

Chapter 7 语义分析:

生成中间代码

基于属性文法的语义计算

语法制导: 伴随语法分析过程, 在语法分析
每步推导或归约时刻, 增加相应的语义分析
和代码生成处理之含义。(语法分析和语义分析
属性文法: 同时进行)

属性文法:

变量的属性: 类型值

文法: 附上 对非终结符的解释 \rightarrow 属性文法 的计算

区分产生式中不同的非终结符, 加上标记。

终结符的属性是用来计算非终结符的属性

且终结符的属性不能由其它计算出来。

~~type~~ ~~value~~ \rightarrow type value @ (n) 语法规则

一个属性文法: (G, V, F) G 为文法, V 为文法符号
属性集, F : 语法规则集

S 属性文法: 只有综合属性的文法 (是 L 属性文法的特例)

L 属性文法: 每条产生式中每个非终结符的属性, 要么
为综合属性, 要么要依赖于父结点或左边兄弟结点的继
承属性

基于属性文法的语义计算: 继承性

在自下而上的语法分析过程中可以同时进行语义分析

① 增加一个语义栈, 存放分析栈中文法符号的属性值

② 根据产生式的语义规则, 在归约时进行属性值

基于 L 属性文法的语义计算: 自顶向下, 深度优先

因为依赖于左边的兄弟结点 + 父结点 \rightarrow 继承性

采用序遍历, 中左右, 即可保证得到所有的继承属性

继承属性: 由父亲结点或兄弟结点计算的 (c)

综合属性: 自底向上, 由孩子结点计算的 (s)

综合属性: 需要所有的孩子属性计算完, 再计算

出综合属性, 从下到上, 回溯的计算为综合属性

终结符只有综合属性

综合属性:

符号栈 \rightarrow 语法分析用, (带有状态栈)
语义栈: 内部数据结构, 记录语义

归约时, 语义也要根据属性文法的内容更新

LL(1) 分析法完成 L 属性文法的语义计算

A, i
A
A-S

先把综合属性开栈, 再开栈非终结
符, 最后开栈继承属性

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

$E \rightarrow TE' \quad E' \rightarrow +TE' \mid \varepsilon \quad T \rightarrow i$

7.2 基于翻译模式的语义计算:

翻译模式: 实现过程/实现方案, 面向实现

语义规则集合可出现在表达式右端任何位置, 可以

显式表达动作和属性计算的次序.

综合属性: 语义动作置于表达式末尾.

基于翻译模式的语义计算: 只有综合属性

继承属性: 位于该符号前.

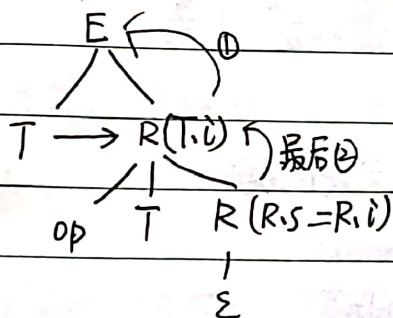
增加一个语义栈, 将综合属性压栈. 归约时.

可计算 (从下而上的语法分析)

基于翻译模式的语义计算:

自上而下的语法分析方法:

例: PPT 22: 左递归的消法: 先修改文法, 消除左递归



自底向上的语法分析过程:

语义动作作用一个非终结符代替, 称为

标记. 标记直接推为 $M.i$: 左边的继承

属性 $M.S = M.i$, 然后将综合属性放到栈内



chapter 8. 静态语义分析和中间代码生成

符号表: 保存符号的各种信息, 体现作用域和可见性。 8.3 中间代码生成: 与机器无关

变量用其在符号表中的位置表示

常见形式: AST: 抽象语法树

TAC: 四元式 $(+, a, b, c) \Rightarrow C = a + b;$

SSA: 静态的单赋值形式

符号表的实现:

常用的数据结构:

① 一般的线性表

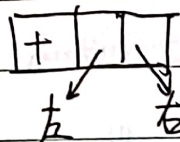
② 有序表

③ 二叉树

④ hash表

AST:

改进型AST: 有向无环图通过优化得到

AST可看作一个三元式 

符号有作用域

每个作用域都建立自己的符号表或者块操作

用域共用一个全局符号表

作用域和可见性:

TAC: 四元式, 也称三地址码:

$(op, op_1, op_2, result)$

会产生大量的临时变量

8.2 静态语义分析:

静态语义分析:

① 类型检查

② 作用域分析:

③ 控制流检查

④ 唯一性检查

⑤ 名字的上文相关性检查

SSA: 程序中的变量只有一处赋值

生成AST:

翻译: 源 \rightarrow 目标

 三元组

目标: AST中的一个结点 ptr 代表结点的位置(地址)

8.2.2 类型检查: 编译时:

生成TAC:

PPT 17: 二叉树的遍历, 有点问题 S.break 跳过

~~place~~ ~~break~~

类型检查: 关心 Type

属性 E.place: id 的存储位置

S.code: 对应于 S 的 TAC 语句序列

gen: 生成一条TAC语句

newtemp: 新建一个临时变量

ll: TAC语句序列的链接运算

布尔表达式

直接对布尔表达式求值: 与算术表达式类似

通过控制流体现布尔表达式的语义:

每一个布尔表达式都可能含有 true 和 false

比如 $E_1 \wedge E_2$

E_1 的真出口: 应该为 E_2 的第条指令? 用一假标号

E_1 的假出口: 应该为 E_1 的假出口

链接和代码回填 PPT 51

chapter 9 运行时的存储组织

变量的偏移值: 在编译时设计好

编译时认为有一个逻辑地址空间, 变量用偏移表示, 操作系统执行时给一个基址, 分配空间。

过程调用:

活动记录: 运行在栈上的栈帧 (frame)

chapter 10. 代码优化和目标代码生成

节约空间
运行速度

基本块: 只有一个入口语句和一个出口语句的语句序列
对每个基本块进行局部优化, 划分出该基本块

原则:

入口语句: 程序的第一个语句, 转移语句的目标语句, 转移语句之后的语句

出口语句: 程序的最后一个语句, 转移语句, 下一个入口之前的语句

- ① 代码等价性: 功能相同
- ② 合并: 优化要有效果。

与机器无关的优化:

程序块图: 结点是基本块。

例1: 删除公共子表达式

两个基本块 i 和 j $[i] \rightarrow [j]$

例2: 代码块: 循环优化

基本块 i 执行完会到 j 执行。

例3: 强类型弱

① i 的最后一条语句不是无条件转移语句:

例4: 变换循环控制条件

i 的最后一条语句和 j 的第一条语句 t: $SH = j$

例5: 合并已知量

② t 为转移语句, t 为目标地址。

例6: 常量传播: 引用其他的值

例7: 删除无用赋值

循环:

DAg优化步骤:

回边: 指回到前置结点

根据四元式序列

US

结点序列 α 的入口结点: 对于任意一个结点序列 α , 如果在结点序列之外有一个结点指向序列中的结点 V , 或结点序列中的结点 V 是程序首结点, 则称 V 为结点序列 α 的入口结点.

m 是 n 的必经结点, n 是 t 的必经结点则 m 是 t 的必经结点.

计算必经结点集 $D(n)$ 的算法:

① 初始值: n 本身是自己的必经结点

② 初始值: 令有结点的 $D(n) = n$ 本身, 其余结点的 $D(n) = \emptyset$ (有结点), 之后更新各点 $D(n)$.

直到 $D(n)$ 不再发生变化, 更新方法:

$$D(n) = D(n) \cup \bigcap \{ D(m) \mid m \text{ 是 } n \text{ 的前置结点} \}$$

回边: 如果 $m \text{ Dom } n$ 且 $n \rightarrow m$, 则称 $n \rightarrow m$

是该图 G 的回边.

从 n 出发, 找前置结点, 可以找出循环

局部优化: 一个基本块内的四元式是有序的, 利用DAg优化来优化 (有向无环图).