OVER;
AGAIN: i := i + 1;
OVER: if i ≤ LIMIT then
begin $S^{(1)}$; goto AGAIN end;

3. 语义动作
F_1→for i := $E^{(1)}$
{GEN(:=, $E^{(1)}$.PLACE, －, ENTRY(i));
F_1.PLACE := ENTRY(i) }
F_2→F_1 to $E^{(2)}$
{LIMIT := NEWTEMP;
GEN(:=, $E^{(2)}$.PLACE, －, LIMIT);
OVER := NXQ+2;

GEN(j, －, －, OVER);
F_2.QUAD := NXQ;
GEN(+, F_1.PLACE, 1, F_1.PLACE);
q := NXQ;
GEN(j≤, F_1.PLACE, LIMIT, q+2);
F_2.CHAIN := NXQ;
GEN(j, －,－,0) }
S→F_2 do $S^{(1)}$
{BACKPATCH($S^{(1)}$.CHAIN, F_2.QUAD);
GEN(j, －,－,F_2.QUAD);
S.CHAIN := F_2.CHAIN }

4. 例子:

for I := 1 to N do M := M + I
的四元式代码如下:
(1) (:=, 1, －, I)
(2) (:=, N, －, LIMIT)
(3) (j, －, －, 5)
(4) (+, I, 1, I)
(5) (j≤, I, LIMIT, 7)
(6) (j, －, －, 0)
(7) (+, M, I, T_1)
(8) (:=, T_1, －, M)
(9) (j, －, －, 4)

阅读: 讲义和教材 6.6, 6.7 节

作业 10::

1. 把下列布尔表达式 A∨(B∧¬ (C∨D))翻译成转移语句四元式序列.
2. 用本课程所讲的方法,将下面的语句翻译成四元式序列:

```
while A < C ∧ B < D do
  if A = 1 then C := C + 1 else
    while A ≤ D do A := A + 2 .
```

七. 语句(switch or case statement)

1. 语法:

Case E of
C_1 : S_1;
C_2 : S_2;
……
C_{n-1}: S_{n-1};
otherwise: S_n
end

2. 中间代码结构

计算 E 的值放入 T 中的代码
Goto TEST

L_1: 关于 S_1 的中间代码
Goto NEXT
L_2: 关于 S_2 的中间代码
Goto NEXT
……
L_{n-1}: 关于 S_{n-1} 的中间代码
Goto NEXT
L_n: 关于 S_n 的中间代码
Goto NEXT
TEST: IF T = C_1 Goto L_1
IF T = C_2 Goto L_2
……
IF T = C_{n-1} Goto L_{n-1}
Goto L_n
NEXT:

§4.8 数组元素引用(array element reference)

一. 数组元素的地址计算(addressing array elements)

首先讨论以行为序(row-major or row by row)存放方式的数组元素地址计算法.假定数组每维的下限均为 1, 每个元素只占一个机器字, 而且, 目标机器的存储器是以字编址的.

假定 A 是一个 $d_1 \times d_2 \times \cdots \times d_n$ 的 n 维数组, 数组元素 $A[i_1, i_2, \ldots, i_n]$ 地址 D 的计算公式如下:

D = CONSPART + VARPART

CONSPART = a－C

$C = d_2d_3\ldots d_n + d_3d_4\ldots d_n + \ldots + d_n + 1$

$VARPART = i_1d_2d_3\ldots d_n + i_2d_3d_4\ldots d_n + \ldots + i_{n-1}d_n + i_n$

其中: a 是数组存储区的实际起始地址.

*举例说明以行为序排列(见下图), 设 A 是 3×3×3 的数组.

A[1,1,1]	A[2,1,1]	A[3,1,1]
A[1,1,2]	A[2,1,2]	A[3,1,2]
A[1,1,3]	A[2,1,3]	A[3,1,3]
A[1,2,1]	A[2,2,1]	A[3,2,1]
A[1,2,2]	A[2,2,2]	A[3,2,2]
A[1,2,3]	A[2,2,3]	A[3,2,3]
A[1,3,1]	A[2,3,1]	A[3,3,1]
A[1,3,2]	A[2,3,2]	A[3,3,2]
A[1,3,3]	A[2,3,3]	A[3,3,3]

*越往右下标变化越快, 越往左下标变化越慢.

addr(A[3,1,2]) = (3－1)×3×3 + (1－1)×3 + (2－1) + a = a + 19

$D = (i_1－1)d_2d_3\ldots d_n+(i_2－1)d_3d_4\ldots d_n+\ldots+(i_{n-1}－1)d_n+(i_n－1)+a = a－C + VARPART$

$C = (\cdots((d_2 + 1)d_3 + 1)d_4 \cdots + 1)d_n + 1$

$VARPART = (\cdots((i_1d_2 + i_2)d_3 + i_3)d_4 \cdots + i_{n-1})d_n + i_n$

VARPART 的算法:

VARPART := i_1;

k := 1;
WHILE k < n DO
BEGIN
VARPART := VARPART × d_{k+1} + i_{k+1} ;
k := k+1
END

*CONSPART 中的 C 只依赖于数组各维的维长 d_i，它和数组元素各维的下标 i_1, i_2, …, i_n 是无关的，它可以在处理数组说明时算出，填在符号表中.

计算数组元素的地址主要指计算它的 VARPART.

*解释教材中的计算方法

二. 数组元素引用的中间代码

*访问数组元素 A[i_1, i_2, …, i_n]，可设想为：把它的 VARPART 计算在某一变址器 T 中，用 CONSPART 作为基地址，然后以变址方式访问存储单元: CONSPART[T]

引入变址存数和变址取数的四元式:

(=[ ], T_1[T], －, X) 相当于 X := T_1[T]

([ ]=, X, －, T_1[T]) 相当于 T_1[T] := X .

三. 赋值句中数组元素的翻译

1. 语法:

A→V := E
V→i[elist] | i
elist→elist, E | E
E→E+E | (E) | V

2. 语义变量和语义过程

elist.ARRAY: 表示数组名的符号表入口.
elist.DIM: 下标式个数(数组维数)计数器.
elist.PLACE: 表示存放业已形成的 VARPART 的中间结果的单元名字.
LIMIT(ARRAY, k): 这是一个函数过程，它给出数组 ARRAY 的第 k 维上限 d_k . 其中 ARRAY 是数组名在符号表中的入口.
V.PLACE 和 V.OFFSET: 若 V 是一个简单变量名 i，则
V.PLACE 就是指此名的符号表入口，而 V.OFFSET 为 null.
若 V 是一个下标变量名，则 V.PLACE 就是指保存
CONSPART 的临时变量名的整数码，而 V.OFFSET 则指保存
VARPART 的临时变量名整数码.

3. 语义动作:

(1) A→V := E
{IF (V.OFFSET = null) THEN
GEN(:=, E.PLACE, －, V.PLACE)
ELSE
GEN([ ]=, E.PLACE,－,V.PLACE[V.OFFSET])}

(2) E→$E^{(1)}$ + $E^{(2)}$
{T := NEWTEMP;
GEN(+, $E^{(1)}$.PLACE, $E^{(2)}$.PLACE, T);
E.PLACE := T}

(3) E→($E^{(1)}$)
{E.PLACE := $E^{(1)}$.PLACE }

(4) E→V
{IF (V.OFFSET = null) THEN
E.PLACE := V.PLACE
ELSE
BEGIN T := NEWTEMP;
GEN(=[ ], V.PLACE[V.OFFSET], －, T);
E.PLACE := T
END }

(5) V→ elist]
{T := NEWTEMP;
GEN(－, elist.ARRAY, C, T);
V.PLACE := T;
V.OFFSET := elist.PLACE}

(6) V→i
{V.PLACE := ENTRY(i);
V.OFFSET := null }

(7) elist→$elist^{(1)}$, E
{T := NEWTEMP;
k := $elist^{(1)}$.DIM + 1;
d_k := LIMIT($elist^{(1)}$.ARRAY, k);
GEN(*, $elist^{(1)}$.PLACE, d_k, T);
GEN(+, T, E.PLACE, T);
elist.ARRAY := $elist^{(1)}$.ARRAY;
elist.PLACE := T;
elist.DIM := k}

(8) elist→i[E
{elist.PLACE := E.PLACE;
elist.DIM := 1;
elist.ARRAY := ENTRY(i) }

4. 例子
设 A 是 10×20 的数组, B 是 20×10×30 的数组, 翻译
A[I, J+1] := B[I+1, J+2, k*3] + M . 四元式序列如下:

(1) (+, J, 1, T_1) (9) (*, K, 3, T_7)
(2) (*, I, 20, T_2) (10) (*, T_6, 30, T_8)
(3) (+, T_2, T_1, T_2) (11) (+, T_8, T_7, T_8)
(4) (－, A, 21, T_3) (12) (－, B, 331, T_9)
(5) (+, I, 1, T_4) (13) (=[], T_9[T_8],－,T_{10})
(6) (+, J, 2, T_5) (14) (+, T_{10}, M, T_{11})
(7) (*, T_4, 10, T_6) (15) ([]=,T_{11},－,T_3[T_2])
(8) (+, T_6, T_5, T_6)

四. 按列为序(column-major)存放数组元素的情形

1. 地址计算

设 A 是 $d_1 \times d_2 \times \ldots \times d_n$ 数组,计算 A[i_1, i_2, …, i_n]的地址:

D = CONSPART + VARPART

CONSPART = a－C

$C = 1 + d_1 + d_1d_2 + \cdots + d_1d_2\ldots d_{n-1}$

$VARPART = i_1 + i_2d_1 + i_3d_1d_2 + \cdots + i_nd_1d_2\ldots d_{n-1}$

$= i_1+d_1(i_2+d_2(i_3+\cdots+d_{n-2}(i_{n-1} + i_nd_{n-1})\ldots))$

2. 语义动作

(1)~(4) 同前

(5) V→i[elist]

```
{IF ENTRY(i).DIM≠elist.DIM THEN ERROR ELSE
    BEGIN
        VP := Pop(elist.STACK);
        K := elist.DIM;
        WHILE K > 1 DO
        BEGIN   K := K－1;
            dk := LIMIT(ENTRY(i), K);
            TERM := Pop(elist.STACK);
            T := NEWTEMP;
            GEN(*, VP, dk, T);
            GEN(+, T, TERM, T);
            VP := T
        END;
        V.OFFSET := VP; T := NEWTEMP;
        GEN(－, ENTRY(i), C, T);
        V.PLACE := T
    END   }
```

(6)同前

(7)elist→elist$^{(1)}$, E

{elist.DIM := elist$^{(1)}$.DIM + 1;

把 E.PLACE 推进栈 elist$^{(1)}$.STACK;

elist.STACK := elist$^{(1)}$.STACK }

(8)elist→E

{elist.DIM := 1;
建立一个关于 elist 的空栈 STACK;
把 E.PLACE 推进栈 elist.STACK }

§4.9 过程调用

一. 参数传递方式(parameter-passing method)

我们假定参数传递方式是“传地址”(call by reference)方式. 如果实参是一个变量或数组元素, 那末就直接传递它的地址.如果实参是其它表达式, 那么,就先把它的值计算出来,并存放在某个临时单元 T 中,然后传递 T 的地址.

相应的每个形式参数都有一个单元,用来存放相应的实在参数的地址.

二. 转子指令的中间代码形式

例: CALL S(A+B, Z)
中间代码:
计算 A+B 置于 T 中的代码
par T
par Z
call S
*解释转子指令

三. 过程调用的四元式产生

语义动作:
(1) S→call i(arglist)
{FOR 队列 arglist.QUEUE 的每一项 p DO
GEN(par, —, — , p);
GEN(call, —, —, ENTRY(i)) }
(2) arglist→arglist$^{(1)}$, E
{把 E.PLACE 排在队列 arglist$^{(1)}$.QUEUE 的末端;
arglist.QUEUE := arglist$^{(1)}$.QUEUE }
(3) arglist→E
{建立一个队列 arglist.QUEUE, 它只包含一项 E.PLACE}

阅读: 讲义和教材 6.4.3, 6.4.4, 6.8, 6.9 节

作业 11::
1. 假设 A 是 10×20 的数组, B 是 20×10×30 的数组,且数组元素以行为序存放, 将以下赋值语句翻译成四元式.
(1) A[I,J] := A[I+1, J+1] + B[I+2, J+2, 2*K]
(2) X := A[I+1, B[I, J, K]]

§4.10 自上而下分析制导翻译概说

一. 语法:

(1) S→if E then S
(2) | if E then S else S
(3) | while E do S
(4) | begin L end
(5) | A
(6) L→L; S
(7) | S

为了无回溯的递归下降分析，把产生式(1),(2)改为

S→if E then S TAIL
TAIL→else S | ε

把(6),(7)改为右递归

L→S; L | S

提左因子

L→S LTAIL
LTAIL→; L | ε

二. 递归下降分析的语义动作:

```
PROCEDURE S;
BEGIN SYM := 下一输入记号;
  CASE SYM OF
  "if": BEGIN
        (E.TC, E.FC) := E;
        BACKPATCH(E.TC, NXQ);
        IF 下一输入记号为"then" THEN
          S(1).CHAIN := S
        ELSE ERROR;
        IF 下一输入记号为"else" THEN
        BEGIN
          q := NXQ;
          GEN(j, －,－, 0);
          BACKPATCH(E.FC, NXQ);
          S(1).CHAIN := MERG(S(1).CHAIN, q);
          S(2).CHAIN := S;
          q := MERG(S(1).CHAIN, S(2).CHAIN);
          RETURN(q);
        END
        ELSE
          RETURN(MERG(E.FC, S(1).CHAIN)
        END;
  "while": BEGIN
                QUAD := NXQ;
```

```
                    (E.TC, E.FC) := E;
                    BACKPATCH(E.TC, NXQ);
                    IF 下一输入记号为“do”THEN
                    BEGIN
                      S.CHAIN := S;
                      BACKPATCH(S.CHAIN, QUAD);
                      GEN(j, －,－, QUAD);
                      RETURN(E.FC)
                    END
                    ELSE ERROR
                    END;
  “begin”: BEGIN
                    L.CHAIN := L;
                    RETURN(L.CHAIN);
                    IF SYM ≠ “end”THEN ERROR
                    ELSE SYM := 下一输入记号;
                    END;
        “i”: BEGIN
              A;
              RETURN(0);
              END
      OTHERWISE: ERROR
      END OF CASE;
      END OF S;
```

三. 表达式的递归下降分析

1. 语法:

   E→T{+T}
   T→F{*F}
   F→ i | (E)

2. 语义动作

```
PROCEDURE   F;
      IF SYM = “i”THEN
      BEGIN ADVANCE; RETURN(ENTRY(i)) END
      ELSE
      IF SYM =“(”THEN
      BEGIN   ADVANCE;
            PLACE := E;
            IF SYM =“)”THEN
            BEGIN ADVANCE; RETURN(PLACE) END
            ELSE ERROR
      END
      ELSE ERROR;
```

```
PROCEDURE    T;
BEGIN
    T(1).PLACE := F;
    WHILE SYM = "*" DO
    BEGIN    ADVANCE;
        T(2).PLACE := F;
        T1 := NEWTEMP;
        GEN(*, T(1).PLACE, T(2).PLACE, T1);
        T(1).PLACE := T1;
    END;
    RETURN(T(1).PLACE);
END OF T;
PROCEDURE    E;
BEGIN
    E(1).PLACE := T;
    WHILE SYM = "+" DO
    BEGIN    ADVANCE;
        E(2).PLACE := T;
        T1 := NEWTEMP;
        GEN(+, E(1).PLACE, E(2).PLACE, T1);
        E(1).PLACE := T1
    END;
    RETURN(E(1).PLACE);
END OF E;
```

§4.11 说明的翻译(Translation of Declarations)

一. 翻译说明的任务

将被说明的名字及其属性记入符号表.翻译时不产生中间代码.

二. 过程中的说明

1. 语义变量和语义过程

offset: 变量名在内存中的相对地址, offset 在处理第一个说明语句之前置为 0.

enter(name, type, offset): 为 name 建立符号表条目, 它的类型是 type, 它在数据区的相对地址是 offset.

type: 表示非终结符的类型.

width: 表示非终结符的宽度(该类型的对象所需的存储单元数).

2. 语义动作:

```
P→MD
M→ ε      {offset := 0}
D→D; D
D→id : T    {enter(id.name, T.type, offset);
              offset := offset + T.width }
T→integer    {T.type := integer;
              T.width := 4 }
```

T→real {T.type := real;
T.width := 8 }
T→array [num] 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 }

三. 作用域信息的保存

1. 语法:

P→D
D→D; D | id : T | proc id; D; S

2. 过程说明的处理

当看见过程说明 D→proc id; D_1; S 时，建立新的符号表, D_1 说明的变量在该新符号表中建立条目.新的表回指直接外围过程的符号表，由 id 代表的过程名是局部于该外围过程的.

3. 符号表的例子

(见黑板)

4. 语义变量和语义过程

(1) mktable(previous): 建立新的符号表，并返回新符号表指针. 变元 previous 指向先前建立的符号表，即直接外围过程的符号表.
(2) enter(table, name, type, offset): 在 table 指向的符号表中为名字 name 建立新条目.
(3) addwidth(table, width): 把符号表 table 所有条目的累加宽度记录在该符号表的首部.
(4) enterproc(table, name, newtable): 为过程名 name 在 table 指向的符号表建立新条目.变元 newtable 指向这个过程 name 本身的符号表.
(5) tblptr: 栈 tblptr 用来保存指向所有外围过程符号表的指针.

四. 语义动作:

P→MD {addwidth(top(tblptr), top(offset));
pop(tblptr); pop(offset) }
M→ ε {t := mktable(nil);
push(t, tblptr); push(0, offset) }
D→D_1; D_2
D→proc id; N D_1
{ t := top(tblptr);
addwidth(t, top(offset));
pop(tblptr); pop(offset);
enterproc(top(tblptr), id.name, t) }

D→id : T {enter(top(tblptr), id.name, T.type, top(offset));
top(offset) := top(offset) + T.width }
N→ ε {t := mktable(top(tblptr));
push(t, tblptr); push(0, offset) }

§4.12 S 属性与 L 属性(S-Attributes and L-Attributes)

一. 综合属性(synthesized attributes)与继承属性(inherited attributes)

1. 定义: 在语法制导定义中，每个文法符号有一组属性，每个文法产生式 $A\rightarrow\alpha$ 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 $c_1, c_2, \ldots, c_k$ 是该产生式的文法符号的属性，并且

   (1) 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，那末 b 叫做文法符号 A 的综合属性.

   (2) 如果 b 是产生式右部某个文法符号 X 的属性，$c_1, c_2, \ldots, c_k$ 是 A 的属性或右部文法符号的属性，那么 b 叫做文法符号 X 的继承属性.

   在这两种情况下，我们都说属性 b 依赖于属性 $c_1, c_2, \ldots, c_k$.每个文法符号的综合属性集和继承属性集的交应为空.

2. 例子:

(1) 综合属性

产生式	语义规则
L→En	{print(E.val) }
$E\rightarrow E_1+T$	{E.val := E_1.val + T.val }
E→T	{E.val := T.val }
$T\rightarrow T_1*F$	{T.val := T_1.val × F.val }
T→F	{T.val := F.val }
F→(E)	{F.val := E.val }
F→num	{F.val := num.lexval }

(2) 继承属性

产生式	语义规则
D→TL	{L.in := T.type }
T→int	{T.type := integer }
T→real	{T.type := real }
$L\rightarrow L_1$, id	{L_1.in := L.in; addtype(id.entry, L.in) }
L→id	{addtype(id.entry, L.in) }

二. 综合属性与 S 属性定义

1. S 属性定义(S-Attributed Definitions):

   仅仅使用综合属性的语法制导定义(SDD)叫做 S 属性定义.

2. 例子:

   前面我们所讲的语法制导翻译的语义规则基本上都是 S 属性定义.

   例如: 3*5+4n 的注释分析树(见书 P308)

3. S 属性定义与自下而上翻译

   如果我们的语法规则可以写成 S 属性定义，则该语言可以方便地用自下而上分析途径翻译.

   分析器可以保存与栈中文法符号有关的综合属性，每当进行归约时，新的属性值就由栈中正在归约的产生式右部符号的属性值计算出来.

阅读: 讲义和教材 6.3, 5.1 节.

三. L 属性定义

1. L 属性定义(L-Attributed Definition):

语法制导定义是 L 属性的, 如果每个产生式 $A \rightarrow X_1X_2\cdots X_n$ 的每条语义规则中每个属性都是综合属性, 或是 X_j 的继承属性,$1 \leqslant j \leqslant n$, 它仅依赖

(1) 该产生式中 X_j 左边符号 $X_1, X_2, \ldots, X_{j-1}$ 的属性;

(2) A 的继承属性; 和

(3) X_j 的其它综合属性或继承属性,并且这些属性的依赖关系不构成圈.

*显然, S-属性定义都是 L-属性定义.

2. 非 L-属性定义的例子:

产生式	语义规则
A→LM	{L.i := l(A.i); M.i := m(L.s); A.s := f(M.s) }
A→QR	{R.i := r(A.i); Q.i := q(R.s); A.s := f(Q.s) }

Q.i := q(R.s) 不合定义.

3. L-属性定义的例子:

产生式	语义规则
S→B	{B.ps := 10; S.ht := B.ht; S.dp := B.dp }
$B \rightarrow B_1B_2$	{B_1.ps := B.ps; B_2.ps := B.ps; B.ht := max(B_1.ht, B_2.ht); B.dp := max(B_1.dp, B_2.dp) }
$B \rightarrow B_1 sub B_2$	{B_1.ps := B.ps; B_2.ps := 0.7×B.ps; B.ht := max(B_1.ht, B_2.ht−0.25×B.ps); B.dp := max(B_1.dp, B_2.dp+0.25×B.ps) }
$B \rightarrow (B_1)$	{B_1.ps := B.ps; B.ht := B_1.ht; B.dp := B_1.dp }
B→text	{B.ht := getHt(B.ps, text.lexval); B.dp := getDp(B.ps, text.lexval) }

*这是数学排版语言 EQN 的语义动作.

继承属性 ps(点的大小)影响公式的高度.产生式 B→text 的规则用正文的正常高度乘以点的大小来得到正文的实际高度. text 的正常高度根据属性 text.lexval 查表得到. 当使用产生式 $B \rightarrow B_1B_2$ 时, 通过复写规则, B_1 和 B_2 继承了 B 的点的大小. B 的高度由综合属性 ht 表示, 它取 B_1 和 B_2 高度的极大值. B 的深度由综合属性 dp 表示, 它的计算与 ht 类似.

4. 语法制导翻译方案(Syntax-Directed Translation Schemes)

语法制导翻译方案(SDT's)是上下文无关文法以及插入在产生式右部的一些程序段. 翻译方案和语法制导定义不同的是它的语义动作放在括号{ }内, 且可以插在产生式右部的任何地方.

例: 下面的翻译方案把有加法的中缀表达式映射到后缀表达式.

E→TR

R→addop T {print(addop.lexeme)}R_1 | ε

T→num {print(num.val)}

若 L-属性定义同时有继承和综合属性, 按以下规则设计翻译方案:

(1) 产生式右部符号的继承属性必须在先于这个符号的动作中计算.

(2) 一个动作不能引用该动作右边符号的综合属性

(3) 左部非终结符的综合属性只能在它所引用的所有属性都计算完后再计算. 计算该属性的动作通常放在产生式右部的末端.

例子: 翻译方案

S → {B.ps := 10} B {S.ht := B.ht; S.dp := B.dp}

B → {B_1.ps := B.ps} B_1 {B_2.ps := B.ps} B_2
{B.ht := max(B_1.ht, B_2.ht); B.dp := max(B_1.dp, B_2.dp) }

B → {B_1.ps := B.ps} B_1 sub {B_2.ps := 0.7×B.ps} B_2
{B.ht := max(B_1.ht, B_2.ht－0.25×B.ps);
B.dp := max(B_1.dp, B_2.dp+0.25×B.ps) }

B → ({B_1.ps := B.ps} B_1) {B.ht := B_1.ht; B.dp := B_1.dp}

B → text { B.ht := getHt(B.ps, text.lexval);
B.dp := getDp(B.ps, text.lexval) }

*修改文法, 在适当时机归约.

§4.13 继承属性的自下而上计算

*本节提出在自下而上分析的框架中实现 L-属性定义的方法.它能实现任何基于 LL(1)文法的 L-属性定义, 也能实现许多(但不是所有)基于 LR(1)文法的 L-属性定义.

一. 删除翻译方案中嵌入的动作

虽然 L-属性定义的继承属性计算需要嵌在产生式右部的不同地方, 但是可以通过修改文法而把翻译方案中所有的嵌入动作都变成只出现在产生式右端.

例:

E→TR

R→+T {print('+')} R | －T {print('－')} R | ε

T→num {print(num.val)}

变换成

E→TR

R→+ T M R | －T N R | ε

T→num {print(num.val)}

M→ ε {print('+')}

N→ ε {print('－')}

二. 分析栈上的继承属性

例: 给出翻译方案

D→T {L.in := T.type}
L

T→int {T.type := integer}

T→real {T.type := real }
L→ {L_1.in := L.in}
L_1, id {addtype(id.entry, L.in)}
L→id {addtype(id.entry, L.in)}
当处理 int p, q, r 时, 继承属性按以下方式传递:
(见黑板)

移进归约的过程如下: (见黑板)

*当用产生式 L→id 归约时, id.entry 在栈顶 val[top], 而 T.type 在 val[top－1].
当用产生式 L→L, id 归约时, id.entry 在栈顶 val[top], 而 L.in 恰好是 T.type 在 val[top－3].
翻译方案可以用以下代码段实现:

产生式	代码段
D→TL;	
T→int	val[ntop] := integer
T→real	val[ntop] := real
L→L, id	addtype(val[top], val[top－3])
L→id	addtype(val[top], val[top－1])

三. 模拟继承属性的计算

*上面讲的分析栈中取属性值的办法,只有在文法允许属性值的位置可预测时才行得通.

例: 一个不能预测位置的实例.

产生式	语义规则
S→aAC	C.i := A.s
S→bABC	C.i := A.s
C→c	C.s := g(C.i)

*上式第 1,2 个产生式有 C.i 的值在 val[top－1]或在 val[top－2]中, 但不能确定是哪一个.
修改文法和语义规则:

产生式	语义规则
S→aAC	C.i := A.s
S→bABMC	M.i := A.s; C.i := M.s
C→c	C.s := g(C.i)
M→ ε	M.s := M.i

*第二个产生式,先算出 M.i 来,不管是第一个产生式还是第二个产生式, 算 C.i 时,都取 val[top－1]的值.
计算属性值不是用复写规则的情形:

产生式	语义规则
S→aAC	C.i := f(A.s)

可把产生式改写为

产生式	语义规则
S→aANC	N.i := A.s; C.i := N.s
N→ ε	N.s := f(N.i)

*N.i 可在 val[top－1]找到. 计算 N.s 时,可将 val[top－1]中的内容作 f 计算放在 val[top]中. C.i 的值在 val[top－1]中.
最后看一下数学排版语言 EQN 的例子.

产生式	语义规则
S→L B	B.ps := L.s; S.ht := B.ht; S.dp := B.dp
L→ ε	L.s := 10
$B→B_1$ M B_2	B_1.ps := B.ps; M.i := B.ps;
	B_2.ps := M.s;
	B.ht := max(B_1.ht, B_2.ht);
	B.dp := max(B_1.dp, B_2.dp)
M→ ε	M.s := M.i
$B→B_1$ sub N B_2	B_1.ps := B.ps; N.i := B.ps;
	B_2.ps := N.s;
	B.ht := max(B_1.ht, B_2.ht－0.25×B.ps);
	B.dp := max(B_1.dp, B_2.dp+0.25×B.ps)
N→ ε	N.s := 0.7×N.i
B→text	B.ht := getHt(B.ps, text.lexval);
	B.dp := getDp(B.ps, text.lexval)

用以下代码段实现

产生式	代码段
S→L B	ht[ntop] := ht[top]; dp[ntop] := dp[top]
L→ ε	ht[ntop] := 10; dp[ntop] := 10
$B→B_1$ M B_2	ht[ntop] := max(ht[top－2], ht[top]);
	dp[ntop] := max(dp[top－2], dp[top])
M→ ε	ht[ntop] := ht[top－2];
	dp[ntop] := dp[top－2]
$B→B_1$ sub N B_2	ht[ntop] := max(ht[top－3], ht[top]－0.25×ht[top－4]);
	dp[ntop] := max(dp[top－3], dp[top]+0.25×dp[top－4])
N→ ε	ht[ntop] := 0.7×ht[top－3];
	dp[ntop] := 0.7×dp[top－3]
B→text	ht[ntop] := getHt(ht[top－1], ht[top]);
	dp[ntop] := getDp(dp[top－1], dp[top])

*第一次用 B→text 归约时, ht[top－1]和 dp[top－1]中是 L.s 即 B.ps,而 ht[top]和 dp[top]中为 text.lexval.

以后每次用第 3 和第 5 个产生式时,假设其中 B_1(或 B_2)在 ht[top]和 dp[top]中, 则 ht[top－1]和 dp[top－1]中是 B_1.ps(或 B_2.ps).

阅读: 讲义和教材 5.2, 5.3, 5.4, 5.5 节.

第五章 运行时存储空间组织

*编译程序必须分配目标程序运行时的数据空间. 程序语言关于名字的作用域和生存期的定义规则决定了分配目标程序数据空间的基本策略.

运行时存储分配策略分为静态分配策略和动态分配策略两种.动态分配策略又分为栈式动态分配策略和堆式动态分配策略.

§5.1 静态存储管理—FORTRAN 存储分配

一. FORTRAN 语言的特点:

1. 不容许过程的递归性
2. 每个数据名所需的存储空间大小都是常量
3. 所有数据名的性质是完全确定的
4. 整个程序所需数据空间的总量在编译时完全确定

*不考虑 FORTRAN 过程哑元的可调数组

二. 数据区

1. 数据区的设置

对于每个程序段和公用块都定义一个对应的数据区，每个数据区有一个编号.地址分配时,在符号表中，对每个数据名将登记上它是属于哪一个数据区的,以及在该区中的相对位置.

程序段的局部区可直接安排在该段的指令代码和常数单元之后.具名公用区和无名公用区安排在目标程序的最后端.

编译程序必须累计每个数据区的体积.

2. 数据区的内容

局部数据区一般含有下列诸项: (见黑板)

临时变量 数　　组 简单变量
形式单元
寄存器保护区
返回地址

*解释各项内容.

§5.2 一个简单栈式存储分配的实现

一. 适用的语言:

假设这种语言没有分程序结构，过程定义不允许嵌套，但允许过程的递归调用，允许过程含可变数组.

例如: C 语言

例子:

```
program    MAIN;
   全局变量或数组的说明
   proc   R;
   ……..
   end R;
   proc   Q;
   ……...
   end Q;
主程序执行语句
end   MAIN;
```

二. 活动记录:

使用栈式存储分配, 运行时每当进入一个过程就有一个相应的活动记录累筑于栈顶. 此记录含有连接数据、形式单 元、局部变量、局部数组的内情向量和临时工作单元等等. 在进入过程后执行过程的可执行语句之前, 再把局部数组所需空间累筑于栈顶.

当一个过程工作完毕返回时, 它在栈顶的数据区也随即不复存在.

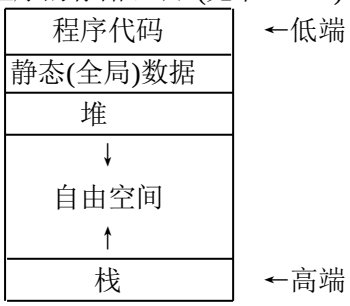
三. 活动记录的内容

1. SP 指针与 TOP 指针

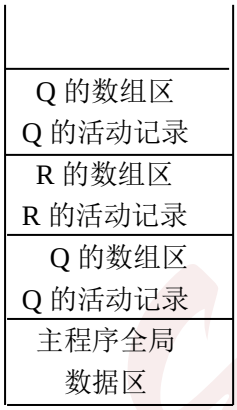
SP 总是指向现行活动记录的起点, TOP 则始终指向(已占用的)栈顶单元.

2. 栈的组织

C++程序的存储组织: (见书 P428)



*(按我们的讲法)栈的结构: (下面是低端,上面是高端)



3. C 的活动记录内容

1) 连接数据, 有两个
   a) 老 SP 值,即前一活动记录的地址
   b) 返回地址
2) 参数个数
3) 形式单元(存放实在参数的值或地址)
4) 过程的局部变量,数组的内情向量和临时工作单元

*在过程段中,对任何局部变量 X 的引用可表示为变址访问 X[SP], 此处 X 代表变量 X 的相对地址.

三. C 的过程调用,过程进入,数组空间分配和过程返回

1. 过程调用的四元式序列:

```
par   T1
par   T2
⋮
par   Tn
call  P, n
```

2. 转子之前的工作:

```
par   Ti 的工作
(i+3)[TOP] := Ti          (传递参数的值)
或 (i+3)[TOP] := addr(Ti)  (传递参数的地址)
call  P, n                翻译成
1[TOP] := SP              (保护现行 SP)
3[TOP] := n               (传送参数个数)
JSR   P                   (转子指令,转向 P 的第一条指令)
```

3. 进入过程 P 后的工作:

```
SP := TOP+1               (定义新 SP)
1[SP] := 返回地址          (保存返回地址)
TOP := TOP + L            (定义新 TOP)
```

4. 对动态数组的处理

*所有数组空间都分配在活动记录的顶上.紧接上述指令之后应是对数组进行存储分配的指令.

对于每个数组说明,相应的目标指令组将做以下几件工作:

(1) 计算各维的上、下限

(2) 调用数组空间分配子程序, 其参数是:

a) 各维的上、下限

b) 内情向量单元的首地址

*数组空间分配子程序计算并填好内情向量的所有信息, 然后在 TOP 所指的位置之上留出数组所需的空间, 并将 TOP 调整为指向数组区的顶端.

5. 过程返回时所做的工作

```
return(E)
```

将 E 值计算出来,放在特定寄存器中

```
TOP := SP－1      (恢复调用过程的 TOP 指针)
SP := 0[SP]       (恢复调用过程的 SP 指针)
x := 2[TOP]       (x 为返回地址)
UJ   0[x]         (返回到 x 执行)
```

§5.3 嵌套过程语言的栈式实现

一. 过程嵌套

容许过程嵌套定义, 但仍假定不含分程序. 例如: Pascal 语言.

* 但 Pascal 中含有文件和指针数据类型带来了复杂性. 这里假设语言中不含有这些数据类型.

过程的嵌套层次. 主程序是第 0 层, 如果过程 Q 是在层数为 i 的过程 P 内定义的, 并且 P 是包围 Q 的最小过程, 那么, Q 的层数为 i+1.

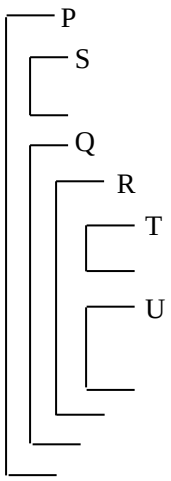
*使用计数器 level 计算嵌套层数.

过程的层数将作为过程名的一个重要属性登记在符号表中.

二. 嵌套层次显示表 DISPLAY 和活动记录

一个过程 Q 可能引用它的任一外层过程 P 的最新活动记录中的某些数据, 因此, 过程 Q 运行时必须知道它的所有外层过程的最新活动记录的地址.

例:



*解释作用域的定义

1. 嵌套层次显示表 DISPLAY

嵌套层次显示表为每个过程保存它的所有外层过程的最新活动记录的 SP 值. DISPLAY 是一个小栈.

例: 上例中, 过程 R 的外层是 Q, Q 的外层是主程序 P, R 运行时的 DISPLAY 内容为:

2	R 的现行活动记录的地址
1	Q 的最新活动记录的地址
0	P 的活动记录的地址

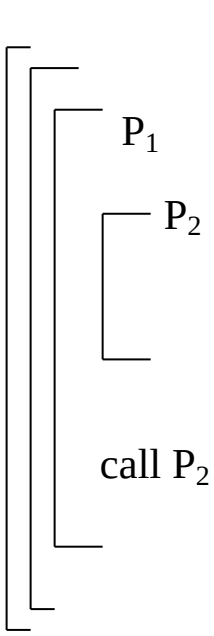
*由于过程的嵌套层数可静态确定, 因此, 每个过程的 DISPLAY 表的体积在编译时即可知道.

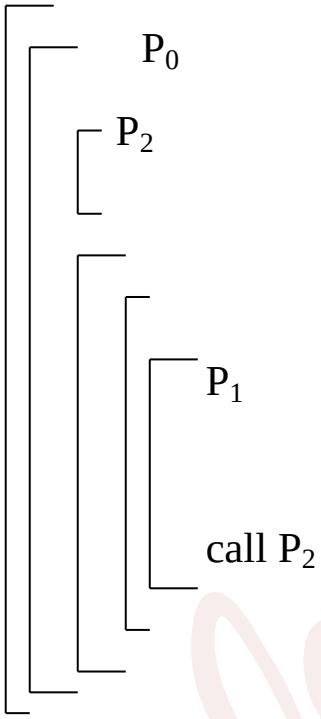
2. 活动记录的组织

TOP→		临时变量 内情向量 简单变量
	d	DISPLAY
		形式单元
	3	参数个数
	2	全局 DISPLAY
	1	返回地址
SP→	0	老 SP

3. 为被调用过程构造 DISPLAY 表

1) 两种过程调用情形





2) 确定 P_2 过程的直接外层过程 P_0

3) 传递 P_1 的 DISPLAY 地址作为全局 DISPLAY

4) 构造 P_2 的 DISPLAY 表.

设 l_2 为 P_2 的层数，只须从 P_1 的 DISPLAY 中自底向上地取 l_2 个单元，再在顶端加上进入 P_2 后新建立的 SP 值，就构成了 P_2 的 DISPLAY.

三. 过程调用、过程进入

```
par   Ti                翻译成
(i+4)[TOP] := Ti        或者
(i+4)[TOP] := addr(Ti)
call   P, n             翻译成
1[TOP] := SP;
3[TOP] := SP + d        /*现行 DISPLAY 地址*/
4[TOP] := n
JSR   P
```

当转入过程 P 并定义了新的 SP 和 TOP，以及保护了返回地址之后，应按第三项连接数据所提供的全局 DISPLAY 地址，自底向上抄录 l 个单元的内容(l 为 P 的层数)，再加上新的 SP 值，以形成现行过程的 DISPLAY(l+1 个单元).

```
return( E )     翻译成
TOP := SP－1
SP := 0[SP]
X := 2[TOP]
UJ   0[X]
```

四. 参数传递

1. par T, T 为数组

在这种情况下，根据不同语言的要求，或者传送数组 T 的首地址，或者传送它的内情向量地址.

2. par T, T 为过程

过程 P 把过程 T 作为实在参数传递给过程 Q，随后，Q 又通过引用相应的形参调用 T. 进入 T 之后，为建立 T 的 DISPLAY，需要知道 T 的直接外层的 DISPLAY.

T 的 DISPLAY 乃是由 P 的 DISPLAY 的前 l 个单元(l 为 T 的层数)的内容和 SP 的现行值所组成.

par T 可刻划为：建立两个相继的临时单元，

第一个临时单元 B_1：过程 T 的入口地址

第二个临时单元 B_2：现行的 DISPLAY 地址.

(i+4)[TOP] := addr(B_1)

过程 Q 执行转子指令时，把 B_2 的值作为全局 DISPLAY 传递给 T 过程.

3. par T, T 为标号

过程 P 把标号 T 作为实参传递给过程 Q，随后,Q 又通过引用相应的形式参数把控制转移到标号 T 所指的地方.

设 T 是过程 P_0 定义的(P_0 是 P 自身或 P 的某一外层)，那么，当 Q 要转移到 T 时，必须首先把 P_0 的活动记录变成现行活动记录.

par T 需要传两项内容:

B_1：标号 T 的地址

B_2: P_0 的活动记录地址.

在过程 Q 的某一时刻执行

goto Z (Z 是对应 T 的形参)

逐级恢复 SP 和 TOP，直至 SP 指向 P_0 的活动记录.

阅读: 讲义和教材 7.1, 7.2, 7.3 节

第七章 代码优化

一. 什么是代码优化?

对程序进行各种等价变换，使得从变换后的程序出发，能生成更有效的目标代码.

所谓等价，是指不改变程序的运行结果；所谓有效主要指目标代码运行的时间较短，以及占用的存储空间较小.

二. 代码优化的时机

1. 在目标代码生成以前，对语法分析后的中间代码进行;
2. 在生成目标代码时进行.

§7.1 优化概述

本节我们通过例子，说明对中间代码可能进行的优化.

例子:

```
PROD := 0;
For I := 1 to 20 do
    PROD := PROD + A[I]*B[I]
```

(中间代码见图 1)

*这个中间代码程序是在第四章算法产生的中间代码基础上略经优化而得到的.

一. 删除多余运算(删除公共子表达式)

(3)和(6)条

二. 代码外提

(4)和(7)条 (优化后见图 2)

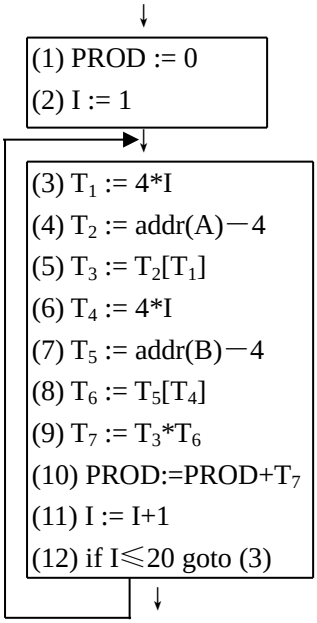


图 1

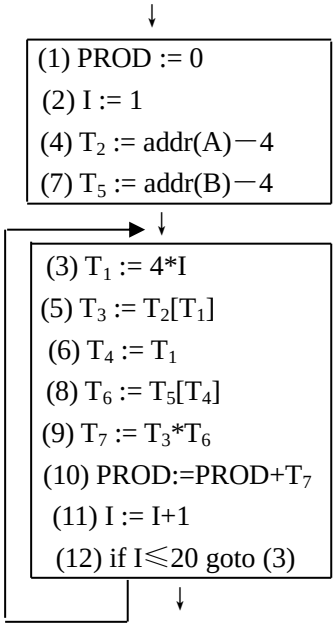


图 2

三. 强度削弱

(3)式 (优化后见图 3)

四. 变换循环控制条件

(12)式

五. 合并已知量

(3)式

六. 复写传播

(6)式和(8)式 (优化后见图 4)

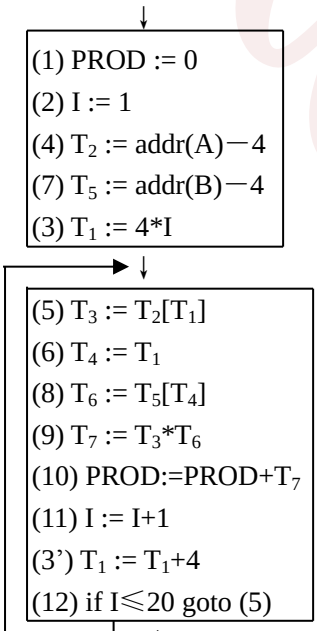


图 3

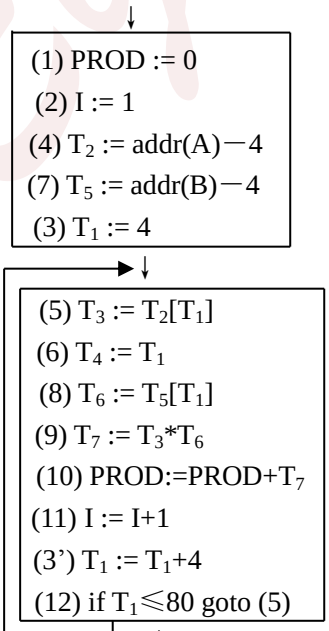


图 4

七. 删除无用赋值

(6)、(2)、(11)式 (优化后见图 5)

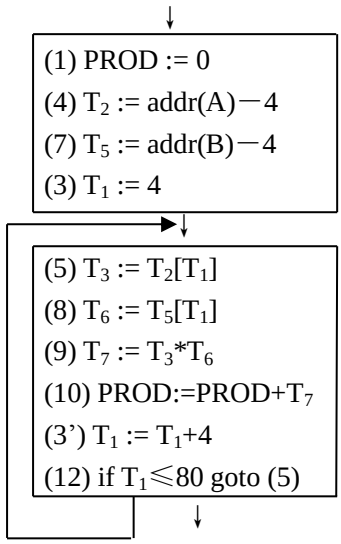


图 5

八. 优化的种类

1. 局部优化
2. 循环优化
3. 全局优化

§7.2 局部优化

一. 基本块

是指程序中一顺序执行的语句序列, 其中只有一个入口和一个出口, 入口就是其中第一个语句, 出口就是其中最后一个语句. 对一个基本块来说, 执行时只能从其入口进入, 从其出口退出.

二. 局部优化的概念

局限于基本块范围的优化称为局部优化.

三. 基本块的划分

1. 算法:

a) 求出四元式程序中各个基本块的入口语句, 它们是:
   (1) 程序的第一个语句; 或者
   (2) 能由条件转移和无条件转移语句转移到的语句; 或者
   (3) 紧跟在条件转移语句后面的语句.

b) 对以上求出的每一个入口语句, 构造其所属的基本块. 它是由该入口语句到下一入口语句(不包括下一入口语句), 或到一转移语句(包括该转移语句), 或到一停语句(包括该停语句)之间的语句序列组成的.

c) 凡未被纳入任何基本块中的语句, 都是程序中控制流无法到达的语句, 从而也是不会被执行到的语句, 我们可以把它们从程序中删除.

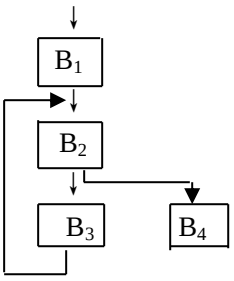
2. 例子:

(1) read X
(2) read Y　　B_1

(3) R := X MOD Y
(4) if R = 0 goto (8)　　B_2

(5) X := Y
(6) Y := R　　B_3
(7) goto (3)

(8) write Y
(9) halt　　B_4



流图

四. 基本块内的优化

1. 合并已知量
2. 删除无用赋值
3. 删除多余运算

五. 几种无用赋值

1. 对某变量 A 赋值后，该 A 值在程序中不被引用.
2. 对某变量 A 赋值后，在该 A 值被引用前，又对 A 重新赋值.
3. 对某变量 A 进行递归赋值，如 A := A+C，且该 A 值在程序中仅在此递归运算中被引用.

*在基本块中可对第二种无用赋值进行优化.

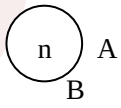
§7.3 基本块的 DAG 表示及其应用

一. DAG

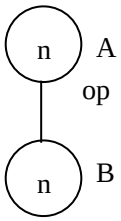
有向图、前驱、后继，通路、环路
无环路有向图称为 DAG.
祖先、后代
例子: (见黑板)

二. 基本块的 DAG 表示

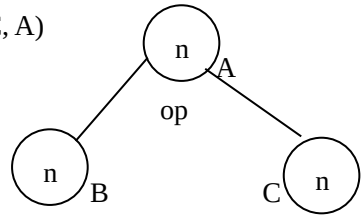
(0) A := B　　(:=, B, —, A)



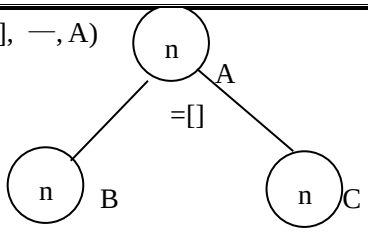
(1) A := op B　　(op, B, —, A)



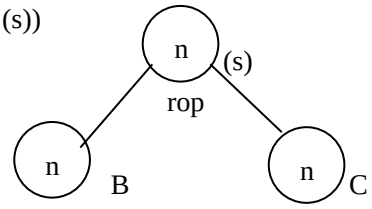
(2) A := B op C　　(op, B, C, A)



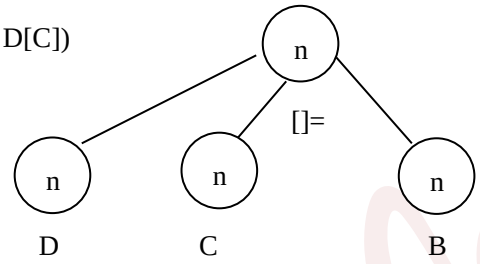
(3) A := B[C]　　(=[ ], B[C], —, A)



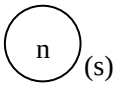
(4) if B rop C goto (s)　(j$_{rop}$, B, C, (s))



(5) D[C] := B　([ ]=, B, —, D[C])



(4)　　goto (s)　(j,—, —, (s))



三.　构造基本块的 DAG 的算法

设后继结点个数为 n 的称为 n 型四元式. 设 A 为一变量名, NODE(A)或者是一个结点的编号, 或者无定义. 前一种情况代表 DAG 中存在一个结点 n, A 是其上的标记或附加标识符.

开始, DAG 为空.

对基本块中每一四元式依次执行以下步骤:

1. 如果 NODE(B)无定义, 则构造一标记为 B 的叶结点, 并定义 NODE(B)为这个结点;
   如果当前四元式是 0 型, 则记 NODE(B)的值为 n, 转 4.
   如果当前四元式是 1 型, 则转 2(1).
   如果当前四元式是 2 型, 则: (i) 如果 NODE(C)无定义, 则构造一标记为 C 的叶结点, 并定义 NODE(C)为这个结点;
   (ii) 转 2(2).

2. (1) 如果 NODE(B)是标记为常数的叶结点, 则转 2(3), 否则转 3(1).
   (2) 如果 NODE(B)和 NODE(C)都是标记为常数的叶结点, 则转 2(4), 否则转 3(2).
   (3) 执行 op B , 令得到的新常数为 p. 如果 NODE(B)是处理当前四元式时新构造出来的结点, 则删除它. 如果 NODE(p)无定义, 则构造一用 p 做标记的叶结点 n. 置 NODE(p) = n,转 4 .
   (4) 执行 B op C, 令得到的新常数为 p. 如果 NODE(B)或 NODE(C)是处理当前四元式时新构造出来的结点, 则删除它. 如果 NODE(p)无定义, 则构造一用 p 做标记的叶结点 n. 置 NODE(p) = n, 转 4.

3. (1) 检查 DAG 中是否已有一结点, 其唯一后继为 NODE(B)且标记为 op. 如果没有, 则构造该结点 n, 否则就把已有的结点作为它的结点, 并设该结点为 n, 转 4.
   (2) 检查 DAG 中是否已有一结点, 其左后继为 NODE(B), 右后继为 NODE(C), 且标记为 op. 如果没有, 则构造该结点 n, 否则就把已有的结点作为它的结点并设该结点为 n. 转 4.

4. 如果 NODE(A)无定义，则把 A 附加在结点 n 上，并令 NODE(A) = n；否则先把 A 从 NODE(A)结点上的附加标识符集中删除(注意，如果 NODE(A)是叶结点，则其标记 A 不删除)，把 A 附加到新结点 n 上，并令 NODE(A) = n. 转处理下一四元式.

四. 例子

(1) $T_0 := 3.14$

(2) $T_1 := 2*T_0$

(3) $T_2 := R + r$

(4) $A := T_1*T_2$

(5) $B := A$

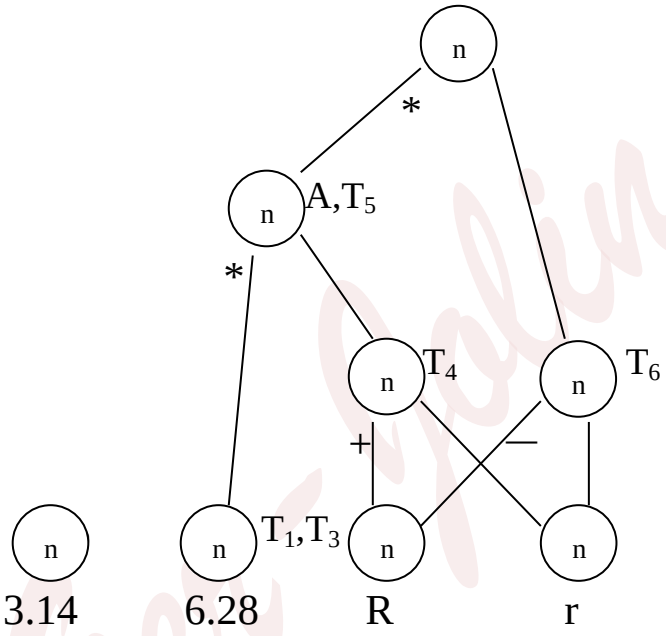
(6) $T_3 := 2*T_0$

(7) $T_4 := R+r$

(8) $T_5 := T_3*T_4$

(9) $T_6 := R-r$

(10) $B := T_5*T_6$



(DAG 见旁边的图)

五. DAG 的应用

1. 从 DAG 到四元式

(1) $T_0 := 3.14$

(2) $T_1 := 6.28$

(3) $T_3 := 6.28$

(4) $T_2 := R+r$

(5) $T_4 := T_2$

(6) $A := 6.28*T_2$

(7) $T_5 := A$

(8) $T_6 := R-r$

(9) $B := A*T_6$

* 与原来的基本块比较

2. 算法的优化作用

(1) 对任何一个四元式，如果其中参与运算的对象都是编译时的已知量，那么，算法的步骤 2 并不生成计算该结点值的内部结点，而是执行该运算，用计算出的常数生成一个叶结点. 所以步骤 2 的作用是实现合并已知量.

(2) 如果某变量被赋值后，在它被引用前又被重新赋值，那么，算法的步骤 4 已把该变量从具有前一个

值的结点删除. 也即算法的步骤 4 具有删除第二种无用赋值的作用.

(3) 算法的步骤 3 的作用是检查公共子表达式. 对具有公共子表达式的所有四元式, 它只产生一个计算该表达式值的内部结点, 而把那些被赋值的变量标识符附加到该结点上.

3. 进一步的优化信息

(1) 在基本块外被定值并在基本块内被引用的所有标识符, 就是作为叶子结点上标记的那些标识符.

(2) 在基本块内被定值且该值能在基本块后面被引用的所有标识符, 就是 DAG 各结点上的那些附加标识符.

*如果某个变量在基本块后面不被引用, 并且在基本块内不引用它的值, 则不产生计算该变量的代码.

例: 在上例中, 设 $T_0, T_1, \ldots, T_6$ 在基本块后不会被引用, 则产生代码:

(1) $S_1 := R+r$

(2) $A := 6.28*S_1$

(3) $S_2 := R-r$

(4) $B := A*S_2$

六. DAG 构造算法的讨论

如果考虑数组元素的引用

例: (1) X := A[I]

(2) A[J] := Y

(3) Z := A[I]

四元式序列:

(1) $S_1 := addr(A)-1$

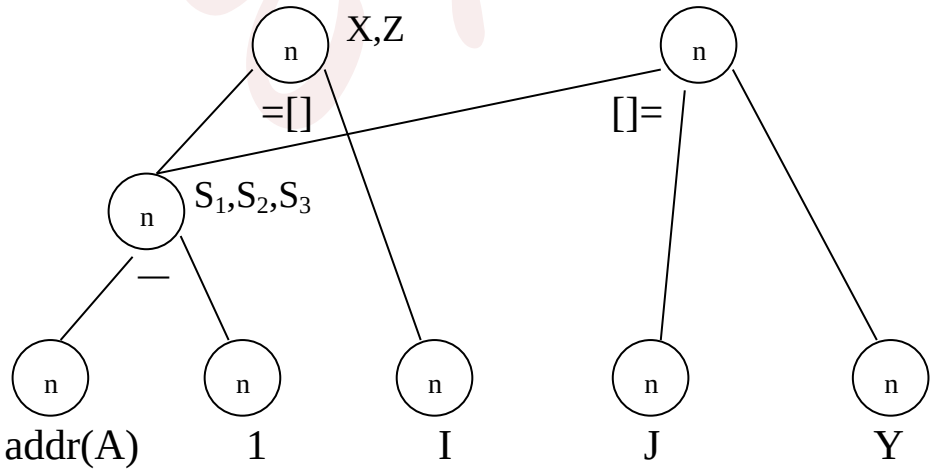
(2) $X := S_1[I]$

(3) $S_2 := addr(A)-1$

(4) $S_2[J] := Y$

(5) $S_3 := addr(A)-1$

(6) $Z := S_3[I]$



恢复四元式序列:

(1) $S_1 := addr(A)-1$

(2) $S_2 := S_1$

(3) $S_3 := S_1$

(4) X := S_1[I]

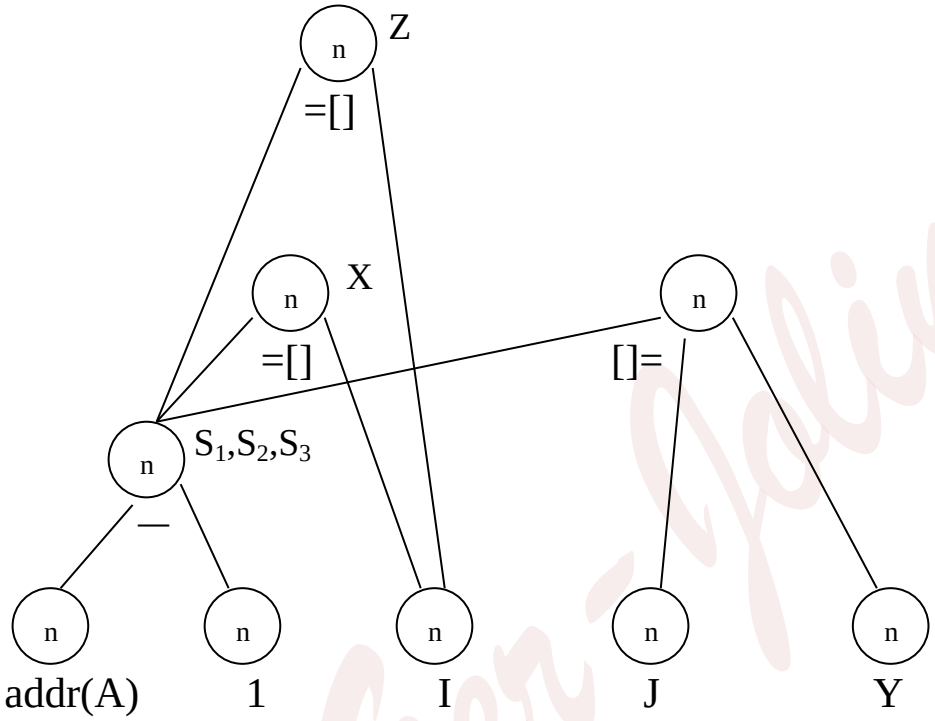
(5) Z := X

(6) S_1[J] := Y

当 I = J 时造成问题.

*解决问题的办法: 可把 DAG 中 addr(A)的祖先结点凡标记为=[]者都予以注销. 一个结点被注销, 意味着在 DAG 构造过程中, 不可以再选它作为已有的结点来代替要构造的新结点.

上例的 DAG 改为:



阅读: 讲义和教材 8.4, 8.5 节.

作业 12::

1. 试把以下程序划分成基本块, 并画出它的流图.

read C
A := 0
B := 1
L_1: A := A+B
if B≥C goto L_2
B := B+1
goto L_1
L_2: write A
halt

2. 给出基本块:

B := 3
D := A+C
E := A*C
F := D+E
G := B*F
H := A+C

I := A*C
J := H+I
K := B*5
L := K+J
M := L

应用 DAG 对其进行优化，并就以下两种情况分别写出优化后的四元式序列:
(1) 假设只有 G, L, M 在基本块后面还要被引用;
(2) 假设只有 L 在基本块后面还要被引用.

第八章 代码生成

一. 几种目标代码

1. 能够立即执行的机器代码
2. 待装配的机器语言模块
3. 汇编语言代码

二. 代码生成的三个主要任务

1. 指令的选择
2. 寄存器的分配与安排
3. 指令的排序

§7.1 一个简单的目标机器模型

一. 机器指令系统:

1. 装入指令: LD dst, addr
   dst 为寄存器, addr 为内存地址，表示 dst := addr. Addr 有时也可以是寄存器.
2. 存数指令: ST x, r
   r 为寄存器, x 为内存地址，表示 x := r .
3. 计算指令: OP dst, src1, src2
   OP 是算符，例如: ADD, SUB. src1, src2 为存放操作数的寄存器, dst 为存放运算结果的寄存器. 例如: SUB r_1, r_2, r_3 表示 $r_1 := r_2 - r_3$. 对一元算符 OP，设指令没有 src2 项.
4. 无条件转移指令:
   BR L
   转移到标号为 L 的指令去执行.
5. 条件转移指令:
   B_{cond} r, L
   r 为寄存器, L 是标号. 表示当 r 中的值满足 cond 时，转到标号 L 去执行. 例如: BLTZ r, L, 表示当 r 中的值小于零时，转到 L 去执行，否则顺序执行.

二. 目标机器允许以下的寻址方式.

1. 地址为变量名 x，表示访问该变量所对应的内存单元.
2. 变址访问 a(r): a 是变量, r 是寄存器.
   表示访问以 a 的值加上 r 中的内容为地址的内存单元.
   例如: LD R_1, a(R_2) 表示: R_1:=contents(a+contents(R_2)).
3. 直接数变址访问:

例: LD R_1, 100(R_2) 表示 R_1:=contents(100+contents(R_2)).

4. 间接寻址: *r
   表示访问以 r 的内容为地址的那个内存单元.
   例如: LD R_1, *r 表示 R_1:=contents(contents(r)).
   我们还可以使用间接加变址的寻址方式. 例如:
   LD R_1, *100(R_2)表示 R_1:=contents(contents(100+contents(R_2)))
5. 直接数访问:
   例如: LD R_1, #100 表示 R_1 := 100
   再如: ADD R_1, R_1, #100 表示 R_1 := contents(R_1)+100

三. 例子:

例 1: x := y－z 机器代码为

LD R_1, y
LD R_2, z
SUB R_1, R_1, R_2
ST x, R_1

例 2: b := a[i] (假设机器以字节为单位寻址, 一个机器字占 8 个字节), 机器代码为:

LD R_1, i
MUL R_1, R_1, #8
LD R_2, a(R_1)
ST b, R_2

例 3: a[j] := c 机器代码为

LD R_1, c
LD R_2, j
MUL R_2, R_2, #8
ST a(R_2), R_1

例 4: x := *p 机器代码为

LD R_1, p
LD R_2, 0(R_1)
ST x, R_2

例 5: *p := y 机器代码为

LD R_1, p
LD R_2, y
ST 0(R_1), R_2

例 6: if x < y goto L 机器代码为

LD R_1, x
LD R_2, y
SUB R_1, R_1, R_2
BLTZ R_1, M

其中, M 是标号 L 的地址.

§7.2 一个简单代码生成器

一. 基本原则

*在基本块中，当生成计算某变量值的目标代码时,尽可能地让该变量的值保留在寄存器中，直到该寄存器必须用来存放别的变量值或者已到达基本块出口为止．后续的目标代码尽可能地引用变量在寄存器中的值，而不访问主存.

二. 待用信息

1. 定义:

如果在一基本块中，四元式 i 对 A 定值，四元式 j 要引用 A 值，而从 i 到 j 之间没有 A 的其它定值，那么，我们称 j 是四元式 i 的变量 A 的待用信息(即下一个引用点)

2. 约定:

把基本块中所有临时变量均看作基本块出口之后的非活跃变量，而把所有非临时变量均看作基本块出口之后的活跃变量.

3. 计算待用信息的算法

1) 开始时，把基本块中各变量的符号表登记项中的待用信息栏填为“非待用”，并根据该变量在基本块出口之后是不是活跃的，把其中的活跃信息填为“活跃”或“非活跃”.
2) 从基本块出口到基本块入口自后向前依次处理各个四元式．对每一四元式 i: A := B op C 依次执行下述步骤:
   (1) 把符号表中变量 A、B 和 C 的待用信息和活跃信息附加到四元式 i 上;
   (2) 把符号表中 A 的待用信息和活跃信息分别置为“非待用”和“非活跃”;
   (3) 把符号表中 B 和 C 的待用信息均置为 i，活跃信息均置为“活跃”.

三. 寄存器描述和地址描述

1. 寄存器描述数组 RVALUE
2. 变量地址描述数组 AVALUE

四. 代码生成算法:

*函数 getReg(I): 为三地址指令 I 中的变量(内存单元)选择寄存器.

运算的机器指令

对每个形如 I: x := y+z 的三地址指令，完成下列步骤:

1) 使用 getReg(x := y+z)为 x, y, z 选择寄存器．我们把这些寄存器称为 R_x, R_y 和 R_z.
2) 如果(根据 R_y 的寄存器描述符)y 不在 R_y 中，那么生成指令: LD R_y, y'．其中 y'是存放 y 的内存单元之一(y'可以根据 y 的地址描述符得到).
3) 类似地，如果 z 不在 R_z 内，生成指令: LD R_z, z'，其中 z'是存放 z 的位置之一.
4) 生成指令: ADD R_x, R_y, R_z .

复制语句的机器指令

复制语句形如 I: x := y. 我们假设 getReg 总是为 x 和 y 选择同一寄存器．如果 y 没有在寄存器 R_y 中，那么生成机器指令: LD R_y, y. 如果 y 已经在 R_y 中，我们不需要做任何事情．我们只需要修改 R_y 的寄存器描述符，表明 R_y 中也存放了 x 的值.

基本块的收尾处理

在代码结束的时候，基本块中使用的变量可能仅存放在某个寄存器中．如果这个变量是一个只在基本块内部使用的临时变量，那就没问题；当基本块结束时，我们可以忘记这些临时变量的值，并假设这些寄存器是空的．但如果一个变量在基本块出口处活跃，或者我们不知道哪些变量在出口处活跃，那么就必须假设这个变量

的值会在以后被用到. 在这种情况下, 对于每个变量 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, 我们必须生成指令: ST x, R, 其中 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数组

当代码生成算法生成加载、保存和其它机器指令时, 它必须同时更新寄存器和地址描述数组(符), 修改规则如下:

1. 对于指令: LD R, x
   (1) 修改 RVALUE(R), 使之只包含 x;
   (2) 修改 AVALUE(x), 将寄存器 R 加入 AVALUE(x)这个集合中;
   (3) 从任何不同于 x 的变量 y 的 AVALUE(y)中删除 R.

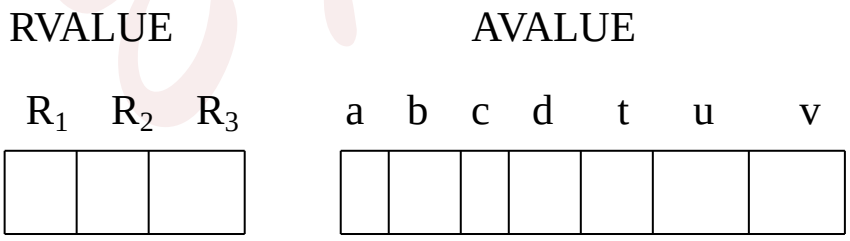
2. 对于指令: ST x, R, 修改 AVALUE(x), 使之只包含元素 x.

3. 对于实现三地址指令: x := y+z 的 ADD R_x, R_y, R_z 这样的运算而言:
   (1) 改变 RVALUE(R_x), 使之只包含 x;
   (2) 改变 AVALUE(x)使得它只包含 R_x. 注意:现在 x 的地址描述符中不包含 x 的内存位置;
   (3) 从任何不同于 x 的变量 y 的 AVALUE(y)中删除 R_x.

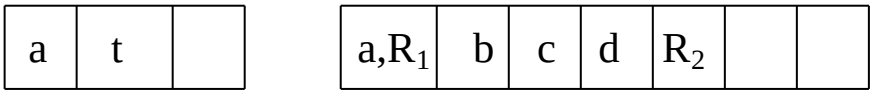
4. 当我们处理复制语句 x := y 时, 如果有必要生成把 y 加载入 R_y 的加载指令, 那么在生成加载指令并(按照规则 1)像处理所有的加载指令那样处理完各个描述符之后, 再进行下面的处理:
   (1) 把 x 加入集合 RVALUE(R_y)中;
   (2) 修改 AVALUE(x), 使得它只包含唯一的 R_y.

五. 例子: 假设 t,u,v 是临时变量, 而变量 a,b,c,d 在基本块出口处活跃.

t := a－b
u := a－c
v := t + u
a := d
d := v + u



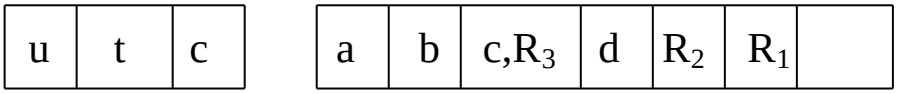
t:=a－b
LD R_1, a
LD R_2, b
SUB R_2, R_1, R_2



u:=a－c

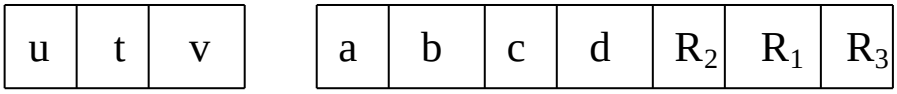
LD R_3, c

SUB R_1, R_1, R_3



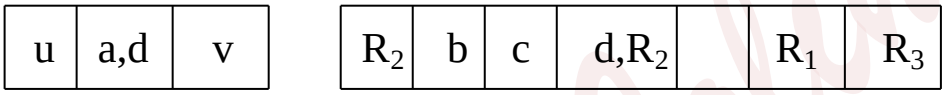
v:= t + u

ADD R_3, R_2, R_1



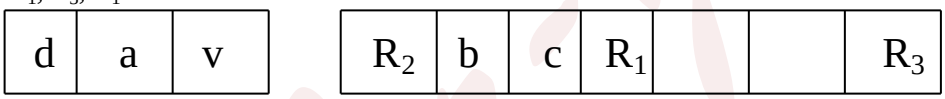
a := d

LD R_2, d



d:= v+u

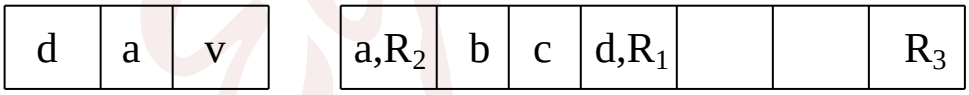
ADD R_1, R_3, R_1



exit

ST a, R_2

ST d, R_1



六. getReg 算法:

对于指令 I: x:= y+z, 分别为 y 和 z 选择寄存器(以 y 为例, z 也类似).

1. 如果 y 当前就在寄存器中, 则选择一个已包含了 y 的寄存器作为 R_y. 而不需要生成一个机器指令来把 y 加载到这个寄存器中.
2. 如果 y 不在寄存器中, 但当前存在一个空寄存器, 那么选择这个空寄存器作为 R_y.
3. 如果当前 y 不在寄存器中, 且当前也没有空寄存器. 设 R 是一个候选寄存器, 且假设 v 是 R 的寄存器描述表明的已位于 R 中的变量. 我们需要保证要么 v 的值已经不会被再次使用, 要么我们还可以到别的地方获取 v 的值. 可能的情况如下:
   1) 如果 AVALUE(v)说明 v 还保存在 R 之外的其它地方, 我们就完成了任务;
   2) 如果 v 是 x, 且 x 不同时是指令 I 的运算分量之一(比如: z), 那么我们就完成了任务. 因为在这种情况下, x 的当前值决不会被再次使用;
   3) 如果 v 不会在此后被使用(即指令 I 之后不会再次使用 v, 且如果 v 在基本块的出口处活跃, 那么 v 的值必然在基本块中被重新计算), 那么我们就完成了任务;

4) 如果前三个条件都不满足，我们就需要生成保存指令: ST v, R, 来把 v 的值复制到它自己的内存单元中去.

因为在那个时刻 R 可能存放了多个变量的值，所以我们需要对每个这样的变量 v 重复上述步骤. 最后，R 的“得分”是我们需要生成的保存指令的个数. 选择一个具有最低得分的寄存器.

现在考虑寄存器 R_x 的选择. 其中的难点和可选项几乎和选择 R_y 时的一样，因此我们只给出其中的区别.

1. 因为 x 的一个新值正在被计算，因此可选择只存放了 x 值的寄存器作为 R_x, (如果 x = y 或 x = z 仍成立).
2. 如果(像上面对变量 v 的描述那样)y 在指令 I 之后不再使用，且(在必要时加载 y 之后)R_y 仅仅保存了 y 的值，那么 R_y 同时也可以用作 R_x. 对 z 和 R_z 也有类似的选择.

当 I 是复制指令 x := y 时，我们用上面描述的方法选择 R_y，然后让 $R_x = R_y$.

*用 getReg 的算法重新回故上面的例子.

7.3 窥孔优化

一. 什么是窥孔优化?

所谓窥孔优化，就是每次只考虑目标代码中的几条指令(不一定是连续的)，并对它们进行优化.

*窥孔优化可在中间代码一级进行，也可在目标代码一级进行.

二. 删除多余指令

在目标程序中，如果有指令:

LD R_0, a
ST a, R_0

可删除第二条指令，但注意:这两条指令必须在同一基本块中才可删除. 指令:

ST a, R_0
LD R_0, a

也类似.

三. 消除不可达代码

为调试，设

```
DEBUG := 0
  ⋮
if DEBUG = 1 goto L1
goto L2
L1: print debugging information
L2:
```

上述代码可优化为

```
DEBUG := 0
  ⋮
if DEBUG <> 1 goto L2
print debugging information
L2:
```

通过常数传播，变为:

```
if 0 <> 1 goto L2
print debugging information
L2:
```

第一个语句条件总为真,可改为 goto L2, 打印调试信息的所有语句都变成不可达语句, 可逐一被消除.

四. 控制流优化

例 1:
```
    goto  L1
      ⋮
L1: goto  L2
```
替换为
```
    goto  L2
      ⋮
L1: goto  L2
```
例 2:
```
    goto  L1
      ⋮
    goto  L4
L1: if a < b goto L2
L3:
```
替换为:
```
    if a < b goto L2
    goto  L3
      ⋮
    goto  L4
L3:
```

五. 代数化简

例如: x := x*1

或 x := x+0

可以删除.

又例如: x^2 可用 x*x 代替.

六. 使用机器特有的指令

例如: x := x+1

阅读: 8.1, 8.2, 8.6, 8.7 节

EGer-Jolin