编译原理

2018 年 6 月 23 日

目录

1 词法分析

目的 将字符流转为符号流.

符号定义

- 巴克斯范式
- 正则表达式
- 混合 (如 lex 的实现)

解析方式 基本通过正则表达式语言和有限状态自动机的等价原理, 将符号定义转为正则表达式, 之后变为有限状态自动机, ϵ-NFA 到 NFA 到 DFA.

通常采取最长匹配方法, 即允许自动机向前偷看一个符号. 如果当前满足一个符号, 但是按照偷看符号转移后不满足任何符号, 则返回当前匹配的符号.

2 语法分析

目的 将符号串解析为符合程序语法的生成树

语法描述 通过上下文无关语言描述语法. 后记此语法为 $G[S]$

惯例

- 终结符串最后还认为有一个终止符号 \$ (课程中是 #)
- $\alpha, \beta \ldots \in (V \cup T)^*$
- $w \in T^*$
- $a, b \ldots \in T$
- $A, B \ldots \in V$
- $X, Y \ldots \in V \cup T$

2.1 自顶向下分析

从起始符号 S 开始, 不断利用产生式展开非终结符, 直到原串.

不确定性 这样的方法有两个不确定性

1. 选择哪一个非终结符展开
2. 选择哪一个产生式

第一个问题很容易, 每次都选择当前符号串 $wA\alpha$ 的最左边的非终结符 A 展开.

2.1.1 产生式的选择

向前查看 对于 $wA\alpha$, 选择产生式如果只考虑 A, 则没有任何线索, 能够确定选择哪一个产生式.

设 $\alpha = X_1X_2\dots X_l$. 由上, 选择产生式时, 不仅考虑 A, 还考虑 $X_1, X_2\dots X_k$, $(k < l)$, 选择的产生式 p 是它们的函数 $p = f(A, X_1\dots X_k)$.

称这种方法为 LL(k) 方法. 下述 LL(1) 方法.

First 函数 First 是符号串的函数, 定义如下

$$First(\alpha) = \{a \in T \mid \exists\beta : \alpha \Rightarrow^* a\beta\}$$

特别地, 如果 $\alpha \Rightarrow^* \epsilon$, 则说 $\epsilon \in First(\alpha)$. 注意 $First$ 定义给 α, 可能包含 ϵ.

First 集合的意义是符号串对应的终结符串的首个终结符的取值集合.

Follow 函数 Follow 是非终结符的函数, 定义如下

$$Follow(A) = \{a \in T \mid \exists\alpha, \beta : S\$ \Rightarrow^* \alpha A\beta\$, a \in First(\beta\$)\}$$

注意 $Follow$ 定义给 A, 可能包含 \$.

Follow 集合的意思是 $G[S]$ 句型中中, 可能跟在 A 的后面的终结符所有取值.

选择产生式 希望展开 A, 并且 lookahead $= a$, 则选择的产生式 $A \to \beta$ 应当满足

$$a \in PS(A \to \beta) = \begin{cases} First(\beta) & \epsilon \notin First(\beta) \\ First(\beta) \cup Follow(A) - \{\epsilon\} & \epsilon \in First(\beta) \end{cases}$$

如果 $\forall A \in V, a \in T$ 有 $\left|\{A \to \beta \mid a \in PS(A \to \beta)\}\right| = 1$ (即产生式的选择唯一), 则称 $G[S]$ 是 LL(1) 文法, 可以按照上述方法解析.

2.1.2 First 和 Follow 的计算

First 的计算 考虑计算 $First(\alpha)$, 设 $\alpha = X_1 X_2 \dots X_k$.

若 $X_1 \dots X_{l-1}$ 可空, X_l 不可空 (显然终结符不可空), 有

$$First(\alpha) = \cup_{1 \le i \le l} \left(First(X_i) - \{\epsilon\}\right)$$

若 $X_1 \dots X_k$ 都可空, 则

$$First(\alpha) = \cup_{1 \le i \le k} First(X_i)$$

非终结符 $First(A) = \cup_{\beta \,:\, A \to \beta} First(\beta)$.

通过迭代即可求解如上的集合约束问题.

Follow 的计算 考虑计算 $Follow(A)$. 首先寻找所有形如 $B \to \alpha A \beta$ 的产生式.

如果 $\epsilon \notin First(\beta)$, 则 $Follow(A) \supseteq First(\beta)$.

如果 $\epsilon \in First(\beta)$, 则 $Follow(A) \supseteq First(\beta) \cup Follow(B) - \{\epsilon\}$.

通过迭代求解如上的集合约束问题, 要求 $Follow(A)$ 取最小 (即满足上述条件的最小集合).

2.1.3 递归下降程序

每个非终结符的解析都是一个函数. 其中根据 lookahead 选择产生式, 匹配非终结符就调用对应函数, 匹配非终结符就直接 `match_token`.

常见的递归下降程序如

```
void parse_B()
{
```

```
        switch (lookahead) {
            case c:
                // B -> c A d,      First(c A d) = {c}
                match_token(c);
                parse_A()
                match_token(d);
                break;
            case b:
                // B -> epsilon,    Follow(B) = {b}
                break;
            default:
                // no production matches here
                report error;
        }
    }
```

2.1.4 表驱动程序

记 $M[A, a]$ 表示希望展开 A, lookahead $= a$ 时, 按照上述方法得到的可用产生式.

表驱动利用一个栈, 最初栈中只有 $S\$$, S 在栈顶. 之后重复检查栈顶 X 和输入 lookahead $= a$

- $X = \$$, 分析完成
- $X \in T$, 此时应有 X 等于 a. 符合则消耗 a, 否则报错
- $X \in V$, 设 $M[X, a] = X \to \alpha$, 则 X 出栈, α 从右到左入栈

2.1.5 语法变换

通过消除左递归和公共左公因子, 能将很多非 LL(1) 语言转换为 LL(1) 语言, 如 $S \to Sa|b$.

去除直接左递归 原为

$$A \to A\alpha_1 \,|\, A\alpha_2 \,|\, \ldots \,|\, \beta_1 \,|\, \beta_2 \,|\, \ldots$$

改写为

$$A \to \beta_1 B \,|\, \beta_2 B \,|\, \dots$$
$$B \to \alpha_1 B \,|\, \alpha_2 B \,|\, \dots \,|\, \epsilon$$

去除间接左递归 要求原文法无环 $A \not\Rightarrow^+ A$, 无 ϵ 产生式 $A \to \epsilon$.

首先将非终结符编号为 $A_1, A_2 \dots A_n$, 然后

```
for i = 1 to n
    for j = 1 to i-1
        对于所有 A_i -> A_j \alpha,
            代替以 A_i -> \beta \alpha,
            其中 A_j -> \beta
    消除A_i的直接递归
```

消除左公因子 直接将左公因子作为新的符号即可.

2.2 Bottom-up parsing

从输入串开始, 不停将串中一个子串规约成一个非终结符, 直到规约成 S.

不确定性 有 2 个不确定性

1. 选择哪个子串
2. 选择哪个产生式, 如果有多个产生式的右边一样

采用一类称为 LR 分析的解决此问题. 每次都选择 “句柄”, 选择的产生式通过 LR 分析表确定.

2.2.1 概念

以下是语法本身的概念

短语 如果 $S \Rightarrow^* \alpha A \beta$ 且 $A \Rightarrow^+ \gamma$, 则称 γ 是 $\alpha A \beta$ 中 A 的短语.

在 $\alpha A \beta$ 确定的情况下, 可以称 γ 是 A 的短语. 这时在 $S \Rightarrow^* \alpha\gamma\beta$ 的分析树中, γ 是 A 的子树的叶遍历, 并且要求 A 本身不能是叶子.

直接短语 $S \Rightarrow^* \alpha A\beta$ 且 $A \Rightarrow \gamma$, 则称 γ 是 $\alpha A\beta$ 中 A 的直接短语.

直接短语就是可以一步推出的短语. 在 $\alpha\gamma\beta$ 的分析树中, 对应的是 A 所有儿子都是叶子的情况.

句柄 $S \Rightarrow^* \alpha Aw$ 且 $A \Rightarrow \beta$, 则称 β 是 αAw 中 A 的句柄.

句柄就是从左到右第一个出现的直接短语. 可以理解为 "最右推导" 反过来就是 "最左规约".

2.2.2 通用框架

所有 LR 分析, 分析框架是一样的, 只有分析表是不一样的.

基本分析方法就是基于一个栈, 从左到右扫描输入串, 每次根据当前控制状态, 栈中内容 (不限于栈顶), 下一个输入符号来决定动作, 并且转移到新的控制状态.

所有 LR 分析都有一个 $action[i, a]$ 表

$action[i, a]$	含义
shift j	将a移入栈中, 状态变为j
reduce $A \to \alpha$	此时栈顶应为α, 规约栈顶. 之后A入栈, 状态变为$goto[i, A]$
success	分析成功, 即将原输入串规约成了S
error	出错

所有 LR 分析都有一个 $goto[i, A]$ 表. $goto[i, A] = j$ 表示当前状态为 i, 一个非终结符 A 入栈后, 那么状态变为 j.

所有 LR 分析都是用增广文法, 即加入产生式 $S' \to S$ 的文法 $G[S']$. $S' \to .S$ 对应的状态为初状态.

2.2.3 LR(0) 分析表

item 加点的产生式. 为了方便认识用方括号括起.

closure 考虑 item 间的关系 R, 为 $[A \to \alpha.B\beta]\ R\ [B \to .\gamma]$. 定义 $closure(I)$ 等于 I 在关系 R 的传递闭包, I 是 item 集合.

go I is closure, $X \in V \cup T$

$$go(I, X) = closure(\{[A \to \alpha X.\beta] \mid [A \to \alpha.X\beta] \in I\})$$

action

$$action[i,a] = \begin{cases} \text{shift } j & J = go(I,a) \\ \text{reduce } A \to \beta & [A \to \beta.] \in I \\ \text{accept} & [S' \to S.] \in I \\ \text{error} & \text{otherwise} \end{cases}$$

goto

$$goto[i,A] = j \quad J = go(I,A)$$

2.2.4 SLR(1) 分析表

基本同 LR(0), 但是完成 reduce 的条件变得更精确.

原理: 完成规约 $A \to \alpha$ 时, 下一个输入 a 应当满足 $a \in Follow(A)$.

action

$$action[i,a] = \begin{cases} \text{shift } j & J = go(I,a) \\ \text{reduce } A \to \beta & [A \to \beta.] \in I \quad \wedge \quad a \in Follow(A) \\ \text{accept} & [S' \to S.] \in I \quad \wedge \quad a = \$ \\ \text{error} & \text{otherwise} \end{cases}$$

2.2.5 LR(1) 分析表

item 之前是一遇到可规约的 $A \to \alpha$. 就规约 (规则 2.), 但是不一定是这样 (前看 α 之后一个符号得到的信息被忽略了).

item 的表示是 $[A \to \alpha.\beta, a], \quad a \in T$. 如果 $\beta \neq \epsilon$, 那么一切和 SLR 一样.

如果 $\beta = \epsilon$, 那么当且仅当 $a \in Follow(A)$ 时, 才进行规约.

closure $[A \to \alpha.B\beta, a]$ **R** $[B \to .\gamma, b] \quad b \in \mathbf{FIRST}(\beta a)$ 如上关系的闭包.

构造 go $go(I,X) = closure([A \to \alpha X.\beta] \mid [A \to \alpha.X\beta] \in I)$

构造 item 集 从 $closure([S' \to .S, \$])$ 按照 go 拓展得图.

初态 $I_0 = closure(\{[S' \to .S,\ \$]\})$

action 更精确的描述了完成 reduce 的条件.

$$action[I,a] = \begin{cases} \text{shift } go(I,a) & [A \to \alpha.a\beta,\ b] \in I \\ \text{reduce } A \to \alpha & [A \to \alpha.,\ a] \in I,\ A \to \alpha \neq S' \to S \\ \text{accept} & [S' \to S.,\$] \in I \\ \text{error} & \text{otherwise} \end{cases}$$

LR(1) 但非 SLR(1) 的例子

$$\begin{aligned} S &\to L = R \quad | \quad R \\ L &\to *R \quad | \quad \text{id} \\ R &\to L \end{aligned}$$

问题就出在, $S \to R$ 和 $L \to *R$ 中的 R 是不一样的 R. 如果把它们分成分开变成两个非终结符 R_1 和 R_2, 容易发现 $Follow(R_1) = \{\$\}$, 而 $Follow(R_2) = \{\$, =\}$. 也就是说, 分析的过程中就已经限定了非终结符的一些性质, 在用其一般的 $Follow$ 去套, 自然就可能出现问题.

2.2.6 LALR(1)

和 LR(0) 一样状态数目. 其余方法和 LR(1) 相同.

合并 LR(1) 同芯状态 同芯: $[A \to \alpha.\beta, a]$ 中 $A \to \alpha.\beta$ 相同

3 符号表

基本概念 不同的作用域有自己的符号表, 某处可见的符号是其自身包含所有父亲作用域 (亦称开作用域) 符号表的并.

实现 可以实现为全局的一个符号表, 也可以实现成各个作用域符号表的栈.

4 基于语法的语义计算

4.1 基于属性文法

4.1.1 基本概念

定义 在 CFG 基础上, 对每个 $X \in V$ 关联属性, 记为 $X.a, X.b$ 等.

每个产生式 $A \to X_1X_2 \dots X_N$, $X_i \in V \cup T, X_0 = A$ 关联一个语法动作, 语法动作是若干个属性计算的序列, 每个属性计算形如 $X_i.a = f(X_j.b \mid j \neq i, b是X_j的属性)$, $0 \leq m \leq N$.

综合属性 $A \to X_0 \dots X_N$ 关联的语法动作是 $A.a = f(X_i.b)$ 则称 $A.a$ 是综合属性. 综合属性代表语法树中自下而上传递的信息.

继承属性 $A \to X_0 \dots X_N$ 关联的语法动作是 $X_i.a = f(X_j.b), X_i \neq A$ 则称 $X_i.a$ 是继承属性. 继承属性代表语法树中自上而下传递的信息.

S-属性文法 只包含综合属性的属性文法称 S-属性文法.

L-属性文法 允许综合属性和继承属性, 但是语法动作要求有 $X_i.a = f(X_{\neq i}.b) = f(X_{<i}.b)$. 即产生式中某符号的属性不能依赖产生式中位于它之后的符号的属性.

4.1.2 两趟方法的属性计算

生成语法树之后, 计算属性的大致步骤如下

1. 分析语法树中结点属性 (即符号属性) 的依赖关系
2. 如果依赖关系不存在环, 则按照依赖关系的拓扑顺序计算属性值.
   如果依赖关系有环, 认为属性语法不是良定义的, 不予处理.

两趟方法通用, 但是效率开销较大.

4.1.3 一趟方法的属性计算

S-属性文法 采用自底向上文法分析 (如 LR), 每次按 $A \to X_1X_2 \dots X_N$ 进行规约时, 一定有 $X_1, X_2 \dots X_N$ 是栈顶 N 个元素. 因此将元素属性一并存

放在栈中, 每次规约时直接通过栈顶元素 N 个元素 $X_1, X_2 \dots X_N$ 的属性计算 $A.a$ 即可.

L-属性文法 采用自顶向下文法分析 (如 LL(1) 递归下降), 将继承属性作为参数传入子符号的分析过程, 并且要求子符号分析过程返回子符号的综合属性.

即将 $\mathbf{Parse}(A) \to \texttt{void}$ 改为 $\mathbf{Parse}(A, A.i) \to A.s$, 其中 $A.i$ 表示 A 的继承属性, $A.s$ 表示 A 的综合属性.

4.2 基于翻译模式

4.2.1 基本概念

翻译模式 类似属性文法, 但是允许语法动作出现在产生式中任何地方, 表示匹配到该处时即刻执行该语法动作.

消除左递归 讨论消除左递归时, 保持翻译动作等价. 以直接左递归为例

$$A \to A_1\alpha \ \{A.s = f(A_1.s, \alpha.s)\}$$
$$A \to \beta \ \{A.s = g(\beta.s)\}$$

按照消除直接左递归的方法变换之后, 变为

$$A \to \beta \ \{R.i = g(\beta.s)\} \ R \ \{A.a = f(R.s)\}$$
$$R \to \epsilon \ \{R.s = R.i\}$$
$$R \to \alpha \ \{R_1.i = f(R.i, \alpha.s)\} \ R_1 \ \{R.s = R_1.s\}$$

可以认为 $R.i$ 中保存了原来式子中所有其左边的所有信息.

4.2.2 自上而下计算

类似 4.1.3, 但是计算属性在 $\mathbf{parse}(\mathbf{A})$ 过程中进行.

4.2.3 自下而上计算

使用从下而上分析方法, 将综合属性和符号一起存储在栈中. 这样有一个问题, 无法访问继承属性, 因为需要在确定非终结符前完成语法动作. 因此需要提供一种方法实现继承属性的访问.

可以将语法稍作变换, 添加若干空非终结符, 由它们完成中间的语法规则的计算, 使得产生式中间只有复写规则 $X_i.i = X_j.s, \quad j < i$.

这样可以沿着复写规则, 对于继承属性 $X_i.i$, 最终找到综合属性 $X_j.s$, 可以用 $X_j.s$ 来代替 $X_i.i$.

注意如果有多个产生式中都复写了 $A.i = B.s$, 则需要保证这些产生式中, A 和 B 的相对位置不变, i.e. A 和 B 之间隔多少个符号是定值, 这样生成规约时代码 (参见图 1) 片段.

5 中间代码生成

5.1 静态语义分析

包含类型检查, 作用域分析, 控制流检查等. 通过翻译模式实现.

5.2 中间代码生成

课程中只考虑三地址码的生成.

5.2.1 布尔表达式的翻译

直接翻译 不处理短路特性. 相当于普通表达式的计算.

L-属性文法 每个表达式 E 有继承属性 $E.true$ 和 $E.false$, 这些属性分别都是标号. 具体如图. 实现如图 2 如果翻译循环等, 还需要加入 $S.next$ 等.

S-属性文法 将标号作为综合属性, 待顶层布尔表达式翻译完后, 将所有下层表达式中未知的标号回填. 称为拉链方法 (backpatching).

表达式 E 有两种综合属性 $E.truelist$ 和 $E.falselist$, 包含未确定的, 但是知道是跳转到真还是假的标号.

还包含函数如表 1 实现如图 3

5.2.2 控制流的翻译

L-属性文法 如图 4, 循环通过继承属性 $S.next$ 实现, break 语句通过继承属性 $S.break$ 传递.

$N \to .\ \{\ S.f := 1\}\ \ S\ \ \{\ print(S.v)\ \}$
$S \to\ \{\ B.f := S.f\ \}\ B\ \{\ S_1.f := S.f + 1\ \}\ S_1\ \{S.v := S_1.v + B.v\ \}$
$S \to \varepsilon\ \{\ S.v := 0\ \}$
$B \to 0\ \ \{\ B.v := 0\ \}$
$B \to 1\ \ \{\ B.v := 2\hat{}(-B.f)\ \}$

(a) 最初的翻译模式

$N \to .\ M\ \{\ S.f := M.s\ \}\ \ S\ \ \{\ print(S.v)\ \}$
$S \to \{\ B.f := S.f\ \}\ B\ \{\ P.i := S.f\ \}\ P\ \{\ S_1.f := P.s\ \}\ S_1\ \{S.v := S_1.v + B.v\ \}$
$S \to \varepsilon\ \{\ S.v := 0\ \}$
$B \to 0\ \ \{\ B.v := 0\ \}$
$B \to 1\ \ \{\ B.v := 2\hat{}(-B.f)\ \}$
$M \to \varepsilon\ \{\ M.s := 1\}$
$P \to \varepsilon\ \{\ P.s := P.i + 1\ \}$

(b) 转换为适合自下而上的翻译模式

产生式	依产生式归约时语义计算的代码片断
$N \to .\ M\ S$	$print(val\ [top]\ .v)$
$S \to B\ P\ S_1$	$val\ [top-2].v := val\ [top].v + val\ [top-2].v$
$S \to \varepsilon$	$val\ [top+1].v := 0$
$B \to 0$	$val\ [top].v := 0$
$B \to 1$	$val\ [top].v := 2\hat{}(-val\ [top-1].s)$
$M \to \varepsilon$	$val\ [top+1].s := 1$
$P \to \varepsilon$	$val\ [top+1].s := val\ [top-1].s+1$

(分析栈val 存放文法符号的综合属性，top为栈顶指针)

(c) 规约时代码片段

图 1: 自下而上计算的例子

– 翻译布尔表达式至短路代码（L-翻译模式）

$E \to \{ E_1.true := E.true;\ E_1.false := newlabel \}\ E_1 \vee$
$\{ E_2.true := E.true;\ E_2.false := E.false \}\ E_2$
$\{ E.code := E_1.code \,||\, gen(E_1.false\ ':') \,||\, E_2.code \}$

$E \to \{ E_1.false := E.false;\ E_1.true := newlabel \}\ E_1 \wedge$
$\{ E_2.false := E.false;\ E_2.true := E.true \}\ E_2$
$\{ E.code := E_1.code \,||\, gen(E_1.true\ ':') \,||\, E_2.code \}$

$E \to \neg \{ E_1.true := E.false;\ E_1.false := E.true \}\ E_1 \{ E.code := E_1.code \}$

$E \to (\ \{ E_1.true := E.true;\ E_1.false := E.false \}\ E_1\ )\ \{ E.code := E_1.code \}$

$E \to \underline{id}_1\ \underline{rop}\ \underline{id}_2\ \{ E.code := gen(\text{'if'}\ \underline{id}_1.place\ \underline{rop}.op\ \underline{id}_2.place\ \text{'goto'}$
$E.true) \,||\, gen(\text{'goto'}\ E.false) \}$

$E \to true\ \{ E.code := gen(\text{'goto'}\ E.true) \}$

$E \to false\ \{ E.code := gen(\text{'goto'}\ E.false) \}$

图 2: L-属性文法的布尔短路翻译

函数名	行为
$makelist(initial\_entry) \to list$	创建一个待回填的表
$merge(list, list) \to list$	返回两个表的合并
$backpatch(list, lbl)$	确定 $list$ 中的标号为 lbl

表 1: 拉链方法的函数

S-属性文法 如图 5, 循环通过综合属性 $S.nextlist$ 实现, break 语句通过综合属性 $S.breaklist$ 传递.

$E \to E_1 \vee M\ E_2$	{ *backpatch*(E_1.*falselist*, *M.gotostm*) ; *E.truelist* := *merge*(E_1.*truelist*, E_2.*truelist*) ; *E.falselist* := E_2.*falselist* }
$E \to E_1 \wedge M\ E_2$	{ *backpatch*(E_1.*truelist*, *M.gotostm*) ; *E.falselist* := *merge*(E_1.*falselist*, E_2.*falselist*) ; *E.truelist* := E_2.*truelist* }
$E \to \neg\ E_1$	{ *E.truelist* := E_1.*falselist* ; *E.falselist* := E_1.*truelist* }
$E \to (\ E_1\ )$	{ *E.truelist* := E_1.*truelist* ; *E.falselist* := E_1.*falselist* }
$E \to \underline{\text{id}}_1$ rop $\underline{\text{id}}_2$	{ *E.truelist* := *makelist* (*nextstm*); *E.falselist* := *makelist* (*nextstm*+1); *emit* ('if' $\underline{\text{id}}_1$.*place* rop.op $\underline{\text{id}}_2$.*place* 'goto _'); *emit* ('goto _') }
$E \to$ true	{ *E.truelist* := *makelist* (*nextstm*); *emit* ('goto _') }
$E \to$ false	{ *E.falselist* := *makelist* (*nextstm*); *emit* ('goto _') }
$M \to \varepsilon$	{ *M.gotostm* := *nextstm* }

图 3: S-属性文法的布尔短路翻译

$P \rightarrow$ D ; { S.next := newlabel; S.break := newlabel } S
{ gen(S.next ':') }

$S \rightarrow$ if { E.true := newlabel; E.false := S.next } E then
{ S_1.next := S.next; S_1.break := S.break } S_1
{ S.code := E.code || gen(E. true ':') || S_1.code }

$S \rightarrow$ if { E.true := newlabel; E.false := newlabel } E then
{ S_1.next := S.next; S_1.break := S.break } S_1 else
{ S_2.next := S.next; S_2.break := S.break } S_2
{ S.code := E.code || gen(E.true ':') || S_1.code ||
gen('goto' S.next) || gen(E.false ':') || S_2.code }

$S \rightarrow$ while { E.true := newlabel; E.false := S.next } E do
{ S_1.next := newlabel; S_1.break := S.next } S_1
{ S.code := gen(S_1.next ':') ||
E.code || gen(E.true ':') || S_1.code ||
gen('goto' S_1.next) }

$S \rightarrow$ { S_1.next := newlabel; S_1.break := S.break } S_1 ;
{ S_2.next := S.next; S_2.break := S.break } S_2
{ S.code := S_1.code || gen(S_1.next ':') || S_2.code }

$S \rightarrow$ break ; { S.code := gen('goto' S.break) }

图 4: L-属性文法的控制流翻译

$P \rightarrow D\ ;\ S\ M$ { backpatch(S.nextlist, M.gotostm) ;
backpatch(S.breaklist, M.gotostm) }

$S \rightarrow$ if E then $M\ S_1$ { backpatch(E.truelist, M.gotostm) ;
S.nextlist := merge(E.falselist, S_1.nextlist) ;
S.breaklist := S_1.breaklist }

$S \rightarrow$ if E then $M_1\ S_1\ N$ else $M_2\ S_2$ { backpatch(E.truelist, M_1.gotostm) ;
backpatch(E.falselist, M_2.gotostm) ;
S.nextlist := merge(S_1.nextlist, merge(N.nextlist, S_2.nextlist) ;
S.breaklist := merge(S_1.breaklist, S_2.breaklist) }

$S \rightarrow$ while $M_1\ E$ then $M_2\ S_1$
{ backpatch(S_1.nextlist, M_1.gotostm) ;
backpatch(E.truelist, M_2.gotostm) ;
S.nextlist := merge(E.falselist, S_1.breaklist);
S.breaklist := “”; emit(‘goto’, M_1.gotostm) }

$S \rightarrow S_1\ ;\ M\ S_2$ { backpatch(S_1.nextlist, M.gotostm) ;
S.nextlist := S_2.nextlist ;
S.breaklist := merge(S_1.breaklist, S_2.breaklist) }

$S \rightarrow$ break ; { S.breaklist t := makelist (nextstm) ; S.nextlist := “”;
emit (‘goto _’) }

$M \rightarrow \varepsilon$ { M.gotostm := nextstm }

$N \rightarrow \varepsilon$ { N.nextlist := makelist(nextstm); emit(‘goto _’) }

图 5: S-属性文法的控制流翻译

6 运行时存储组织

6.1 存储分配策略

决定变量如何安排在内存中. 可以分为静态和动态分配, 其中动态分配又分为栈式和堆式.

静态分配 编译期确定变量在内存中的位置, 常用于全局变量或`static` 变量. 这样无法处理递归函数.

栈式分配 空间的分配根据进入退出函数在运行期确定. 可有效实现动态嵌套的程序结构.

堆式分配 最灵活, 需要操作系统和程序员或运行时内存系统共同管理.

6.2 活动记录

活动记录即栈帧, 其中保存一个函数调用的所有相关数据, 包括实参, 返回地址, 局部变量, 保存的寄存器等等.

6.2.1 不允许嵌套函数

这种情况下, 进入一个函数分配一个栈帧, 从函数中返回就回收其栈帧. 每个函数能看到的符号只有全局符号和自己栈帧中的符号.

6.2.2 允许嵌套函数

基本约定 称被嵌套的函数是子函数, 嵌套其他函数的函数是父函数. 一个子函数有且仅有一个父函数, 只有父函数的函数体中才能调用子函数.

子函数能够访问父函数的变量[1].

问题 一个函数能看到的符号除了全局符号和自己栈帧中的符号, 还可能有父函数的符号. 因此需要查找父函数的栈帧.

[1]这里变量声明都类似 Pascal 语言, 是在函数体前声明的

函数继承树 将所有函数按照父子函数关系建立一颗树, 其中main 为树根, 深度为 0. 注意此处的main 不像 C 的main 是一个全局函数, 而相当于整个程序, 可以参考 Pascal 的语法

```
program ...
...
begin
    (* main *)
end;
```

所以全局变量是main 的变量, 全局函数是main 的子函数.

容易证明, 在程序运行的任何时刻, 深度为 k 的所有函数至多有一个是活动的, 即深度为 k 的函数至多有一个, 其栈帧在当前运行栈中. 并且如果深度为 k 的某函数是活动的, 则一定有深度是 $k-1$ 的某函数也是活动的, 且后者是前者的父函数.

Display 表方法 维护一个表 D_n. D_n 表示从main 到当前函数在函数继承树上构成的链中, 当前函数深度为 n 的祖先函数的栈帧位置.

如对于以下程序

```
program main;

    function A(params): ret
        function B(params): ret
            function C(params): ret
            begin
                ... C(args); ...
            end; (* end function C *)
        begin
            ... C(args); ...
        end; (* end function B *)
    begin
        ... B(args); ...
    end; (* end function A *)
```

```
    function A1(params): ret
    begin
        ... A(args); ...
    end; (* end function A1 *)

begin
    ... A1(args); ...
end.
```

第二次运行C 时, 栈应当类似于

	whose stackframe	depth
$D_3 \rightarrow$	C	3
	C	3
$D_2 \rightarrow$	B	2
$D_1 \rightarrow$	A	1
	A1	1
$D_0 \rightarrow$	main	0

维护 Display 考虑到每次对 Display 表的修改至多一项, 所以可以在栈帧中直接保存被替换的 Display 表项.

静态动态表方法 除了最初的main 外每个函数的栈帧都额外保存两个指针, 叫静态链和动态链. 某函数的动态链指向本次运行中调用该函数的函数栈帧; 静态链指向本次运行栈中其父函数, 并且是在该函数栈帧之前的第一个父函数.

6.3 面向对象系统的实现

6.3.1 对象的存储组织

朴素方法 当前对象的所有特性, 包括继承的特性 (域和方法), 都复制到对象存储区内.

类存储 将每个类的结构的描述保存在类的存储空间, 使用父类指针完成继承. 之后每个对象除了自己的成员数据外, 只需要访问自己类的结构描述得到方法.

虚函数表 将每个类的结构的描述保存在类的存储空间, 但是将父类方法也直接保存到自己类的描述. 之后每个对象除了自己的成员数据外, 只需要访问自己类的结构描述得到方法.

7 目标代码生成

7.1 数据流分析

7.1.1 概念

假设程序是 TAC 语句的序列.

基本块 如果将程序按照语句访问组织成图, 顺序语句和无条件跳转有且只有一条出边, 有条件跳转有两条出边. 定义基本块是其中一条极大链, 链内部非首尾节点入度出度都为 1.

流图 原程序的图进行基本块缩链后, 得到的以基本块为结点的图. 程序首条语句所在的流图结点成为流图首结点. 一般都去掉首结点不可达的语句.

支配节点集 流图中, 称 u 是 v 的支配节点, 当且仅当任何从流图首结点到 v 的路径都需要经过 u, 记为 $u\mathrm{DOM}v$

回边 若 $u\mathrm{DOM}v$ 且存在边 $v \to u$, 则称 $v \to u$ 是一条回边.

循环 $v \to u$ 是一条回边, 则定义其对应的自然循环为 $\{v, u\} \cup \{t \mid \exists P : t \to v, u \notin P\}$

7.1.2 正向数据流分析

定值和引用 对于语句 I_n : lhs = rhs, 称 lhs 中变量都被定值, rhs 中所有变量都被引用, 分别记为 I_n **Def** lhs, I_n **Ref** rhs. I_n 称为 lhs 的一个定值点.

同一条语句中, 引用发生在定值之前.

如果 I_n **Def** $A \ \wedge \ I_m$ **Ref** A, 并且存在路径 $P : I_n \to I_m$, 满足 $\forall I \in P, I \neq I_n : \neg(I \ \textbf{Def} \ A)$, 则称 A 的定值点 I_n 可达引用点 I_m.

基本集合

- $GEN[B]$, 是 B 中定值点的集合, 其可达 B 的出口.
- $KILL[B]$, 是 B 外定值点的集合, 其可达 B 的入口, 但不能达到 B 的出口.
- $IN[B]$, 是可达 B 入口的定值点的集合.
- $OUT[B]$, 是可达 B 出口的定值点的集合.

其中, GEN 和 $KILL$ 可以直接通过流图得到. 计算时可以取 $KILL$ 为所有可能被 B 中定值点覆盖的外部定值点.

到达-定值分析 数据流方程

$$IN[B] = \bigcup_{C \in \mathrm{Prev}[B]} OUT[C]$$

$$OUT[B] = IN[B] \cup GEN[B] - KILL[B]$$

计算集合约束问题, 即得诸集合的值.

7.1.3 后向数据流分析

活跃变量 对于语句 I_n 和变量 A, 如果存在路径 $P : I_n \to I_m$, 满足 I_m **Ref** A, 则称 A 在 I_n 处是活跃的.

基本集合

- $Def[B]$, 是变量的集合, 其在 B 中被引用前被定值
- $LiveUse[B]$, 是变量的集合, 其在 B 中被定值前被引用
- $LiveIn[B]$, 是变量的集合, 其在 B 的入口寸前是活跃的
- $LiveOut[B]$, 是变量的集合, 其在 B 的出口寸后是活跃的

一定有 $Def \cap LiveUse = \emptyset$

活跃变量分析 数据流方程

$$LiveOut[B] = \bigcup_{C \in \mathrm{Succ}[B]} LiveIn[B]$$
$$LiveIn[B] = LiveOut[B] \cup LiveUse[B] - Def[B]$$

计算集合约束问题, 即得诸集合的值.

7.1.4 其他流信息

UD 链 考虑 I_n 点引用了 A 的值, 则定义 A 在 I_n 的引用-定值链为

$$\{I \mid I \textbf{ Def } A \wedge I\text{可达}I_n\}$$

显然, 如果 $A \in Def[B]$, 那么其 UD 链就是 B 中定值 A 的那条语句. 否则, 应有 $A \in LiveIn[B]$, 则其 UD 链就是 $IN[B]$ 中所有 A 的定值点.

DU 链 考虑 I_n 点定义了 A 的值, 则定义 A 在 I_n 的定值-引用链为

$$\{I \mid I \textbf{ Ref } A \wedge I_n\text{可达}I\}$$

待用信息和活跃信息 待用信息追踪变量在一个基本块中下一次引用的位置. 考虑加上待用信息注解, 则需要对于基本块中每个语句其中每个变量, 找到该基本块中其下一次被引用的位置, 然后标记在右上角标. 位置用语句的编号表示, 编号从 1 开始, 如果没有下一次引用则置为 0.

活跃信息追踪变量的一个值的生命周期. 考虑加上活跃信息注解, 即对基本块中每个语句中每个变量加一个右上角标, 为 L 或者 F. 如果是 L, 表示此变量在这条语句之后, 还会在被定值前被引用; 为 F 则表示在这条语句之后, 此变量的这个值已经不会被引用了. 初始条件可以从基本块的 $LiveOut$ 信息得到.

7.2 基本块的 DAG 表示

约定 基本块中, 只考虑三种语句, 如图 之后, 对于每一个语句. 如果其 rhs 中, 有变量没有对应的结点, 则为其新建一个结点.

之后构建过程中, 完成的优化有合并常量, 删除冗余运算, 合并已知量, 删除无用赋值. 例如下图.

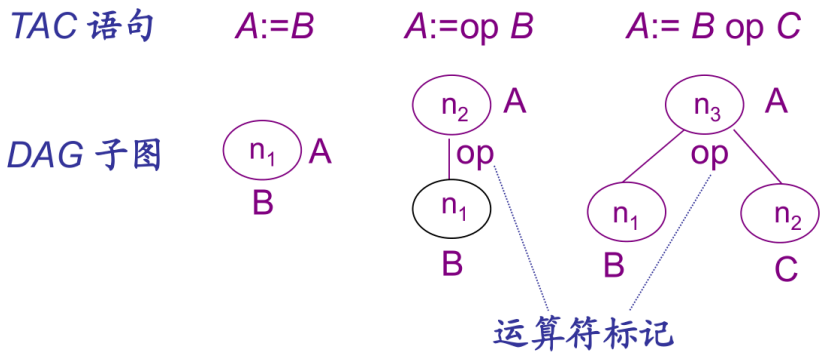

图 6: 三种语句, 以及其 DAG 子图表示

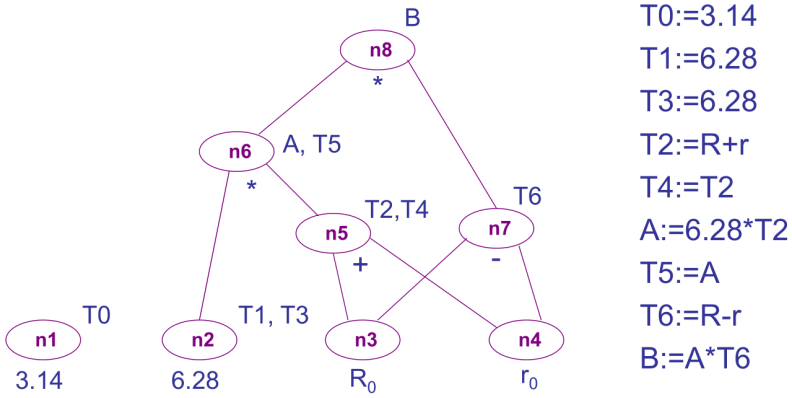

图 7: 右边的基本块生成左边的 DAG

7.3 代码生成

7.3.1 指令选择

要求 首先要求正确性, 即意思不改变. 其次是代价, 包括减少语句条数, 减少内存访问等等. 具体实现通过 AVALUE 和 RVALUE 来完成, 即及时维护一个寄存器-变量的一对多关系.

7.3.2 Ershov 数

一个表达式的 Ershov 数定义为其求值所需最少多少寄存器.

可以递归地计算, 考虑表达式树 (DAG 亦可)T, 左右儿子是 L, R. 有

$$\mathrm{Ershov}(T) = \begin{cases} 1 & L = R = \mathrm{NULL} \\ \mathrm{Ershov}(R) & L = \mathrm{NULL} \\ \mathrm{Ershov}(L) & R = \mathrm{NULL} \\ \mathrm{Ershov}(L) + 1 & \mathrm{Ershov}(L) = \mathrm{Ershov}(R) \\ max\{\mathrm{Ershov}(L), \mathrm{Ershov}(R)\} & \mathrm{Ershov}(L) \neq \mathrm{Ershov}(R) \end{cases}$$

由 Ershov 数的定义容易得到求值算法 (Sethi-Ullman 算法)

7.3.3 寄存器分配

即将伪寄存器映射到物理寄存器的方法, 不考虑到泄露到内存的情况. 采用寄存器相干图. 原理是, 如果在 A 的定值点寸后 B 是活跃的, 则 A, B 应当分配到不同的物理存储器, 否则对 A 的定值会修改 B 的值. 于是将伪寄存器作为结点建图, 求图色数就是程序最少使用的物理存储器.