$PL/0\epsilon$ 编译器实现

从零开始设计并实现一个 PL/0 扩展语法的编译器

Jinghui Hu <hujinghui@buaa.edu.cn>

release v0.9.0 on 2024-03-06

† 空白页

目录

插图

表格

```
+------------------------------------------------------------------------------+
|                  _ _   _____   _           __ ___                            |
|                ( | ) |  __ \ | |         / // _ \                           |
|                 V V  | |__) || |        / /| | | |                          |
|                      |  ___/ | |       / / | | | |                          |
|                      | |     | |____  / /  | |_| |                          |
|                      |_|     |______|/_/    \___/                          |
|                  ______        _                    _           _           |
|                |  ____|      | |                  | |         | |          |
|                | |__   __  __| |_  ___  _ __    __| |  ___   __| |          |
|                |  __|  \ \/ /| __|/ _ \| '_ \  / _` | / _ \ / _` |          |
|                | |____  >  < | |_|  __/| | | || (_| ||  __/| (_| |          |
|                |______|/_/\_\ \__|\___||_| |_| \__,_| \___| \__,_|          |
|                                                                              |
+------------------------------------------------------------------------------+
```

本书配套视频课程《PLOE 编译器实现》在 B 站已经上线，点击 link 跳转观看。**如果觉得写的不错，请作者喝一杯咖啡也是极好的**。好心的捐赠大佬可以添加一下备注 “pl0e+ 昵称”，我会在收到后记录在本书的读者列表中，感谢！

第一章　前言

1.1　背景介绍

笔者在撰写本文的时间点是 2023 年 12 月左右。在撰写本文的十年前，也就是 2013 年 11 月到 2014 年初，笔者在 **非常吃力**地完成一个编译器的课程设计 [1] ，具体的时间点可以参考下面采样的 git 提交日志。

```
git --no-pager log --before='2014-01-01' --pretty=format:"[%ad] %h %s" \
    --date=format:'%Y-%m-%d %H:%M:%S' | sed -n '10,13p;25,30p;55,$p'
```

```
[2013-12-16 02:08:23] a653cdb add dag optimization
[2013-12-11 02:03:13] d944698 start bblock
[2013-12-11 00:31:47] 7533a55 add type checking
[2013-12-10 23:18:42] 10fb9bc Merge branch 'optimized'
[2013-11-29 01:50:05] 7390f01 add array reference
[2013-11-28 17:05:42] 024ee8d add reference files
[2013-11-28 03:08:57] fbcc5b0 asm.io
[2013-11-28 01:54:47] 8d84647 remove some unuseful files
[2013-11-28 01:32:01] 0d2f76d add mult and div
[2013-11-27 22:15:55] da06bb6 fix display macro bug
[2013-11-06 15:51:36] 509313d 词法分析第一版完成
[2013-11-06 13:51:14] ff2273b 修改了文法解读文档中关于过程调用及函数调用的一些bug
[2013-11-05 13:23:01] 8785d90 文法解读完成
[2013-11-05 13:20:17] a348249 文法解读完成
[2013-11-05 12:11:37] d69ce53 finish grammar diagram
[2013-11-03 20:32:07] 758d223 fit a bug in grammar, start to write getToken
[2013-11-03 04:23:44] 896fc31 starting writing code
[2013-11-03 00:32:50] 99bfff0 add a grammar doc
[2013-11-03 00:24:44] b5ea0c4 first commit
```

当时完成的编译器可以在 Ubuntu 12.04 操作系统中运行，使用到 nasm 2.09.10 作为汇编器、gcc 4.6.3 作为连接器，最终实现将源代码文件编译链接成可执行的二进制文件。虽然之前笔者的编程基础比较差，经过了无数个挑灯夜战，功夫不负有心人，最终跌跌撞撞地完成编译器的设计与实现。下面是当时完成后的最终感想：

[1] 编译器代码仓库 https://github.com/Jeanhwea/Compiler.git

进行了几周的努力，终于完成了拓展的 PL/0 编译器的建设。期间遇到的难关都在不断摸索中慢慢变得清晰。下面总结一下自己这几周的工作。

1. 进行了 PL/0 文法的解读和分析
2. 自己设计和不断修改四元式
3. 9000 多行的 c 代码
4. 自己学习了 x86 汇编（运行栈太难调了）
5. 熟悉了 Linux 编程，和使用 gcc 调试汇编

回想这些日子以来自己不知不觉地已经做了这么多的工作，编译器在自己的工作下一天天的强大，感觉很好。但是自己还是没有时间做太多的优化，首先是自己当时没有组织好数据结构。白白浪费了很多时间重整数据结构，这是比较繁琐的。四元式的设计也是不断迭代才得到的最终版。

学习是循序渐进的过程，我没有奢求一次就完成整个编译器的构建的野心。只有在不断调试之后我才获得更好的实现方式，同时自己的代码能力也是在不断提高。

最后的话，写这个编译器是很值的。

现在再看来，不免有点感慨 **时间如白驹过隙**，当人年纪大了就会怀恋逝去的时光，所以笔者打算重构一下之前编写的代码，并完成一本讨论编译器编程实践书籍，给有志于实现属于自己的编译器的读者提供参考。

1.2 前置基础

编译原理的学习需要一定的前置基础，主要包括以下几个方面：

1. 离散数学：编译原理中涉及到很多离散数学的概念，如图论、集合论、自动机等。掌握这些概念对于理解和实现编译器至关重要。
2. 数据结构与算法：编译器的实现需要用到各种数据结构和算法，如哈希表、栈、队列、深度优先搜索、广度优先搜索等。熟悉这些数据结构和算法的原理和实现方法对于编译器的设计和实现有很大的帮助。
3. 形式语言与自动机理论：编译原理的核心是形式语言与自动机理论，包括正则表达式、有限自动机、上下文无关文法、下推自动机、图灵机等。掌握这些理论可以帮助我们更好地理解编译器的原理和构造方法。
4. 计算机组成原理：编译器的输出是目标代码，而目标代码是与计算机硬件紧密相关的。因此，熟悉计算机组成原理，特别是指令系统、寄存器、内存等硬件组件的工作原理，对于理解编译器的目标代码生成和优化有很大的帮助。
5. 操作系统：编译器的实现需要与操作系统进行交互，如文件读写、进程管理、系统调用等。了解操作系统的基本原理和实现方法可以帮助我们更好地理解编译器的实现和运行环境。
6. 编程语言基础：掌握至少一门编程语言的基础语法和编程技巧，可以帮助我们更好地理解编译器的输入和输出，以及编译过程中各个阶段的作用和实现方法。

入门并实现一个简单的编译器并不需要太大门槛，在掌握如下技能后即可阅读本书。

1. 熟悉 C 语言
2. 理解基本数据结构实现
3. 熟悉 Linux 编程环境

1.3 编译阶段

在计算机世界中，源代码纯文本文件需要通过编译器处理才能获取可以直接在操作系统中运行的二进制可执行文件。这个编译器内部运行机理相当复杂，图 1.1 展示编译器的执行逻辑关系，编译过程可分为以下阶段。

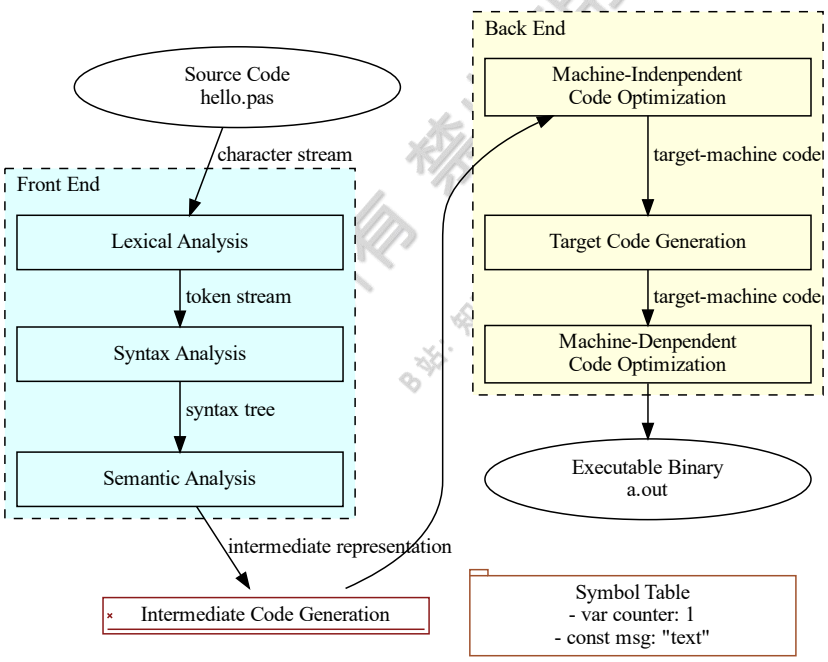


图 1.1: 编译器的执行阶段图

1. 词法分析（Lexical Analysis）：输入源程序，对构成源程序的字符串进行扫描和分解，识别出一个个的 token（亦称单词符号或简称符号），如基本字（begin、end、if、for、while），标识符、常数、运算符和界符（标点符号、左右括号）。单词符号是语言的基本组成成分，是人们理解和编写程序的基本要素。
2. 语法分析（Syntax Analysis）：根据语言的语法规则，由单词符号形成语法单位（如“短语”、“句子”、“程序段”、“程序” 等），由小到大，一层一层地逐步进行。通过语法分析，检查源程序在语法上是否正确，把源程序分解成语法的正确成分，即程序语句。
3. 语义分析（Semantic Analysis）：编译程序的语义分析阶段要对源程序的语法结构进行静态分析，检查源程序中的语义错误，并收集类型信息供后面的代码生成阶段使用。
4. 中间代码产生（Intermediate Code Generation）：中间代码是源程序的一种内部表示，或称

“中间语言”。这个中间语言使得编译器可以被分为前端 (Front End) 和后端 (Back End)。编译器前端负责产生中间代码，而后端负责生成目标代码。这样对于一个新出现的语言只需写出它的前端就可以了。中间代码也是编译器的前端和后端的分界点。

5. 代码优化 (Code Optimization)：编译程序中优化阶段的任务是对前阶段产生的中间代码进行变换或进行改造，目的是使生成的目标代码更为高效，即运行时间更短，占用的空间更小。其中代码优化根据是否与目标代码相关又分成以下两类：
   - 机器无关代码优化 (Machine-Independent Code Optimization)：它与特定的计算机体系结构无关，可以在多种不同的计算机上运行。这种优化技术主要关注源代码本身的逻辑结构和算法，通过优化算法和数据结构来提高程序的性能，而不考虑目标机器的具体实现。
   - 机器相关代码优化 (Machine-Dependent Code Optimization)：它针对特定的计算机体系结构进行优化，以生成更加高效的目标代码。这种优化技术需要考虑计算机的系统结构、指令系统、寄存器的分配以及内存的组织等因素，以充分利用计算机硬件的特性来提高程序的性能。
6. 目标代码生成 (Target Code Generation)：目标代码生成是编译的最后一个阶段。在生成目标代码时要考虑以下几个问题：计算机的系统结构、指令系统、寄存器的分配以及内存的组织等。

另外，在编译过程过程中，编译器需要维护一个符号表 (Symbol Table)，以记录程序中各个标识符的信息。当编译器遇到一个标识符时，它会在符号表中查找该标识符的信息，以确定该标识符的名称、类型和作用域等。这样，编译器就可以正确地处理程序中的各种语言构造，如变量声明、函数调用、控制结构等。

1.4 文法解读

1.4.1 文法定义

文法是指用于描述语言的语法结构的形式规则。任何一种语言都有它自己的文法，不管它是机器语言还是自然语言。文法可以判断句子结构是否符合规范，也就是说，根据一些规则，来确定编程语言的语法，从而实现编译器的功能。

PL/0 是 Pascal 语言 [2] 的一个子集，是一种较简单的程序设计语言。它具有简单易学、语法严谨、结构清晰等特点，是学习编程语言和编译器原理的入门语言之一。

PL/0 语言以赋值语句为基础，构造概念有顺序、条件和重复（循环）三种。PL/0 中唯一的数据类型是整型，可以用来说明该类型的常量和变量。本文根据 PL/0 文法扩展字符和字符串类型以及输入输出必要语法，称为 $PL/0\epsilon$ (PL/0 Extended)，作为后续实现的依据，扩展后的文法如下：

[2]Pascal 语言是一种高级编程语言，由法国数学家和工程师尼古拉斯·沃斯 (Niklaus Wirth) 于 1968 年设计并发布。Pascal 语言最初被设计用于教育和教学目的，以培养良好的编程风格和习惯。它具有严格的结构化和自顶向下的程序设计方法，强调程序的清晰性和可靠性。

$$
\begin{aligned}
\textit{program} &\rightarrow \textit{block}\ . && (1.1)\\
\textit{block} &\rightarrow [\ \textit{constdec}\ ][\ \textit{vardec}\ ]\{[\ \textit{procdec}\ ]|[\ \textit{fundec}\ ]\}\ \textit{compstmt} && (1.2)\\
\textit{constdec} &\rightarrow \textbf{const}\ \ \textit{constdef}\ \{,\ \textit{constdef}\ \};\ && (1.3)\\
\textit{constdef} &\rightarrow \textit{ident}\ =\ \textit{const} && (1.4)\\
\textit{vardec} &\rightarrow \textbf{var}\ \ \textit{vardef}\ ;\{\ \textit{vardef}\ ;\} && (1.5)\\
\textit{vardef} &\rightarrow \textit{ident}\ \{,\ \textit{ident}\ \}:\ \textit{type} && (1.6)\\
\textit{procdec} &\rightarrow \textit{prochead}\ \ \textit{block}\ \{;\ \textit{prochead}\ \ \textit{block}\ \};\ && (1.7)\\
\textit{prochead} &\rightarrow \textbf{procedure}\ \ \textit{ident}\ \ `(' [\ \textit{paralist}\ ]\ `)' ; && (1.8)\\
\textit{fundec} &\rightarrow \textit{funhead}\ \ \textit{block}\ \{;\ \textit{funhead}\ \ \textit{block}\ \};\ && (1.9)\\
\textit{funhead} &\rightarrow \textbf{function}\ \ \textit{ident}\ \ `(' [\ \textit{paralist}\ ]\ `)' :\ \textit{basictype}\ ; && (1.10)\\
\textit{paralist} &\rightarrow [\ \textbf{var}\ ]\ \textit{ident}\ \{,\ \textit{ident}\ \}:\ \textit{basictype}\ \{;\ \textit{paralist}\ \} && (1.11)\\
\textit{type} &\rightarrow \textit{basictype}\ |\ \textbf{array}\ \ `[' \ \textit{unsign}\ \ `]'\ \ \textbf{of}\ \ \textit{basictype} && (1.12)\\
\textit{basictype} &\rightarrow \textbf{integer}\ |\ \textbf{char} && (1.13)\\
\textit{const} &\rightarrow [+|-]\ \textit{unsign}\ |\ \textit{character} && (1.14)\\
\textit{character} &\rightarrow\ `\ \textit{letter}\ '\ |\ `\ \textit{digit}\ ' && (1.15)\\
\textit{string} &\rightarrow \texttt{"\{Printable ASCII characters exclude double quote\}"} && (1.16)\\
\textit{unsign} &\rightarrow \textit{digit}\ \{\ \textit{digit}\ \} && (1.17)\\
\textit{letter} &\rightarrow a|b|c|...|z|A|B|C|...|Z && (1.18)\\
\textit{digit} &\rightarrow 0|1|2|3|...|9 && (1.19)\\
\textit{expression} &\rightarrow [+|-]\ \textit{term}\ \{\ \textit{addop}\ \ \textit{term}\ \} && (1.20)\\
\textit{term} &\rightarrow\ \textit{factor}\ \{\ \textit{multop}\ \ \textit{factor}\ \} && (1.21)\\
\textit{factor} &\rightarrow\ \textit{ident}\ |\ \textit{ident}\ \ `[' \ \textit{expression}\ \ `]'\ |\ \textit{unsign}\\
&\quad |\ `(' \ \textit{expression}\ \ `)'\ |\ \textit{fcallstmt} && (1.22)\\
\textit{addop} &\rightarrow +|- && (1.23)\\
\textit{multop} &\rightarrow *|/ && (1.24)\\
\textit{statement} &\rightarrow\ \textit{assignstmt}\ |\ \textit{ifstmt}\ |\ \textit{repeatstmt}\ |\ \textit{pcallstmt}\\
&\quad |\ \textit{compstmt}\ |\ \textit{readstmt}\ |\ \textit{writestmt}\ |\ \textit{forstmt}\ |\ \textit{nullstmt} && (1.25)\\
\textit{assignstmt} &\rightarrow\ \textit{ident}\ :=\ \textit{expression}\ |\ \textit{funident}\ :=\ \textit{expression}\\
&\quad |\ \textit{ident}\ \ `[' \ \textit{expression}\ \ `]'\ :=\ \textit{expression} && (1.26)\\
\textit{funident} &\rightarrow\ \textit{ident} && (1.27)\\
\textit{ident} &\rightarrow\ \textit{letter}\ \{\ \textit{letter}\ |\ \textit{digit}\ \} && (1.28)\\
\textit{fcallstmt} &\rightarrow\ \textit{ident}\ \ `(' [\ \textit{arglist}\ ]\ `)' && (1.29)\\
\textit{arglist} &\rightarrow\ \textit{argument}\ \{,\ \textit{argument}\ \} && (1.30)\\
\textit{argument} &\rightarrow\ \textit{expression} && (1.31)\\
\textit{condition} &\rightarrow\ \textit{expression}\ \ \textit{relop}\ \ \textit{expression} && (1.32)
\end{aligned}
$$

$$relop \rightarrow \ < \mid <= \mid > \mid >= \mid = \mid <> \tag{1.33}$$

$$ifstmt \rightarrow \textbf{if} \ condition \ \textbf{then} \ statement \mid \textbf{if} \ condition \ \textbf{then} \ statement \ \textbf{else} \ statement \tag{1.34}$$

$$repeatstmt \rightarrow \textbf{repeat} \ statement \ \textbf{until} \ condition \tag{1.35}$$

$$forstmt \rightarrow \textbf{for} \ ident := expression \ (\ \textbf{to} \mid \textbf{downto} \ ) \ expression \ \textbf{do} \ statement \tag{1.36}$$

$$pcallstmt \rightarrow ident \ `(' \ [ \ arglist \ ] \ `)' \tag{1.37}$$

$$compstmt \rightarrow \textbf{begin} \ statement \ \{; \ statement \ \} \ \textbf{end} \tag{1.38}$$

$$readstmt \rightarrow \textbf{read} \ `(' \ ident \ \{, \ ident \ \} \ `)' \tag{1.39}$$

$$writestmt \rightarrow \textbf{write} \ `(' \ string \ , \ expression \ `)' \mid \textbf{write} \ `(' \ string \ `)' \mid \textbf{write} \ `(' \ expression \ `)' \tag{1.40}$$

$$nullstmt \rightarrow \tag{1.41}$$

上下文相关文法的概念由诺姆·乔姆斯基（Avram Noam Chomsky）[3] 在 1950 年代提出，用于描述自然语言的语法。在自然语言中，一个单词是否可以出现在特定位置上，依赖于其上下文。

一个文法根据是否具有上下文相关特性可以分成：上下文相关文法（Context-Sensitive Grammar）和上下文无关文法（Context-Free Grammar）。

上下文相关文法是一种比上下文无关文法更加一般性的形式文法，但仍然足够有序，可以被线性有界自动机所解析。上下文相关文法是一种更一般性的形式文法，能够更好地描述自然语言的语法和语义信息，例如：我们常见的英语和汉语都是上下文相关文法。

上下文无关文法定义了一个形式语言中所有可能的合法句子的结构和规则。上下文无关文法是由生成规则和终结符集合组成的，能够产生符合特定语法规则的句子。

上下文无关文法的组成部分包括：终结符（包括所有出现在句子中的单词或标点符号）、非终结符（表示一组可能的词序列的符号）、起始符号（整个文法的根节点）以及产生式规则（定义了由一个非终结符生成一个字符串的过程）。

上下文无关文法的特点是，语法范畴（或语法单位）是完全独立于这种范畴可能出现的环境。例如，在程序设计语言中，当碰到一个算术表达式时，我们完全可以 “就事论事” 处理，而不必考虑它所处的上下文。然而，在自然语言中，随便一个词甚至一个字的意思在不同的上下文中都可能有不同的意思。

上下文无关文法的应用包括但不限于编译器设计、自然语言处理、形式语言理论等。通过建立适当的文法规则，可以对分析出的句子进行语法上的分析，从而进一步实现对语义信息的抽取和理解。总之，上下文无关文法是一种重要的基础语法规则，它对于自然语言处理等领域的技术应用具有重要的意义。

$PL/0\epsilon$ 的文法属于上下文无关文法，后序的小节将会对该文法描述的语言进行详细解读。

[3]美国哲学家、语言学家、政治家和心理学家，是美国历史上最具影响力的知识分子之一。他被誉为是现代语言学之父，是 20 世纪理论语言学研究上最伟大的贡献者之一

1.4.2 程序

文法（1.1）定义了 $PL/0\epsilon$ 的程序基本结构。为了更好地理解语言的语法结构，我们这里介绍一种可以直观地表示语法结构的图示，也就是语法图。它通常由节点和边组成，节点表示语法元素，边表示语法关系。图 1.2 定义了 $PL/0\epsilon$ 的程序 program 的基本构成，即程序是由分程序 block 加 . 组成，其中 . 可以判定程序的结束。

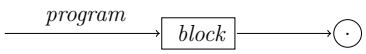


图 1.2: 程序 program 语法图

下面是一个 $PL/0\epsilon$ 程序的例子，它只有一行就是一个完整的程序。

```
const counter = 2; .
```

它定义了一个常量 counter 并将其初始化成 2 ，最后的 . 标记程序结束。

1.4.3 分程序

文法（1.2）-（1.11）定义了分程序 block 的结构。图 1.3 是分程序的语法图，通过分析可以得出，分程序 block 先进行常量定义 constdec，然后是变量定义 vardef ，接着是过程的声明 procdec 和函数声明 fundec 最终的 block 通过一个复合语句 compstmt 退出。这样的定义顺序是不能改变的。

我们通过一个具体分程序定义的例子来分析其结构，每个部分的说明如下：

1. 首先是常量和变量的定义
   - 常量定义必须在变量定义前面，这种顺序不能改变，
   - 常量是可以连续定义的，之间使用逗号隔开，最后以分号结束常量的定义，
   - 变量的定义也可以连续定义，不同变量定义之间使用逗号隔开，
   - 变量定义使用冒号后跟类型来说明定义的变量类型，
   - 变量的定义的结束是使用分号。
2. 接着是过程和函数的声明，文法中描述过程和函数定义顺序是可以改变的。
   - 过程定义以 procedure 关键字起始，然后定义参数等，
   - 函数定义以 function 关键字起始，然后定义参数等，
   - 过程和函数不同的是，函数具有返回值，但是过程没有返回值，
   - 过程和函数都可以右参数列表，用于传入参数。
3. 然后是复合语句，这个复合语句也是分程序的执行入口
4. 常量定义，变量定义，过程声明，函数声明对一个分程序来说是可有可无的，只有复合语句是必须部分。

```
const a = 0, b = 1;          { 常量定义 }
var i, j, sum: integer;      { 变量定义 }

procedure hello();           { 过程声明部分 }
```

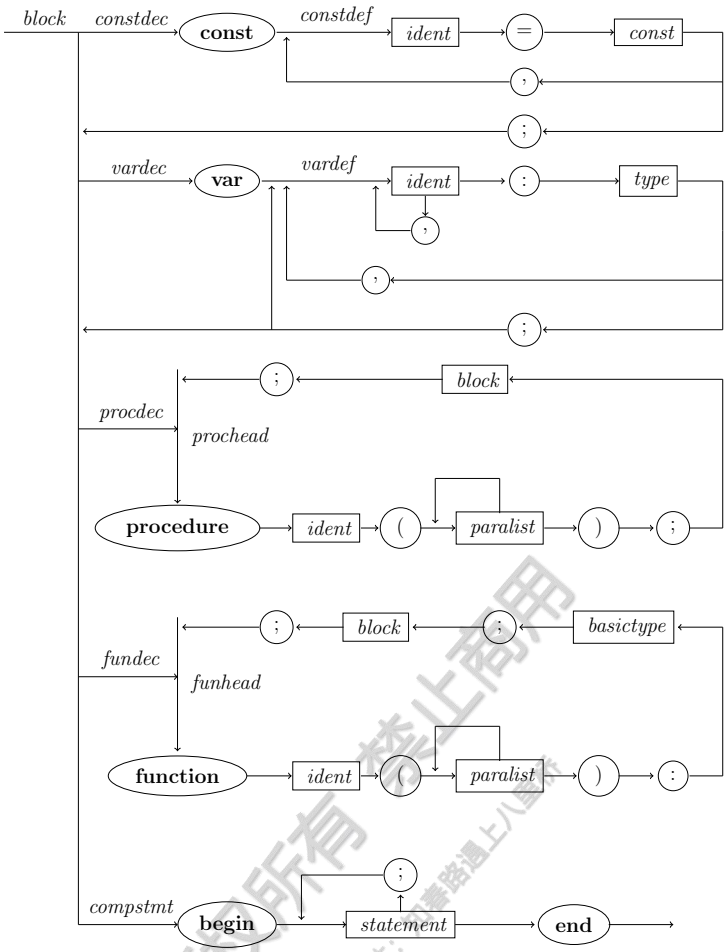


图 1.3: 分程序 block 语法图

```
5   begin write("hello") end;
6
7  function add():integer;      { 函数声明部分 }
8    begin
9      sum := a + b;
10     add := sum
11   end;
12
13 begin                        { 复合语句部分 }
14   i := sum
15 end
```

1.4.4 关键字

关键字是被编程语言所预留并有特殊含义的标识符和字。它们是编程语言的一部分，用于表示特定的功能或操作。关键字也被称为保留字，这些保留字不能作为常量名、变量名或其他标识符名称。$PL/0\epsilon$ 语言所使用的关键字见表 1.1。

表 1.1: $PL/0\epsilon$ 语言所有关键字列表

array	**begin**	**char**	**const**	**do**
downto	**else**	**end**	**for**	**function**
if	**integer**	**of**	**procedure**	**read**
repeat	**then**	**to**	**until**	**var**
write				

1.4.5 类型系统

类型系统是编程语言中用于定义数据类型、操作和约束的规则和机制。类型系统的主要目的是提供一种方式来确保程序的安全性和正确性，通过在编译时或运行时检查数据类型，以防止错误的数据类型操作或访问。

类型系统可以分为静态类型系统和动态类型系统。

- 静态类型系统：在编译时进行类型检查。这种类型的系统要求程序员在编写代码时明确指定每个变量的类型，并在编译过程中进行类型检查。如果变量的类型不匹配，编译器将报错。静态类型系统的代表语言包括 C、C++、Java 等。
- 动态类型系统：在运行时进行类型检查。这种类型的系统不需要程序员在编写代码时指定变量的类型，而是在运行时动态确定变量的类型。动态类型系统的代表语言包括 Python、Ruby 等。

文法 (1.12) - (1.19) 定义了 $PL/0\epsilon$ 的类型系统。$PL/0\epsilon$ 是静态类型系统，由于简单仅支持两种类型，具体类型系统的含义说明如下：

1. $PL/0\epsilon$ 的类型分为两类：基本类型、字符串和数组。
2. 基本类型包括整型和字符型，分别由 **integer** 和 **char** 关键字修饰
   - **integer** 表示整型，它包含正负的整数 [4] 。
   - **char** 表示字符型，字符型包含数字位和大小写字母，以单引号 ' 隔开。
3. 字符串包含可以打印的 ASCII 字符，但是不包括双引号 " 字符
   - 其字符范围由十进制从 32 到 126（不包含 34）的所有的值，参考表 1.2 。
   - 这样有利于词法分析的状态机的设计。
   - 另外字符串不是一种基本类型，只用于写语句的打印。
4. 常量的定义可以是正负整数，也可以是字符。若是字符，则会使用 ASCII 码值进行运算。

1.4.6 表达式、项和因子

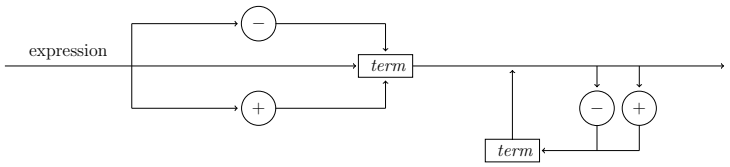
文法 (1.20) - (1.24) 定义了表达式 expression、项 term 和因子 factor 的语法，它们之间有着密切的关系，通过它们之间的关系我们可以描述小学数学中的加减乘除算术四则运算，并且表达式、项和因子都是有值的，通过定义它们的求值方式我们就可以构成编程语言中的求值操作。

图 1.4(a) 描述了表达式的语法结构。表达式是一个数学表达式，它由数字、变量、运算符和

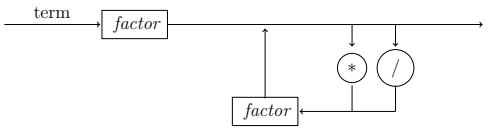
[4] 在 32 位机器上，这里的整数指的是 32 位整数，它的取值范围为 $[-2^{31}, 2^{31}-1]$ 。

表 1.2: ASCII 表: 从 32 到 126 (去除 34) 范围字符是字符串的合法字符

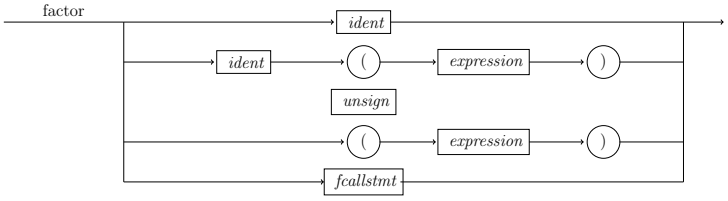
0	nul	1	soh	2	stx	3	etx	4	eot	5	enq	6	ack	7	bel
8	bs	9	ht	10	nl	11	vt	12	np	13	cr	14	so	15	si
16	dle	17	dc1	18	dc2	19	dc3	20	dc4	21	nak	22	syn	23	etb
24	can	25	em	26	sub	27	esc	28	fs	29	gs	30	rs	31	us
32	sp	33	!	34	"	35	#	36	$	37	%	38	&	39	'
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	del



(a) 表达式 expression 语法图



(b) 项 term 语法图



(c) 因子 factor 语法图

图 1.4: 表达式、项和因子的语法图

括号等组成。表达式可以代表一个数值或一个方程式，其目的是将信息组织起来并进行计算。下面都是合法的表达式。

```
22
9*y
4*x + y/5
```

图 1.4(b) 描述了项的语法结构。项是表达式的组成部分，它由一个或多个因子组成，并由一个运算符或一个括号分隔。每个项可以是一个单独的数字或变量，也可以是几个数字和变量的乘积。项在表达式的计算中扮演着重要的角色。下面都是合法的项。

```
3.14
y
x*y
x*(y+1)
```

图 1.4(c) 描述了因子的语法结构。因子是项的基本组成部分，它可以是数字或变量。因子可以被视为一个独立的单位，可以与其他的因子组合成项。每个项可以包含一个或多个因子，而每个因子也可以是其他项的一部分。下面都是合法的因子。

```
3.11
x
a[0-1]
(x+8)
add(x,y)
```


1.4.7 语句

文法 (1.25) - (1.41) 定义了语句 statement 的语法。语句是构成源程序的基本单位，它们代表了程序中的各个命令和操作。图 1.5 描述了语句的语法结构，可以看出语句包含赋值语句 assignstmt，条件语句 ifstmt，重复语句 repeatstmt，过程调用语句 pcallstmt，复合语句 compstmt，读语句 readstmt，写语句 writestmt，循环语句 forstmt 和空语句 nullstmt 。

$PL/0\epsilon$ 的语句结构有以下注意点。

1. 赋值语句可以是表达式给变量的赋值，可以是表达式给数组赋值，还可以给表达式给函数给返回值。
2. 条件语句的 **else** 悬挂的解决方法是总将 **else** 和最近的 **if** 进行匹配。
3. 循环语句的步长设为一，通过 **for** 开始， **to** 表示变量值加一， **downto** 表示变量值减一。
4. 读语句，写语句用于对查询进行输入输出操作。
5. 过程调用语句用于唤醒其他的过程定义。
6. 复合语句被 **begin** 和 **end** 围起来，之间是以分号 ; 隔开。
7. 空语句指什么都没有定义的语句，例如: ;; 也是合法的语句，两个分号之间的就是空语句。

```
const a=1, b=2, c=11, i=1;
```

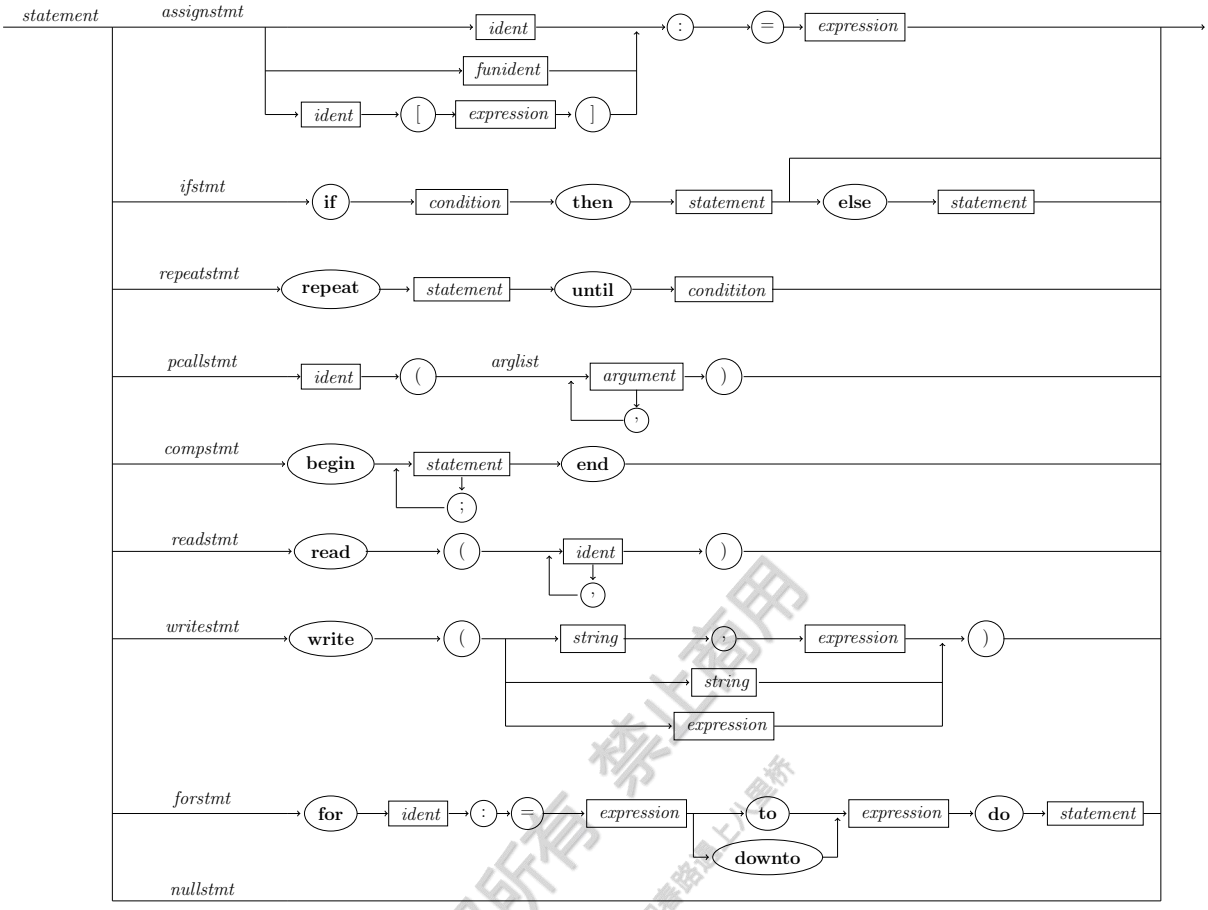


图 1.5: 语句 statement 语法图

```
var sum, k: integer; A:array[3] of integer;
procedure testproc(); begin write("...") end;
begin
  { 赋值语句 }
  sum := a + b;
  A[i] := b;
  { 条件语句 }
  if a > b then
    sum := a;
  else
    sum := b;
  { 重复语句 }
  repeat
    sum := sum + 1;
  until sum > 4
  { for 语句 (步长为一) }
  for i := 1 to 10 do
    sum := sum + i
```

```
20    { 过程调用语句 }
21    testproc();
22    { 读语句 }
23    read (k, sum);
24    { 写语句 }
25    write("hello world!");
26    write(sum)
27  end.
```

上述代码片段是一些合法的语句。宏观上看，第 1 到 3 行定义了一些常量和变量和一个过程，第 4 行到第 27 行是一个复合语句。微观上来看，复合语句包含许多常见的语句。其中第 6 行是一个赋值语句，将 a+b 结果写入变量 sum 中，第 7 行也是一个赋值语句，将 b 的值写入数组 A[i] 中。第 9 到 12 行是条件语句，通过关键字 **if** 、**then** 和 **else** 定义了一个完整的条件控制，如果 a>b 条件满足，则执行第 10 行的语句，否则执行第 12 行的语句。第 14 到 16 行是重复语句，它的语义是对变量 sum 进行累加，当满足条件 sum > 4 后跳出重复执行。第 18 到 19 行是循环语句，变量 i 是循环变量，它的语义是对 sum 进行累加 1 到 10 这个十个数字。第 21 行是过程调用语句，调用的过程名为 testproc 。最后，第 23 到 26 行是读语句和写语句，第 23 行读取变量 k 和 sum 到变量中，第 25 行输出 hello world! 字符串，第 26 行输出变量 sum 的值。

1.5 本章总结

本章主要介绍了实现编译器的背景，分析了编译器各个阶段的关联关系，最后解读了 $PL/0\epsilon$ 语言的文法。

第二章　词法分析

2.1　词法分析

词法分析是编译器的一个关键组成部分，如图 2.1 所示，它负责将输入的源代码分解成一系列的词法单元或 token。词法分析器通常被称为扫描器或词法器。另外，词法分析还可以使用 Lex [1] 等工具自动生成。

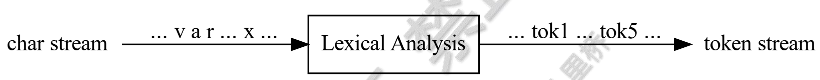


图 2.1：词法分析示意图

以下是词法分析的一般说明：

1. 输入：词法分析器接收源代码作为输入。这可以是一段用特定编程语言编写的代码。
2. 词法分析：词法分析器将源代码分解成一系列的词法单元。每个词法单元都是源代码中的一个原子元素，例如关键字、标识符、运算符、常量、标点符号等。
3. 词法规则：词法分析器使用一组预定义的词法规则来识别和分类词法单元。这些规则通常定义在语言的语法规范中。
4. 标记化：词法分析器将每个词法单元转换成一个标记（token）。每个标记都有一个类型和一个值。类型表示词法单元的类别（例如，关键字、标识符等），值表示词法单元的实际内容。
5. 错误处理：在词法分析过程中，可能会遇到语法错误，例如未定义的标识符、无效的字符等。词法分析器需要能够识别和处理这些错误，通常是通过报告错误位置和类型。
6. 输出：词法分析器的输出是一组标记的序列，这个序列代表了源代码的词法结构。这个输出通常传递给语法分析器，后者进一步处理这些标记以构建语法树。

词法分析是编译过程中的第一步，它为后续的语法分析和语义分析提供了基础。通过将源代码分解成一系列的词法单元，编译器能够更容易地理解和处理代码的结构和语义。其中 $PL/0\epsilon$ 语言的涉及到的 token 见表 2.1 ，从表中可以看出一共有 50 个 token 类型，通过使用功能分成终结符，关键字，多字节和特殊字符四个子类。

对于词法分析中 token 的定义在文件 `include/lexical.h` 中，其核心就是定义了一个 `token_t` 的枚举类型，用来标记词法分析中可以失败到的所以 token 类型。具体代码如下：

[1]注意 flex 一个是 Lex 的开源实现

表 2.1: $PL/0\epsilon$ 语言所有 token 列表

终结符	0	ENDFILE	1	ERROR			
关键字	2	KW_ARRAY	3	KW_BEGIN	4	KW_CHAR	5	KW_CONST
	6	KW_DO	7	KW_DOWNTO	8	KW_ELSE	9	KW_END
	10	KW_FOR	11	KW_FUNCTION	12	KW_IF	13	KW_INTEGER
	14	KW_OF	15	KW_PROCEDURE	16	KW_READ	17	KW_REPEAT
	18	KW_THEN	19	KW_TO	20	KW_UNTIL	21	KW_VAR
	22	KW_WRITE					
多字节	23	MC_ID	24	MC_CH	25	MC_UNS	26	MC_STR
特殊字符	27	SS_PLUS	28	SS_MINUS	29	SS_STAR	30	SS_OVER
	31	SS_EQU	32	SS_LST	33	SS_LEQ	34	SS_GTT
	35	SS_GEQ	36	SS_NEQ	37	SS_COMMA	38	SS_SEMI
	39	SS_ASGN	40	SS_LPAR	41	SS_RPAR	42	SS_LBRA
	43	SS_RBRA	44	SS_LBBR	45	SS_RBBR	46	SS_SQUO
	47	SS_DQUO	48	SS_COLON	49	SS_DOT		


```c
// Define all token enumeration
typedef enum _token_enum {
        // Book-keeping Token
        /*  0 */ ENDFILE = 0,
        /*  1 */ ERROR = 1,
        // 省略部分行
        /* 48 */ SS_COLON = 48,
        /* 49 */ SS_DOT = 49
} token_t;
```

2.2 文件行缓冲

文件行缓冲是一种读取文件的方式，它使用缓冲区来存储已经读取的行，然后一次性返回缓冲区中的所有行。这种方式可以提高读取文件的效率，特别是在处理大文件时，可以减少 IO 操作的次数，从而提高程序的性能。具体实现见 `include/scan.h` 和 `source/scan.c` 。在 `scan.c` 中定义了缓冲变量，其中 `linebuf` 数组作为当前读取行的缓冲区

```c
char linebuf[MAXLINEBUF];
int bufsize = 0;
// when meet EOF, then set done to TRUE
static bool fileend = FALSE;
// hold file scan postion (line, column)
int lineno = 0;
int colmno = 0;
```

在 scan.h 中还定义了读取一个字符的函数 readc() 以及逆操作函数 unreadc()

```
static int readc(bool peek);
static void unreadc(void);
```

使用缓冲区按行读取文件时，readc() 检查当的列数 colmno 释放超出 bufsize，如果未超出则返回一个字符，否则调用 fgets() 库函数来加载文件中的一行数据到 linebuf 缓冲区中。它的具体代码如下：

```
// read a character
static int readc(bool peek)
{
        if (colmno < bufsize) {
                goto ready;
        }

        lineno++;
        if (fgets(linebuf, MAXLINEBUF - 1, source) == NULL) {
                fileend = TRUE;
                return EOF;
        }
        dbg("source L%03d: %s", lineno, linebuf);

        bufsize = strlen(linebuf);
        colmno = 0;
        goto ready;

ready:
        return (peek) ? linebuf[colmno] : linebuf[colmno++];
}

// unread a charachter
static void unreadc(void)
{
        if (colmno <= 0) {
                panic("unread at line postion zero!");
        }
        if (!fileend) {
                colmno--;
        }
}
```

需要注意的是，在使用文件行缓冲时，unread() 函数可能会因为 linebuf 中没有数据而失败。

2.3 关键字及 MC_ID 解析

$PL/0\epsilon$ 中包含一些关键字，具体见表 1.1 。在 scan.c 中定义了关键字表，具体代码如下：

```
static struct _pl0e_keywords_struct {
        // keyword string
        char *str;
        // represented token
        token_t tok;
} PL0E_KEYWORDS[] = {
        /*  0 */ { "array", KW_ARRAY },
        /*  1 */ { "begin", KW_BEGIN },
        /*  2 */ { "char", KW_CHAR },
        /*  3 */ { "const", KW_CONST },
        /*  4 */ { "do", KW_DO },
        /*  5 */ { "downto", KW_DOWNTO },
        /*  6 */ { "else", KW_ELSE },
        /*  7 */ { "end", KW_END },
        /*  8 */ { "for", KW_FOR },
        /*  9 */ { "function", KW_FUNCTION },
        /* 10 */ { "if", KW_IF },
        /* 11 */ { "integer", KW_INTEGER },
        /* 12 */ { "of", KW_OF },
        /* 13 */ { "procedure", KW_PROCEDURE },
        /* 14 */ { "read", KW_READ },
        /* 15 */ { "repeat", KW_REPEAT },
        /* 16 */ { "then", KW_THEN },
        /* 17 */ { "to", KW_TO },
        /* 18 */ { "until", KW_UNTIL },
        /* 19 */ { "var", KW_VAR },
        /* 20 */ { "write", KW_WRITE }
};
```

getkw() 函数通过遍历方式搜索关键字表 PL0E_KEYWORDS 中的每一个关键字，如果发现是关键字返回对应的 token 类型，否则返回 MC_ID 表示是一个标识符。

```
#define MAXRESERVED (sizeof(PL0E_KEYWORDS) / sizeof(PL0E_KEYWORDS[0]))

static token_t getkw(char *s)
{
        int i;
        for (i = 0; i < MAXRESERVED; i++) {
                if (!strcmp(s, PL0E_KEYWORDS[i].str)) {
                        return PL0E_KEYWORDS[i].tok;
```

```
                }
        }
        return MC_ID;
}
```

2.4 gettok 有限状态机

有限状态机（Finite State Machine，FSM）是一种数学模型，用于描述有限个状态以及这些状态之间的转移和动作等行为。它是一种离散数学中的概念，广泛应用于计算机科学、自动化控制、通信系统等领域。

有限状态机由一组有限的状态集合、一个起始状态、一组输入、一组转换函数和一个或多个终止状态组成。当系统接收到输入时，它会根据当前状态和输入，按照转换函数的规则，转移到下一个状态。状态机可以表示为一个有向图，其中节点表示状态，边表示状态之间的转换。

有限状态机的行为可以描述为：在给定当前状态和输入的情况下，有限状态机会根据转换函数的规则，更新其状态并执行相应的动作。转换函数通常包括输入条件、当前状态和下一个状态之间的关系。

$PL/0\epsilon$ 的词法分析中也需要包含一个有限状态机。它的状态在 scan.h 文件中定义，代码如下：

```
// gettok states
typedef enum _state_enum {
        /* 0 */ START,
        /* 1 */ INSTR,
        /* 2 */ INUNS,
        /* 3 */ INIDE,
        /* 4 */ INLES,
        /* 5 */ INCOM,
        /* 6 */ INGRE,
        /* 7 */ INCHA,
        /* 8 */ INCMT,
        /* 9 */ DONE
} state_t;
```

状态机的初始状态是 START ，结束状态是 DONE 。其它的一系列以 IN 开头的状态是中间状态，它们的含义如下：

- INSTR 表示当前处于解析字符串
- INUNS 表示当前解析无符号数字
- INIDE 表示当前解析标识符（identity）
- INLES 表示当前解析小于号 <
- INCOM 表示当前解析逗号 ,

- INGRE 表示当前解析大于号 >
- INCHA 表示当前解析字符类型
- INCMT 表示当前解析注释

除了对状态机的状态定义，图 2.2 还描述了状态之间的迁移关系，称为状态迁移图。图中的节点表示一个状态，边表示一个状态迁移到下一个状态，边中还包含迁移条件。值得注意的的是，同一个状态自身到自身的迁移也是合法的。

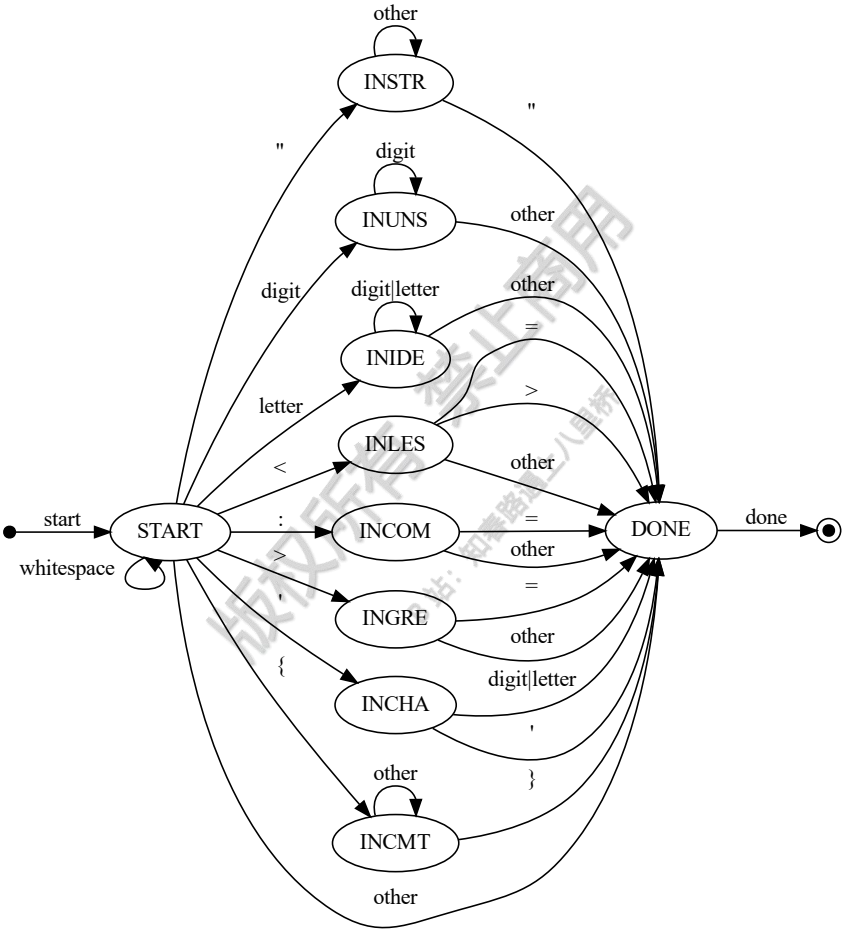


图 2.2: 词法分析状态迁移图

词法解析的核心函数是 gettok()，它返回一个 token_t 类型作为当前的 token。并将当前 token 的字符串放入 tokbuf[] [2] 数组中缓冲，同时记录 token 所在的行号，将行号值存入 toklineno 变量中。

gettoken() 的实现细节比较冗长，但是其思想还是比较简单，主要是实现图 2.2 中所描述的状态机。第 18 行开始的 while 循环是读取状态的主循环，每次通过 readc() 读取一个字符放入 ch 变量中，然后第 22 到 197 行都是对不同状态迁移的处理，细节是使用 switch-case 和 if-else 语句对 ch 变量的各种情况检查，完成逻辑分支切换。每次处理一个字符后，在第 200 到

[2]在编译原理中，这也被称作词位（lexeme）

205 行中将字符到 tokbuf[] 数组，以备后序使用。与通用的处理不同，第 208 到 213 行是后置处理工作，如果发现是 MC_ID 类型的 token 还需要额外调用 getkw() 进一步判断是否为关键字。

```
1  char tokbuf[MAXTOKSIZE + 1];
2  int toklineno;
3
4  // get next token
5  token_t gettok(void)
6  {
7          // token buffer index
8          int i = 0;
9          // current token
10         token_t curr;
11         // whether save current character to tokbuf[...]
12         bool save;
13
14         // the state of our state machine
15         state_t state = START;
16
17         // the state machine main loop
18         while (state != DONE) {
19                 int ch g= readc(FALSE);
20                 save = TRUE;
21                 // state machine
22                 switch (state) {
23                 case START:
24                         if (isspace(ch)) {
25                                 save = FALSE;
26                         } else if (isdigit(ch)) {
27                                 state = INUNS;
28                         } else if (ch == '"') {
29                                 save = FALSE;
30                                 state = INSTR;
31                         } else if (ch == '\'') {
32                                 save = FALSE;
33                                 state = INCHA;
34                         } else if (ch == '{') {
35                                 save = FALSE;
36                                 state = INCMT;
37                         } else if (isalpha(ch)) {
38                                 state = INIDE;
39                         } else if (ch == ':') {
40                                 state = INCOM;
```

```
41                      } else if (ch == '>') {
42                              state = INGRE;
43                      } else if (ch == '<') {
44                              state = INLES;
45                      } else {
46                              state = DONE;
47                              switch (ch) {
48                              case EOF:
49                                      save = FALSE;
50                                      curr = ENDFILE;
51                                      break;
52                              case '.':
53                                      curr = SS_DOT;
54                                      break;
55                              case '+':
56                                      curr = SS_PLUS;
57                                      break;
58                              case '-':
59                                      curr = SS_MINUS;
60                                      break;
61                              case '*':
62                                      curr = SS_STAR;
63                                      break;
64                              case '/':
65                                      curr = SS_OVER;
66                                      break;
67                              case '=':
68                                      curr = SS_EQU;
69                                      break;
70                              case ',':
71                                      curr = SS_COMMA;
72                                      break;
73                              case ';':
74                                      curr = SS_SEMI;
75                                      break;
76                              case '(':
77                                      curr = SS_LPAR;
78                                      break;
79                              case ')':
80                                      curr = SS_RPAR;
81                                      break;
82                              case '[':
```

```
83                                      curr = SS_LBRA;
84                                      break;
85                              case ']':
86                                      curr = SS_RBRA;
87                                      break;
88                              case '{':
89                                      curr = SS_LBBR;
90                                      break;
91                              case '}':
92                                      curr = SS_RBBR;
93                                      break;
94                              default:
95                                      curr = ERROR;
96                                      break;
97                              }
98                      }
99                      break;
100             case INCMT: /* in comment */
101                     save = FALSE;
102                     if (ch == EOF) {
103                             state = DONE;
104                             curr = ENDFILE;
105                     } else if (ch == '}') {
106                             state = START;
107                     }
108                     break;
109             case INSTR: /* in string */
110                     if (ch == '"') {
111                             state = DONE;
112                             save = FALSE;
113                             curr = MC_STR;
114                     } else if (isprint(ch)) {
115                             // only allow printable character
116                     } else if (!isprint(ch)) {
117                             panic("unprintable character");
118                     } else {
119                             state = DONE;
120                             if (ch == EOF) {
121                                     save = FALSE;
122                                     i = 0;
123                                     curr = ENDFILE;
124                             }
```

```
125                     }
126                     break;
127             case INCHA: /* in character */
128                     if (ch == '\'') {
129                             state = DONE;
130                             save = FALSE;
131                             curr = MC_CH;
132                     } else if (isdigit(ch) || isalpha(ch)) {
133                             // skip case
134                     } else {
135                             if (ch == EOF) {
136                                     save = FALSE;
137                                     i = 0;
138                                     curr = ENDFILE;
139                                     state = DONE;
140                             }
141                     }
142                     break;
143             case INUNS: /* in unsign number */
144                     if (!isdigit(ch)) {
145                             unreadc();
146                             save = FALSE;
147                             state = DONE;
148                             curr = MC_UNS;
149                     }
150                     break;
151             case INIDE: /* in identifier */
152                     if (!(isdigit(ch) || isalpha(ch))) {
153                             unreadc();
154                             save = FALSE;
155                             state = DONE;
156                             curr = MC_ID;
157                     }
158                     break;
159             case INLES: /* in less than */
160                     state = DONE;
161                     if (ch == '=') {
162                             curr = SS_LEQ;
163                     } else if (ch == '>') {
164                             curr = SS_NEQ;
165                     } else {
166                             unreadc();
```

```
167                             save = FALSE;
168                             curr = SS_LST;
169                     }
170                     break;
171             case INCOM: /* in comma */
172                     state = DONE;
173                     if (ch == '=') {
174                             curr = SS_ASGN;
175                     } else {
176                             unreadc();
177                             save = FALSE;
178                             curr = SS_COLON;
179                     }
180                     break;
181             case INGRE: /* in great than */
182                     state = DONE;
183                     if (ch == '=') {
184                             curr = SS_GEQ;
185                     } else {
186                             unreadc();
187                             save = FALSE;
188                             curr = SS_GTT;
189                     }
190                     break;
191             case DONE:
192             default:
193                     dbg("error state = %d", state);
194                     state = DONE;
195                     curr = ERROR;
196                     break;
197             }
198
199             // save ch to tokbuf[...]
200             if ((save) && (i <= MAXTOKSIZE)) {
201                     tokbuf[i++] = (char)ch;
202                     tokbuf[i] = '\0';
203             } else if (i > MAXTOKSIZE) {
204                     dbg("token size is too long, lineno = %d\n", lineno);
205             }
206
207             // post-processing works
208             if (state == DONE) {
```

```
209                         tokbuf[i] = '\0';
210                         toklineno = lineno;
211                         if (curr == MC_ID) {
212                                 curr = getkw(tokbuf);
213                         }
214                 }
215         }
216
217         dbg("token=%2d, buf=[%s], pos=%d:%d\n", curr, tokbuf, lineno, colmno);
218         return curr;
219 }
```

2.5 测试用例

我们通过一个简单的 $PL/0\epsilon$ 代码来测试词法分析的实现是否正常，测试程序为 ch2tok.c 文件，它循环读取文件，并通过 gettok() 获取 token 并将其打印出来。

```
1  int main(int argc, char *argv[])
2  {
3          echo = 0;
4          silent = 1;
5          init(argc, argv);
6          int counter = 0;
7
8          token_t tok;
9          while ((tok = gettok()) != ENDFILE) {
10                 printf("%03d: token=%d, buf=[%s]\n", ++counter, tok, tokbuf);
11         }
12
13         return 0;
14 }
```

编写一个简单的测试用例，将如下代码写入 simple.pas 文件中。

```
1  var ans : integer;
2  begin
3     ans := 1 + 2;
4  end
5  .
```

ch2tok 可以将识别到的 token 流结果打印出来，运行结果如下：

```
./bin/ch2tok ./examples/simple.pas
```

```
001: token=21, buf=[var]
002: token=23, buf=[ans]
003: token=48, buf=[:]
004: token=13, buf=[integer]
005: token=38, buf=[;]
006: token=3, buf=[begin]
007: token=23, buf=[ans]
008: token=39, buf=[:=]
009: token=25, buf=[1]
010: token=27, buf=[+]
011: token=25, buf=[2]
012: token=38, buf=[;]
013: token=9, buf=[end]
014: token=49, buf=[.]
```

另外，drawtok 可视化解析每行代码的程序，可以通过其输出更好的理解词法分析是如何把字符流转成 token 流的。下面是解析 simple.pas 的输出结果

```
./bin/drawtok ./examples/simple.pas

----------------------------------------------------------
#001 LINE001: var ans : integer;
              ^^^
                 `-- token=21, buf=[var]
----------------------------------------------------------
#002 LINE001: var ans : integer;
                  ^^^
                     `-- token=23, buf=[ans]
----------------------------------------------------------
#003 LINE001: var ans : integer;
                      ^
                       `-- token=48, buf=[:]
----------------------------------------------------------
#004 LINE001: var ans : integer;
                        ^^^^^^^
                               `-- token=13, buf=[integer]
----------------------------------------------------------
#005 LINE001: var ans : integer;
                               ^
                                `-- token=38, buf=[;]
----------------------------------------------------------
#006 LINE002: begin
              ^^^^^
                   `-- token=3, buf=[begin]
```

```
------------------------------------------------------------
#007 LINE003:    ans := 1 + 2;
                 ^^^
                    `-- token=23, buf=[ans]
------------------------------------------------------------
#008 LINE003:    ans := 1 + 2;
                     ^^
                       `-- token=39, buf=[:=]
------------------------------------------------------------
#009 LINE003:    ans := 1 + 2;
                        ^
                         `-- token=25, buf=[1]
------------------------------------------------------------
#010 LINE003:    ans := 1 + 2;
                          ^
                           `-- token=27, buf=[+]
------------------------------------------------------------
#011 LINE003:    ans := 1 + 2;
                            ^
                             `-- token=25, buf=[2]
------------------------------------------------------------
#012 LINE003:    ans := 1 + 2;
                              ^
                               `-- token=38, buf=[;]
------------------------------------------------------------
#013 LINE004: end
              ^^^
                 `-- token=9, buf=[end]
------------------------------------------------------------
#014 LINE005: .
              ^
               `-- token=49, buf=[.]
```


2.6 本章总结

本章主要介绍了编译原理的词法分析阶段功能、源文件读取和解析 gettok 有限状态机的实现细节，最后通过测试的方式来验证词法分析功能的正确性。

第三章　语法分析

3.1　语法分析

如图 3.1 所示，语法分析是编译过程中的一个逻辑阶段，主要任务是在词法分析的基础上，将单词序列组合成各类语法短语，如 “程序”、“语句”、“表达式” 等等。语法分析程序判断源程序在结构上是否正确，源程序的结构由上下文无关文法描述。

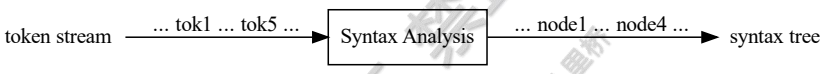


图 3.1: 语法分析示意图

语法分析程序可以用 YACC 等工具自动生成。完成语法分析任务的程序称为语法分析器，或语法分析程序。按照源语言的语法规则，从词法分析的结果中识别出相应的语法范畴，同时进行语法检查。

3.2　抽象语法树

抽象语法树（Abstract Syntax Tree）是源代码的抽象语法结构的树状表现形式 [1]，它的每个节点都表示源代码中的一种结构。这里的 “抽象” 是因为抽象语法树并不会表示出真实语法出现的每一个细节，比如说，嵌套括号被隐含在树的结构中，并没有以节点的形式呈现。

抽象语法树是一种以树形的方式表现语法结构的形式，它通常由语法分析器（parser）在将源代码转换成抽象语法的过程中生成。抽象语法树的每个节点都表示源代码中的一种结构，例如表达式、语句、函数定义等。

抽象语法树对于编译器和解释器的设计和实现非常重要。它可以帮助编译器和解释器更好地理解源代码的结构和语义，从而进行正确的编译或解释。同时，抽象语法树也可以用于代码生成、代码优化、静态分析等方面。

在构造抽象语法树时，需要考虑节点类型的选择、节点的属性以及节点的关系等问题。同时，还需要考虑如何处理错误和异常情况，以保证抽象语法树的正确性和完整性。

[1]树状表现形式是计算机中极其常见的数据结构，例如：二叉树、红黑树等都是树状表现形式的实例。

抽象语法树可以帮助我们更好地理解和分析源代码的结构和语义，从而更好地设计和实现编译器和解释器。为了直观理解抽象语法树，我编写了一段简单的求两数之和的程序代码 `twosum.pas`，具体代码如下：

```
1 const a = 1, b = 2;
2 var x : integer;
3 begin
4    x := a + b
5 end.
```

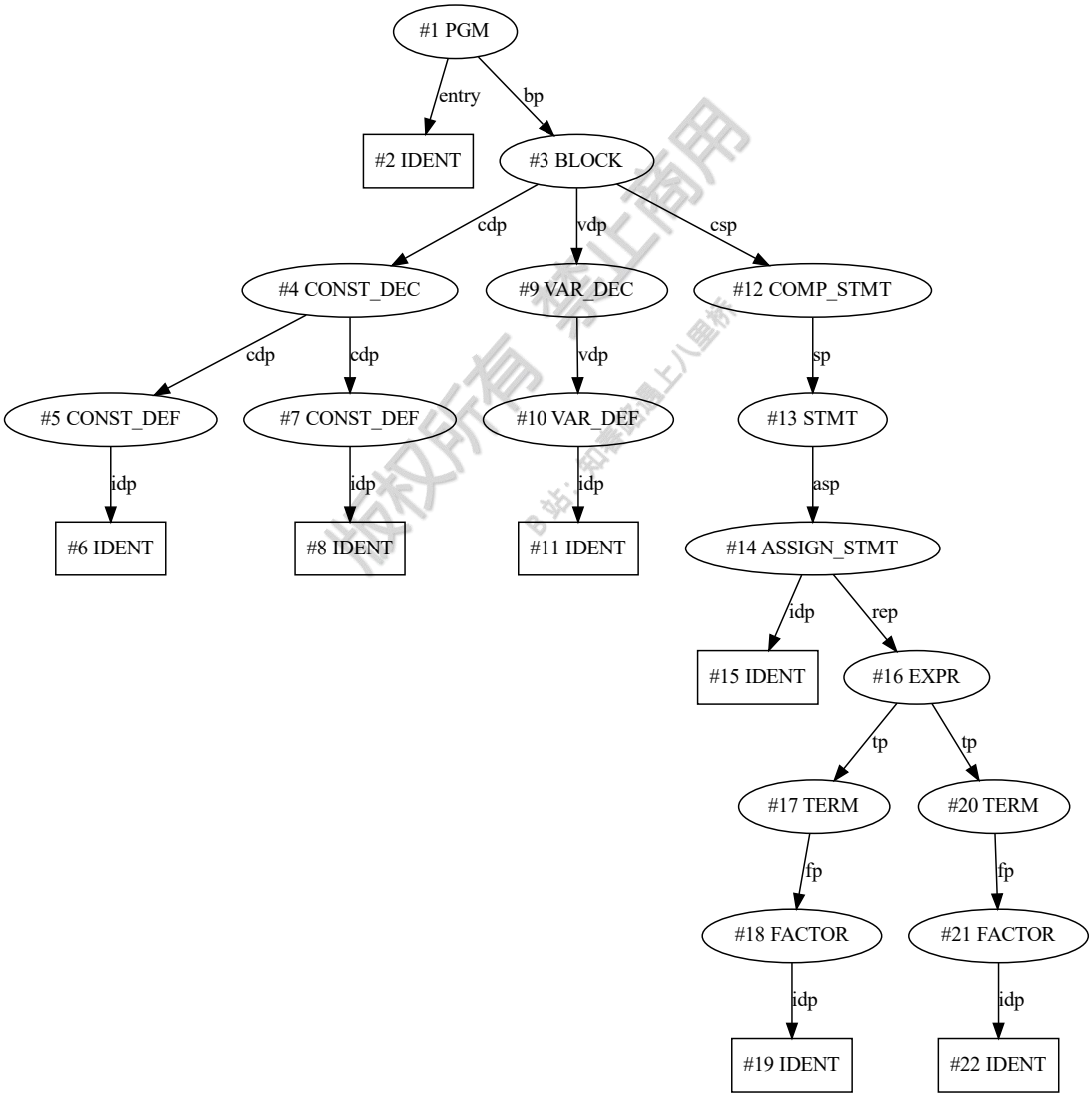


图 3.2: 源文件 twosum.pas 构成的抽象语法树

图 3.2 是与 `twosum.pas` 对应的抽象语法树，它清晰地描述了代码的逻辑结构。为了方便后序的说明，这里先说明图中的一些符号含义。每个节点都表示源代码中的一种结构，比如 PGM 表

示程序、VAR 表示变量等。每个边表示节点之间的关系，例如：bp 表示 PGM 对 BLOCK 的引用。另外，每个节点可以通过 **#n** 的标记法来表示，其中 **n** 是一个正整数。在每个语法树中，每个节点对应的 **n** 是唯一的，所以 **n** 也称节点的顺序号。例如：图 3.2 中 **#17** 和 **#20** 节点都是项 TERM 的节点，但是它们的顺序号不同。

3.3 文法歧义

文法歧义是指同一个文法规则或输入序列可以有多个不同的语法分析树或输出结果。简单来说，就是同一个语法规则或者输入可以对应多个不同的解析结果。为了说明歧义的存在性，我们先定义一个示例文法 (3.1) - (3.4)，然后在这个文法的基础上将一个表达式解析成语法树。

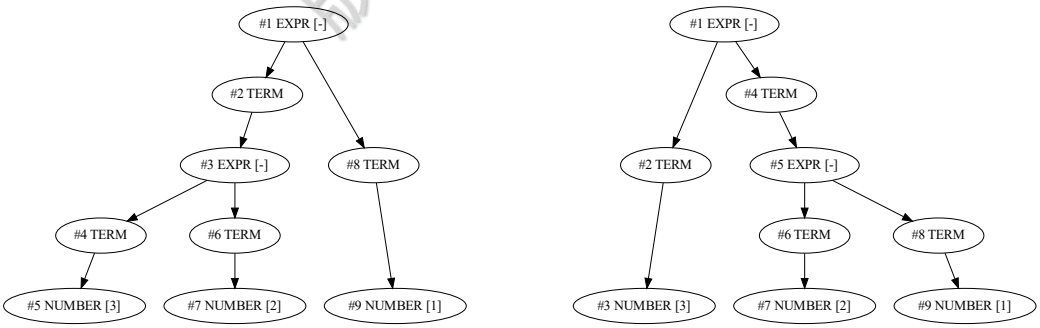
$$expr \rightarrow [+|-]\ term\ \{\ addop\ \ term\ \} \tag{3.1}$$

$$term \rightarrow number \mid expr \tag{3.2}$$

$$addop \rightarrow +|- \tag{3.3}$$

$$number \rightarrow \texttt{Unsigned Number} \tag{3.4}$$

我们以表达式 3-2-1 为例构建语法树，构建好的语法树见图 3.3，在满足文法 (3.1) - (3.4) 的前提下，可能会出现两种不同情况的语法树，分别为图 3.3(a) 和3.3(b) ，显然两个语法树结构式不一样的。



(a) 第一种情形　　(b) 第二种情形

图 3.3: 表达式 3-2-1 的两种不同的语法树

进一步讨论，如果我定义了求值顺序为自底而上求值顺序，然后分别对上述两种情景求值。两种计算过程如下。我们首先定义一个可以对节点进行求值的 eval 函数，eval(节点) 满足求值规则 (3.5) 。

$$eval(\text{节点}) = \begin{cases} eval(\text{左子节点})\ addop\ eval(\text{右子节点}) & \textbf{if}\ \text{节点是}\ expr\ \text{类型}, \\ eval(\text{子节点}) & \textbf{if}\ \text{节点是}\ term\ \text{类型}, \\ \text{子节点的数值} & \textbf{if}\ \text{节点是}\ number\ \text{类型} \end{cases} \tag{3.5}$$

图 3.3(a) 中，通过 `eval` 函数来求值根节点的演算步骤如下：

$$
\begin{aligned}
eval(\#\mathbf{1}) &= eval(\#\mathbf{2}) - eval(\#\mathbf{8}) \\
&= eval(\#\mathbf{3}) - eval(\#\mathbf{8}) \\
&= (eval(\#\mathbf{4}) - eval(\#\mathbf{6})) - eval(\#\mathbf{8}) \\
&= (eval(\#\mathbf{5}) - eval(\#\mathbf{6})) - eval(\#\mathbf{8}) \\
&= (eval(\#\mathbf{5}) - eval(\#\mathbf{7})) - eval(\#\mathbf{8}) \\
&= (eval(\#\mathbf{5}) - eval(\#\mathbf{7})) - eval(\#\mathbf{9}) \\
&= (3-2) - 1 \\
&= 1 - 1 \\
&= 0
\end{aligned} \tag{3.6}
$$

相应的，图 3.3(b) 中，通过 `eval` 函数来求值根节点的演算步骤如下：

$$
\begin{aligned}
eval(\#\mathbf{1}) &= eval(\#\mathbf{2}) - eval(\#\mathbf{4}) \\
&= eval(\#\mathbf{3}) - eval(\#\mathbf{4}) \\
&= eval(\#\mathbf{3}) - eval(\#\mathbf{5}) \\
&= eval(\#\mathbf{3}) - (eval(\#\mathbf{6}) - eval(\#\mathbf{8})) \\
&= eval(\#\mathbf{3}) - (eval(\#\mathbf{7}) - eval(\#\mathbf{8})) \\
&= eval(\#\mathbf{3}) - (eval(\#\mathbf{7}) - eval(\#\mathbf{9})) \\
&= 3 - (2 - 1) \\
&= 3 - 1 \\
&= 2
\end{aligned} \tag{3.7}
$$

可以对求值结果 (3.6) 和(3.7) 进行对比，发现它们并不只是相等的，可见如果文法定义存在歧义构造成不同的语法树会导致语言存在二义性 [2] 。幸运的是，我们需要实现的 $PL/0\epsilon$ 的文法是不存在二义性的文法。

3.4 语法树节点

图 3.1 表示了一个抽象语法树，与 “抽象” 对应的就是 “具体” 的语法树，我们编写的 $PL/0\epsilon$ 语言中需要定义具体语法树，首先我们来介绍语法树的节点。每个语法树节点代表了源代码中的一个语法成分，例如一个运算符、一个变量、一个函数调用等。语法树的根节点通常对应于源代码的起始符号，而其他节点则根据语法规则和它们在源代码中的位置进行组织。

一个语法树节点通常包含以下信息：

[2] 编译原理中有很多通过修改文法消除歧义的理论，为了避免目标发散，我们这里暂不做讨论。

1. 节点类型：表示该节点所代表的语法成分的类型。例如，节点类型可以是变量、运算符、函数调用等。
2. 节点值：表示该节点所代表的语法成分的值。例如，如果节点类型是变量，那么节点值就是变量的名称；如果节点类型是运算符，那么节点值就是运算符的符号。
3. 子节点：表示该节点的直接子节点。对于一个语法树节点，可以有一个或多个子节点，子节点按照它们在源代码中的顺序进行组织。

参考 $PL/0\epsilon$ 语言的文法，我们可以对具体语法树节点定义。由于语法树节点之间有循环引用，需要将所有节点声明统一放在 `include/parse.h` 中。具体声明的节点类型代码如下：

```
// Declaretion of a bundle of tree node
typedef struct _pgm_node pgm_node_t;
typedef struct _block_node block_node_t;
typedef struct _const_dec_node const_dec_node_t;
typedef struct _const_def_node const_def_node_t;
typedef struct _var_dec_node var_dec_node_t;
typedef struct _var_def_node var_def_node_t;
typedef struct _pf_dec_list_node pf_dec_list_node_t;
typedef struct _proc_dec_node proc_dec_node_t;
typedef struct _proc_def_node proc_def_node_t;
typedef struct _proc_head_node proc_head_node_t;
typedef struct _fun_dec_node fun_dec_node_t;
typedef struct _fun_def_node fun_def_node_t;
typedef struct _fun_head_node fun_head_node_t;
typedef struct _stmt_node stmt_node_t;
typedef struct _assign_stmt_node assign_stmt_node_t;
typedef struct _if_stmt_node if_stmt_node_t;
typedef struct _repe_stmt_node repe_stmt_node_t;
typedef struct _for_stmt_node for_stmt_node_t;
typedef struct _pcall_stmt_node pcall_stmt_node_t;
typedef struct _fcall_stmt_node fcall_stmt_node_t;
typedef struct _comp_stmt_node comp_stmt_node_t;
typedef struct _read_stmt_node read_stmt_node_t;
typedef struct _write_stmt_node write_stmt_node_t;
typedef struct _expr_node expr_node_t;
typedef struct _term_node term_node_t;
typedef struct _factor_node factor_node_t;
typedef struct _cond_node cond_node_t;
typedef struct _ident_node ident_node_t;
typedef struct _para_list_node para_list_node_t;
typedef struct _para_def_node para_def_node_t;
typedef struct _arg_list_node arg_list_node_t;
```

每个节点通过 C 语言的 `typedef` 定义，它们的命名规则为 `xxx_node_t` 形式。每个节点

都包含一个 nid 的 int 类型的整数，用于记录节点的 ID 值。每个节点具体的结构体定义在 include/syntax.h 文件中，具体的定义的代码如下：

```
/* program */
typedef struct _pgm_node {
        int nid;
        block_node_t *bp;
        ident_node_t *entry;
} pgm_node_t;

/* block */
typedef struct _block_node {
        int nid;
        const_dec_node_t *cdp;
        var_dec_node_t *vdp;
        pf_dec_list_node_t *pfdlp;
        comp_stmt_node_t *csp;
} block_node_t;
typedef struct _const_dec_node {
        int nid;
        const_def_node_t *cdp;
        const_dec_node_t *next;
} const_dec_node_t;
typedef struct _const_def_node {
        int nid;
        ident_node_t *idp;
} const_def_node_t;
typedef struct _var_dec_node {
        int nid;
        var_def_node_t *vdp;
        var_dec_node_t *next;
} var_dec_node_t;
typedef struct _var_def_node {
        int nid;
        ident_node_t *idp;
        var_def_node_t *next;
} var_def_node_t;

typedef enum _pf_dec_enum { FUN_PFDEC, PROC_PFDEC } pf_dec_t;
typedef struct _pf_dec_list_node {
        int nid;
        pf_dec_t kind;
        proc_dec_node_t *pdp;
```

```
41          fun_dec_node_t *fdp;
42          pf_dec_list_node_t *next;
43  } pf_dec_list_node_t;
44  typedef struct _proc_dec_node {
45          int nid;
46          proc_def_node_t *pdp;
47          proc_dec_node_t *next;
48  } proc_dec_node_t;
49  typedef struct _proc_def_node {
50          int nid;
51          proc_head_node_t *php;
52          block_node_t *bp;
53  } proc_def_node_t;
54  typedef struct _proc_head_node {
55          int nid;
56          ident_node_t *idp;
57          para_list_node_t *plp;
58  } proc_head_node_t;
59  typedef struct _fun_dec_node {
60          int nid;
61          fun_def_node_t *fdp;
62          fun_dec_node_t *next;
63  } fun_dec_node_t;
64  typedef struct _fun_def_node {
65          int nid;
66          fun_head_node_t *fhp;
67          block_node_t *bp;
68  } fun_def_node_t;
69  typedef struct _fun_head_node {
70          int nid;
71          ident_node_t *idp;
72          para_list_node_t *plp;
73  } fun_head_node_t;
74
75  /* statement */
76  typedef enum _stmt_enum {
77          ASSGIN_STMT,
78          IF_STMT,
79          REPEAT_STMT,
80          PCALL_STMT,
81          COMP_STMT,
82          READ_STMT,
```

```
        WRITE_STMT,
        FOR_STMT,
        NULL_STMT
} stmt_t;
typedef struct _stmt_node {
        int nid;
        stmt_t kind;
        assign_stmt_node_t *asp;
        if_stmt_node_t *ifp;
        repe_stmt_node_t *rpp;
        for_stmt_node_t *frp;
        pcall_stmt_node_t *pcp;
        comp_stmt_node_t *cpp;
        read_stmt_node_t *rdp;
        write_stmt_node_t *wtp;
} stmt_node_t;
typedef enum _assgin_enum { NORM_ASSGIN, FUN_ASSGIN, ARRAY_ASSGIN } assgin_t;
typedef struct _assign_stmt_node {
        int nid;
        assgin_t kind;
        ident_node_t *idp;
        expr_node_t *lep;
        expr_node_t *rep;
} assign_stmt_node_t;
typedef struct _if_stmt_node {
        int nid;
        cond_node_t *cp;
        /* then */
        stmt_node_t *tp;
        /* else */
        stmt_node_t *ep;
        symtab_t *stab;
} if_stmt_node_t;
typedef struct _repe_stmt_node {
        int nid;
        stmt_node_t *sp;
        cond_node_t *cp;
        symtab_t *stab;
} repe_stmt_node_t;
typedef enum _for_enum { TO_FOR, DOWNTO_FOR } for_t;
typedef struct _for_stmt_node {
        int nid;
```

```
125         for_t kind;
126         ident_node_t *idp;
127         expr_node_t *lep;
128         expr_node_t *rep;
129         stmt_node_t *sp;
130         symtab_t *stab;
131 } for_stmt_node_t;
132 typedef struct _pcall_stmt_node {
133         int nid;
134         ident_node_t *idp;
135         arg_list_node_t *alp;
136 } pcall_stmt_node_t;
137 typedef struct _fcall_stmt_node {
138         int nid;
139         ident_node_t *idp;
140         arg_list_node_t *alp;
141         symtab_t *stab;
142 } fcall_stmt_node_t;
143 typedef struct _comp_stmt_node {
144         int nid;
145         stmt_node_t *sp;
146         comp_stmt_node_t *next;
147 } comp_stmt_node_t;
148 typedef struct _read_stmt_node {
149         int nid;
150         ident_node_t *idp;
151         read_stmt_node_t *next;
152 } read_stmt_node_t;
153 typedef enum _write_enum { STRID_WRITE, STR_WRITE, ID_WRITE } write_t;
154 typedef struct _write_stmt_node {
155         int nid;
156         write_t type;
157         /* string pointer */
158         char sp[MAXSTRLEN];
159         expr_node_t *ep;
160         symtab_t *stab;
161 } write_stmt_node_t;
162
163 /* expression term factor condition */
164 typedef enum _addop_enum {
165         NOP_ADDOP,
166         NEG_ADDOP,
```

```
167         ADD_ADDOP,
168         MINUS_ADDOP
169 } addop_t;
170 typedef struct _expr_node {
171         int nid;
172         addop_t kind;
173         term_node_t *tp;
174         expr_node_t *next;
175         symtab_t *stab;
176 } expr_node_t;
177 typedef enum _multop_enum { NOP_MULTOP, MULT_MULTOP, DIV_MULTOP } multop_t;
178 typedef struct _term_node {
179         int nid;
180         multop_t kind;
181         factor_node_t *fp;
182         term_node_t *next;
183         symtab_t *stab;
184 } term_node_t;
185 typedef enum _factor_enum {
186         ID_FACTOR,
187         ARRAY_FACTOR,
188         UNSIGN_FACTOR,
189         CHAR_FACTOR,
190         EXPR_FACTOR,
191         FUNCALL_FACTOR
192 } factor_t;
193 typedef struct _factor_node {
194         int nid;
195         factor_t kind;
196         ident_node_t *idp;
197         expr_node_t *ep;
198         // value: store unsigned int or char
199         int value;
200         fcall_stmt_node_t *fcsp;
201         symtab_t *stab;
202 } factor_node_t;
203 typedef enum _rela_enum {
204         EQU_RELA,
205         NEQ_RELA,
206         GTT_RELA,
207         GEQ_RELA,
208         LST_RELA,
```

```
209         LEQ_RELA
210 } rela_t;
211 typedef struct _cond_node {
212         int nid;
213         expr_node_t *lep;
214         rela_t kind;
215         expr_node_t *rep;
216         symtab_t *stab;
217 } cond_node_t;
218
219 /* ident parameter argument*/
220 typedef enum _ident_enum {
221         // Normal Identifier
222         /*  0 */ INIT_IDENT,
223         /*  1 */ PROC_IDENT,
224         /*  2 */ INT_FUN_IDENT,
225         /*  3 */ CHAR_FUN_IDENT,
226         // Const Identifier
227         /*  4 */ INT_CONST_IDENT,
228         /*  5 */ CHAR_CONST_IDENT,
229         // Variable Identifier
230         /*  6 */ INT_VAR_IDENT,
231         /*  7 */ CHAR_VAR_IDENT,
232         /*  8 */ INT_ARRVAR_IDENT,
233         /*  9 */ CHAR_ARRVAR_IDENT,
234         // Parameter Identifier, (by value, by address)
235         /* 10 */ INT_BYVAL_IDENT,
236         /* 11 */ CHAR_BYVAL_IDENT,
237         /* 12 */ INT_BYADR_IDENT,
238         /* 13 */ CHAR_BYADR_IDENT
239 } idekind_t;
240 typedef struct _ident_node {
241         int nid;
242         idekind_t kind;
243         char name[MAXSTRLEN];
244         int value;
245         int length;
246         int line;
247         syment_t *symbol;
248 } ident_node_t;
249
250 typedef struct _para_list_node {
```

```
251         int nid;
252         para_def_node_t *pdp;
253         para_list_node_t *next;
254 } para_list_node_t;
255 typedef struct _para_def_node {
256         int nid;
257         ident_node_t *idp;
258         para_def_node_t *next;
259 } para_def_node_t;
260 typedef struct _arg_list_node {
261         int nid;
262         expr_node_t *ep;
263         arg_list_node_t *next;
264         // link to referred variable or array
265         syment_t *refsym; // referred parameter
266         syment_t *argsym; // local argument
267         expr_node_t *idx; // array reference index
268 } arg_list_node_t;
```

节点的定义的明细比较繁琐，这里对一些需要注意的节点定义进行讲解。

1. 有些节点的定义比较简单，
   - 例如：pgm_node_t 节点描述程序，它的核心属性就只有指向分程序的 bp 指针。
2. 有些节点的定义包含多次重复出现的可能，
   - 例如：const_dec_node_t 节点描述常量声明，它除了 cdp 指向常量定义以外，还需要通过 next 指向下一个常量声明，这也是一个链表的设计。
3. 有些节点的定义需要通过 kind 种类属性来进行分类，
   - 例如：stmt_node_t 节点描述一个语句，通过 kind 指出何种语句，
   - 再例如：expr_node_t 节点描述一个表达式，通过 kind 指出是空运算种类，或者加运算种类，或者减运算种类的表达式。
4. 最后 ident_node_t 是一个非常重要的节点类型，它描述一个标识符（identifier），标识符除了需要通过 kind 记录语法分析中的种类，还需要记录一下额外的信息，具体如下：
   - name 表示标识符名称
   - value 表示标识符的值
   - length 表示数组标识符的值
   - line 表示标识符所在行数

最后，还有一些 symbol 和 stab 等属性是为后续分析阶段提供数据查找所记录的信息，这里暂时不需要深入理解。

图 3.2 描述了对 twosum.pas 代码的抽象语法树，如果在上述语法树节点的 struct 结构完全定义的前提下，我们就可以对具体语法树进行描述，图 3.4 就是在每个节点定义的情况下完成具体语法树的构建，图中的一个很重要的信息是添加了 nid 属性，这样就可以将节点和具体的内存进行对应。

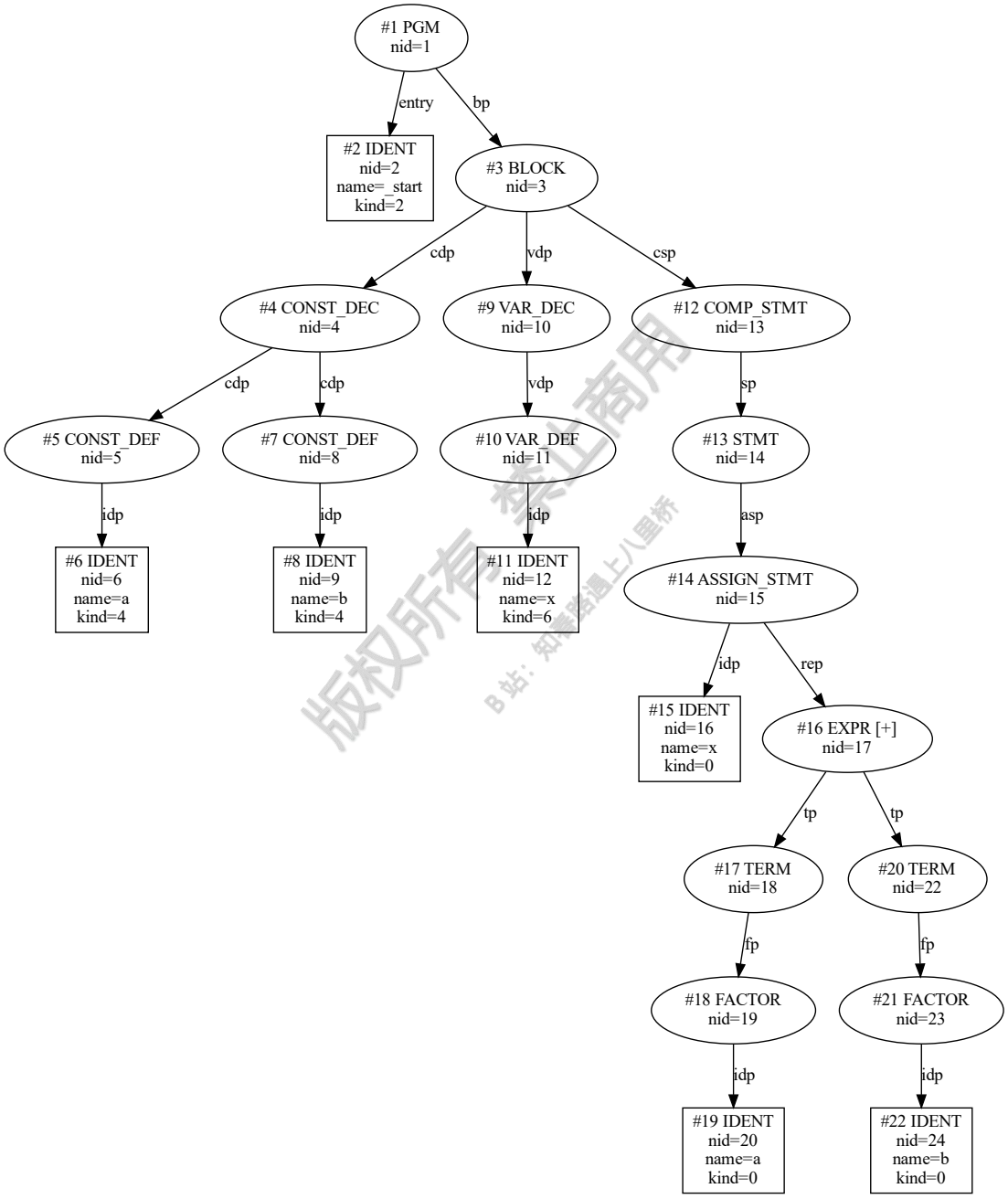


图 3.4: 源文件 twosum.pas 构成的具体语法树

这里还需要对语法树中的一些符号的含义进行介绍。图 3.4 中的节点根据形状可以分成两类：椭圆和矩形。椭圆表示的是程序必需的语法结构，通常是语法树的非叶子节点。矩形表示的是程序的标识符 identifier，通常是语法树的叶子节点。

椭圆的大写字母标记可以描述节点所属的节点结构，例如 **#1** 的 PGM 可以和代码中定义的 `pgm_node_t` 结构体对应，再例如 **#14** 的 ASSGIN_STMT 可以和代码中定义的 `assign_stmt_node_t` 结构体对应。有些节点中还存在一些额外的信息，例如节点 **#16** 中包含 [+] 表示当前表达式节点的种类为 `ADD_ADDOP` ，即加法运算。

矩形的大写字母标记都是 IDENT ，表示叶子节点都是 `ident_node_t` 结构体，但是 IDENT 有着丰富的属性，其中 `kind` 表示标识符的种类，表 3.1 列出了所有的种类，这些种类可以通过语法分析得出。`name` 也是 IDENT 的重要属性，它记录的是标识符的名称，例如 **#19** 中的 `name=a` 可以与第 4 行代码 `x := a + b` 表达式中的 `a` 是一一对应的 [3]。

表 3.1: $PL/0\epsilon$ 语言所有标识符种类

常规种类	0	INIT_IDENT	1	PROC_IDENT
	2	INT_FUN_IDENT	3	CHAR_FUN_IDENT
常量种类	4	INT_CONST_IDENT	5	CHAR_CONST_IDENT
变量种类	6	INT_VAR_IDENT	7	CHAR_VAR_IDENT
	8	INT_ARRVAR_IDENT	9	CHAR_ARRVAR_IDENT
参数种类	10	INT_BYVAL_IDENT	11	CHAR_BYVAL_IDENT
	12	INT_BYADR_IDENT	13	CHAR_BYADR_IDENT

有了节点的定义后，我们还需要新建节点，并为节点申请内存空间。代码中是通过两个宏定义完成这个操作。第一个宏 `INITMEM` 定义在 `include/common.h` 中，它完成了两个功能：

1. 通过库函数 `malloc()` 来申请内存
2. 如果内存申请成功，调用库函数 `memset()` 将内存清零。否则会抛出内存不足的异常。

```
1  // Initialize struct, allocate memory
2  //     INITMEM(s: struct, v: variable, sturct pointer)
3  #define INITMEM(s, v)                                                   \
4          do {                                                            \
5                  v = (s *)malloc(sizeof(s));                             \
6                  if (v == NULL) {                                        \
7                          panic("OUT_OF_MEMORY");                         \
8                  };                                                      \
9                  memset(v, 0, sizeof(s));                                \
10         } while (0)
```

第二个宏 `NEWNODE` 是创建节点的宏，定义在 `include/parse.h` 文件中，它调用 `INITMEM` 宏初始化节点内存，然后给 `nid` 属性赋成唯一值，即当前的 `nidcnt` 值。这样，在语法分析的过程中就可以通过 `NEWNODE` 宏创建并初始化语法树节点。

[3]语法分析读取到的是 token，语法树节点一般不能和源文件对应，但是标识符要和源代码中的结构保持查找关系，所以必需要和代码中的词位一一对应。

```
extern int nidcnt;

// Create New Node
#define NEWNODE(s, v)                                   \
        do {                                            \
                INITMEM(s, v);                          \
                v->nid = ++nidcnt;                      \
        } while (0)
```

语法树在编译的各个阶段中都需要使用，为了简单起见，这里不对语法树节点进行释放操作，最终语法树会在编译程序执行结束由操作系统统一回收。

3.5 递归下降分析法

递归下降分析法（Recursive Descent Parsing）是一种自顶向下的语法分析方法，也被称为预测分析法。这种方法的基本思想是为每一个非终结符编写一个函数或过程，当这个函数或过程被调用时，它按照产生式的右部顺序逐一识别输入符号，若识别成功则继续识别下一个输入符号，否则调用出错处理程序。

递归下降分析法的优点是算法简单明了，易于理解和实现。然而，它的缺点是对于某些文法，可能无法找到有效的预测分析算法，即无法确定下一个可能的输入符号。此外，当文法中存在左递归或公共前缀时，递归下降分析法可能会导致无限循环或无法正确识别输入。

为了克服这些缺点，可以使用一些改进的方法，如回溯法、左因子提取和引入虚拟符号等。其中，回溯法是在预测失败时回溯到上一个状态重新进行预测；左因子提取是将具有公共前缀的产生式进行合并，消除公共前缀；引入虚拟符号是为了解决某些文法无法找到有效预测算法的问题。

总之，递归下降分析法是一种简单直观的语法分析方法，但需要根据具体文法的特点进行适当的改进和优化。

3.5.1 parse_xxx 解析函数家族

通过精巧的设计，$PL/0\epsilon$ 语言的文法适合使用递归下降分析法进行语法分析。首先我们对每个语法树节点实现一个 `parse_xxx()` 命名格式的解析函数。这些函数的声明在 `include/parse.h` 文件中，具体代码如下：

```
// ID read mode, for parse_ident()
typedef enum _idreadmode_enum { READCURR, READPREV } idreadmode_t;

// Define a bundle of parse function
static pgm_node_t *parse_pgm(void);
static block_node_t *parse_block(void);
static const_dec_node_t *parse_const_dec(void);
```

```
8  static const_def_node_t *parse_const_def(void);
9  static var_dec_node_t *parse_var_dec(void);
10 static var_def_node_t *parse_var_def(void);
11 static pf_dec_list_node_t *parse_pf_dec_list(void);
12 static proc_dec_node_t *parse_proc_dec(void);
13 static proc_def_node_t *parse_proc_def(void);
14 static proc_head_node_t *parse_proc_head(void);
15 static fun_dec_node_t *parse_fun_dec(void);
16 static fun_def_node_t *parse_fun_def(void);
17 static fun_head_node_t *parse_fun_head(void);
18 static stmt_node_t *parse_stmt(void);
19 static assign_stmt_node_t *parse_assign_stmt(void);
20 static if_stmt_node_t *parse_if_stmt(void);
21 static repe_stmt_node_t *parse_repe_stmt(void);
22 static for_stmt_node_t *parse_for_stmt(void);
23 static pcall_stmt_node_t *parse_pcall_stmt(void);
24 static fcall_stmt_node_t *parse_fcall_stmt(void);
25 static comp_stmt_node_t *parse_comp_stmt(void);
26 static read_stmt_node_t *parse_read_stmt(void);
27 static write_stmt_node_t *parse_write_stmt(void);
28 static expr_node_t *parse_expr(void);
29 static term_node_t *parse_term(void);
30 static factor_node_t *parse_factor(void);
31 static cond_node_t *parse_cond(void);
32 static ident_node_t *parse_ident(idreadmode_t mode);
33 static para_list_node_t *parse_para_list(void);
34 static para_def_node_t *parse_para_def(void);
35 static arg_list_node_t *parse_arg_list(void);
```


解析函数 parse_xxx() 几乎都是不接受入参的函数，每个函数返回分析后得出的语法树节点。其中 parse_ident() 函数需要获取 token 的词位字符串，由于递归下降分析法会向前预读一个 token 值，所以存在两种情况的词位读取位置，这里通过传入一个读取模式的 mode 参数来区分两种场景，即：READCURR 表示读取当前词位，READPREV 表示读取前一个词位。

3.5.2 match 函数

在词法分析中我们已经介绍了 gettok() 函数可以获取下一个 token，在语法分析中需要不断获取 token 来决策后续需要调用哪个解析函数，当 token 和预期的匹配后还需要跳过当前的 token，并需要完成一些后置操作。这里的 match() 函数就是用来匹配 token 并进行搬运操作，它的代码实现在 source/parse.c ，具体细节如下：

```
1 // current token
2 static token_t currtok;
```

```
// hold previous token
static token_t prevtok;
static char prevtokbuf[MAXTOKSIZE + 1];
static int prevlineno;

// match an expected token, and skip to next token
static void match(token_t expected)
{
        // check if token matched
        if (currtok != expected) {
                char buf[MAXSTRBUF];
                sprintf(buf, "UNEXPECTED_TOKEN: LINE%d [%s]", lineno, tokbuf);
                panic(buf);
        }

        // store previous token
        strcopy(prevtokbuf, tokbuf);
        prevtok = currtok;
        prevlineno = toklineno;

        // read next token
        currtok = gettok();
}
```


在语法分析过程中需要缓存几个静态变量，它们通过 static 关键字修饰：

- currtok 表示当前正在处理的 token
- prevtok 表示上一个 token
- prevtokbuf[] 数组记录上一个 token 的词位字符串
- prevlineno 记录上一个 token 的行号

有了上述变量的介绍，语法分析实现的 match() 函数功能如下：

1. 判断 currtok 是否是预期的 expected 的 token
2. 如果不符合预期则抛出异常，语法解析失败
3. 如果 currtok 符号预期，则将当前 token，词位字符串，token 行号搬运到 prev 开头的 3 个变量中存储
4. 接着调用 gettok() 读取下一个 token 并存储到 currtok 变量中

3.5.3 程序的解析

所有的 parse_xxx() 系列解析函数的实现都在 source/parse.c 中，这里从语法分析的入口函数 parse() 说起，它完成了如下功能：

1. 调用 gettok() 函数初始化第一个 token 。

2. 然后调用 parse_pgm() 函数开启解析程序，将解析得到的语法树放入 pgm 全局变量中。
3. 如果解析没有出错，将 phase 改成下一个语法分析阶段 SEMANTIC 。
4. 同时既然语法解析完成，后续不需要读取源代码文件了，这里调用 fclose() 关闭源文件释放资源。

```
// syntax tree
pgm_node_t *pgm;
int nidcnt = 0;

// 省略一些代码 ...

pgm_node_t *parse(void)
{
        currtok = gettok();
        pgm = parse_pgm();
        chkerr("parse fail and exit.");
        phase = SEMANTIC;
        fclose(source);
        return pgm;
}
```

parse_pgm() 是解析程序的入口函数，它大概有以下几个功能：

1. 它首先调用 NEWNODE 初始化了节点 t 。
2. 接着设置好了入口点 entry 标识符，入口点函数的名字是固定的，即 MAINFUNC 宏定义的 "_start" 字符串。
3. 接着最关键的一步，通过调用 parse_block() 函数用来解析分程序。
4. 在分程序结束后，还需要调用 match(SS_DOT) 匹配程序的最后一个 . ，这一步也是必需的，如果不调用 match() 函数匹配就会导致文法解析不完整。

```
/**
 * program ->
 *        block .
 */
static pgm_node_t *parse_pgm(void)
{
        pgm_node_t *t;
        NEWNODE(pgm_node_t, t);

        // setup entrypoint
        ident_node_t *entry;
        NEWNODE(ident_node_t, entry);
        entry->kind = INT_FUN_IDENT;
        entry->value = 0;
```

```
        entry->length = 0;
        entry->line = 0;
        strcpy(entry->name, MAINFUNC);
        t->entry = entry;

        t->bp = parse_block();
        match(SS_DOT);
        return t;
}
```

3.5.4 分程序的解析

分程序的解析入口函数为 parse_block() ，该函数根据 $PL/0\epsilon$ 文法的定义递归调用子函数进行解析，子函数包括：

1. parse_const_dec() 解析常量声明。
2. parse_var_dec() 解析变量声明。
3. parse_pf_dec_list() 解析过程或函数声明列表。
4. parse_comp_stmt() 解析复杂语句。

需要注意的是，为了提升代码阅读体验，我们在 include/parse.h 中定义了一些 TOKANYn [4] 的宏（其中 n 表示接受参数数量），来判断 currtok 是否在给定参数列表中。

```
// use like:
//   if (TOKANY(a, b, c, ...)) { ... }
#define TOKANY(a) (currtok == (a))
#define TOKANY2(a, b) (currtok == (a) || currtok == (b))
#define TOKANY3(a, b, c) (currtok == (a) || currtok == (b) || currtok == (c))
#define TOKANY4(a, b, c, d)                                                \
        (currtok == (a) || currtok == (b) || currtok == (c) || currtok == (d))
#define TOKANY5(a, b, c, d, e)                                             \
        (currtok == (a) || currtok == (b) || currtok == (c) ||             \
         currtok == (d) || currtok == (e))
#define TOKANY6(a, b, c, d, e, f)                                          \
        (currtok == (a) || currtok == (b) || currtok == (c) ||             \
         currtok == (d) || currtok == (e) || currtok == (f))
```

后续的解析函数都是通过递归下降的方式来进行调用，如果发现有可以判断的非终结符，则调用对应的解析函数。具体解析函数的实现代码如下：

```
/**
 * block ->
 *         [constdec] [vardec] [pfdeclist] compstmt
```

[4] 为方便记忆 if(TOKANY(...)) 读作 if current token is any of ... ，如果当前 token 是 ... 其中的任何一个的话。另外这个 ... 也被称为 first 集。

```
 */
static block_node_t *parse_block(void)
{
        block_node_t *t;
        NEWNODE(block_node_t, t);

        if (TOKANY(KW_CONST)) {
                t->cdp = parse_const_dec();
        }

        if (TOKANY(KW_VAR)) {
                t->vdp = parse_var_dec();
        }

        if (TOKANY2(KW_FUNCTION, KW_PROCEDURE)) {
                t->pfdlp = parse_pf_dec_list();
        }

        if (TOKANY(KW_BEGIN)) {
                t->csp = parse_comp_stmt();
        }

        return t;
}

/**
 * constdec ->
 *        CONST constdef {, constdef};
 */
static const_dec_node_t *parse_const_dec(void)
{
        const_dec_node_t *t, *p, *q;
        NEWNODE(const_dec_node_t, t);

        match(KW_CONST);
        t->cdp = parse_const_def();

        for (p = t; TOKANY(SS_COMMA); p = q) {
                match(SS_COMMA);
                NEWNODE(const_dec_node_t, q);
                p->next = q;
                q->cdp = parse_const_def();
```

```
46          }
47          match(SS_SEMI);
48
49          return t;
50  }
51
52  /**
53   * constdef ->
54   *          ident = const
55   */
56  static const_def_node_t *parse_const_def(void)
57  {
58          const_def_node_t *t;
59          NEWNODE(const_def_node_t, t);
60
61          if (TOKANY(MC_ID)) {
62                  t->idp = parse_ident(READCURR);
63          }
64
65          match(SS_EQU);
66
67          if (TOKANY4(SS_PLUS, SS_MINUS, MC_UNS, MC_CH)) {
68                  switch (currtok) {
69                  case SS_PLUS:
70                          match(SS_PLUS);
71                          t->idp->kind = INT_CONST_IDENT;
72                          t->idp->value = atoi(tokbuf);
73                          match(MC_UNS);
74                          break;
75                  case SS_MINUS:
76                          match(SS_MINUS);
77                          t->idp->kind = INT_CONST_IDENT;
78                          t->idp->value = -atoi(tokbuf);
79                          match(MC_UNS);
80                          break;
81                  case MC_UNS:
82                          t->idp->kind = INT_CONST_IDENT;
83                          t->idp->value = atoi(tokbuf);
84                          match(MC_UNS);
85                          break;
86                  case MC_CH:
87                          t->idp->kind = CHAR_CONST_IDENT;
```

```
88                          t->idp->value = (int)tokbuf[0];
89                          match(MC_CH);
90                          break;
91                  default:
92                          unlikely();
93                  }
94          } else {
95                  unlikely();
96          }
97
98          return t;
99  }
100
101 /**
102  * vardec ->
103  *        VAR vardef; { vardef;}
104  */
105 static var_dec_node_t *parse_var_dec(void)
106 {
107         var_dec_node_t *t, *p, *q;
108         NEWNODE(var_dec_node_t, t);
109
110         match(KW_VAR);
111         t->vdp = parse_var_def();
112         match(SS_SEMI);
113
114         for (p = t; TOKANY(MC_ID); p = q) {
115                 NEWNODE(var_dec_node_t, q);
116                 p->next = q;
117                 q->vdp = parse_var_def();
118                 match(SS_SEMI);
119         }
120
121         return t;
122 }
123
124 /**
125  * vardef ->
126  *        ident {, ident} : type
127  */
128 static var_def_node_t *parse_var_def(void)
129 {
```

```
130         var_def_node_t *t, *p, *q;
131         NEWNODE(var_def_node_t, t);
132 
133         int arrlen = 0;
134         t->idp = parse_ident(READCURR);
135 
136         for (p = t; TOKANY(SS_COMMA); p = q) {
137                 match(SS_COMMA);
138                 NEWNODE(var_def_node_t, q);
139                 p->next = q;
140                 q->idp = parse_ident(READCURR);
141         }
142 
143         match(SS_COLON);
144 
145         switch (currtok) {
146         case KW_INTEGER:
147                 match(KW_INTEGER);
148                 for (p = t; p; p = p->next) {
149                         p->idp->kind = INT_VAR_IDENT;
150                 }
151                 break;
152         case KW_CHAR:
153                 match(KW_CHAR);
154                 for (p = t; p; p = p->next) {
155                         p->idp->kind = CHAR_VAR_IDENT;
156                 }
157                 break;
158         case KW_ARRAY: // array[10] of integer
159                 match(KW_ARRAY);
160                 match(SS_LBRA);
161                 if (TOKANY(MC_UNS)) {
162                         arrlen = atoi(tokbuf);
163                         match(MC_UNS);
164                 } else {
165                         unlikely();
166                 }
167                 match(SS_RBRA);
168                 match(KW_OF);
169                 if (TOKANY(KW_INTEGER)) {
170                         match(KW_INTEGER);
171                         for (p = t; p; p = p->next) {
```

```
172                             p->idp->kind = INT_ARRVAR_IDENT;
173                             p->idp->length = arrlen;
174                     }
175             } else if (TOKANY(KW_CHAR)) {
176                     match(KW_CHAR);
177                     for (p = t; p; p = p->next) {
178                             p->idp->kind = CHAR_ARRVAR_IDENT;
179                             p->idp->length = arrlen;
180                     }
181             } else {
182                     unlikely();
183             }
184             break;
185     default:
186             unlikely();
187     }
188
189     return t;
190 }
191
192 /**
193  * pfdeclist ->
194  *         { procdec | fundec }
195  */
196 static pf_dec_list_node_t *parse_pf_dec_list(void)
197 {
198     pf_dec_list_node_t *t, *p, *q;
199
200     for (p = t = NULL; TOKANY2(KW_FUNCTION, KW_PROCEDURE); p = q) {
201             NEWNODE(pf_dec_list_node_t, q);
202             if (!p) {
203                     t = q;
204             } else {
205                     p->next = q;
206             }
207             switch (currtok) {
208             case KW_PROCEDURE:
209                     q->kind = PROC_PFDEC;
210                     q->pdp = parse_proc_dec();
211                     break;
212             case KW_FUNCTION:
213                     q->kind = FUN_PFDEC;
```

```
214                         q->fdp = parse_fun_dec();
215                         break;
216                 default:
217                         unlikely();
218                 }
219         }
220
221         return t;
222 }
223
224 /**
225  * procdec ->
226  *         procdef {; procdef};
227  */
228 static proc_dec_node_t *parse_proc_dec(void)
229 {
230         proc_dec_node_t *t, *p, *q;
231         NEWNODE(proc_dec_node_t, t);
232
233         t->pdp = parse_proc_def();
234         match(SS_SEMI);
235
236         for (p = t; TOKANY(KW_PROCEDURE); p = q) {
237                 NEWNODE(proc_dec_node_t, q);
238                 p->next = q;
239                 q->pdp = parse_proc_def();
240                 match(SS_SEMI);
241         }
242
243         return t;
244 }
245
246 /**
247  * procdef ->
248  *         prochead block
249  */
250 static proc_def_node_t *parse_proc_def(void)
251 {
252         proc_def_node_t *t;
253         NEWNODE(proc_def_node_t, t);
254         t->php = parse_proc_head();
255         t->bp = parse_block();
```

```
256         return t;
257 }
258
259 /**
260  * prochead ->
261  *         PROCEDURE ident '(' [paralist] ')' ;
262  */
263 static proc_head_node_t *parse_proc_head(void)
264 {
265         proc_head_node_t *t;
266         NEWNODE(proc_head_node_t, t);
267
268         match(KW_PROCEDURE);
269         t->idp = parse_ident(READCURR);
270         t->idp->kind = PROC_IDENT;
271
272         match(SS_LPAR);
273         if (TOKANY2(KW_VAR, MC_ID)) {
274                 t->plp = parse_para_list();
275         }
276         match(SS_RPAR);
277         match(SS_SEMI);
278
279         return t;
280 }
281
282 /**
283  * fundec ->
284  *         fundef {; fundef};
285  */
286 static fun_dec_node_t *parse_fun_dec(void)
287 {
288         fun_dec_node_t *t, *p, *q;
289         NEWNODE(fun_dec_node_t, t);
290
291         t->fdp = parse_fun_def();
292         match(SS_SEMI);
293
294         for (p = t; TOKANY(KW_FUNCTION); p = q) {
295                 NEWNODE(fun_dec_node_t, q);
296                 p->next = q;
297                 q->fdp = parse_fun_def();
```

```
298                 match(SS_SEMI);
299         }
300
301         return t;
302 }
303
304 /**
305  * fundef ->
306  *         funhead block
307  */
308 static fun_def_node_t *parse_fun_def(void)
309 {
310         fun_def_node_t *t;
311         NEWNODE(fun_def_node_t, t);
312
313         t->fhp = parse_fun_head();
314         t->bp = parse_block();
315
316         return t;
317 }
318
319 /**
320  * funhead ->
321  *         FUNCTION ident '(' [paralist] ')' : basictype ;
322  */
323 static fun_head_node_t *parse_fun_head(void)
324 {
325         fun_head_node_t *t;
326         NEWNODE(fun_head_node_t, t);
327
328         match(KW_FUNCTION);
329         t->idp = parse_ident(READCURR);
330         match(SS_LPAR);
331         if (TOKANY2(KW_VAR, MC_ID)) {
332                 t->plp = parse_para_list();
333         }
334         match(SS_RPAR);
335         match(SS_COLON);
336
337         switch (currtok) {
338         case KW_INTEGER:
339                 match(KW_INTEGER);
```

```
340                 t->idp->kind = INT_FUN_IDENT;
341                 break;
342         case KW_CHAR:
343                 match(KW_CHAR);
344                 t->idp->kind = CHAR_FUN_IDENT;
345                 break;
346         default:
347                 unlikely();
348         }
349         match(SS_SEMI);
350
351         return t;
352 }
```

这里上述解析分程序代码中的一些重要的函数处理流程进行说明。

文法（1.3）描述了常量声明的结构，它表示一个常量声明可以由一个或者多个常量定义来表示，所以在 parse_const_dec() 函数的第 41 到第 46 行通过 for 循环的方式来解析常量定义，如果可以匹配到 SS_COMMA ，则申请新的常量定义节点 q 并通过尾插法将节点插入 p->next 。后序的常量定义节点 q->cdp 通过递归调用 parse_const_def() 函数来解析。

函数 parse_var_def() 函数是解析变量定义的函数，在代码的第 148 至 150 行、第 154 至 156 行对链表遍历设置对应的 p->idp->kind 值，原因是变量定义出现形如 var x, y, z :integer 时，在解析到 x 标识符时无法确定其种类，只有当 y 和 z 都解析完成，最后在解析到 integer 这个 token 时才能知道 x 的种类，所以必需通过后置处理来赋值。数组变量类型的定义也是类似的逻辑。另外，第 158 至 184 行是解析数组类型的定义，根据文法（1.12）中对数组类型的定义，数组定义是形如 array[10] of char 类型，这里还需要读取数组长度，存在 arrlen 变量中，最后 arrlen 会赋值到 p->idp->length 中存储。

在文法（1.2）中规定，过程和函数可以交替定义，即如下源代码片段是合法的。

```
1 procedure p1(); begin end;
2 procedure p2(); begin end;
3 function  f1(): integer; begin end;
4 function  f2(): char; begin end;
5 procedure p3(); begin end;
```

所以我们对文法进行了小幅度的修改，引入了一个中间节点 pf_dec_list_node_t 结构体，该结构体可以记录过程或者函数，并通过 pf_dec_t kind 属性区分是过程还是函数。与之对应的解析函数 parse_pf_dec_list() 可以解析过程或者函数。后续对过程或者函数的解析虽然比较冗长，但是其思想和上面的常量或变量的解析类似，这里不作过多赘述。

3.5.5 语句的解析

语句是一个程序的基本要素，我们常见的语言都是由一条条语句组成的。语句的解析入口函数是 parse_stmt() ，该函数通过 currtok 来将语句的种类分流，具体有以下几类函数：

1. parse_if_stmt() 解析 if 条件语句。
2. parse_repe_stmt() 解析 repeat 重复循环语句。
3. parse_comp_stmt() 解析复杂语句。
4. parse_read_stmt() 解析读语句。
5. parse_write_stmt() 解析写语句。
6. parse_for_stmt() 解析 for 循环语句。
7. parse_pcall_stmt() 解析过程调用语句。
8. parse_assign_stmt() 解析复制语句。
9. 如果上述情况都不满足，则判定为空语句。

这部分的代码是根据 $PL/0\epsilon$ 的文法对照编写的，处理逻辑不是很复杂，具体代码如下：

```
/**
 * statement ->
 *         assignstmt | ifstmt | repeatstmt | pcallstmt | compstmt
 *                 readstmt | writestmt | forstmt | nullstmt
 */
static stmt_node_t *parse_stmt(void)
{
        stmt_node_t *t;
        NEWNODE(stmt_node_t, t);

        switch (currtok) {
        case KW_IF:
                t->kind = IF_STMT;
                t->ifp = parse_if_stmt();
                break;
        case KW_REPEAT:
                t->kind = REPEAT_STMT;
                t->rpp = parse_repe_stmt();
                break;
        case KW_BEGIN:
                t->kind = COMP_STMT;
                t->cpp = parse_comp_stmt();
                break;
        case KW_READ:
                t->kind = READ_STMT;
                t->rdp = parse_read_stmt();
                break;
```

```
28          case KW_WRITE:
29                  t->kind = WRITE_STMT;
30                  t->wtp = parse_write_stmt();
31                  break;
32          case KW_FOR:
33                  t->kind = FOR_STMT;
34                  t->frp = parse_for_stmt();
35                  break;
36          case MC_ID:
37                  match(MC_ID);
38                  if (TOKANY(SS_LPAR)) {
39                          t->kind = PCALL_STMT;
40                          t->pcp = parse_pcall_stmt();
41                  } else if (TOKANY2(SS_ASGN, SS_LBRA)) {
42                          t->kind = ASSGIN_STMT;
43                          t->asp = parse_assign_stmt();
44                  } else if (TOKANY(SS_EQU)) {
45                          t->kind = ASSGIN_STMT;
46                          t->asp = parse_assign_stmt();
47                          rescue(ERRTOK, "L%d: bad token, = may be :=", lineno);
48                  } else {
49                          unlikely();
50                  }
51                  break;
52          default:
53                  t->kind = NULL_STMT;
54                  break;
55          }
56
57          return t;
58  }
59
60  /**
61   * remember in the statement build function
62   * we have read the token from ident to := or '['
63   *
64   * assignstmt ->
65   *         ident := expression | funident := expression
66   *                 | ident '[' expression ']' := expression
67   */
68  static assign_stmt_node_t *parse_assign_stmt(void)
69  {
```

```
70          assign_stmt_node_t *t;
71          NEWNODE(assign_stmt_node_t, t);
72
73          switch (currtok) {
74          case SS_ASGN:
75                  t->kind = NORM_ASSGIN;
76                  t->idp = parse_ident(READPREV);
77                  match(SS_ASGN);
78                  t->lep = NULL;
79                  t->rep = parse_expr();
80                  break;
81          case SS_LBRA:
82                  t->kind = ARRAY_ASSGIN;
83                  t->idp = parse_ident(READPREV);
84                  match(SS_LBRA);
85                  t->lep = parse_expr();
86                  match(SS_RBRA);
87                  match(SS_ASGN);
88                  t->rep = parse_expr();
89                  break;
90          case SS_EQU: // bad case
91                  t->kind = NORM_ASSGIN;
92                  t->idp = parse_ident(READPREV);
93                  match(SS_EQU);
94                  t->lep = NULL;
95                  t->rep = parse_expr();
96                  break;
97          default:
98                  unlikely();
99          }
100
101         return t;
102 }
103
104 /**
105  * ifstmt ->
106  *        IF condition THEN statement |
107  *                IF condition THEN statement ELSE statement
108  */
109 static if_stmt_node_t *parse_if_stmt(void)
110 {
111         if_stmt_node_t *t;
```

```
112         NEWNODE(if_stmt_node_t, t);
113
114         match(KW_IF);
115         t->cp = parse_cond();
116
117         match(KW_THEN);
118         t->tp = parse_stmt();
119
120         if (TOKANY(KW_ELSE)) {
121                 match(KW_ELSE);
122                 t->ep = parse_stmt();
123         }
124
125         return t;
126 }
127
128 /**
129  * repeatstmt ->
130  *         REPEAT statement UNTIL condition
131  */
132 static repe_stmt_node_t *parse_repe_stmt(void)
133 {
134         repe_stmt_node_t *t;
135         NEWNODE(repe_stmt_node_t, t);
136
137         match(KW_REPEAT);
138         t->sp = parse_stmt();
139         match(KW_UNTIL);
140         t->cp = parse_cond();
141
142         return t;
143 }
144
145 /**
146  * forstmt ->
147  *         FOR ident := expression ( TO | DOWNTO ) expression DO statement
148  */
149 static for_stmt_node_t *parse_for_stmt(void)
150 {
151         for_stmt_node_t *t;
152         NEWNODE(for_stmt_node_t, t);
153
```

```
154         match(KW_FOR);
155         t->idp = parse_ident(READCURR);
156         match(SS_ASGN);
157
158         t->lep = parse_expr();
159
160         switch (currtok) {
161         case KW_TO:
162                 match(KW_TO);
163                 t->kind = TO_FOR;
164                 break;
165         case KW_DOWNTO:
166                 match(KW_DOWNTO);
167                 t->kind = DOWNTO_FOR;
168                 break;
169         default:
170                 unlikely();
171         }
172
173         t->rep = parse_expr();
174         match(KW_DO);
175         t->sp = parse_stmt();
176
177         return t;
178 }
179
180 /**
181  * remember in the statement build function
182  * we have read the token from ident to (
183  *
184  * pcallstmt ->
185  *         ident '(' [arglist] ')'
186  */
187 static pcall_stmt_node_t *parse_pcall_stmt(void)
188 {
189         pcall_stmt_node_t *t;
190         NEWNODE(pcall_stmt_node_t, t);
191
192         t->idp = parse_ident(READPREV);
193         match(SS_LPAR);
194
195         if (TOKANY6(MC_ID, MC_CH, SS_PLUS, SS_MINUS, MC_UNS, SS_LPAR)) {
```

```
196                 t->alp = parse_arg_list();
197         }
198         match(SS_RPAR);
199 
200         return t;
201 }
202 
203 /**
204  * remember in the factor build function
205  * we have read the token from ident to (
206  *
207  * fcallstmt ->
208  *         ident '(' [arglist] ')'
209  */
210 static fcall_stmt_node_t *parse_fcall_stmt(void)
211 {
212         fcall_stmt_node_t *t;
213         NEWNODE(fcall_stmt_node_t, t);
214         t->idp = parse_ident(READPREV);
215         match(SS_LPAR);
216 
217         if (TOKANY6(MC_ID, MC_CH, SS_PLUS, SS_MINUS, MC_UNS, SS_LPAR)) {
218                 t->alp = parse_arg_list();
219         }
220         match(SS_RPAR);
221 
222         return t;
223 }
224 
225 /**
226  * compstmt ->
227  *         BEGIN statement {; statement} END
228  */
229 static comp_stmt_node_t *parse_comp_stmt(void)
230 {
231         comp_stmt_node_t *t, *p, *q;
232         NEWNODE(comp_stmt_node_t, t);
233         match(KW_BEGIN);
234         t->sp = parse_stmt();
235 
236         for (p = t; TOKANY(SS_SEMI); p = q) {
237                 match(SS_SEMI);
```

```
238                 NEWNODE(comp_stmt_node_t, q);
239                 p->next = q;
240                 q->sp = parse_stmt();
241         }
242
243         match(KW_END);
244
245         return t;
246 }
247
248 /**
249  * readstmt ->
250  *         READ '(' ident {, ident} ')'
251  */
252 static read_stmt_node_t *parse_read_stmt(void)
253 {
254         read_stmt_node_t *t, *p, *q;
255         NEWNODE(read_stmt_node_t, t);
256
257         match(KW_READ);
258         match(SS_LPAR);
259         t->idp = parse_ident(READCURR);
260         for (p = t; TOKANY(SS_COMMA); p = q) {
261                 match(SS_COMMA);
262                 NEWNODE(read_stmt_node_t, q);
263                 p->next = q;
264                 q->idp = parse_ident(READCURR);
265         }
266         match(SS_RPAR);
267
268         return t;
269 }
270
271 /**
272  * writestmt ->
273  *         WRITE '(' string, expression ')' | WRITE '(' string ')' |
274  *                 WRITE '(' expression ')'
275  */
276 static write_stmt_node_t *parse_write_stmt(void)
277 {
278         write_stmt_node_t *t;
279         NEWNODE(write_stmt_node_t, t);
```

```
280
281         match(KW_WRITE);
282         match(SS_LPAR);
283         if (TOKANY(MC_STR)) {
284                 t->type = STR_WRITE;
285                 strcopy(t->sp, tokbuf);
286                 match(MC_STR);
287         } else if (TOKANY6(MC_ID, MC_CH, SS_PLUS, SS_MINUS, MC_UNS, SS_LPAR)) {
288                 t->type = ID_WRITE;
289                 t->ep = parse_expr();
290         } else {
291                 unlikely();
292         }
293         if (TOKANY(SS_COMMA) && t->type == STR_WRITE) {
294                 match(SS_COMMA);
295                 t->type = STRID_WRITE;
296                 t->ep = parse_expr();
297         }
298         match(SS_RPAR);
299
300         return t;
301 }
```

我们对 parse_assign_stmt() 函数读取标识符的情景进行一下说明，判断是赋值语句通常有以下两种情况，即普通赋值和数组赋值。

```
1 x    := 1;       { 普通赋值 }
2 a[2] := x + 1;   { 数组赋值 }
```

这两种情况需要记录的标识符时，currtok 已经读取到 SS_ASGN 或者 SS_LBRA ，所以保存的标识符需要上一个词位，即调用 parse_ident(READPREV) 函数。

在上述代码第 90 至 96 行中处理了 SS_EQU 的情景，并标记了 bad case （错误场景），其原因是：当出现 x = 1; 这样的语句时，它属于一个错误的语句，应该修改成 x := 1; 这样的赋值操作。这里在 parse_stmt() 函数的第 44 至 47 行中对这种常见错误进行了处理，即在第 47 行调用 rescue() 函数标记错误，然后当调用 parse_assign_stmt() 函数是把 SS_EQU 当初 SS_ASGN 来继续后面处理。

3.5.6 表达式和条件的解析

表达式的解析是处理加减乘除四则运算，它的解析函数包括：

1. parse_expr() 函数解析表达式。
2. parse_term() 函数解析项。
3. parse_factor() 函数解析因子。

条件解析仅通过 parse_cond() 函数进行处理。

```
/**
 * expression ->
 *        [+|-] term { addop term }
 */
static expr_node_t *parse_expr(void)
{
        expr_node_t *t, *p, *q;
        NEWNODE(expr_node_t, t);

        // left-most part
        switch (currtok) {
        case SS_PLUS:
                match(SS_PLUS);
                t->kind = ADD_ADDOP;
                t->tp = parse_term();
                break;
        case SS_MINUS:
                match(SS_MINUS);
                t->kind = NEG_ADDOP;
                t->tp = parse_term();
                break;
        case MC_ID:
        case MC_CH:
        case MC_UNS:
        case SS_LPAR:
                t->kind = NOP_ADDOP;
                t->tp = parse_term();
                break;
        default:
                unlikely();
        }

        // reminder part
        for (p = t; TOKANY2(SS_PLUS, SS_MINUS); p = q) {
                NEWNODE(expr_node_t, q);
                p->next = q;
                switch (currtok) {
                case SS_PLUS:
                        match(SS_PLUS);
                        q->kind = ADD_ADDOP;
                        q->tp = parse_term();
```

```
42                      break;
43              case SS_MINUS:
44                      match(SS_MINUS);
45                      q->kind = NEG_ADDOP;
46                      q->tp = parse_term();
47                      break;
48              default:
49                      unlikely();
50              }
51      }
52
53      return t;
54  }
55
56  /**
57   * term ->
58   *         factor { multop factor}
59   */
60  static term_node_t *parse_term(void)
61  {
62      term_node_t *t, *p, *q;
63      NEWNODE(term_node_t, t);
64
65      t->kind = NOP_MULTOP;
66      t->fp = parse_factor();
67
68      for (p = t; TOKANY2(SS_STAR, SS_OVER); p = q) {
69              NEWNODE(term_node_t, q);
70              p->next = q;
71              switch (currtok) {
72              case SS_STAR:
73                      match(SS_STAR);
74                      q->kind = MULT_MULTOP;
75                      q->fp = parse_factor();
76                      break;
77              case SS_OVER:
78                      match(SS_OVER);
79                      q->kind = DIV_MULTOP;
80                      q->fp = parse_factor();
81                      break;
82              default:
83                      unlikely();
```

```
84                  }
85          }
86
87          return t;
88  }
89
90  /**
91   * factor ->
92   *         ident | ident '[' expression ']' | unsign
93   *                 | '(' expression ')' | fcallstmt
94   */
95  static factor_node_t *parse_factor(void)
96  {
97          factor_node_t *t;
98          NEWNODE(factor_node_t, t);
99
100         switch (currtok) {
101         case MC_UNS:
102                 t->kind = UNSIGN_FACTOR;
103                 t->value = atoi(tokbuf);
104                 match(MC_UNS);
105                 break;
106         case MC_CH:
107                 t->kind = CHAR_FACTOR;
108                 t->value = (int)tokbuf[0];
109                 match(MC_CH);
110                 break;
111         case SS_LPAR:
112                 match(SS_LPAR);
113                 t->kind = EXPR_FACTOR;
114                 t->ep = parse_expr();
115                 match(SS_RPAR);
116                 break;
117         case MC_ID:
118                 match(MC_ID);
119                 if (TOKANY(SS_LBRA)) {
120                         t->kind = ARRAY_FACTOR;
121                         t->idp = parse_ident(READPREV);
122                         match(SS_LBRA);
123                         t->ep = parse_expr();
124                         match(SS_RBRA);
125                 } else if (TOKANY(SS_LPAR)) {
```

```
126                         t->kind = FUNCALL_FACTOR;
127                         t->fcsp = parse_fcall_stmt();
128                 } else {
129                         t->kind = ID_FACTOR;
130                         t->idp = parse_ident(READPREV);
131                 }
132                 break;
133         default:
134                 unlikely();
135         }
136
137         return t;
138 }
139
140 /**
141  * condition ->
142  *         expression relop expression
143  */
144 static cond_node_t *parse_cond(void)
145 {
146         cond_node_t *t;
147         NEWNODE(cond_node_t, t);
148
149         t->lep = parse_expr();
150         switch (currtok) {
151         case SS_EQU:
152                 match(SS_EQU);
153                 t->kind = EQU_RELA;
154                 break;
155         case SS_LST:
156                 match(SS_LST);
157                 t->kind = LST_RELA;
158                 break;
159         case SS_LEQ:
160                 match(SS_LEQ);
161                 t->kind = LEQ_RELA;
162                 break;
163         case SS_GTT:
164                 match(SS_GTT);
165                 t->kind = GTT_RELA;
166                 break;
167         case SS_GEQ:
```

```
168                 match(SS_GEQ);
169                 t->kind = GEQ_RELA;
170                 break;
171         case SS_NEQ:
172                 match(SS_NEQ);
173                 t->kind = NEQ_RELA;
174                 break;
175         default:
176                 unlikely();
177         }
178         t->rep = parse_expr();
179 
180         return t;
181 }
```

上述解析函数最复杂的是解析因子的 parse_factor() 函数，该函数解析了以下类型的因子：

1. 如果出现 MC_UNS 表示无符号的因子。
2. 如果出现 MC_CH 表示字符类型的因子。
3. 如果出现 SS_LPAR 表示出现的时表达式因子。
4. 如果出现 MC_ID 还需要在读取一个 token 并根据以下情景讨论：
   - 如果是 SS_LBRA 表示数组类型因子，
   - 如果是 SS_LPAR 表示函数调用类型因子，
   - 其他的都是标记符类型的因子。

其他的解析函数比较直观，这里不作过多说明。

3.5.7 标识符、形参和实参的解析

这部分包括对标识符、形参和实参的解析，这些解析函数如下：

1. parse_ident() 函数解析标识符。
2. parse_para_list() 函数解析形参列表。
3. parse_arg_list() 函数解析实参列表。

具体的实现代码如下：

```
1 /**
2  * construct a identifier
3  */
4 static ident_node_t *parse_ident(idreadmode_t mode)
5 {
6         ident_node_t *t;
7         NEWNODE(ident_node_t, t);
8 
```

```
9           switch (mode) {
10          case READCURR:
11                  t->kind = INIT_IDENT;
12                  t->value = 0;
13                  t->length = 0;
14                  t->line = lineno;
15                  strcopy(t->name, tokbuf);
16                  match(MC_ID);
17                  break;
18          case READPREV:
19                  t->kind = INIT_IDENT;
20                  t->value = 0;
21                  t->length = 0;
22                  t->line = prevlineno;
23                  strcopy(t->name, prevtokbuf);
24                  break;
25          default:
26                  unlikely();
27          }
28          return t;
29  }
30
31  /**
32   * paralist ->
33   *         paradef {; paradef }
34   */
35  static para_list_node_t *parse_para_list(void)
36  {
37          para_list_node_t *t, *p, *q;
38          NEWNODE(para_list_node_t, t);
39
40          t->pdp = parse_para_def();
41          for (p = t; TOKANY(SS_SEMI); p = q) {
42                  match(SS_SEMI);
43                  NEWNODE(para_list_node_t, q);
44                  p->next = q;
45                  q->pdp = parse_para_def();
46          }
47
48          return t;
49  }
50
```

```
51  /**
52   * paradef ->
53   *        [VAR] ident {, ident} : basictype
54   */
55  static para_def_node_t *parse_para_def(void)
56  {
57          para_def_node_t *t, *p, *q;
58          NEWNODE(para_def_node_t, t);
59
60          // VAR mean call by reference
61          bool byref = FALSE;
62          if (TOKANY(KW_VAR)) {
63                  byref = TRUE;
64                  match(KW_VAR);
65          }
66
67          t->idp = parse_ident(READCURR);
68
69          for (p = t; TOKANY(SS_COMMA); p = q) {
70                  match(SS_COMMA);
71                  NEWNODE(para_def_node_t, q);
72                  p->next = q;
73                  q->idp = parse_ident(READCURR);
74          }
75          match(SS_COLON);
76
77          switch (currtok) {
78          case KW_INTEGER:
79                  match(KW_INTEGER);
80                  for (p = t; p; p = p->next) {
81                          p->idp->kind =
82                                  byref ? INT_BYADR_IDENT : INT_BYVAL_IDENT;
83                  }
84                  break;
85          case KW_CHAR:
86                  match(KW_CHAR);
87                  for (p = t; p; p = p->next) {
88                          p->idp->kind =
89                                  byref ? CHAR_BYADR_IDENT : CHAR_BYVAL_IDENT;
90                  }
91                  break;
92          default:
```

```
93                  unlikely();
94          }
95
96          return t;
97  }
98
99  /**
100  * arglist ->
101  *         argument {, argument}
102  *
103  * argument ->
104  *         expression
105  */
106 static arg_list_node_t *parse_arg_list(void)
107 {
108         arg_list_node_t *t, *p, *q;
109         NEWNODE(arg_list_node_t, t);
110
111         t->ep = parse_expr();
112
113         for (p = t; TOKANY(SS_COMMA); p = q) {
114                 match(SS_COMMA);
115                 NEWNODE(arg_list_node_t, q);
116                 p->next = q;
117                 q->ep = parse_expr();
118         }
119
120         return t;
121 }
```


在 parse_ident() 函数中设置了标识符名字 t->name 为 token 的词位。设置了标识符所在行数 t->line （方便后续阶段追踪)。当读取模式为 READCURR 时，还需要调用 match() 函数跳过已经读取的 token 。

在 parse_para_def() 函数中，需要注意的是第 61 到 66 行处理了 var 关键字的语句，这部分在后续的语言分析中还会进一步讨论。然后关于形参 parameter 和实参 argument 的解析就是按照文法的定义进行递归下降解析。

3.6 语法分析细节调试

为了理解递归下降分析法的精髓，我们通过调试的方式来说明语法分析的解析函数调用细节。编译好所有代码后会在根目录得到 pcc 可执行文件，通过 gdb 启动 pcc 编译器来解析 twosum.pas

的代码。对应的命令如下：

```
gdb --args ./pcc ./example/twosum.pas
```

当程序运行后，我们对程序进行如下的调试，最终程序解析到节点 **#6** ，其对应的 `nid=6` ，执行路径见图 3.5 ，其中灰底的标记的是已经解析的节点。

```
$ gdb --args ./pcc ./example/twosum.pas
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./pcc...
(gdb) b parse_ident   <= 添加断点
Breakpoint 1 at 0x862e: file source/parse.c, line 910.
(gdb) r               <= 启动程序
Starting program: ./pcc ./example/twosum.pas
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
compiler pcc start, version v0.18.0
reading file ./example/twosum.pas

Breakpoint 1, parse_ident (mode=READCURR) at source/parse.c:910
910             NEWNODE(ident_node_t, t);
(gdb) bt              <= 打印当前调用栈
#0  parse_ident (mode=READCURR) at source/parse.c:910
#1  0x000055555555a308 in parse_const_def () at source/parse.c:128
#2  0x000055555555a1a7 in parse_const_dec () at source/parse.c:105
#3  0x000055555555a0bb in parse_block () at source/parse.c:77
#4  0x000055555555a017 in parse_pgm () at source/parse.c:62
#5  0x000055555555cc7a in parse () at source/parse.c:1029
#6  0x0000555555562ecd in main (argc=2, argv=0x7fffffffd828) at source/main.c:15
(gdb) n               <= 执行 parse_ident 的 NEWNODE 宏初始化节点
```

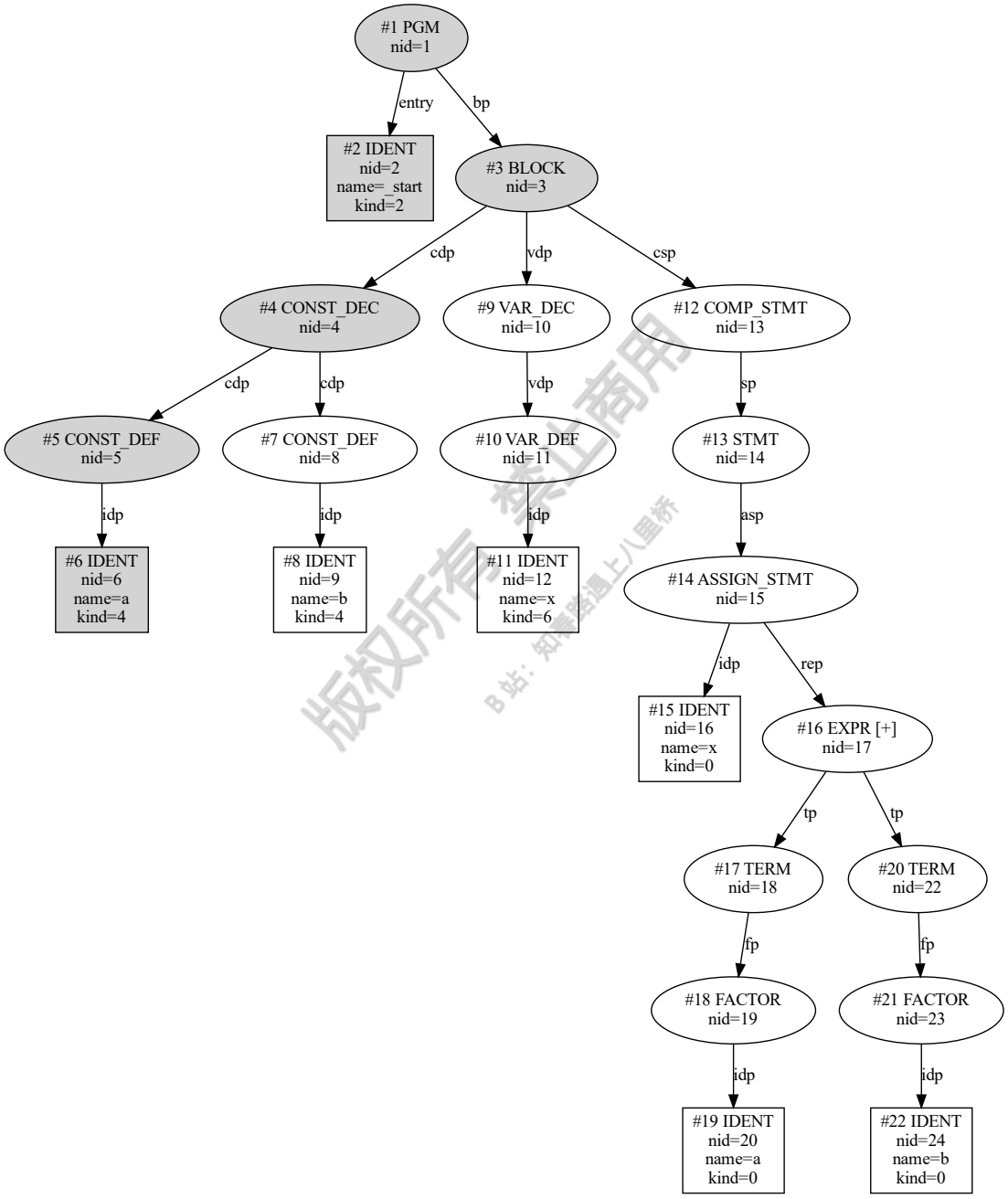


图 3.5: 源文件 twosum.pas 解析到常量 a 标识符时的语法树快照

```
912             switch (mode) {
(gdb) p t->nid        <= 打印节点的 nid, 其值为 6
$1 = 6
(gdb)
```

我们研究一下解析函数的调用栈，它们分别是解析图 3.5 中的节点

1. parse_ident (mode=READCURR) at source/parse.c:910 解析节点 **#6**
2. parse_const_def () at source/parse.c:128 解析节点 **#5**
3. parse_const_dec () at source/parse.c:105 解析节点 **#4**
4. parse_block () at source/parse.c:77 解析节点 **#3**
5. parse_pgm () at source/parse.c:62 解析节点 **#1**
6. parse () at source/parse.c:1029
7. main (argc=2, argv=0x7fffffffd828) at source/main.c:15

此时的递归深度为 5，解析访问路径为：

$$\#6 \leftarrow \#5 \leftarrow \#4 \leftarrow \#3 \leftarrow \#1$$

从图 3.5 解析过程来看，parse_xxx() 函数之间调用是递归的，并且在解析过程顺序是对语法树从根节点依次往下深入解析，结合上述两点，递归下降分析法名称的由来也就不言而喻了。

继续之前的调试，我们在 parse_factor() 函数处打一个断点，让程序继续执行到第一个因子的解析函数中，调试过程如下：

```
(gdb) i b             <= 查看当前断点
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000055555555c62e in parse_ident at source/parse.c:910
        breakpoint already hit 1 time
(gdb) del 1           <= 删除 parse_ident 函数的断点
(gdb) b parse_factor <= 添加 parse_factor 断点
Breakpoint 2 at 0x55555555c296: file source/parse.c, line 819.
(gdb) i b             <= 查看确保断点设置成功
Num     Type           Disp Enb Address            What
2       breakpoint     keep y   0x000055555555c296 in parse_factor at source/parse.c:819
(gdb) c               <= 继续运行
Continuing.
Breakpoint 2, parse_factor () at source/parse.c:819
819             NEWNODE(factor_node_t, t);
(gdb) n               <= 运行初始化宏 NEWNODE
821             switch (currtok) {
(gdb) bt              <= 查看调用栈
#0  parse_factor () at source/parse.c:821
#1  0x000055555555c149 in parse_term () at source/parse.c:787
#2  0x000055555555bf50 in parse_expr () at source/parse.c:748
#3  0x000055555555b45e in parse_assign_stmt () at source/parse.c:498
```

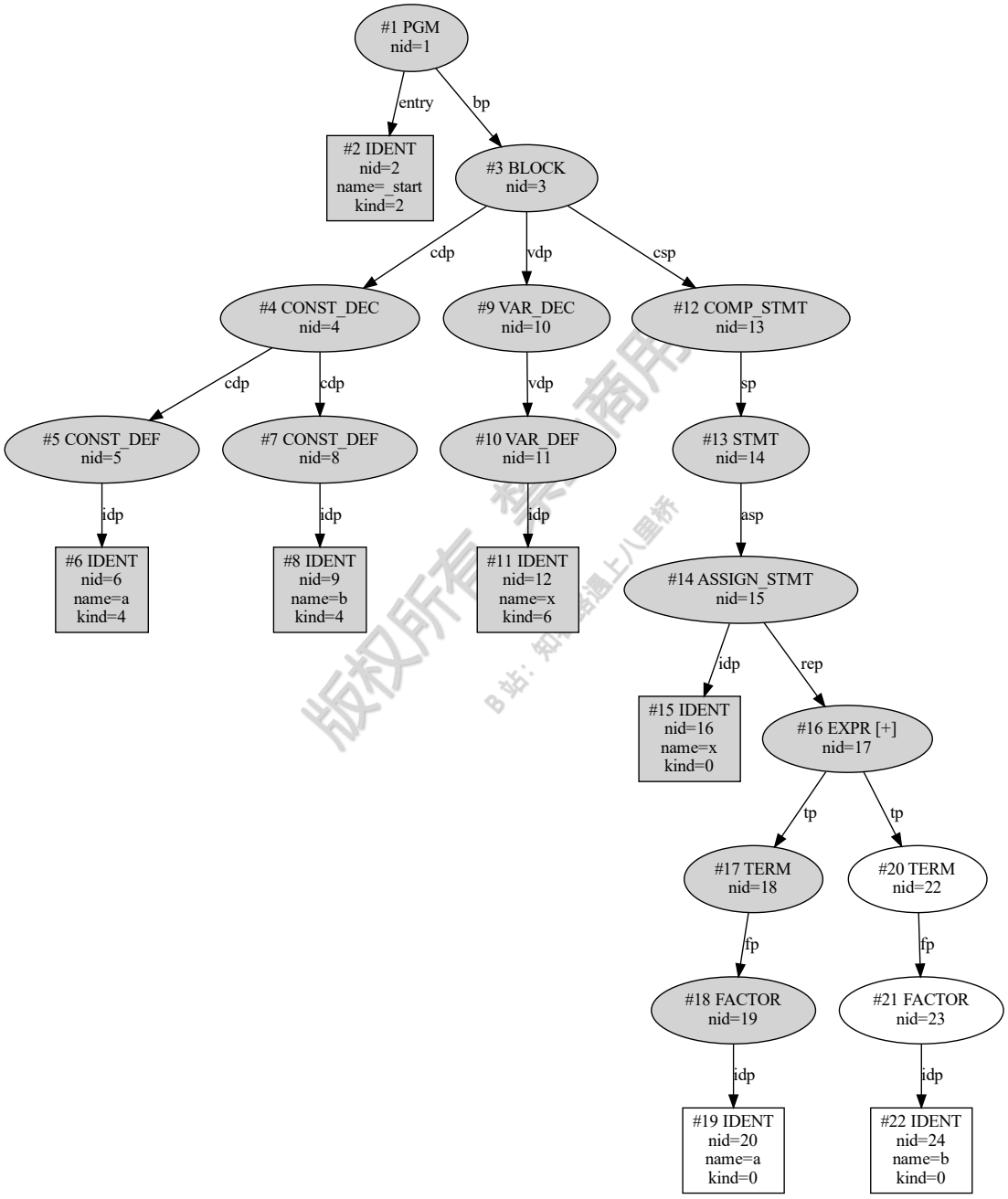


图 3.6: 源文件 twosum.pas 解析到第一个因子时的语法树快照

```
#4  0x000055555555b2d0 in parse_stmt () at source/parse.c:462
#5  0x000055555555ba70 in parse_comp_stmt () at source/parse.c:653
#6  0x000055555555a10e in parse_block () at source/parse.c:89
#7  0x000055555555a017 in parse_pgm () at source/parse.c:62
#8  0x000055555555cc7a in parse () at source/parse.c:1029
#9  0x0000555555562ecd in main (argc=2, argv=0x7fffffffd828) at source/main.c:15
(gdb) p t->nid       <= 查看当前的 nid
$2 = 19
(gdb)
```

此时语法树解析到的节点见图 3.6，当前进入第一个因子的解析，处于图中的节点 **#18** （该节点 nid=19 ），我们继续将调用栈中的解析函数和语法树节点对应一下：

1. parse_factor () at source/parse.c:821 解析节点 **#18**
2. parse_term () at source/parse.c:787 解析节点 **#17**
3. parse_expr () at source/parse.c:748 解析节点 **#16**
4. parse_assign_stmt () at source/parse.c:498 解析节点 **#14**
5. parse_stmt () at source/parse.c:462 解析节点 **#13**
6. parse_comp_stmt () at source/parse.c:653 解析节点 **#12**
7. parse_block () at source/parse.c:89 解析节点 **#3**
8. parse_pgm () at source/parse.c:62 解析节点 **#1**
9. parse () at source/parse.c:1029
10. main (argc=2, argv=0x7fffffffd828) at source/main.c:15

此时的递归深度为 8，解析访问路径为：

$$\textbf{\#18} \leftarrow \textbf{\#17} \leftarrow \textbf{\#16} \leftarrow \textbf{\#14} \leftarrow \textbf{\#13} \leftarrow \textbf{\#12} \leftarrow \textbf{\#3} \leftarrow \textbf{\#1}$$

综上所述，我们已经可以直观地理解使用递归下降分析法来进行语法分析了，并且可以对词法分析产生的 token 流构建出唯一的语法树。

3.7 更多语法树示例

之前的小节都是使用 twosum.pas 这个作为例子讲解语法树，为了加深语法树的理解，这里给出更多的示例来说明语法树的情景。

第一个例子是累加器的代码，它涉及到函数定义和函数调用，对应的语法树见图 3.7，下面是具体的 syn01.pas 的代码

```
1 var u, v, ans: integer;
2 function adder(var x, y : integer):integer;
3 begin
4    adder := x + y
5 end;
```

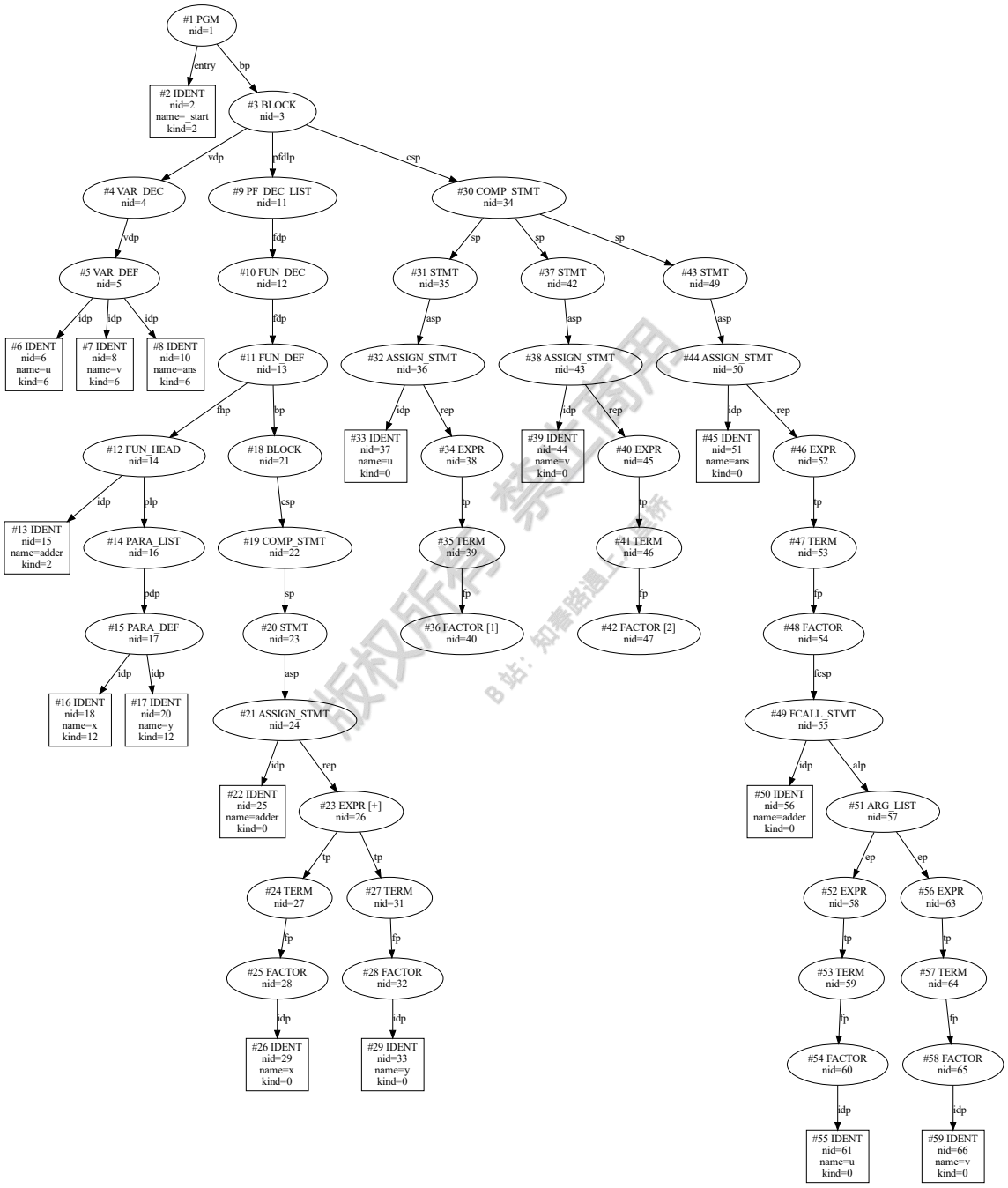


图 3.7: 语法树示例一：函数定义及使用

```
begin
   u := 1;
   v := 2;
   ans := adder(u, v)
end.
```

第二个例子是一个表达式求值的代码，它涉及到数组定义，数组寻址和表达式、项和因子的运算，其核心就是第 4 行的赋值语句，其中包含多种因子的形式。具体实现代码如下，对应的语法树见图 3.8。下面是具体的 syn02.pas 的代码

```
var u, v, ans: integer;
    a : array[10] of integer;
begin
   ans := u / a[v] + (u * v) + a[u+v]
end.
```

第三个例子是一个冒泡排序，它相较于前两个例子更加接近真实场景。代码实现了一个 swap() 过程用于交换两个参数值，主程序通过两个 for 循环对数组 a[] 进行排序。具体实现代码如下，对应的语法树见图 3.9。下面是具体的 syn03.pas 的代码

```
var
   a:array[5] of integer {= (4, 5, 2, 7, 0)};
   i, j: integer;
procedure swap(var x,y:integer);
var
   t: integer;
begin
   t := x;
   x := y;
   y := t
end;
begin { main }
   a[0] := 4; a[1] := 5; a[2] := 2; a[3] := 7; a[4] := 0;
   for i := 0 to 4 do
   begin
      for j := i to 4 do
      begin
         if a[i] > a[j] then
         begin
            swap(a[i], a[j])
         end;
      end;
   end;
   for i := 0 to 4 do
```

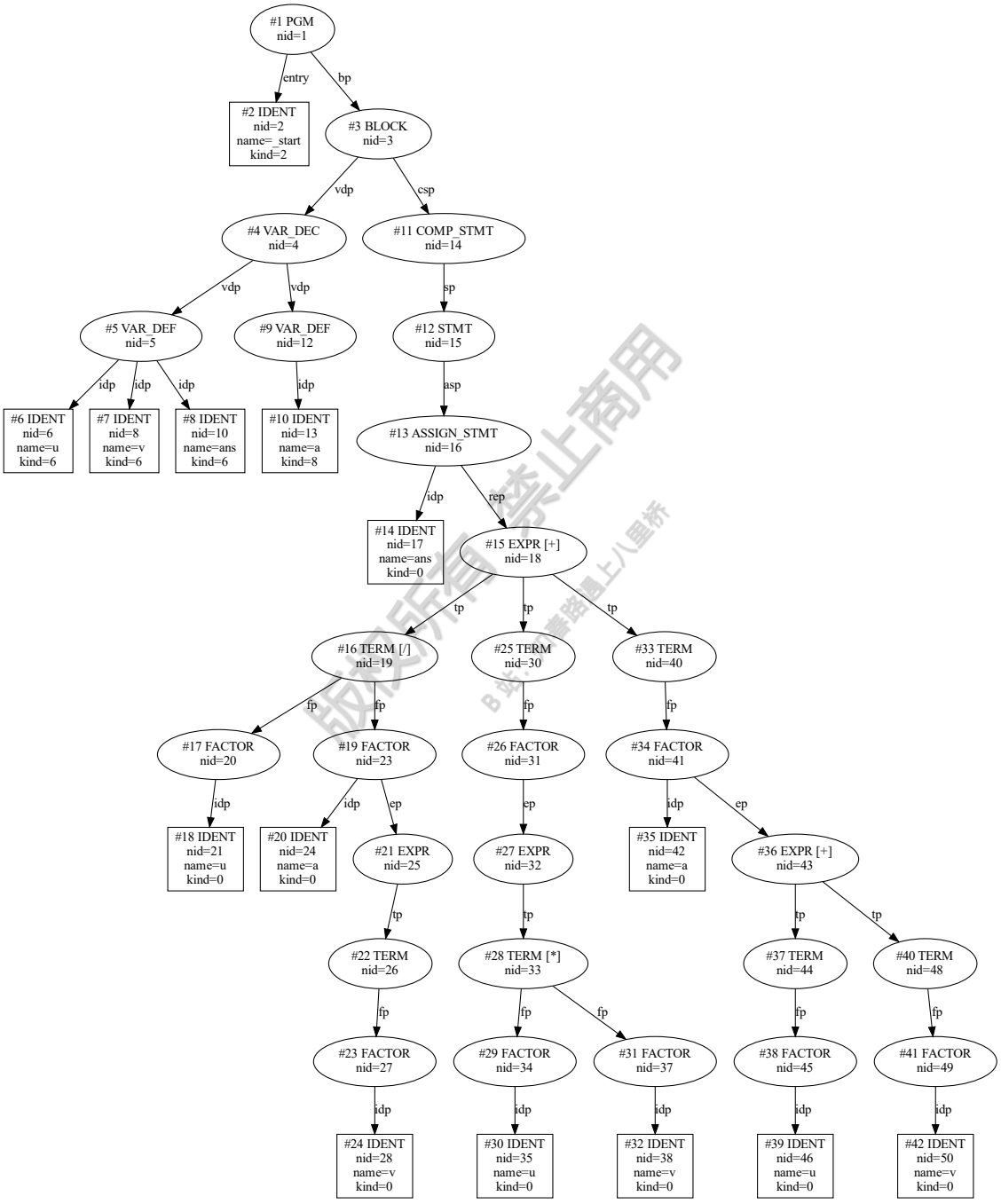


图 3.8: 语法树示例二：带数组的表达式求值

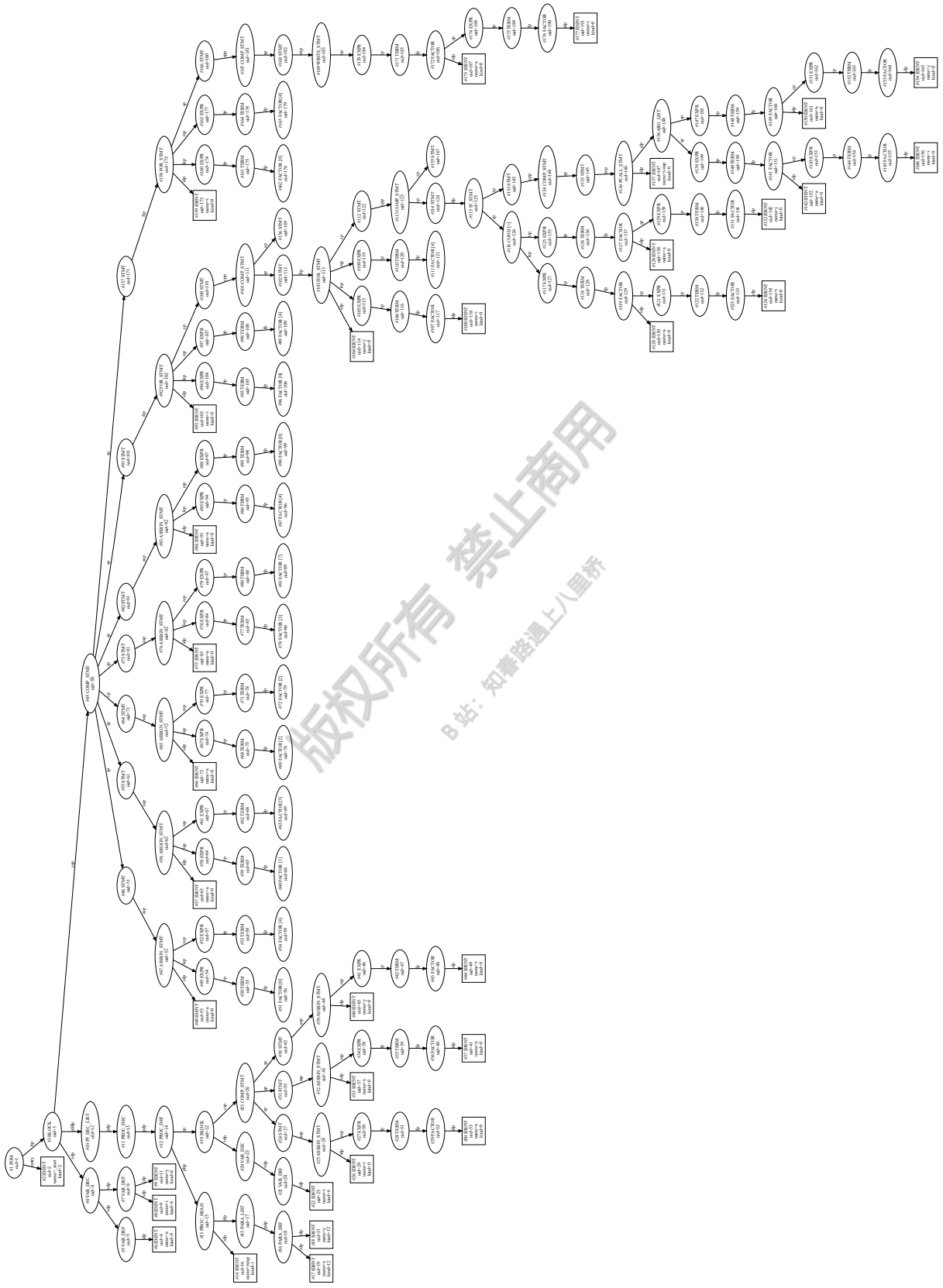


图 3.9：语法树示例三：冒泡排序

```
25    begin
26       write(a[i])
27    end
28 end.
```

3.8 思考题

1. 既然文法 (3.1) - (3.4) 是歧义的，能否修改一下文法从而避免歧义?

2. $PL/0\epsilon$ 语言中描述条件 condition 的是文法 (1.32) - (1.33)，在这样的文法中我们能否使用下面类似 C 语言的条件写法? 如果不能，我们该如何解决这一问题?

```
1  if (x > 0 && x < 10) {
2          // do something
3  }
```

3.9 本章总结

本章介绍了语法分析的细节。包括抽象语法树、文法歧义等一系列语法分析中的理论基础。后续介绍了语法树节点的实现，介绍了 $PL/0\epsilon$ 的递归下降分析法的实现代码。最后对语法分析进行调试并给出更多语法树示例。

第四章　语义分析

4.1　语义分析

如图 4.1 所示，语义分析处于语法分析过后的下一个编译阶段，其目的是对源代码进行语义上的检查，确保其符合语言的规则，并生成相应的语义信息。这个阶段通常包括类型检查、语义分析和静态语义分析等任务。

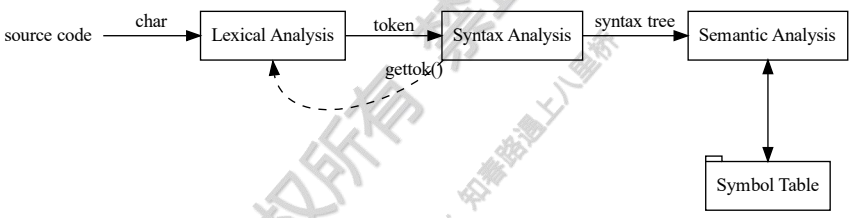


图 4.1：语义分析示意图

1. 类型检查：类型检查是编译过程中对源代码进行语义分析的一个重要部分。它确保源代码中的每个操作符和函数调用都与正确的数据类型一起使用。类型检查可以捕获许多常见的编程错误，例如类型不匹配或未声明的变量。
2. 语义分析：语义分析阶段进一步检查源代码的语义含义，确保其符合语言的语义规则。这包括检查变量和表达式的定义、控制流语句的正确性、函数调用的正确性等。此外，语义分析还负责生成中间代码，这是源代码和目标代码之间的过渡表示。
3. 静态语义分析：静态语义分析是编译器在编译时进行的检查，以确保源代码的语义是正确的。与动态语义分析不同，静态分析在程序运行之前完成。它检查程序中的错误，例如未声明的变量、类型不匹配、未初始化的变量等。静态语义分析有助于提高程序的可靠性和安全性。
4. 符号表：在语义分析过程中，编译器使用符号表来跟踪源代码中定义和引用的标识符的语义信息。符号表是一个数据结构，用于存储标识符的类型、作用域和链接信息等。在编译过程中，符号表用于解决标识符的解析问题，即确定标识符的引用位置和类型。
5. 错误和警告：在语义分析阶段，编译器可能会报告错误或警告消息。错误消息表示源代码中存在无法修复的错误，导致编译失败。警告消息则表示源代码中可能存在问题或不安全的操作，但编译器仍然可以生成可执行文件。

总的来说，语义分析是对源代码进行深入检查的阶段，以确保其符合语言的规则和语义约束。这个阶段的结果是生成中间代码和符号表，为后续的优化和目标代码生成提供必要的信息。

4.2 符号表

4.2.1 符号项

符号表在编译过程中起到了关键的作用，主要用于存储和解析程序中的符号信息。符号表的一条记录被称之为符号项 (Symbol Table Entry)，符号项存储以下几类属性：

1. 通用属性：每个符号都必要的属性，具体如下：
   - `sid` 符号的唯一标记 ID。
   - `name` 当前符号的名称。
   - `cate` 当前符号的分类[1]：标记符号是变量、常量还是函数等，所有分类见表 4.1 。
   - `type` 当前符号的类型：标记符号是整型、字符型等，所有类型见表 4.2 。
2. 附加属性：对于某特定符号扩展出的属性，具体如下：
   - 常量需要有初始值 `initval` 。
   - 数组需要有初始长度 `arrlen` 。
   - 字符串需要有字面量值 `str[]` 。
   - 函数需要有参数列表 `phead` 和 `ptail` 。
3. 链接属性：用于生成目标代码时的地址分配的依据，具体如下：
   - `off` 记录着临时变量和变量的偏移值。
   - `label[]` 记录汇编语言的标号。
4. 调试属性：用于调试或快速查找的属性，具体如下：
   - `lineno` 记录符号所在源文件行数。
   - `stab` 记录符号所在符号表的地址。

表 4.1：符号项的 cate 分类属性值列表

基本分类	0	NOP_OBJ	1	CONST_OBJ	2	VAR_OBJ	3	PROC_OBJ
	4	FUN_OBJ	5	ARRAY_OBJ	6	BYVAL_OBJ	7	BYREF_OBJ
扩充分类	8	TMP_OBJ	9	LABEL_OBJ	10	NUM_OBJ	11	STR_OBJ

表 4.2：符号项的 type 类型属性值列表

基本类型	0	VOID_TYPE	1	INT_TYPE	2	CHAR_TYPE	3	STR_TYPE

这些信息在编译过程中是必要的，有助于编译器进行语义分析和代码生成。符号表还为上下文语义的合法性检查提供依据。例如，通过符号表，编译器可以确定某个标识符在特定作用域内的含义，从而确保程序的语义正确性。在生成目标代码时，符号表中的信息可以用于地址分配。编译器可以通过符号表中的信息确定每个符号的内存地址，从而生成有效的目标代码。

4.2.2 符号表的逻辑结构

多个符号项的集合被称为符号表，它除了包含符号项以外，还有一些自身的属性。具体如下：

[1]cate 是英文分类 category 的简写

1. tid 是符号表的唯一 ID 。
2. nspace[] 是符号表的名称。
3. depth 标记符号表的深度，这个在后续章节中会使用。
4. 符号表还包含一些汇编堆栈分配的属性，为生成汇编代码提供依据，具体如下：
   - argoff 参数的最大偏移
   - varoff 变量的最大偏移
   - tmpoff 临时变量的最大偏移

我在上一章的结束提到累加器例子基础上进行分析，它的代码如下：

```
1  var u, v, ans: integer;
2  function adder(var x, y : integer):integer;
3  begin
4     adder := x + y
5  end;
6  begin
7     u := 1;
8     v := 2;
9     ans := adder(u, v)
10 end.
```

在累加器的语法树添加符号表中包含的信息后，可以得到一个新的语法树，见图 4.2 ，与之前的图 3.7 相对比，见图 4.2 中的 IDENT 节点中添加了一些新的属性，它们分别是 label 、cate 和 type 。这些新的属性就是之前符号项中的通用属性。当然，符号表还包括附加属性和链接属性，这里为了保持语法树的美观先不往上堆数据了。

这里介绍一下图 4.2 中节点 **#33** 中一些属性的具体含义，属性 type=1 ，通过查找表 4.2 可以得知该符号是 INT_TYPE，即为一个整数类型。属性 cate=2 ，通过查找表 4.1 可以得知该符号是 VAR_OBJ，即为一个变量对象。

表 4.3: 累加器进入主程序时的符号表

sid	name	cate	type	value	label
1	_start	4	1	0	L001
2	u	2	1	0	L002
3	v	2	1	0	L003
4	ans	2	1	0	L004
5	adder	4	1	0	L005

我们来接着讨论符号表，当语义分析到第 7 行的主程序时，我们程序已经构造出了一个符号表，它里面包含信息见表 4.3 。可以看出表中已经包含了 5 个符号 [2] ，并且列出了一些通用属性。当语义分析解析到第 7 行数时，遇到 u 这个标识符就可以在符号表中查询，从而获取 u 对应的符号项为 sid=2 的数据项，类似于图 4.2 中节点 **#33** 的属性信息，我们可以获取关于 u 的额外的信息，通过这些额外信息的辅助，语义分析器就可以对程序所表达的语义进行正确理解，针

[2]符号 "_start" 是编译器默认添加到全局符号表中的，它表示程序的入口点。

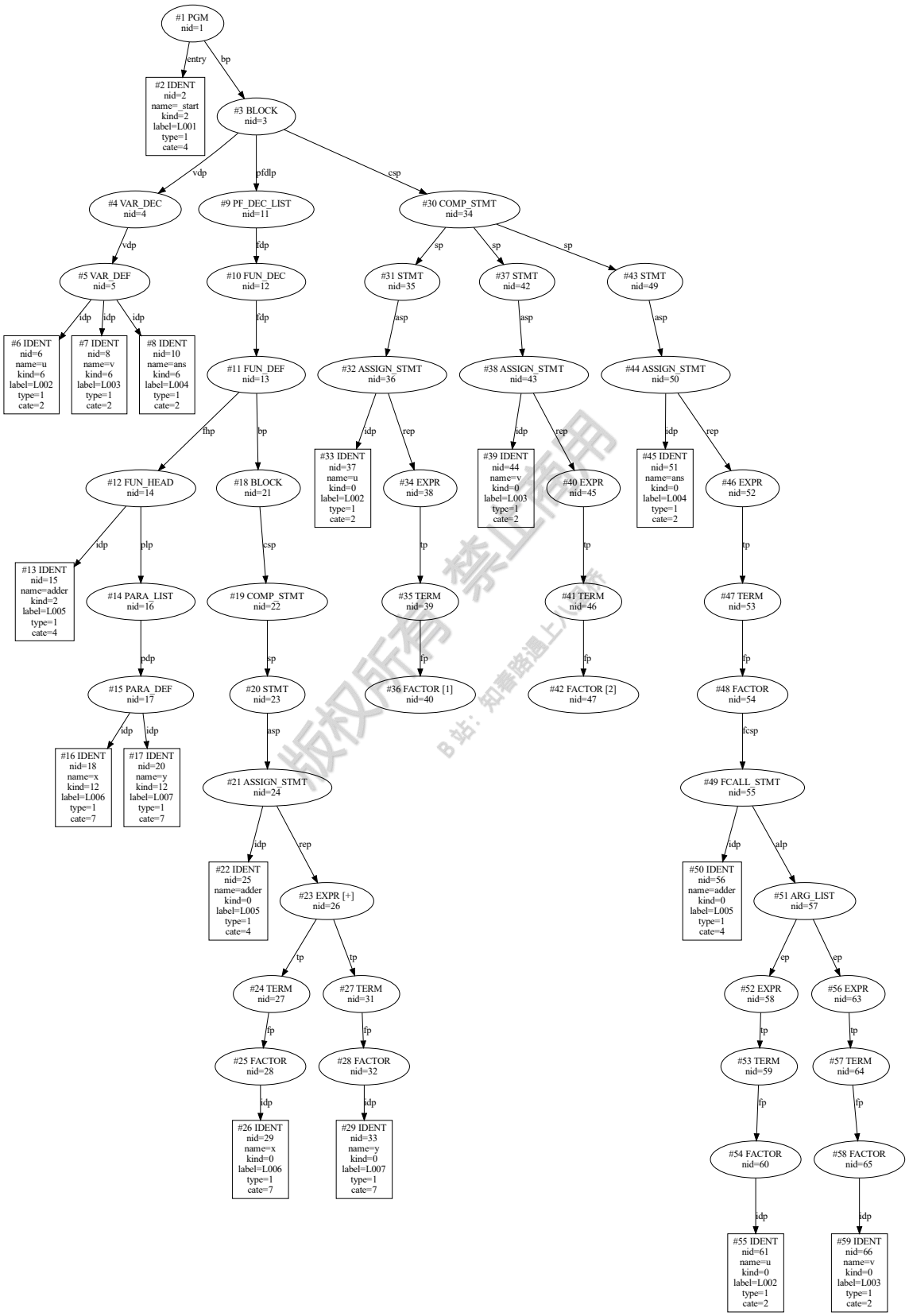


图 4.2: 累加器带符号表的语法树

对于当前这个场景，我们就可以知道 u 是一个 **已经定义的整型变量**。

写到这里的话，可能大家就会对编译原理中复杂的概念产生混淆。同样的一个 u 在不同的地方会被叫做标记（token），词位（lexeme），标识符（identifier）以及符号（symbol）。我们在这里把这些概念辨析一下：

- token 和 lexeme 是词法分析阶段中对 u 的称呼，词法分析中 gettok() 函数的输出是一个 token，在本书中的定义下，u 其实一个枚举类 token_t 的值，具体就是数字 23 ，即 MC_ID。而 lexeme 是这个 token 对应的原始字符串，即 "u" 这样的字符串，它存储在 tokbuf[] 数组中。
- identifier 是语法分析阶段的产物，它表示语法树的一个特殊节点，即 IDENT 节点，确认 identifier 的最好依据就是它会有一个 nid ，并且在语法树上作为一个叶子节点存在。
- symbol 是符号表的附属概念，当 identifier 通过语义分析插入符号表后我们就可以得到一个具体的符号项 [3]，这个符号项一定有一个 sid ，此时我们就可以把这个符号项叫做 symbol 。

另外，u 还有一些其他的称呼，比如：变量 u ，常量 u ，名字（name）u ，参数 u 。这些是编程语言中的概念，相信大家在 c 语言课中已经理解透彻了，这里不花费篇幅进行介绍。

4.2.3 符号表的数据结构

符号表的数据结构定义在文件 include/symtab.h 中，这部分代码中定义了以下结构体类型：

1. 第 39 至 60 行定义了 syment_t 符号项。
   - 第 8 至 23 行定义了 cate_t 符号分类。
   - 第 26 至 31 行定义了 type_t 符号类型。
2. 第 62 至 83 行定义了 symtab_t 符号表。
3. 第 34 至 37 行定义了 param_t 参数项。

其中符号项和符号表的大部分属性含义在之前的章节中已经明确了。参数项是记录过程或函数定义的参数的，它里面包含一个 symbol 指针指向参数所引用的符号项，和一个 next 指向下一个参数。具体代码如下：

```
1  #define MAXBUCKETS 16
2
3  typedef struct _sym_param_struct param_t;
4  typedef struct _sym_entry_struct syment_t;
5  typedef struct _sym_table_struct symtab_t;
6
7  // symbol category
8  typedef enum _sym_cate_enum {
9          // Primary Object
10         /*  0 */ NOP_OBJ,
11         /*  1 */ CONST_OBJ,
```

[3]言下之意是有些 identifier 不能插入符号表，这个我们在后续章节中就可以看到。

```
        /*  2 */ VAR_OBJ,
        /*  3 */ PROC_OBJ,
        /*  4 */ FUN_OBJ,
        /*  5 */ ARRAY_OBJ,
        /*  6 */ BYVAL_OBJ,
        /*  7 */ BYREF_OBJ,
        // Additional
        /*  8 */ TMP_OBJ,
        /*  9 */ LABEL_OBJ,
        /* 10 */ NUM_OBJ,
        /* 11 */ STR_OBJ
} cate_t;

// symbol type
typedef enum _sym_type_enum {
        /* 0 */ VOID_TYPE,
        /* 1 */ INT_TYPE,
        /* 2 */ CHAR_TYPE,
        /* 3 */ STR_TYPE
} type_t;

// signature for procedure and function
typedef struct _sym_param_struct {
        syment_t *symbol;
        param_t *next;
} param_t;

typedef struct _sym_entry_struct {
        // identifier name
        int sid;
        char name[MAXSTRLEN];
        cate_t cate;
        type_t type;
        // const value, initval value
        int initval;
        int arrlen;
        char str[MAXSTRLEN];
        param_t *phead;
        param_t *ptail;
        symtab_t *scope;
        // label for assemble codes
        char label[MAXSTRLEN];
```

```
54          int off; // offset, for local variable stack mapping
55          // referred line number
56          int lineno;
57          // which symbol table
58          symtab_t *stab;
59          syment_t *next;
60  } syment_t;
61
62  typedef struct _sym_table_struct {
63          int tid; // symbol table ID
64
65          // for function scope management
66          int depth; // symbol table nested depth
67          char nspace[MAXSTRLEN]; // namespace
68          symtab_t *inner; // inner scope
69          symtab_t *outer; // outer scope
70
71          // for assembly stack mapping
72          //    1. arguments values
73          //    2. saved ebps
74          //    3. return value
75          //    4. local variables    (varoff)
76          //    5. temporary variables (tmpoff)
77          int argoff; // argument variable offset in total
78          int varoff; // variable offset in total
79          int tmpoff; // temporary variable offset in total
80
81          // entries buckets
82          syment_t buckets[MAXBUCKETS];
83  } symtab_t;
```


由于哈希表不是 c 语言的内置数据结构，所有我参考一般哈希表的实现设计了一个符号表。如图 4.3 所示，符号表的主要设计思路如下：

1. 选择哈希函数：哈希函数用于将键映射到哈希表的索引。理想情况下，哈希函数应尽可能均匀地将键分布到表的大小，以减少冲突的可能性。我们实现了 `hash()` 函数来将符号名字（name 中在字符数组）哈希成一个整数。
2. 确定哈希表大小：哈希表的大小应足以容纳所有可能的键。如果预先知道键的数量，可以设置一个固定大小的哈希表。否则，可以使用动态扩展策略，根据需要增加哈希表的大小。我们这里将标识符通过哈希函数均匀地分布到散列表的各个桶中，即 `buckets[]` 数组。
3. 处理冲突：当两个或更多的键哈希到同一索引时，会发生冲突。常见的处理冲突的方法有链地址法、开放地址法等。我们采取链地址法将所有哈希到同一索引的键存储在该索引的链表中。在我们的设计中，符号项中通过 `next` 指针将一个桶中的所有符号项串联成一个单项链表，这

就是链地址法解决哈希冲突。

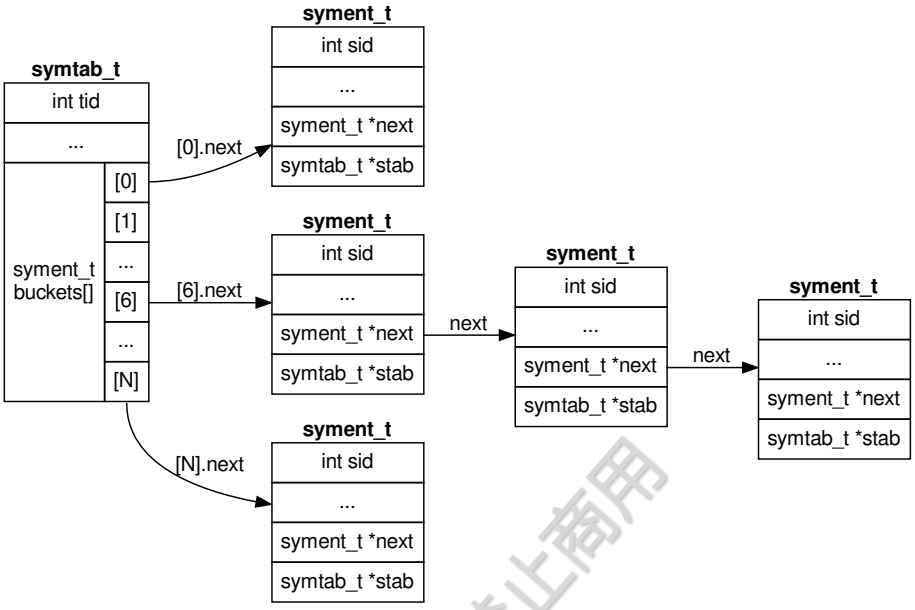


图 4.3: 单个符号表数据结构示意图

4.3 作用域

4.3.1 函数作用域

作用域（scope）是指程序中变量的可见性和生命周期。变量的作用域决定了它可以在哪些范围内被访问和使用。函数作用域是指函数内被定义的变量的可访问性。换句话说，变量的作用域决定了如何使用它。只有变量所在的函数内才可以访问它。函数的作用域也称为词法作用域，因为它是根据代码的书写来确定的，而不是在运行时确定的。

```
1  var u, v, ans: integer;
2  function adder(var x, y : integer):integer;
3  begin
4     adder := x + y
5  end;
6  begin
7     u := 1;
8     v := 2;
9     ans := adder(u, v)
10 end.
```

为了说明函数作用域的细节，这里继续使用累计器作为例子进行说明。语义分析器初始启动时

会新建一个 tid=1 的全局符号表，在分析完第 1 行代码后符号表细节见表 4.4 ，这里我们把 tid 也放入表格中，可以看出表格中只有 u 、v 和 ans 这 3 个有效符号。

表 4.4: 函数作用域：累加器分析完第 1 行代码时的符号表

	sid	name	cate	type	value	label
tid=1	1	_start	4	1	0	L001
	2	u	2	1	0	L002
	3	v	2	1	0	L003
	4	ans	2	1	0	L004

当语义分析器进入 adder() 函数时，会将 adder 作为符号添加入符号表，并且还会新建一个新的符号表，就是 tid=2 的符号表，在进入 adder() 函数后，也就是代码的第 3 行，参数 x 和 y 也会被放入符号表，此时的符号表就变成了表 4.5 。在 adder() 函数里面，也就是代码的第 4 行，程序访问了变量 x 和 y ，这里 x 和 y 已经在 tid=2 的符号表中，所以这里的访问是语义正确的。

表 4.5: 函数作用域：累加器进入 adder() 函数时的符号表

	sid	name	cate	type	value	label
tid=1	1	_start	4	1	0	L001
	2	u	2	1	0	L002
	3	v	2	1	0	L003
	4	ans	2	1	0	L004
	5	adder	4	1	0	L005
tid=2	6	x	7	1	0	L006
	7	y	7	1	0	L007

语义分析器会接着分析，当累加器进入主程序时，也就是结束分析第 5 行，开始分析第 6 行。这时语义分析器会做这样的操作，将 tid=2 的符号表销毁，此时的符号表就变成了表 4.6 。销毁的原因是：x 和 y 作为 adder() 函数的参数，在主程序中是不能被访问到的。

表 4.6: 函数作用域：累加器进入主程序时的符号表

	sid	name	cate	type	value	label
tid=1	1	_start	4	1	0	L001
	2	u	2	1	0	L002
	3	v	2	1	0	L003
	4	ans	2	1	0	L004
	5	adder	4	1	0	L005

下面举一个 c 语言的例子来说明这个问题。我们首先编写一个 adder.c 文件，里面代码如下：

```
1 int u, v, ans;
2
3 int adder(int x, int y)
```

```
4  {
5          return x + y;
6  }
7
8  int main (int argc, char *argv[])
9  {
10         u = 1; v = 2;
11         ans = adder(u, v);
12         ans = x + y;
13         return 0;
14 }
```

在完成上述程序的编码过后，使用编译器编译 adder.c 代码会报错。错误提示 x 和 y 标识符没有声明，结果如下：

```
$ cc adder.c
adder.c:12:8: error: use of undeclared identifier 'x'
        ans = x + y;
              ^
adder.c:12:12: error: use of undeclared identifier 'y'
        ans = x + y;
                  ^
2 errors generated.
$
```

我们这里对累加器的例子进行扩展讨论，将 adder() 函数的传入参数进行修改，修改后的累加器代码称作 adder2() ，具体实现如下：

```
1  var u, v, ans: integer;
2  function adder2(var u, v : integer):integer;
3  begin
4     adder2 := u + v
5  end;
6  begin
7     u := 1;
8     v := 2;
9     ans := adder2(u, v)
10 end.
```

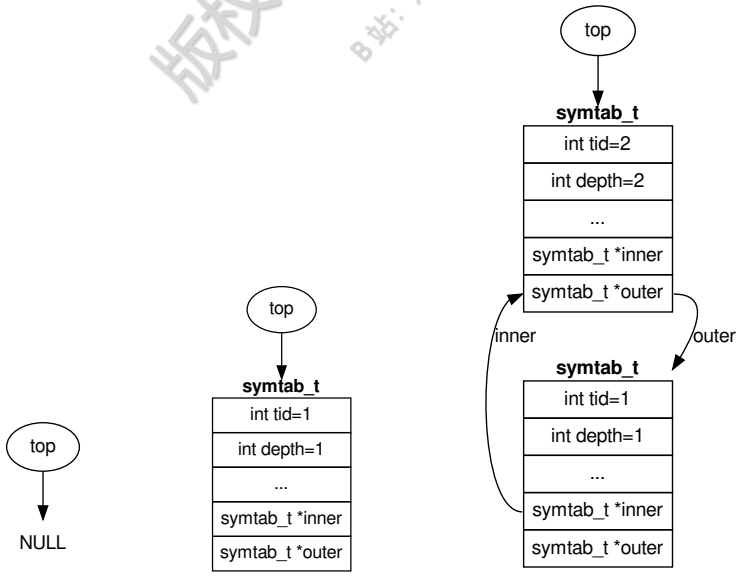
函数 adder2() 和 adder() 的实现是完全一样的，只是入参的名字修改成了 u 和 v 。通过我们的编程知识可以知道，参数的名称实际上不会对函数的逻辑产生影响。但是差之毫厘谬以千里，当语义分析器进入 adder2() 函数后，也就是代码的第 3 行，此时的符号表却不是像表 4.5 中描述的一样，取而代之的是表 4.7 这样的形式。仔细看一下表 4.7 中的数据，你会发现里面竟然有 2 个名称为 u 的符号，幸运的时，这 2 个 u 是出于不同符号表中，分别为 tid=1 和 tid=2 的两个符号表，并且他的 sid 也是不同的，分别为 sid=2 和 sid=6 。当语义分析器分析代码第 4 行

引用到的 u 时，通过符号表查找规则会先就近查找 tid=2 的符号表，然后会首先查找到标准包含的 sid=6 的符号项 u ，所以最终不会引用 sid=2 的符号项 u 。这其实就是作用域的基本概念。

表 4.7: 函数作用域: 累加器进入 adder2() 函数时的符号表

	sid	name	cate	type	value	label
tid=1	1	_start	4	1	0	L001
	2	u	2	1	0	L002
	3	v	2	1	0	L003
	4	ans	2	1	0	L004
	5	adder2	4	1	0	L005
tid=2	6	u	7	1	0	L006
	7	v	7	1	0	L007

当语义分析器进入到一个函数时，相当于进入了一个新的作用域，此时会新建一个符号表，这也就是函数作用域的由来。符号表和作用域操作关系构成了一个符号表栈的数据结构，图 4.4 是符号表入栈场景的全过程。首先，有一个 top 指针记录符号表栈顶，初始时如图 4.4(a) 所示，此时符号表栈没有任何符号表，top=NULL 。当语义分析器开始初始化全局符号表时，tid=1 的符号表入栈，这时变成图 4.4(b) 描述的情景。当进入 adder() 函数时，tid=2 的符号表入栈，这时变成图 4.4(c) 描述的情景，符号表之间通过 outer 和 inner 指针关联。



(a) 符号表栈: 空 (b) 符号表栈: tid=1 压入 (c) 符号表栈: tid=2 压入

图 4.4: 符号表入栈场景

在图 4.4(c) 场景中，语义分析器查找符号 u 的逻辑是这样的:

1. 通过 top 指针获取最近的符号表，即 tid=2 的符号表
2. 在 tid=2 的符号表中查询符号 u

- 如果查询到 u 则返回结果
- 否则通过 tid=2 的 outer 指针获取外层符号表，即 tid=1 的符号表，重复步骤 2 。

上面的逻辑其实就是函数里面对符号的引用会优先查询当前作用域的符号表，如果查询不到再从外部的符号表中查询。这里我们还对符号表引入一个概念，即 depth 深度，depth 记录当前符号表嵌套的层级，例如：tid=2 的符号表的 depth=2 表示它上面还有一级符号表。符号表出栈的情形和入栈正好是互逆的操作，这里就不详细说明了。

4.3.2 嵌套函数作用域

嵌套函数定义是指在某个函数内部定义另一个函数。这种定义方式使得内部函数可以访问外部函数的变量，并且还可以在外部函数之外被调用。c 语言不支持嵌套函数，$PL/0\epsilon$ 的文法定义是支持嵌套函数，这里给出一个嵌套函数定义的代码 nested.pas 作为样例，其代码如下：

```
1  var x: integer;
2  procedure p1();
3    var x: integer;
4    procedure p2();
5      var x: integer;
6      begin x := 2 end;
7    begin
8       x := 1;
9       p2()
10   end;
11 begin
12    x := 0;
13    p1()
14 end.
```


上述代码的第 2 行定义了第一个过程 p1() ，根据文法 (1.7) 的描述，第 3 至 10 行是 p1() 的分程序。需要注意的是，分程序中又定义了一个 p2() 过程，对应于代码的第 4 至 6 行。这时候有意思的事情就发生了，过程 p2() 其实属于 p1() 过程的作用域下。然后第 7 至 10 行是 p1() 的执行体，最后第 11 至 14 行是主函数的执行体，为确保上述程序的语法正确性，如图 4.5 所示，我们可以将程序对应的语法树绘制出来，为了方便查找，图中把 p1() 和 p2() 的 IDENT 节点添加灰底高亮显示出来了，并将它们对应的关系的子树通过虚线圈在一起，在图中可以清晰地看到它们之间的继承关系。

可能读者已经注意到了，我在 nested.pas 中定义的变量名字都叫 x ，但是在 p1() 、p2() 以及主函数中使用的 x 却不尽相同。在这里，我们比较关心程序进入第 6 行，即过程 p2() 执行体时，语义分析器看到的符号表是什么情况？通过推演，我们知道表 4.8 就是这时候的符号表，表中包含三个同名的 x 变量，但是却是 3 个不同的符号（因为 sid 不同）。

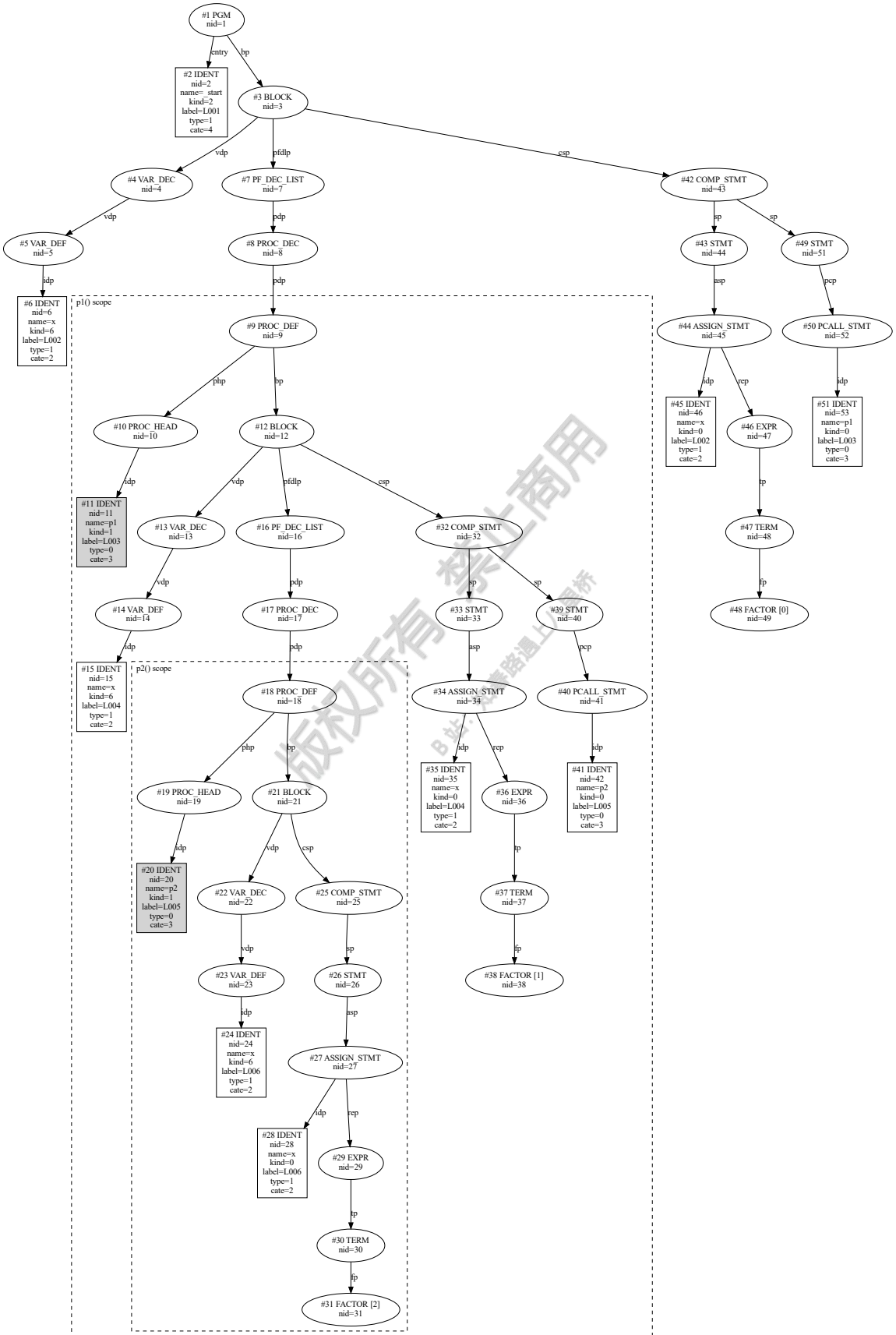


图 4.5: 嵌套函数 nested.pas 的语法树

表 4.8: 嵌套函数作用域: 进入 p2() 过程时的符号表

	sid	name	cate	type	value	label
tid=1	1	_start	4	1	0	L001
	2	x	2	1	0	L002
	3	p1	3	0	0	L003
tid=2	4	x	2	1	0	L004
	5	p2	3	0	0	L005
tid=3	6	x	7	1	0	L006

4.4 符号表的操作函数

通过前面章节的铺垫，大家应该可以建立符号表的基本认知，这一小节我们来介绍符号表的操作函数。符号表的操作函数在文件 source/symtab.c 中实现，这部分代码包含两个部分：符号表管理和符号项管理。

- 符号表管理主要是处理作用域进入和退出时符号表栈的压入和弹出，
- 符号项管理主要是处理符号初始化、插入和查询等操作。

其具体实现代码如下:

```
1  // symbol table management
2  symtab_t *top = NULL;
3  int depth = 0;
4  int tidcnt = 0;
5
6  syment_t *syments[MAXSYMENT];
7  int sidcnt = 0;
8
9  symtab_t *scope_entry(char *nspace)
10 {
11         symtab_t *t;
12         NEWSTAB(t);
13         t->tid = ++tidcnt;
14         t->depth = ++depth;
15         strcopy(t->nspace, nspace);
16         t->varoff = 1; // reserve function return value
17
18         // Push
19         t->outer = top;
20         if (top) {
21                 top->inner = t;
22         }
23         top = t;
```

```
24
25          // trace log
26          dbg("push depth=%d tid=%d nspace=%s\n", t->depth, t->tid, t->nspace);
27          return t;
28  }
29
30  symtab_t *scope_exit(void)
31  {
32          nevernil(top);
33
34          // Pop
35          symtab_t *t = top;
36          top = t->outer;
37          if (top) {
38                  top->inner = NULL;
39          }
40          depth--;
41
42          // trace log
43          //   1. dump table info
44          //   2. dump all table entry
45          dbg("pop depth=%d tid=%d nspace=%s\n", t->depth, t->tid, t->nspace);
46          int i;
47          for (i = 0; i < MAXBUCKETS; ++i) {
48                  syment_t *hair, *e;
49                  hair = &t->buckets[i];
50                  for (e = hair->next; e; e = e->next) {
51                          dbg("sid=%d, name=%s\n", e->sid, e->name);
52                  }
53          }
54          return t;
55  }
56
57  symtab_t *scope_top(void)
58  {
59          nevernil(top);
60          return top;
61  }
62
63  // entry management
64  const int HASHSIZE = 211;
65  const int HASHSHIFT = 4;
```

```
66
67  static inline int hash(char *key)
68  {
69          if (!key) {
70                  panic("BAD_HASH_KEY");
71          }
72
73          int h, i;
74          for (i = h = 0; key[i] != '\0'; i++) {
75                  h = ((h << HASHSHIFT) + key[i]) % HASHSIZE;
76          }
77
78          return h;
79  }
80
81  static syment_t *getsym(symtab_t *stab, char *name)
82  {
83          syment_t *hair, *e;
84          hair = &stab->buckets[hash(name) % MAXBUCKETS];
85          for (e = hair->next; e; e = e->next) {
86                  if (!strcmp(e->name, name)) {
87                          return e;
88                  }
89          }
90          return NULL;
91  }
92
93  static void putsym(symtab_t *stab, syment_t *e)
94  {
95          syment_t *hair = &stab->buckets[hash(e->name) % MAXBUCKETS];
96          e->next = hair->next;
97          hair->next = e;
98
99          // for debugging
100         if (e->sid + 1 >= MAXSYMENT) {
101                 panic("TOO_MANY_SYMBOL_ENTRY");
102         }
103         syments[e->sid] = e;
104
105         dbg("tid=%d nspace=%s sym=%s\n", stab->tid, stab->nspace, e->name);
106 }
107
```

```
108 static void dumptab(symtab_t *stab)
109 {
110         char indent[MAXSTRBUF] = "\0";
111         int i;
112         for (i = 0; i < stab->depth; ++i) {
113                 strcat(indent, "  ");
114         }
115
116         symtab_t *t = stab;
117         msg("%sstab(tid=%d): depth=%d, nspace=%s\n", indent, t->tid, t->depth,
118             t->nspace);
119
120         strcat(indent, "  ");
121         for (i = 0; i < MAXBUCKETS; ++i) {
122                 syment_t *hair, *e;
123                 hair = &t->buckets[i];
124                 for (e = hair->next; e; e = e->next) {
125                         msg("%ssid=%d, name=%s, cate=%d, type=%d, value=%d, label=%s\n",
126                             indent, e->sid, e->name, e->cate, e->type,
127                             e->initval, e->label);
128                 }
129         }
130 }
131
132 syment_t *symget(char *name)
133 {
134         nevernil(top);
135         return getsym(top, name);
136 }
137
138 syment_t *symfind(char *name)
139 {
140         nevernil(top);
141         syment_t *e;
142         symtab_t *t;
143         e = NULL;
144         for (t = top; t; t = t->outer) {
145                 if ((e = getsym(t, name)) != NULL) {
146                         return e;
147                 }
148         }
149         return e;
```

```
150 }
151
152 void symadd(syment_t *entry)
153 {
154         nevernil(top);
155         putsym(top, entry);
156         entry->stab = top;
157 }
158
159 void stabdump()
160 {
161         msg("DUMP SYMBOL TABLE:\n");
162         symtab_t *t;
163         for (t = top; t; t = t->outer) {
164                 dumptab(t);
165         }
166         msg("\n");
167 }
168
169 syment_t *syminit(ident_node_t *idp)
170 {
171         syment_t *e;
172         NEWENTRY(e);
173         e->sid = ++sidcnt;
174
175         strcopy(e->name, idp->name);
176         e->initval = idp->value;
177         e->arrlen = idp->length;
178         e->lineno = idp->line;
179
180         switch (idp->kind) {
181         case PROC_IDENT:
182                 e->cate = PROC_OBJ;
183                 break;
184         case INT_FUN_IDENT:
185         case CHAR_FUN_IDENT:
186                 e->cate = FUN_OBJ;
187                 break;
188         case INT_CONST_IDENT:
189         case CHAR_CONST_IDENT:
190                 e->cate = CONST_OBJ;
191                 break;
```

```
192         case INT_VAR_IDENT:
193         case CHAR_VAR_IDENT:
194                 e->cate = VAR_OBJ;
195                 break;
196         case INT_ARRVAR_IDENT:
197         case CHAR_ARRVAR_IDENT:
198                 e->cate = ARRAY_OBJ;
199                 break;
200         case INT_BYVAL_IDENT:
201         case CHAR_BYVAL_IDENT:
202                 e->cate = BYVAL_OBJ;
203                 break;
204         case INT_BYADR_IDENT:
205         case CHAR_BYADR_IDENT:
206                 e->cate = BYREF_OBJ;
207                 break;
208         default:
209                 e->cate = NOP_OBJ;
210         }
211
212         switch (idp->kind) {
213         case INT_FUN_IDENT:
214         case INT_CONST_IDENT:
215         case INT_VAR_IDENT:
216         case INT_ARRVAR_IDENT:
217         case INT_BYVAL_IDENT:
218         case INT_BYADR_IDENT:
219                 e->type = INT_TYPE;
220                 break;
221         case CHAR_FUN_IDENT:
222         case CHAR_CONST_IDENT:
223         case CHAR_VAR_IDENT:
224         case CHAR_ARRVAR_IDENT:
225         case CHAR_BYVAL_IDENT:
226         case CHAR_BYADR_IDENT:
227                 e->type = CHAR_TYPE;
228                 break;
229         default:
230                 e->type = VOID_TYPE;
231         }
232
233         sprintf(e->label, "L%03d", e->sid);
```

```
234         switch (e->cate) {
235         case NOP_OBJ:
236         case CONST_OBJ:
237                 // no need allocation
238                 break;
239         case VAR_OBJ:
240         case PROC_OBJ:
241         case FUN_OBJ:
242                 e->off = top->varoff;
243                 top->varoff++;
244                 break;
245         case BYVAL_OBJ:
246         case BYREF_OBJ:
247                 e->off = top->argoff;
248                 top->argoff++;
249                 break;
250         case ARRAY_OBJ:
251                 e->off = top->varoff;
252                 top->varoff += e->arrlen;
253                 break;
254         default:
255                 unlikely();
256         }
257
258         symadd(e);
259         return e;
260 }
261
262 syment_t *symalloc(symtab_t *stab, char *name, cate_t cate, type_t type)
263 {
264         syment_t *e;
265         NEWENTRY(e);
266         strcopy(e->name, name);
267         e->sid = ++sidcnt;
268
269         e->cate = cate;
270         e->type = type;
271
272         sprintf(e->label, "T%03d", e->sid);
273         switch (e->cate) {
274         case TMP_OBJ:
275                 // from now on, we will NEVER alloc local variables so just
```

```
276                 // alloc temporary variables
277                 e->off = stab->varoff + stab->tmpoff;
278                 stab->tmpoff++;
279                 break;
280         case LABEL_OBJ:
281         case NUM_OBJ:
282         case STR_OBJ:
283                 // label/number/string never use bytes
284                 break;
285         default:
286                 unlikely();
287         }
288
289         e->stab = stab;
290         putsym(stab, e);
291         return e;
292 }
```

图 4.6 是带作用域的符号表栈数据结构的一个示意图，读者可以结合图来理解符号表操作的代码实现。

上述代码第 9 到 61 行实现了符号表管理功能，该部分主要功能通过下面几个函数实现：

1. `scope_entry(char *nspace)` 函数将通过 `nspace` 创建创建一个符号表 `symtab_t` ，并将其压入符号表栈顶。通常在一个作用域进入时调用。
2. `scope_exit()` 函数将符号表栈顶的一个符号表弹出。通常在一个作用域退出时调用。
3. `scope_top()` 函数获取符号表栈顶的符号表指针。

符号项的实现函数分成两个部分：底层操作函数和上层操作函数。底层操作实现真实操作功能，上层操作对底层操作进行封装，为用户提供调用接口。上述代码第 63 到 106 行实现例符号项的底层操作，这部分是 `static` 修饰的静态函数。这些函数的介绍如下：

1. `int hash(char *key)` 是将符号名字转化成数字的哈希函数。
2. `syment_t *getsym(symtab_t *stab, char *name)` 通过符号 `name` 查询 `stab` 符号表，如果存在返回符号项。
3. `void putsym(symtab_t *stab, syment_t *e)` 将符号项 `e` 通过头插法放入符号表 `stab` 对应的桶中。
4. `void dumptab(symtab_t *stab)` 打印所有符号表中的所有符号项，主要是用于调试。

上述代码第 132 到 292 行实现了上层操作接口，这些函数功能介绍如下：

1. 符号项的构造函数
   - `syment_t *syminit(ident_node_t *idp)` 通过 `idp` 节点创建符号项
   - `syment_t *symalloc(symtab_t *stab, char *name, cate_t cate, type_t type)` 通过一些入参创建符号项，主要用于临时变量或标号的创建
2. `void symadd(syment_t *entry)` 添加符号项

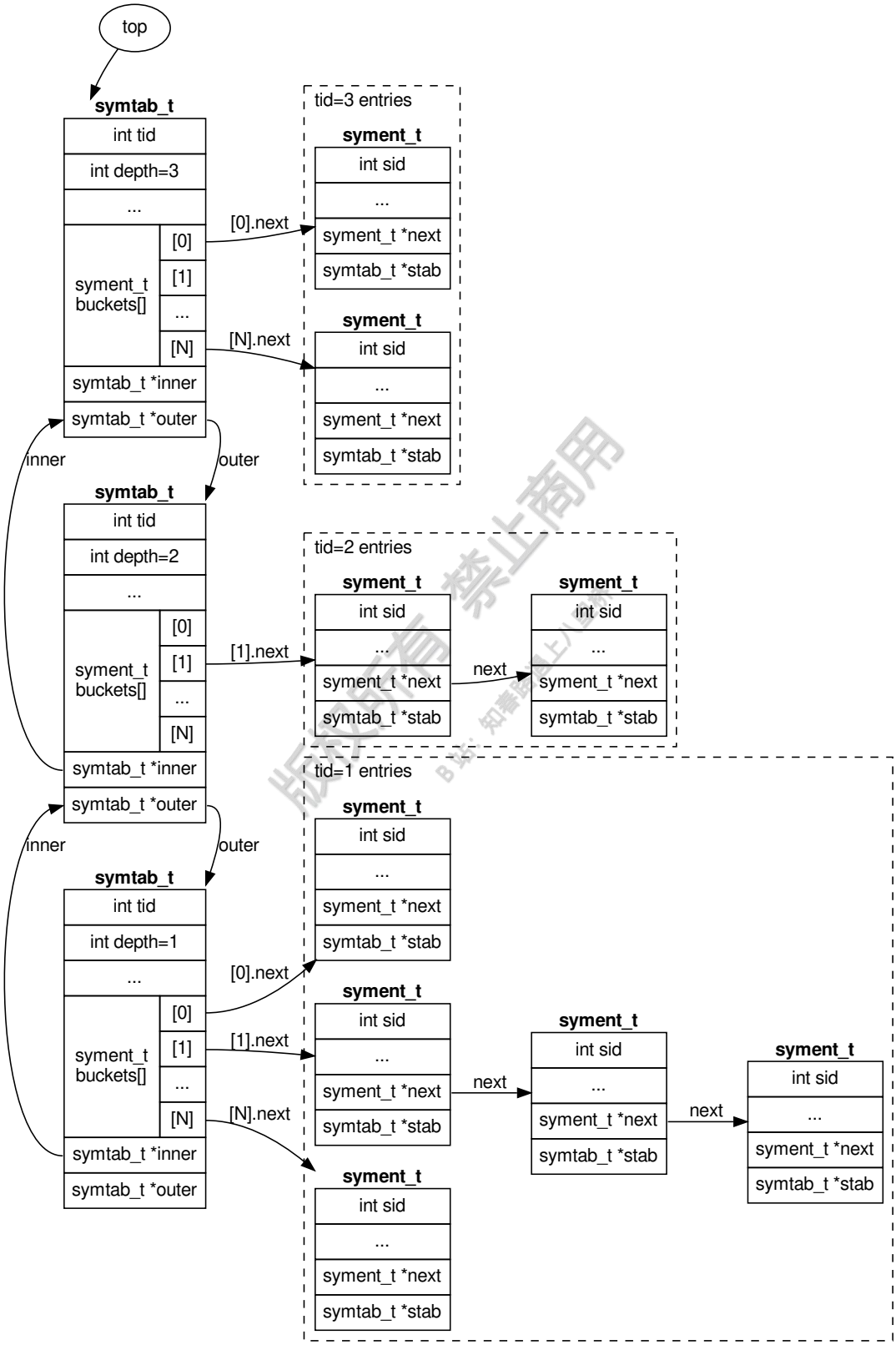


图 4.6: 带作用域的符号表栈数据结构示意图

3. 查找符号项
   - `syment_t *symget(char *name)` 只搜索符号表栈顶的符号
   - `syment_t *symfind(char *name)` 搜索符号表栈顶及外部的符号表
4. `stabdump()` 打印出当前所有的符号表栈中的数据

至此，我就将符号表的全部功能及实现介绍完毕了。

4.5 形参和实参

形参（也称为形式参数）和实参（也称为实际参数）都是与函数和方法的参数相关的概念。在 $PL/0\epsilon$ 语言中，它们的具体说明如下：

1. 形参是形式参数（parameter）的简称，它具有以下特性：
   - 形参是在定义函数或方法时声明的参数。
   - 它用于接收传递给函数或方法的值或数据。
   - 在函数或方法被调用之前，形参会被分配内存。
   - 形参的作用域仅限于函数或方法内部。
   - 它主要用于初始化函数的局部变量。
2. 实参是实际参数（argument）的简称，它具有以下特性：
   - 实参是在调用函数或方法时传递给它的参数。
   - 实参可以是常量、变量、表达式或其他数据类型。
   - 实参是实际传递给函数或方法的值或数据。

我们来举例说明一下会更加直观，在下面代码的第 3 至 6 行定义了 `adder()` 函数，它的 `x` 和 `y` 是形参，这里的 `x` 和 `y` 仅仅在 `adder()` 中可以使用。第 11 行调用 `adder()` 函数，传入参数 `u` 和 `v` 是实参。

```
1  var u, v, ans: integer;
2
3  function adder(var x, y : integer):integer;
4  begin
5     adder := x + y
6  end;
7
8  begin
9     u := 1;
10    v := 2;
11    ans := adder(u, v)
12 end.
```

4.6 函数传值和传引用

参数列表有一个比较重要的概念就是传值 (Pass by Value) 和传引用 (Pass by Reference)。以下是它们的具体说明:

1. 传值 (Pass by Value):
   - 在传值中，函数接收的是参数值的副本，而不是参数本身。
   - 修改了副本的值不会影响到原始变量。
   - 传值是一种相对安全的方式，因为原始数据不会被外部函数修改。
   - $PL/0\epsilon$ 语言默认使用传值方式。
2. 传引用 (Pass by Reference):
   - 在传引用中，函数接收的是参数的引用或指针，而不是参数的值。
   - 函数可以直接修改传递给它的参数，因为它是参数的直接引用。
   - 传引用可以提高性能，因为不需要创建参数的副本。
   - $PL/0\epsilon$ 语言默认使用 `var` 关键字修饰达到传引用。

传值传递的是参数的副本，修改副本不会影响原始变量；而传引用传递的是参数的引用或指针，允许函数直接修改原始变量。选择使用哪种方式取决于具体的需求和编程语言的特性。下面代码可以直观说明传值和传引用的区别，代码定义了两个自增函数 `inc1()` 和 `inc2()` 。其中: `inc1()` 的形参 `x` 是传值方式，`inc2()` 的形参 `x` 是传引用方式。在第 8 行调用 `inc1(counter)` 后，第 9 行输出 `counter` 的值为 0 (不变)。在第 10 行调用 `inc2(counter)` 后，第 11 行输出 `counter` 的值为 1 (改变)。

```
1  var counter: integer;
2
3  procedure inc1(x: integer); begin x := x + 1 end;
4  procedure inc2(var x: integer); begin x := x + 1 end;
5
6  begin
7     counter := 0;
8     inc1(counter);
9     write(counter); { 输出 0}
10    inc2(counter);
11    write(counter); { 输出 1}
12 end.
```


4.7 语义分析过程

4.7.1 anlys_xxx 分析函数家族

语义分析的函数都在文件 `include/anlys.h` 中声明，这些分析函数的命名格式为 `anlys_xxx()` 方式。最终语义分析的入口函数是 `analysis()` 函数，相关的分析函数声明具体代码如下:

```
static void anlys_pgm(pgm_node_t *node);
static void anlys_const_decf(const_dec_node_t *node);
static void anlys_var_decf(var_dec_node_t *node);
static void anlys_pf_dec_list(pf_dec_list_node_t *node);
static void anlys_proc_decf(proc_dec_node_t *node);
static void anlys_proc_head(proc_head_node_t *node);
static void anlys_fun_decf(fun_dec_node_t *node);
static void anlys_fun_head(fun_head_node_t *node);
static void anlys_para_list(syment_t *sign, para_list_node_t *node);
static void anlys_comp_stmt(comp_stmt_node_t *node);
static void anlys_stmt(stmt_node_t *node);
static void anlys_assign_stmt(assign_stmt_node_t *node);
static void anlys_if_stmt(if_stmt_node_t *node);
static void anlys_repe_stmt(repe_stmt_node_t *node);
static void anlys_for_stmt(for_stmt_node_t *node);
static void anlys_pcall_stmt(pcall_stmt_node_t *node);
static void anlys_read_stmt(read_stmt_node_t *node);
static void anlys_write_stmt(write_stmt_node_t *node);
static void anlys_expr(expr_node_t *node);
static void anlys_term(term_node_t *node);
static void anlys_factor(factor_node_t *node);
static void anlys_fcall_stmt(fcall_stmt_node_t *node);
static void anlys_cond(cond_node_t *node);
static void anlys_arg_list(syment_t *sign, arg_list_node_t *node);

void analysis();
```

虽然 anlys_xxx() 分析函数和之前的 parse_xxx() 解析函数家族存在类似的命名方式，但是 anlys_xxx() 分析函数家族的功能却完全不同。这些分析函数工作在语法树已经解析完成后，对语法树遍历同时进行语义分析，例如：检查符号是否定义，函数形参和实参是否正确，处理函数作用域等语义相关的工作。这些分析函数的实现在文件 source/anlys.c 中，具体代码如下：

```
static void anlys_pgm(pgm_node_t *node)
{
        scope_entry(MAINFUNC);

        syment_t *e = syminit(node->entry);
        node->entry->symbol = e;
        node->entry->symbol->scope = scope_top();

        nevernil(node->bp);
        block_node_t *b = node->bp;
        anlys_const_decf(b->cdp);
```

```
12          anlys_var_decf(b->vdp);
13          anlys_pf_dec_list(b->pfdlp);
14          anlys_comp_stmt(b->csp);
15
16          scope_exit();
17  }
18
19  static void anlys_const_decf(const_dec_node_t *node)
20  {
21          const_dec_node_t *t;
22          for (t = node; t; t = t->next) {
23                  nevernil(t->cdp);
24                  nevernil(t->cdp->idp);
25                  ident_node_t *idp = t->cdp->idp;
26                  syment_t *e = symget(idp->name);
27                  if (e) {
28                          rescue(DUPSYM, "L%d: const %s already declared.",
29                                  idp->line, idp->name);
30                  } else {
31                          e = syminit(idp);
32                  }
33                  idp->symbol = e;
34          }
35  }
36
37  static void anlys_var_decf(var_dec_node_t *node)
38  {
39          var_dec_node_t *t;
40          var_def_node_t *p;
41          for (t = node; t; t = t->next) {
42                  for (p = t->vdp; p; p = p->next) {
43                          nevernil(p->idp);
44                          ident_node_t *idp = p->idp;
45                          syment_t *e = symget(idp->name);
46                          if (e) {
47                                  rescue(DUPSYM,
48                                          "L%d: variable %s already declared.",
49                                          idp->line, idp->name);
50                          } else {
51                                  e = syminit(idp);
52                          }
53                          idp->symbol = e;
```

```
54                  }
55          }
56  }
57
58  static void anlys_pf_dec_list(pf_dec_list_node_t *node)
59  {
60          pf_dec_list_node_t *t;
61          for (t = node; t; t = t->next) {
62                  switch (t->kind) {
63                  case PROC_PFDEC:
64                          anlys_proc_decf(t->pdp);
65                          break;
66                  case FUN_PFDEC:
67                          anlys_fun_decf(t->fdp);
68                          break;
69                  default:
70                          unlikely();
71                  }
72          }
73  }
74
75  static void anlys_proc_decf(proc_dec_node_t *node)
76  {
77          proc_dec_node_t *t;
78          for (t = node; t; t = t->next) {
79                  nevernil(t->pdp);
80                  nevernil(t->pdp->php);
81                  anlys_proc_head(t->pdp->php);
82
83                  nevernil(t->pdp->bp);
84                  block_node_t *b = t->pdp->bp;
85
86                  anlys_const_decf(b->cdp);
87                  anlys_var_decf(b->vdp);
88                  anlys_pf_dec_list(b->pfdlp);
89                  anlys_comp_stmt(b->csp);
90
91                  scope_exit();
92          }
93  }
94
95  static void anlys_proc_head(proc_head_node_t *node)
```

```
96  {
97          proc_head_node_t *t = node;
98
99          nevernil(t->idp);
100         ident_node_t *idp = t->idp;
101         syment_t *e = symget(idp->name);
102         if (e) {
103                 rescue(DUPSYM, "L%d: procedure %s already declared.", idp->line,
104                         idp->name);
105         } else {
106                 e = syminit(idp);
107         }
108         idp->symbol = e;
109
110         e->scope = scope_entry(idp->name);
111
112         if (t->plp) {
113                 anlys_para_list(idp->symbol, t->plp);
114         }
115 }
116
117 static void anlys_fun_decf(fun_dec_node_t *node)
118 {
119         fun_dec_node_t *t;
120         for (t = node; t; t = t->next) {
121                 nevernil(t->fdp);
122                 nevernil(t->fdp->fhp);
123                 anlys_fun_head(t->fdp->fhp);
124
125                 nevernil(t->fdp->bp);
126                 block_node_t *b = t->fdp->bp;
127
128                 anlys_const_decf(b->cdp);
129                 anlys_var_decf(b->vdp);
130                 anlys_pf_dec_list(b->pfdlp);
131                 anlys_comp_stmt(b->csp);
132
133                 scope_exit();
134         }
135 }
136
137 static void anlys_fun_head(fun_head_node_t *node)
```

```
{
        fun_head_node_t *t = node;

        nevernil(t->idp);
        ident_node_t *idp = t->idp;
        syment_t *e = symget(idp->name);
        if (e) {
                rescue(DUPSYM, "L%d: function %s already declared.", idp->line,
                        idp->name);
        } else {
                e = syminit(idp);
        }
        idp->symbol = e;

        e->scope = scope_entry(idp->name);

        if (t->plp) {
                anlys_para_list(idp->symbol, t->plp);
        }
}

static void anlys_para_list(syment_t *sign, para_list_node_t *node)
{
        para_list_node_t *t;
        para_def_node_t *p;
        for (t = node; t; t = t->next) {
                for (p = t->pdp; p; p = p->next) {
                        nevernil(p->idp);
                        ident_node_t *idp = p->idp;
                        syment_t *e = symget(idp->name);
                        if (e) {
                                rescue(DUPSYM,
                                        "L%d: parameter %s already declared.",
                                        idp->line, idp->name);
                        } else {
                                e = syminit(idp);
                        }
                        idp->symbol = e;

                        param_t *param;
                        NEWPARAM(param);
                        param->symbol = e;
```

```
180
181                         if (!sign->ptail) {
182                                 sign->ptail = param;
183                                 sign->phead = param;
184                         } else {
185                                 sign->ptail->next = param;
186                                 sign->ptail = param;
187                         }
188                 }
189         }
190 }
191
192 static void anlys_comp_stmt(comp_stmt_node_t *node)
193 {
194         comp_stmt_node_t *t;
195         for (t = node; t; t = t->next) {
196                 nevernil(t->sp);
197                 anlys_stmt(t->sp);
198         }
199 }
200
201 static void anlys_stmt(stmt_node_t *node)
202 {
203         switch (node->kind) {
204         case ASSGIN_STMT:
205                 anlys_assign_stmt(node->asp);
206                 break;
207         case IF_STMT:
208                 anlys_if_stmt(node->ifp);
209                 break;
210         case REPEAT_STMT:
211                 anlys_repe_stmt(node->rpp);
212                 break;
213         case FOR_STMT:
214                 anlys_for_stmt(node->frp);
215                 break;
216         case PCALL_STMT:
217                 anlys_pcall_stmt(node->pcp);
218                 break;
219         case COMP_STMT:
220                 anlys_comp_stmt(node->cpp);
221                 break;
```

```
222         case READ_STMT:
223                 anlys_read_stmt(node->rdp);
224                 break;
225         case WRITE_STMT:
226                 anlys_write_stmt(node->wtp);
227                 break;
228         case NULL_STMT:
229                 break;
230         default:
231                 unlikely();
232         }
233 }
234
235 static void anlys_assign_stmt(assign_stmt_node_t *node)
236 {
237         syment_t *e;
238         ident_node_t *idp = node->idp;
239         e = symfind(idp->name);
240         if (!e) {
241                 giveup(BADSYM, "L%d: symbol %s not found.", idp->line,
242                         idp->name);
243         }
244         idp->symbol = e;
245         switch (node->kind) {
246         case NORM_ASSGIN:
247         case FUN_ASSGIN:
248                 anlys_expr(node->rep);
249                 break;
250         case ARRAY_ASSGIN:
251                 anlys_expr(node->lep);
252                 anlys_expr(node->rep);
253                 break;
254         default:
255                 unlikely();
256         }
257 }
258
259 static void anlys_if_stmt(if_stmt_node_t *node)
260 {
261         node->stab = scope_top();
262         anlys_cond(node->cp);
263         if (node->ep) {
```

```
264                 anlys_stmt(node->ep);
265         }
266         anlys_stmt(node->tp);
267 }
268
269 static void anlys_repe_stmt(repe_stmt_node_t *node)
270 {
271         node->stab = scope_top();
272         anlys_stmt(node->sp);
273         anlys_cond(node->cp);
274 }
275
276 static void anlys_for_stmt(for_stmt_node_t *node)
277 {
278         node->stab = scope_top();
279         anlys_expr(node->lep);
280         anlys_expr(node->rep);
281
282         ident_node_t *idp = node->idp;
283         syment_t *e = symfind(idp->name);
284         if (!e) {
285                 giveup(BADSYM, "L%d: symbol %s not found.", idp->line,
286                         idp->name);
287         }
288         idp->symbol = e;
289
290         switch (node->kind) {
291         case TO_FOR:
292         case DOWNTO_FOR:
293                 anlys_stmt(node->sp);
294                 break;
295         default:
296                 unlikely();
297         }
298 }
299
300 static void anlys_pcall_stmt(pcall_stmt_node_t *node)
301 {
302         ident_node_t *idp = node->idp;
303         syment_t *e = symfind(idp->name);
304         if (!e) {
305                 giveup(BADSYM, "L%d: symbol %s not found.", idp->line,
```

```
306                         idp->name);
307         }
308         if (e->cate != PROC_OBJ) {
309                 giveup(BADSYM, "L%d: procedure %s not found.", idp->line,
310                         idp->name);
311         }
312         idp->symbol = e;
313
314         if (node->alp) {
315                 anlys_arg_list(e, node->alp);
316         }
317 }
318
319 static void anlys_read_stmt(read_stmt_node_t *node)
320 {
321         read_stmt_node_t *t;
322         for (t = node; t; t = t->next) {
323                 nevernil(t->idp);
324                 ident_node_t *idp = t->idp;
325                 syment_t *e = symfind(idp->name);
326                 if (!e) {
327                         giveup(BADSYM, "L%d: symbol %s not found.", idp->line,
328                                 idp->name);
329                 }
330                 idp->symbol = e;
331         }
332 }
333
334 static void anlys_write_stmt(write_stmt_node_t *node)
335 {
336         node->stab = scope_top();
337         switch (node->type) {
338         case STR_WRITE:
339                 break;
340         case ID_WRITE:
341         case STRID_WRITE:
342                 anlys_expr(node->ep);
343                 break;
344         default:
345                 unlikely();
346         }
347 }
```

```
348
349 static void anlys_expr(expr_node_t *node)
350 {
351         node->stab = scope_top();
352         expr_node_t *t;
353         for (t = node; t; t = t->next) {
354                 nevernil(t->tp);
355                 anlys_term(t->tp);
356         }
357 }
358
359 static void anlys_term(term_node_t *node)
360 {
361         node->stab = scope_top();
362         term_node_t *t;
363         for (t = node; t; t = t->next) {
364                 nevernil(t->fp);
365                 anlys_factor(t->fp);
366         }
367 }
368
369 static void anlys_factor(factor_node_t *node)
370 {
371         node->stab = scope_top();
372         ident_node_t *idp;
373         syment_t *e;
374         switch (node->kind) {
375         case ID_FACTOR:
376                 nevernil(node->idp);
377                 idp = node->idp;
378                 e = symfind(idp->name);
379                 if (!e) {
380                         giveup(BADSYM, "L%d: symbol %s not found.", idp->line,
381                                 idp->name);
382                 }
383                 switch (e->cate) {
384                 case CONST_OBJ:
385                 case VAR_OBJ:
386                 case TMP_OBJ:
387                 case BYVAL_OBJ:
388                 case BYREF_OBJ:
389                         break;
```

```
390                 default:
391                         giveup(BADCTG, "L%d: symbol %s category is bad.",
392                                 idp->line, idp->name);
393                 }
394                 idp->symbol = e;
395                 break;
396         case ARRAY_FACTOR:
397                 nevernil(node->idp);
398                 idp = node->idp;
399                 e = symfind(idp->name);
400                 if (!e) {
401                         giveup(BADSYM, "L%d: symbol %s not found.", idp->line,
402                                 idp->name);
403                 }
404                 if (e->cate != ARRAY_OBJ) {
405                         giveup(ERTYPE, "L%d: symbol %s type is not array.",
406                                 idp->line, idp->name);
407                 }
408                 idp->symbol = e;
409
410                 nevernil(node->ep);
411                 anlys_expr(node->ep);
412                 break;
413         case UNSIGN_FACTOR:
414                 break;
415         case CHAR_FACTOR:
416                 break;
417         case EXPR_FACTOR:
418                 nevernil(node->ep);
419                 anlys_expr(node->ep);
420                 break;
421         case FUNCALL_FACTOR:
422                 nevernil(node->fcsp);
423                 anlys_fcall_stmt(node->fcsp);
424                 break;
425         default:
426                 unlikely();
427         }
428 }
429
430 static void anlys_fcall_stmt(fcall_stmt_node_t *node)
431 {
```

```
432         node->stab = scope_top();
433         nevernil(node->idp);
434         ident_node_t *idp = node->idp;
435         syment_t *e = symfind(idp->name);
436         if (!e) {
437                 giveup(BADSYM, "L%d: function %s not found.", idp->line,
438                         idp->name);
439         }
440         if (e->cate != FUN_OBJ) {
441                 giveup(ERTYPE, "L%d: symbol %s type is not function.",
442                         idp->line, idp->name);
443         }
444         idp->symbol = e;
445
446         if (node->alp) {
447                 anlys_arg_list(e, node->alp);
448         }
449 }
450
451 static void anlys_cond(cond_node_t *node)
452 {
453         nevernil(node->lep);
454         anlys_expr(node->lep);
455         nevernil(node->rep);
456         anlys_expr(node->rep);
457 }
458
459 static void anlys_arg_list(syment_t *sign, arg_list_node_t *node)
460 {
461         arg_list_node_t *t = node;
462         param_t *p = sign->phead;
463         int pos = 0;
464         for (; t && p; t = t->next, p = p->next) {
465                 pos++;
466                 syment_t *e, *a;
467                 e = p->symbol;
468                 switch (e->cate) {
469                 case BYVAL_OBJ:
470                         nevernil(t->ep);
471                         anlys_expr(t->ep);
472                         t->refsym = e;
473                         break;
```

```
474                 case BYREF_OBJ: // var, arr[exp]
475                         if (!t->ep) {
476                                 goto referr;
477                         }
478                         if (t->ep->kind != NOP_ADDOP) {
479                                 goto referr;
480                         }
481                         expr_node_t *ep = t->ep;
482                         if (ep->next) {
483                                 goto referr;
484                         }
485                         if (!ep->tp || ep->tp->kind != NOP_MULTOP) {
486                                 goto referr;
487                         }
488                         term_node_t *tp = ep->tp;
489                         if (!tp->fp) {
490                                 goto referr;
491                         }
492
493                         factor_node_t *fp = tp->fp;
494                         ident_node_t *idp;
495                         if (fp->kind == ID_FACTOR || fp->kind == ARRAY_FACTOR) {
496                                 idp = fp->idp;
497                                 t->idx = fp->ep;
498                                 anlys_factor(fp);
499                                 goto refok;
500                         }
501                 referr:
502                         giveup(BADREF, "L%d: %s() arg%d has bad reference.",
503                                sign->lineno, sign->name, pos);
504                         continue;
505                 refok:
506                         a = symfind(idp->name);
507                         if (!a) {
508                                 giveup(BADSYM, "L%d: symbol %s not found.",
509                                        idp->line, idp->name);
510                         }
511                         if (fp->kind == ID_FACTOR && a->cate != VAR_OBJ) {
512                                 giveup(OBJREF,
513                                        "L%d: %s() arg%d is not variable object.",
514                                        idp->line, idp->name, pos);
515                         }
```

```
516                         if (fp->kind == ARRAY_FACTOR && a->cate != ARRAY_OBJ) {
517                                 giveup(OBJREF,
518                                         "L%d: %s() arg%d is not array object.",
519                                         idp->line, idp->name, pos);
520                         }
521                         t->argsym = idp->symbol = a;
522                         t->refsym = e;
523                         break;
524                 default:
525                         unlikely();
526                 }
527         }
528
529         if (t || p) {
530                 giveup(BADLEN,
531                         "L%d: %s(...) arguments and parameters length not equal.",
532                         sign->lineno, sign->name);
533         }
534 }
535
536 void analysis()
537 {
538         anlys_pgm(pgm);
539         chkerr("annlysis fail and exit.");
540         phase = IR;
541 }
```

这些分析函数代码是在特定的语义分析场景下编写的，我们在后续的小节中结合具体场景讨论代码的功能。

4.7.2 未定义标识符

未定义标识符通常指的是在程序中使用了一个没有事先定义或声明的变量、函数、类或其他符号。这种错误通常会导致编译失败或运行时错误。未定义的标识符是非常常见的语法错误，并且在我们的日常的编程中也经常遇到这样的错误。sem01.pas 是一个未定义标识符错误演示，具体代码如下：

```
1 var x : integer;
2 begin
3    x := 1;
4    y := 2
5 end.
```

这里通过 fpc 编译器编译一下代码，具体输出结果如下：

```
$ fpc sem01.pas
Free Pascal Compiler version 3.2.2+dfsg-9ubuntu1 [2022/04/11] for x86_64
Copyright (c) 1993-2021 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling sem01.pas
sem01.pas(4,4) Error: Identifier not found "y"
sem01.pas(6) Fatal: There were 1 errors compiling module, stopping
Fatal: Compilation aborted
Error: /usr/bin/ppcx64 returned an error exitcode
$
```

这里的 `Identifier not found "y"` 指的是没有找到标识符 `y` ，这就是未定义标识符错误。所以我们在语法分析的过程中需要对标识符进行检查，检查是否在使用标识符之前已经进行了定义或声明。确保所有的变量、函数、类等在使用前都已经进行了定义或声明。

由于符号表记录了语法分析时所有的符号信息，所以具体的检查标识符是否定义依赖于符号表，如果符号表中不存在的符号均被判定为未定义的标识符。注意：这里需要遍历当前符号表及其外部的符号表中所有的符号。之前在符号表的操作函数中介绍的 `symfind()` 函数可以递归查询所有符号表，所以可以使用这个函数来判断标识符是否定义。一个具体实现的例子就是在 `anlys_assign_stmt()` 函数中使用下面代码片段查询标识符。

```
1 syment_t *e;
2 ident_node_t *idp = node->idp;
3 e = symfind(idp->name);
4 if (!e) {
5         giveup(BADSYM, "L%d: symbol %s not found.", idp->line,
6                idp->name);
7 }
```


4.7.3 重复定义标识符

为变量和函数选择唯一的名称是非常重要的。使用具有描述性的名称可以避免与其他代码冲突，并使代码更易于理解和维护。如果标识符被重复定义，这通常会导致编译错误或运行时错误。这里还是编写一个 `sem02.pas` 的代码实例来说明这个场景：

```
1 var x, y, x : integer;
2 begin
3    x := 1;
4    y := 2
5 end.
```

通过 `fpc` 编译验证可以发现会报出 `Duplicate identifier "x"` 这样的重复标识符定义的错误。具体的验证效果如下：

```
$ fpc sem02.pas
```

```
Free Pascal Compiler version 3.2.2+dfsg-9ubuntu1 [2022/04/11] for x86_64
Copyright (c) 1993-2021 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling sem02.pas
sem02.pas(1,13) Error: Duplicate identifier "x"
sem02.pas(6) Fatal: There were 1 errors compiling module, stopping
Fatal: Compilation aborted
Error: /usr/bin/ppcx64 returned an error exitcode
$
```

需要注意的是标识符在同一作用域中重复定义会出现错误，但是不同作用域中却是可以重复定义的，下面的 sem03.pas 的代码虽然定义了两个 x 变量，但却是可以正确编译的。

```
1  var x, y: integer;
2  procedure p1();
3    var x : integer;
4    begin
5       x := 3
6    end;
7  begin
8     x := 1;
9     y := 2
10 end.
```


具体编译的结果如下，其中的 3 个 Note 表示变量复制并没有使用，这些只是一些提示信息，不是错误。

```
$ fpc sem03.pas
Free Pascal Compiler version 3.2.2+dfsg-9ubuntu1 [2022/04/11] for x86_64
Copyright (c) 1993-2021 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling sem03.pas
sem03.pas(3,7) Note: Local variable "x" is assigned but never used
sem03.pas(1,5) Note: Local variable "x" is assigned but never used
sem03.pas(1,8) Note: Local variable "y" is assigned but never used
Linking sem03
10 lines compiled, 0.0 sec
3 note(s) issued
$
```

综上所述，检查标识符是否重复定义仅需查询当前作用域的符号表，这点和未定义标识符查询是不同的。在之前符号表的操作函数中，我们使用 symget() 函数来查询是否存在重复定义的变量。

anlys_var_decf() 函数是对变量定义进行解析，这里就是使用 symget() 来检查标识符是否重复定义，如果重复定义就报错，否则调用 syminit() 函数构造符号项 e 并插入当前作用域的符

号表中。最终还要将符号项 e 回写到语法树中方便引用。

```
ident_node_t *idp = p->idp;
syment_t *e = symget(idp->name);
if (e) {
        rescue(DUPSYM,
               "L%d: variable %s already declared.",
               idp->line, idp->name);
} else {
        e = syminit(idp);
}
idp->symbol = e;
```

4.7.4 作用域进入及退出

$PL/0\epsilon$ 语言是函数作用域，即每个函数都需要维护自己的符号表，根据之前的介绍作用域嵌套是也就是符号表栈的出栈和入栈。

全局符号表是在 anlys_pgm() 函数中创建的，通过 scope_entry(MAINFUNC) 创建主函数的符号表，并将设置 entry->symbol->scope 指针指向全局符号表。在 anlys_pgm() 函数结束，还需要调用 scope_exit() 来弹出全局符号表。

函数的定义是需要将当前函数的相关声明信息插入当前符号表，当已进入函数的参数定义时则需要新建函数的局部符号表创建。所以，在 anlys_proc_head() 和 anlys_fun_head() 函数中，首先通过 syminit(idp) 将当前函数的符号插入符号表，然后调用 scope_entry(idp->name) 开启新的作用域。

函数作用域的退出是在 anlys_proc_decf() 和 anlys_fun_decf() 中进行处理，当过程或函数的执行体定义完成后，调用 scope_exit() 退出当前作用域。

4.7.5 类型检查

类型检查是指在编译时或运行时检查程序中变量、函数参数和返回值等的类型是否符合预期的过程。类型检查有助于发现和预防类型错误，提高代码的健壮性和可维护性。

编译器会检查代码中使用的变量、函数参数和返回值是否具有正确的类型。如果类型不匹配，编译器将产生错误并阻止代码的编译。

类型检查有助于减少运行时错误，因为类型错误在编译时被捕获，而不是在程序运行时才发现。这使得开发人员能够更快地发现和修复错误，提高开发效率。

此外，类型检查还可以帮助提高代码的可读性和可维护性。通过强制使用特定类型的变量和函数，可以减少代码中的混淆和错误。这使得代码更容易理解和维护，并减少未来的错误和问题。

需要注意的是，过度依赖类型检查可能会限制代码的灵活性和可扩展性。有时候，使用动态类型语言或某些类型的类型推断技术可以提高代码的灵活性和可读性。因此，在编写代码时，需要平

衡类型检查的严格性和灵活性，以满足项目的需求。

在之前类型系统的介绍中，$PL/0\epsilon$ 仅支持 **integer** 整型和 **char** 字符型这两种类型，为了简化编码工作，我们在存储中 **char** 也使用 32 位的整数存储，所以，当出现 **integer** + **char** 这样的场景，我们就是使用强制类型转换规则把 **char** 当成 **integer** 进行计算，类似与下面的 c 语言代码：

```
int x, ans;
char y;
ans = x + (int)y;
```

在下面代码中会将 **char** 类型的 y 标识符隐式转换成 **integer** 类型参与计算：

```
var x, ans : integer; y : char;
begin
   ans := x + y;
end.
```

有了上述假设，我们这里在做语义分析时，就忽略变量的类型检查，只在后续代码翻译中对类型进行隐式转换。

4.7.6 四则算术运算

文法 (1.20) - (1.24) 定义了 $PL/0\epsilon$ 语言支持四则运算，即 + − ∗ / 这四个运算符。然而 + 、− 和 ∗ 对整数域是封闭运算，其中 / 运算却不是，例如：$3/2 = 1.5$ ，所以为了简化计算模型，我们这里定义 / 为整数除法，只取结果的整数部分，所以 $3/2 = 1$ 。

4.7.7 参数校验

函数或过程的形参和实参在调用时也可能存在问题，常见的场景有：

1. 参数数量不匹配
2. 参数类型不匹配
3. 引用传值是的实参存在问题

下面使用一个示例程序 sem04.pas 来说明这些问题：

```
var u, v, ans: integer;
function f1(): integer;
  begin
     f1 := 3
  end;
function f2(x, y : integer): integer;
  begin
     f2 := x + y
  end;
```

```
10 function f3(var x, y : integer): integer;
11   begin
12      f3 := x + y
13   end;
14 begin
15    u := 1; v := 2;
16    ans := f1(u, v);
17    ans := f2(u+v, v);
18    ans := f3(u+v, v)
19 end.
```

第 16 行调用函数 f1() 是传入的参数数量不正确，属于语义错误。第 17 行调用函数 f2() 时，其中函数定义时参数 x 和 y 的为传值类型，所以调用时支持传入 u+v 这样的表达式，调用正确。第 18 行调用函数 f3() 时，其中函数定义时参数 x 和 y 的为传引用类型，所以调用时不支持传入 u+v 这样的表达式，调用错误。

参数检查的代码需要结合 anlys_para_list() 和 anlys_arg_list() 这两个函数，anlys_para_list() 函数是分析形参的逻辑，在分析形参列表时，同时需要构建函数签名 sign 的参数列表，即 sign->phead 和 sign->ptail 。anlys_arg_list() 函数是分析实参的逻辑，处理传引用的实参在 case BYREF_OBJ: 情景中处理，传引用包括：var 和 arr[exp] 这两个类型的情况，当检查到不符合时需要报错。最后在 anlys_arg_list() 检查参数数量是否相同，如果参数相等的话，t 和 p 指针都应该为空，否则报错。

4.8 思考题

1. 斐波那契数列（Fibonacci sequence），又称黄金分割数列，因意大利数学家莱昂纳多·斐波那契（Leonardo Fibonacci）1202 年以兔子繁殖为例子而引入，故又称为“兔子数列”，指的是这样一个数列：

$$1, 1, 2, 3, 5, 8, 13, 21 \cdots$$

(4.1) 是 $fib(n)$ 的递归定义（其中 $\mathbb{N}^+$ 是正整数集合）。

$$fib(n) = \begin{cases} 1 & \textbf{if } n \in \{1, 2\} \\ fib(n-1) + fib(n-2) & \textbf{if } n \in \mathbb{N}^+ \wedge n \notin \{1, 2\} \end{cases} \tag{4.1}$$

下面的是 $fib(n)$ 的代码实现，试着画出 fib(4) 各个递归调用时的符号表的变化。

```
1 function fib(n : integer ): integer; begin
2    if n <= 2 then
3       fib := 1
4    else
5       fib := fib(n-1) + fib(n-2);
6 end;
```

```
7
8  begin
9     fib(4)
10 end.
```

2. **函数重载**是指在同一个作用域内，可以定义多个同名函数，只要它们的参数列表不同（参数类型、数量、顺序不同）。这样，编译器可以根据函数调用时提供的参数类型和数量来决定应该调用哪个函数。下面的代码 `overload.pas` 定义了两个函数的名字相同 `f` 函数，但是参数数量不是相同的。

```
1  var ans : integer;
2  function f(x : integer):integer; begin f := x end;
3  function f(x, y : integer ):integer; begin f := x + y end;
4  begin
5     ans := f(1);
6     write(ans);
7     ans := f(3, 4);
8     write(ans)
9  end.
```

使用 fpc 可以正确编译 `overload.pas` ，编译结果如下：

```
$ fpc overload.pas
Free Pascal Compiler version 3.2.2+dfsg-9ubuntu1 [2022/04/11] for x86_64
Copyright (c) 1993-2021 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling overload.pas
Linking overload
9 lines compiled, 0.0 sec
$
```

然而 pcc 编译却会出现函数重复定义的错误，编译过程如下：

```
$ ./pcc example/overload.pas
compiler pcc start, version v0.18.3
reading file example/overload.pas
ERROR: L3: function f already declared.
L2: f call arguments and parameters length not equal.
$
```

请思考一下 pcc 需要怎样的改造才能支持函数重载？

4.9 本章总结

本章主要介绍语义分析的实现细节。首先介绍了符号表的设计与实现，然后介绍函数作用域的细节，明确函数的相关语义细节。最后通过具体语义分析场景说明语义分析函数家族的实现细节。

第五章　中间代码

5.1　中间代码

中间代码 (Intermediate Representation) 的作用是方便编译器的优化和生成目标代码，如图 5.1 所示，编译原理中的中间代码是源代码在编译过程中生成的中间表示形式，它位于源代码和目标代码之间。另外，$PL/0\epsilon$ 中实现了的前端和后端在图中通过灰底的方式标记出来了。

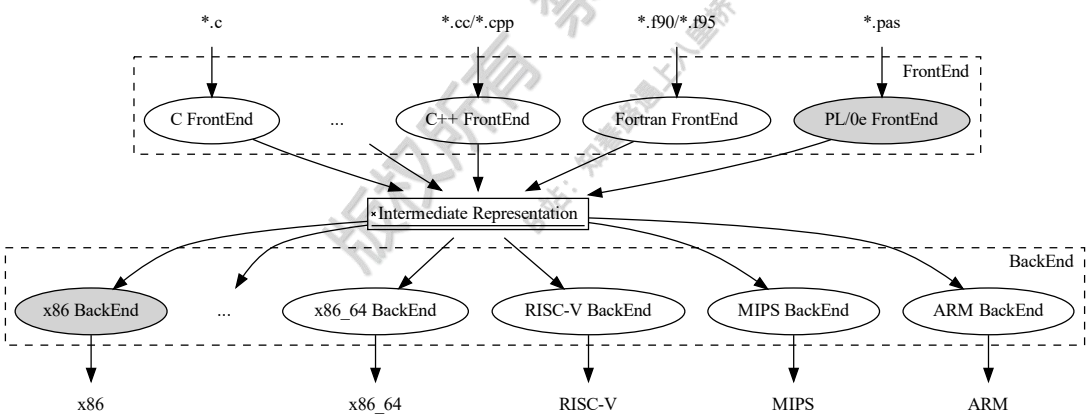


图 5.1: 中间代码示意图

中间代码的最常用形式就是是三地址码和四元式。

三地址码 (Three-Address Code, TAC) 是一种类似于汇编语言的中间代码形式，它由一系列的三元式组成，每个三元式包含三个部分，其形式为:

$$\textbf{operand3} \leftarrow \textbf{operand1} \quad \textbf{operator} \quad \textbf{operand2}$$

其中 operator 是操作符、operand1, operand2 和 operand3 是三个操作数。三地址码的优点是易于生成和理解，但它的缺点是不易于进行高级优化。

我们先列举一个具体场景中来说明三地址码的作用，表达式 (5.1) 是一个我们熟悉的算术表达式，它表示将 $(a \times b) - (c \div d)$ 的值赋给 x 。

$$x := (a * b) - (c/d) \tag{5.1}$$

当对表达式（5.1）进行三地址码翻译时，会生成（5.2）-（5.5）这样的序列。其中 T_1 、T_2 和 T_3 是临时变量，用于记录中间值。$\leftarrow$ 表示赋值操作。

$$T_1 \leftarrow a \times b \tag{5.2}$$
$$T_2 \leftarrow c \div d \tag{5.3}$$
$$T_3 \leftarrow T_1 - T_2 \tag{5.4}$$
$$x \leftarrow T_3 \tag{5.5}$$

现对于表达式（5.1），三地址码（5.2）-（5.5）这样的序列表达的语义更加单一，对计算机来说更加容易理解。

四元式（Quadruple）是另一种中间代码形式，由操作符、结果变量和两个操作数组成。四元式的形式为：

operator　destination, source1, source2

其中 operator 是操作符，destination 是结果变量，source1 和 source2 是操作数。操作符可以是二元操作符（需要两个操作数），也可以是一元操作符（需要一个操作数）或零元操作符（不需要操作数）。

四元式的优点是可以方便地进行代码优化，例如通过合并相邻的四元式、消除公共子表达式等。此外，四元式也方便进行代码生成，因为它类似于汇编语言的格式。然而，四元式的缺点是生成和理解比较复杂，因为需要处理更多的元素和关系。此外，四元式也不易进行高级优化，例如函数内联、循环展开等。

四元式和三地址码可以进行相互转换，当我们使用四元式表示（5.2）-（5.5）三地址码序列会得到（5.6）-（5.9）四元式序列。

$$*\quad T_1,\ a,\ b \tag{5.6}$$
$$/\quad T_2,\ c,\ d \tag{5.7}$$
$$-\quad T_3,\ T_1,\ T_2 \tag{5.8}$$
$$\leftarrow\quad x,\ T_3, \tag{5.9}$$

当然，$PL/0\epsilon$ 语言的中间代码采用四元式表示，后续的章节会详细介绍设计方案。

5.2 四元式

5.2.1 四元式设计

表 5.1 是针对 $PL/0\epsilon$ 设计的四元式，根据功能可以分成以下几类：

1. 算术指令：包含加减乘除四则运算，用于算术运算。
2. 存储指令：包含读取数组值、赋值操作和数组赋值操作，用于存取数值。
3. 条件跳转指令：包含通过条件判断来调整对应 label 的操作，用于计算控制流。
4. 无条件跳转指令：包含直接跳转到 label 的功能。
5. 栈管理指令：包含堆栈入栈，数组地址入栈和出栈操作，用于运行时堆栈管理。
6. 函数调用指令：用于控制函数调用和函数退出等逻辑。
7. 读写指令：用于控制程序的 I/O 操作，可以读取或打印到控制台。
8. 标号：给程序添加 LABEL ，方便跳转时进行地址交叉引用。

为了后续章节说明的方便，这里统一四元式的术语，我们定义表 5.1 中的四元式形式如下：

op d, r, s

其中 op 是操作符，它包含操作符编号 (opnum) 和操作符编码 (opcode)，d 是结果变量，r 和 s 是两个操作数。通过表格可以看出，操作符包含二元操作符，例如：ADD、SUB；一元操作符，例如：NEG，PADR；以及零元操作符，例如：LAB；

我们使用之前的 twosum.pas 作为实例来介绍四元式大概如何使用：

```
1 const a = 1, b = 2;
2 var x : integer;
3 begin
4   x := a + b
5 end.
```

我们直接将 twosum.pas 的语法树翻译成如下的四元式序列，具体包含两个部分：

1. DUMP SYMBOLS
   - 打印目前编译器中的所有符号
   - label 作为符号的标记，在所有表中是唯一的
   - 这里还包含一些其它属性：类型 type，分类 cate 等
2. DUMP INTERMEDIATE REPRESENTATION
   - 打印四元式序列
   - 开始的 #xxx 是四元式的顺序号
   - 每个四元式都至少包含 op 和 d；r 和 s 可能不包含

```
DUMP SYMBOLS:
label=L001 type=1 cate=FUN name=_start off=1 stab=1 depth=1 initval=0 arrlen=0 str=
label=L002 type=1 cate=CONST name=a off=0 stab=1 depth=1 initval=1 arrlen=0 str=
label=L003 type=1 cate=CONST name=b off=0 stab=1 depth=1 initval=2 arrlen=0 str=
label=L004 type=1 cate=VAR name=x off=2 stab=1 depth=1 initval=0 arrlen=0 str=
```

表 5.1: $PL/0\epsilon$ 语言的四元式设计

功能分类	操作符		操作数			功能描述
	opnum	opcode	d	r	s	
算术指令	0	ADD	d	r	s	$d \leftarrow r + s$
	1	SUB	d	r	s	$d \leftarrow r - s$
	2	MUL	d	r	s	$d \leftarrow r \times s$
	3	DIV	d	r	s	$d \leftarrow r \div s$
	4	INC	d			$d \leftarrow d + 1$
	5	DEC	d			$d \leftarrow d - 1$
	6	NEG	d	r		$d \leftarrow -r$
存取指令	7	LOAD	d	r	s	Load value $d \leftarrow r[s]$
	8	ASS	d	r		Assign value $d \leftarrow r$
	9	ASA	d	r	s	Assign value $d[s] \leftarrow r$
条件跳转指令	10	EQU	d	r	s	Jump to label d if $r = s$
	11	NEQ	d	r	s	Jump to label d if $r \neq s$
	12	GTT	d	r	s	Jump to label d if $r > s$
	13	GEQ	d	r	s	Jump to label d if $r \geq s$
	14	LST	d	r	s	Jump to label d if $r < s$
	15	LEQ	d	r	s	Jump to label d if $r \leq s$
无条件跳转指令	16	JMP	d			Jump to label d
栈管理指令	17	PUSH	d			Push value d into stack
	18	PADR	d	r		Push address $\&d[r]$ into stack
	19	POP	d			Pop value of stack, save to d
函数调用指令	20	CALL	d	r		Call function r, and $d \leftarrow r()$
	21	ENT	d			Enter function d
	22	FIN	d			Finish function d
读写指令	23	RDI	d			Read integer, save to d
	24	RDC	d			Read character, save to d
	25	WRI	d			Write integer from d
	26	WRC	d			Write character from d
	27	WRS	d			Write string from d
标号	28	LAB	d			Label d

```
label=T005 type=1 cate=TMP name=@expr/add off=3 stab=1 depth=1 initval=0 arrlen=0 str=

DUMP INTERMEDIATE REPRESENTATION:
#001: ENT       d=L001
#002: ADD       d=T005  r=L002  s=L003
#003: ASS       d=L004  r=T005
#004: FIN       d=L001
```

由于每个四元式的 d, r 和 s 域都会引用一个符号，这里就可以对符号的进行引用，即每个四元式的都有在所有符号中找到一个与之一一对应的符号项。其中四元式打印的解释如下：

1. #001 的操作符是 ENT 表示进入函数，通过 L001 查找符号表，找到 _start 函数，说明是主函数入口。
2. #002 的操作符是 ADD 相加操作，r 和 s 是两个操作数，d 是结果变量：
   - r=L002 查找符号表得到一个 name=a 的 CONST 常量。
   - s=L003 查找符号表得到一个 name=b 的 CONST 常量。
   - d=T005 查找符号表得到一个 type=1 的 TMP 临时变量。
   - 这个四元式的语义就是把 L002 和 L003 相加的结果存入 T005 中。
3. #003 的操作符是 ASS 赋值操作，r 是操作数，d 是结果变量：
   - r=T005 查找符号表得到一个 type=1 的 TMP 临时变量，是 #002 的结果。
   - d=L004 查找符号表得到一个 name=x 的 VAR 变量。
   - 这个四元式就是把 T005 临时变量值取出来放入 L004 变量中。
4. #004 的操作符是 FIN 表示完成退出函数，通过 L001 查找符号表，找到 _start 函数，说明是主函数退出。

5.2.2 四元式数据结构

在之前章节中关于符号表介绍的前提下，四元式的数据结构就显得比较简单易懂，其结构体 inst_t 的定义在文件 source/ir.h 中。里面包括以下定义值：

1. inst_t 类型表示一个具体的四元式，它包含：
   - op 表示操作符。
   - d 表示结果变量，r，s 表示两个操作数，它们都会指向一个符号项。
2. op_t 是枚举类型，它具体定义和表 5.1 中的操作符相对应。
3. prev，next 是指向四元式的前后的指针，因为语法树最终翻译成的四元式构成了双向链表。

这些定义代码如下：

```
1 // Instruction Operator Type
2 typedef enum _inst_op_enum {
3         // Arithmetic
4         /*  0 */ ADD_OP,
5         /*  1 */ SUB_OP,
6         /*  2 */ MUL_OP,
7         /*  3 */ DIV_OP,
```

```
        /*  4 */ INC_OP,
        /*  5 */ DEC_OP,
        /*  6 */ NEG_OP,
        // Load and Store
        /*  7 */ LOAD_OP,
        /*  8 */ ASS_OP,
        /*  9 */ ASA_OP,
        // Conditional Branch
        /* 10 */ EQU_OP,
        /* 11 */ NEQ_OP,
        /* 12 */ GTT_OP,
        /* 13 */ GEQ_OP,
        /* 14 */ LST_OP,
        /* 15 */ LEQ_OP,
        // Unconditional Branch
        /* 16 */ JMP_OP,
        // Stack Management
        /* 17 */ PUSH_OP,
        /* 18 */ PADR_OP,
        /* 19 */ POP_OP,
        // Function Management
        /* 20 */ CALL_OP,
        /* 21 */ ENT_OP,
        /* 22 */ FIN_OP,
        // I/O Management
        /* 23 */ RDI_OP,
        /* 24 */ RDC_OP,
        /* 25 */ WRS_OP,
        /* 26 */ WRI_OP,
        /* 27 */ WRC_OP,
        // Label Marker
        /* 28 */ LAB_OP
} op_t;

// Instruction struct
typedef struct _inst_struct inst_t;

typedef struct _inst_struct {
        int xid;
        op_t op;
        syment_t *d;
        syment_t *r;
```

```
50          syment_t *s;
51          inst_t *prev;
52          inst_t *next;
53  } inst_t;
```

5.2.3 中间代码队列

中间代码发射是指将中间代码插入一个公共的队列中的过程，这个队列的实际结构如图 5.2 所示，它是为后续翻译成真实的汇编代码而准备。

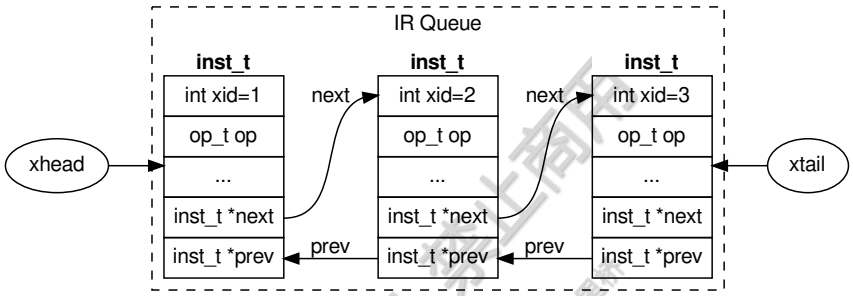


图 5.2: 中间代码队列图

我们在文件 include/ir.h 中定义了队列的队首 xhead 和队尾 xtail 两个指针，通过 xhead 可以遍历整个队列。此外，这里还定义了三个发射函数 emit1() ，emit2() 和 emit3() ，这三个函数的区别是发射的入参数量不同，但是功能完全一致。具体实现代码如下：

```
1  // Constructor
2  #define NEWINST(v) INITMEM(inst_t, v)
3
4  // hold instructions
5  extern inst_t *xhead;
6  extern inst_t *xtail;
7
8  // emit an instruction
9  inst_t *emit1(op_t op, syment_t *d);
10 inst_t *emit2(op_t op, syment_t *d, syment_t *r);
11 inst_t *emit3(op_t op, syment_t *d, syment_t *r, syment_t *s);
```

文件 include/ir.h 中定义了代码的具体实现，其实就是调用 emit() 函数往队尾插入一条 inst_t 结构体。

```
1  // instructions
2  inst_t *xhead;
3  inst_t *xtail;
```

```
4
5   // instruction count
6   int xidcnt = 0;
7
8   static inst_t *emit(op_t op)
9   {
10          inst_t *t;
11          NEWINST(t);
12          t->xid = ++xidcnt;
13          t->op = op;
14
15          if (xtail) {
16                  t->prev = xtail;
17                  xtail->next = t;
18                  xtail = t;
19          } else {
20                  t->prev = xtail;
21                  xhead = xtail = t;
22          }
23
24          dbg("emit xid=%d op=%d\n", t->xid, op);
25          return t;
26  }
27
28  inst_t *emit1(op_t op, syment_t *d)
29  {
30          inst_t *x = emit(op);
31          x->d = d;
32          return x;
33  }
34
35  inst_t *emit2(op_t op, syment_t *d, syment_t *r)
36  {
37          inst_t *x = emit(op);
38          x->d = d;
39          x->r = r;
40          return x;
41  }
42
43  inst_t *emit3(op_t op, syment_t *d, syment_t *r, syment_t *s)
44  {
45          inst_t *x = emit(op);
```

```
46          x->d = d;
47          x->r = r;
48          x->s = s;
49          return x;
50  }
```

5.3 语法树转换四元式

编译器会将优化后的语法树转换为四元式。在转换过程中，编译器会遍历语法树，并为每个节点生成相应的四元式。为了具体说明这个转换过程，后面小节中将会通过具体示例讲解。

5.3.1 表达式转换

我们在语义分析后会得到一个符合语义检查的语法树，这个语法树中有一类表示表达式 EXPR 的节点，我们通常对于这类表达式需要进行求值，求值的一个核心步骤就是生成求值的四元式序列。我们借助之前表达式（5.1）的例子来说明，首先我们将这个表达式翻译成代码 `ir01.pas` ，具体实现如下：

```
1  var a, b, c, d, x: integer;
2  begin
3    x := (a * b) - (c / d)
4  end.
```


如图 5.3 所示，在之前的语义分析后我可以得到一个的包含符号项的语法树，虚线圈出来的子树就是我们需要关注的表达式求值部分，它就是表示上面代码的第 3 行的语法结构。

我们对这部分代码生成的符号表和四元式进行打印，最终得到的结构如下：

```
DUMP SYMBOLS:
label=L001 type=1 cate=FUN name=_start off=1 stab=1 depth=1 initval=0 arrlen=0 str=
label=L002 type=1 cate=VAR name=a off=2 stab=1 depth=1 initval=0 arrlen=0 str=
label=L003 type=1 cate=VAR name=b off=3 stab=1 depth=1 initval=0 arrlen=0 str=
label=L004 type=1 cate=VAR name=c off=4 stab=1 depth=1 initval=0 arrlen=0 str=
label=L005 type=1 cate=VAR name=d off=5 stab=1 depth=1 initval=0 arrlen=0 str=
label=L006 type=1 cate=VAR name=x off=6 stab=1 depth=1 initval=0 arrlen=0 str=
label=T007 type=1 cate=TMP name=@term/mul off=7 stab=1 depth=1 initval=0 arrlen=0 str=
label=T008 type=1 cate=TMP name=@term/div off=8 stab=1 depth=1 initval=0 arrlen=0 str=
label=T009 type=1 cate=TMP name=@expr/sub off=9 stab=1 depth=1 initval=0 arrlen=0 str=

DUMP INTERMEDIATE REPRESENTATION:
#001: ENT       d=L001
#002: MUL       d=T007  r=L002  s=L003
#003: DIV       d=T008  r=L004  s=L005
```

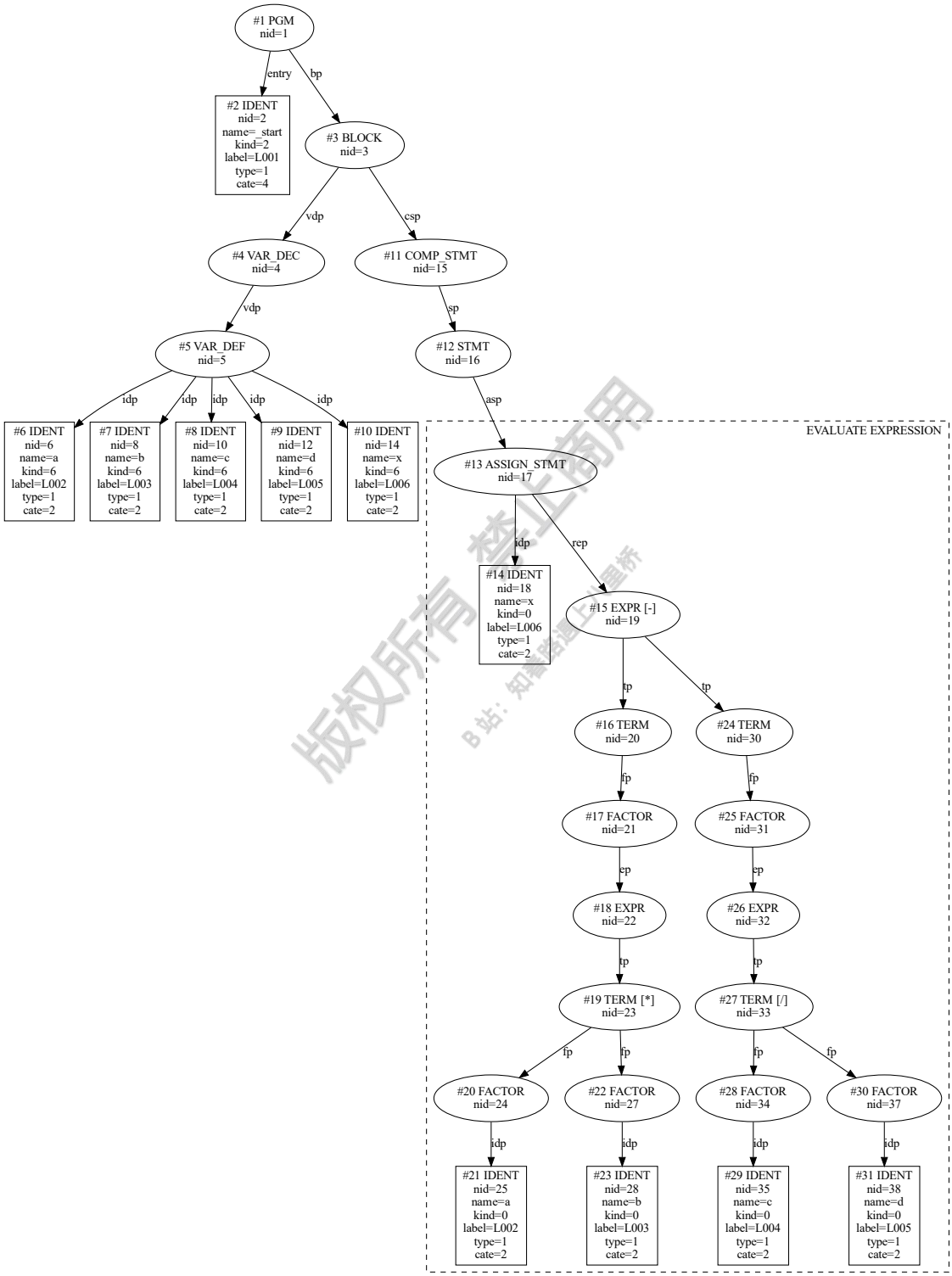


图 5.3：表达式求值示例的语法树

```
#004: SUB       d=T009  r=T007  s=T008
#005: ASS       d=L006  r=T009
#006: FIN       d=L001
```

在中间代码的序列中，通过符号表找到 L002，L003 和 T007 等对应关系。我们可以构造出具体的三地址码，其表示方法如下：

$$T_7 \leftarrow L_2(a) \times L_3(b) \tag{5.10}$$

$$T_8 \leftarrow L_4(c) \div L_5(d) \tag{5.11}$$

$$T_9 \leftarrow T_7 - T_8 \tag{5.12}$$

$$L_6(x) \leftarrow T_9 \tag{5.13}$$

在 (5.10) 中 $L_2(a)$ 表示的实际是 L002 对应的符号项，这个符号项实际是变量 a ，这里通过小括号将代指的变量包裹起来。另外，T 开头的表示临时变量，这些变量在程序代码中不存在，但是存在于四元式中。

5.3.2 控制流：if 条件语句转换

if 条件语句时程序中最常见的控制流语句，我们编写一个 ir02.pas 来说明 if 条件语句的转换，其代码如下：

```
1 var a, b, c, d, x: integer;
2 begin
3    if a > b then
4       x := c
5    else
6       x := d
7 end.
```

上述代码的核心是第 3 到 5 行的 if 条件语句，该部分根据 a>b 这个条件来将变量 x 赋值成 c 或者 d 。通过语义分析可以得到如图 5.4 所示的语法树，其中虚线圈定的是 IF_STMT 子树，里面包含条件 COND 节点，then 语句序列部分和 else 语句序列部分。我们对其符号表和四元式指令序列进行分析得到如下结果：

```
DUMP SYMBOLS:
label=L001 type=1 cate=FUN name=_start off=1 stab=1 depth=1 initval=0 arrlen=0 str=
label=L002 type=1 cate=VAR name=a off=2 stab=1 depth=1 initval=0 arrlen=0 str=
label=L003 type=1 cate=VAR name=b off=3 stab=1 depth=1 initval=0 arrlen=0 str=
label=L004 type=1 cate=VAR name=c off=4 stab=1 depth=1 initval=0 arrlen=0 str=
label=L005 type=1 cate=VAR name=d off=5 stab=1 depth=1 initval=0 arrlen=0 str=
label=L006 type=1 cate=VAR name=x off=6 stab=1 depth=1 initval=0 arrlen=0 str=
label=T007 type=0 cate=LABEL name=@ifthen off=0 stab=1 depth=1 initval=0 arrlen=0 str=
label=T008 type=0 cate=LABEL name=@ifdone off=0 stab=1 depth=1 initval=0 arrlen=0 str=
```

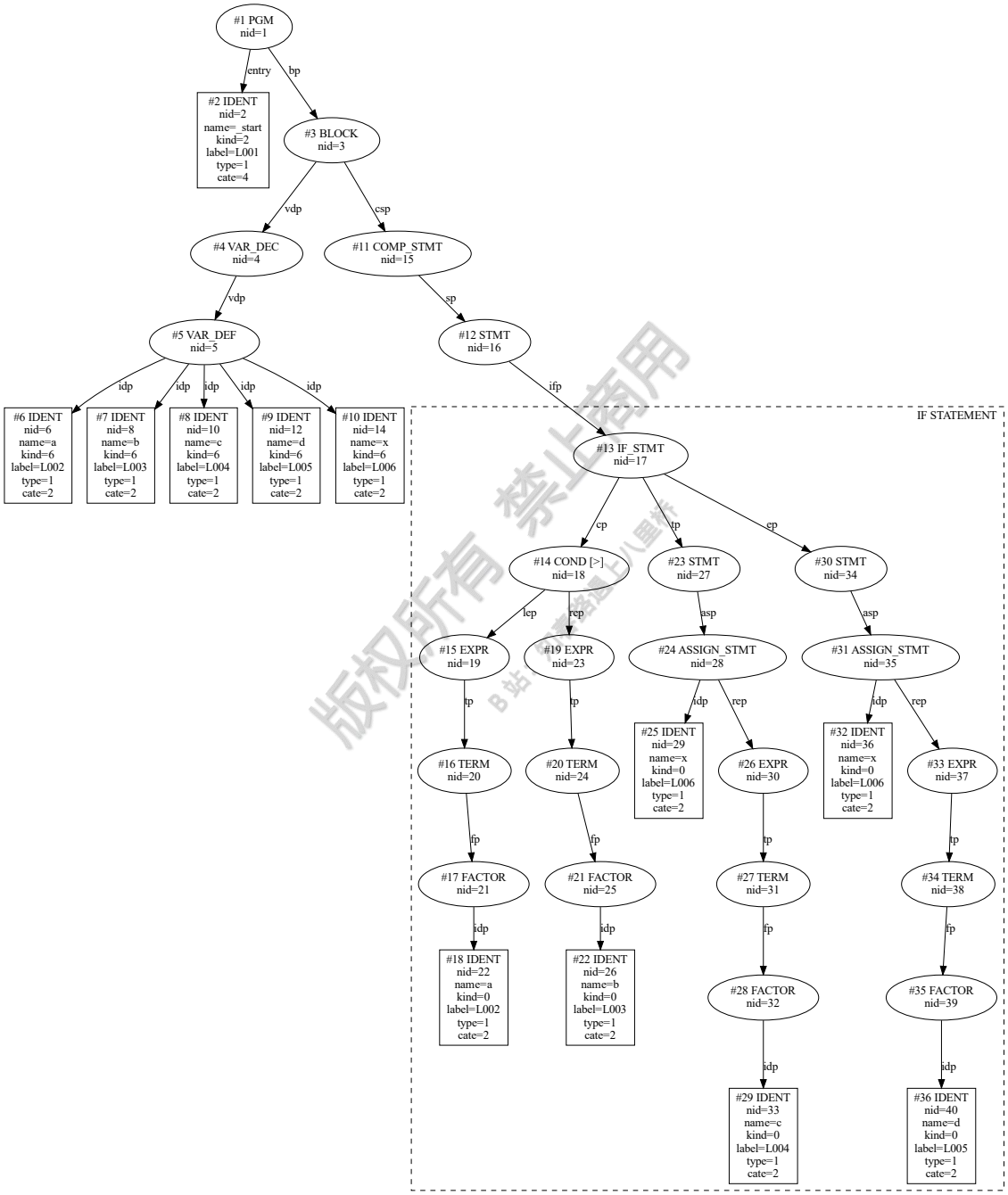


图 5.4: if 条件语句示例的语法树

```
DUMP INTERMEDIATE REPRESENTATION:
#001: ENT       d=L001
#002: GTT       d=T007  r=L002  s=L003
#003: ASS       d=L006  r=L005
#004: JMP       d=T008
#005: LAB       d=T007
#006: ASS       d=L006  r=L004
#007: LAB       d=T008
#008: FIN       d=L001
```

指令 (5.14) - (5.19) 是对 if 条件语句的翻译，(5.14) 判断 a>b 这个条件是否成立，

- 如果成立，则跳转到标号 T_7 中，对应 (5.17) 指令。跳转过后执行指令 (5.18)，将 x 赋值成 c 。
- 如果不成立，则直接执行下一条指令，即 (5.15) 指令，将 x 赋值成 d 。

$$\text{goto } T_7 \text{ if } L_2(a) > L_3(b) \tag{5.14}$$
$$L_6(x) \leftarrow L_5(d) \tag{5.15}$$
$$\text{goto } T_8 \tag{5.16}$$
$$T_7: \tag{5.17}$$
$$L_6(x) \leftarrow L_4(c) \tag{5.18}$$
$$T_8: \tag{5.19}$$

5.3.3 控制流：for 循环语句转换

for 循环语句时程序中常用循环语句，其中 $PL/0\epsilon$ 的循环语句和 Pascal 的 for 循环语义类似，我们编写一个 ir03.pas 来说明 for 循环语句的转换，其代码如下：

```
1 var i, sum: integer;
2 begin
3    for i := 1 to 10 do
4       sum := sum + i;
5 end.
```

上述代码是进行求和操作，具体实现公式 5.20 中的计算功能，这种等差数列非常适合使用 for 循环进行计算，计算的结果为 55 。

$$\begin{aligned}\sum_{n=1}^{10} n &= 1 + 2 + ... + 10 \\ &= \frac{(1+10)\times 10}{2}\end{aligned}$$

$$= 55 \tag{5.20}$$

图 5.5 是 for 循环的语法树，我们对上述求和代码进行分析可以得到下面的符号表和四元式序列：

```
DUMP SYMBOLS:
label=L001 type=1 cate=FUN name=_start off=1 stab=1 depth=1 initval=0 arrlen=0 str=
label=L002 type=1 cate=VAR name=i off=2 stab=1 depth=1 initval=0 arrlen=0 str=
label=L003 type=1 cate=VAR name=sum off=3 stab=1 depth=1 initval=0 arrlen=0 str=
label=T004 type=1 cate=NUM name=@factor/usi off=0 stab=1 depth=1 initval=1 arrlen=0 str=
label=T005 type=1 cate=NUM name=@factor/usi off=0 stab=1 depth=1 initval=10 arrlen=0 str=
label=T006 type=0 cate=LABEL name=@forstart off=0 stab=1 depth=1 initval=0 arrlen=0 str=
label=T007 type=0 cate=LABEL name=@fordone off=0 stab=1 depth=1 initval=0 arrlen=0 str=
label=T008 type=1 cate=TMP name=@expr/add off=4 stab=1 depth=1 initval=0 arrlen=0 str=

DUMP INTERMEDIATE REPRESENTATION:
#001: ENT       d=L001
#002: ASS       d=L002  r=T004
#003: LAB       d=T006
#004: GTT       d=T007  r=L002  s=T005
#005: ADD       d=T008  r=L003  s=L002
#006: ASS       d=L003  r=T008
#007: INC       d=L002
#008: JMP       d=T006
#009: LAB       d=T007
#010: DEC       d=L002
#011: FIN       d=L001
```


指令 (5.21) – (5.29) 是对 for 循环语句的翻译，其中 for 循环需要将循环变量 i 赋初值后进行跳转语义生成。

$$L_2(i) \leftarrow T_4(1) \tag{5.21}$$

$$T_6: \tag{5.22}$$

$$\text{goto } T_7 \text{ if } L_2(i) > T_5(10) \tag{5.23}$$

$$T_8 \leftarrow L_3(sum) + L_2(i) \tag{5.24}$$

$$L_3(sum) \leftarrow T_8 \tag{5.25}$$

$$L_2(i)++ \tag{5.26}$$

$$\text{goto } T_6 \tag{5.27}$$

$$T_7: \tag{5.28}$$

$$L_2(i)-- \tag{5.29}$$

上述的四元式序列的含义如下：

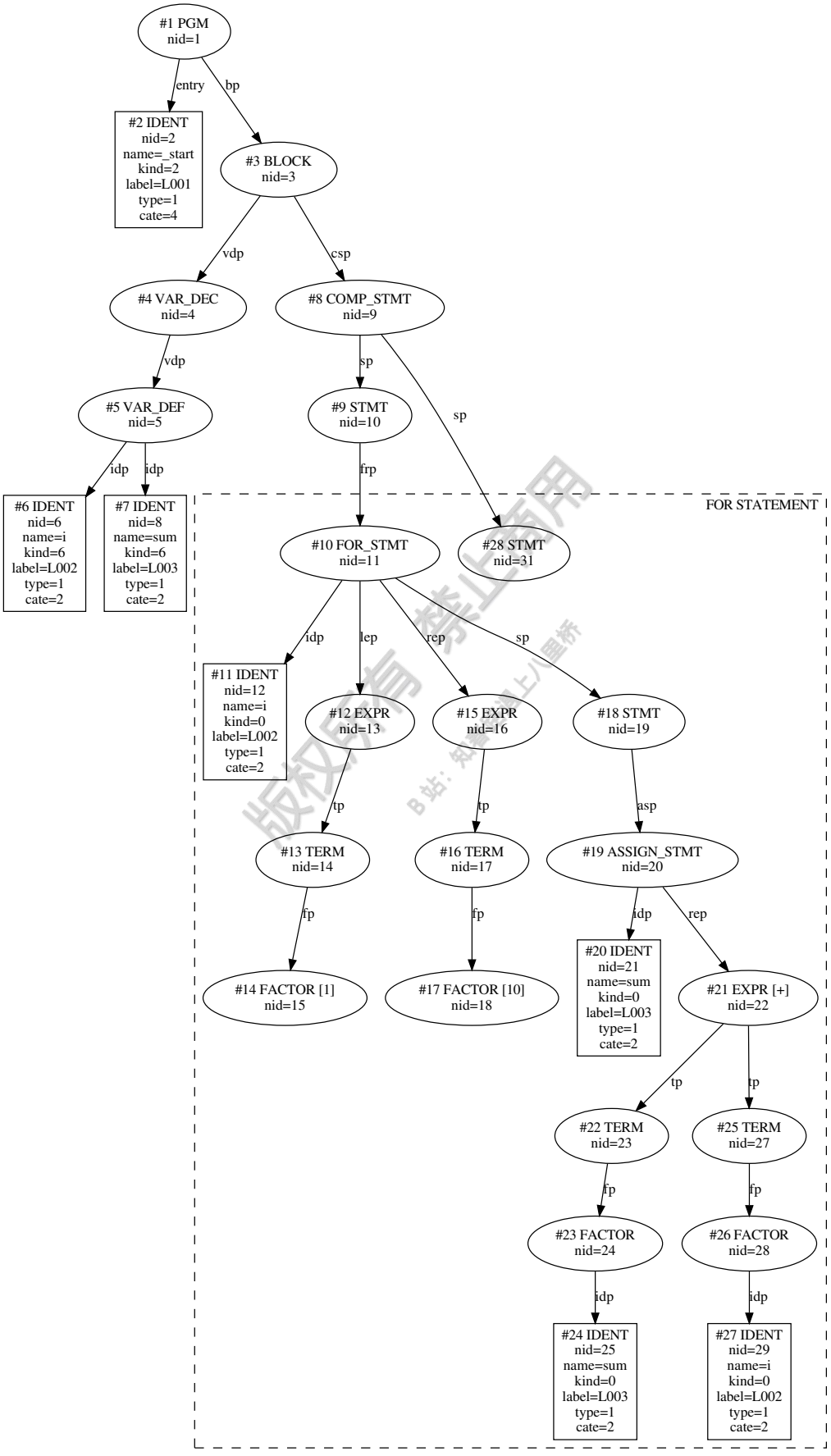


图 5.5: for 循环语句示例的语法树

1. 指令 (5.21) 将变量 $L_2(i)$ 赋初始值 $T_4(1)$ 数字
2. 指令 (5.22) 是 for 循环开始的标号 T_6
3. 指令 (5.23) 判断 $L_2(i) > T_5(10)$ ，即 i>10 是否成立
   - 如果成立，跳转到标号 T_7
   - 否则执行下一条语句 (5.24)
4. 指令 (5.24) 计算 $L_3(sum) + L_2(i)$ 的和，结果值放入 T_8
5. 指令 (5.25) 回写 T_8 到 $L_3(sum)$
6. 指令 (5.26) 将 $L_2(i)$ 的值自增
7. 指令 (5.27) 跳转到 for 循环开始 T_6 标号，进入下一次循环
8. 指令 (5.28) 是 for 循环退出标号 T_7
9. 指令 (5.29) 循环退出还需要将 $L_2(i)$ 的值自减

根据上述分析可知：指令 (5.24) - (5.25) 是 for 循环的执行体，其它部分的指令都是执行控制指令。

5.3.4 函数调用

我们通过累加器这个例子说明函数调用的细节，具体实现 ir04.pas 代码如下：

```
1  var u, v, ans: integer;
2
3  function adder(x, y : integer):integer;
4  begin
5     adder := x + y
6  end;
7
8  begin
9     u := 1;
10    v := 2;
11    ans := adder(u, v)
12 end.
```

我们对上面函数调用的代码的符号表和四元式进行分析，可以得到如下输出：

```
DUMP SYMBOLS:
label=L001 type=1 cate=FUN name=_start off=1 stab=1 depth=1 initval=0 arrlen=0 str=
label=L002 type=1 cate=VAR name=u off=2 stab=1 depth=1 initval=0 arrlen=0 str=
label=L003 type=1 cate=VAR name=v off=3 stab=1 depth=1 initval=0 arrlen=0 str=
label=L004 type=1 cate=VAR name=ans off=4 stab=1 depth=1 initval=0 arrlen=0 str=
label=L005 type=1 cate=FUN name=adder off=5 stab=1 depth=1 initval=0 arrlen=0 str=
label=L006 type=1 cate=BYVAL name=x off=0 stab=2 depth=2 initval=0 arrlen=0 str=
label=L007 type=1 cate=BYVAL name=y off=1 stab=2 depth=2 initval=0 arrlen=0 str=
label=T008 type=1 cate=TMP name=@expr/add off=1 stab=2 depth=2 initval=0 arrlen=0 str=
label=T009 type=1 cate=NUM name=@factor/usi off=0 stab=1 depth=1 initval=1 arrlen=0 str=
```

```
label=T010 type=1 cate=NUM name=@factor/usi off=0 stab=1 depth=1 initval=2 arrlen=0 str=
label=T011 type=1 cate=TMP name=@fcall/ret off=6 stab=1 depth=1 initval=0 arrlen=0 str=

DUMP INTERMEDIATE REPRESENTATION:
#001: ENT       d=L005
#002: ADD       d=T008  r=L006  s=L007
#003: ASS       d=L005  r=T008
#004: FIN       d=L005
#005: ENT       d=L001
#006: ASS       d=L002  r=T009
#007: ASS       d=L003  r=T010
#008: PUSH      d=L003
#009: PUSH      d=L002
#010: CALL      d=T011  r=L005
#011: POP
#012: POP
#013: ASS       d=L004  r=T011
#014: FIN       d=L001
```

输出的结构可以分成两个部分，#001 到 #004 是 adder() 函数的定义，#005 到 #014 是入口函数执行体，其中 #008 到 #013 是函数调用的细节。

$$\Rightarrow L_5(adder) \tag{5.30}$$

$$T_8 \leftarrow L_6(x) + L_7(y) \tag{5.31}$$

$$L_5(adder) \leftarrow T_8 \tag{5.32}$$

$$\Leftarrow L_5(adder) \tag{5.33}$$

adder() 函数的定义指令序列见 (5.30) - (5.33)，其中，指令 (5.30) 表示函数进入，指令 (5.33) 表示函数完成退出。(5.31) 是求和并赋值操作，结果赋值到 T_8 。(5.32) 回写 T_8 到函数的返回值 $L_5(adder)$ 。

$$\text{push } L_3(v) \tag{5.34}$$

$$\text{push } L_2(u) \tag{5.35}$$

$$T_{11} \leftarrow \text{call } L_5(adder) \tag{5.36}$$

$$\text{pop} \tag{5.37}$$

$$\text{pop} \tag{5.38}$$

$$L_4(ans) \leftarrow T_{11} \tag{5.39}$$

adder() 函数调用指令序列见 (5.30) - (5.39)，指令 (5.34) 和指令 (5.35) 将参数逆序压入调用栈，(5.36) 调用函数 $L_5(adder)$ 并将结果写入临时变量 T_{11} 中。指令 (5.37) 和 (5.38)

将入参弹出，保持堆栈平衡。指令（5.39）将函数调用结果 T_{11} 赋值给 $L_4(ans)$ 变量。

5.4 四元式生成函数

5.4.1 gen_xxx 生成函数家族

和语义分析的 anlys_xxx 函数家族类似，四元式的生成函数也是一系列函数，这些函数的定义在文件 include/gen.h 中定义，具体实现代码如下：

```
static void gen_pgm(pgm_node_t *node);
static void gen_pf_dec_list(pf_dec_list_node_t *node);
static void gen_proc_decf(proc_dec_node_t *node);
static void gen_fun_decf(fun_dec_node_t *node);
static void gen_comp_stmt(comp_stmt_node_t *node);
static void gen_stmt(stmt_node_t *node);
static void gen_assign_stmt(assign_stmt_node_t *node);
static void gen_if_stmt(if_stmt_node_t *node);
static void gen_repe_stmt(repe_stmt_node_t *node);
static void gen_for_stmt(for_stmt_node_t *node);
static void gen_pcall_stmt(pcall_stmt_node_t *node);
static void gen_read_stmt(read_stmt_node_t *node);
static void gen_write_stmt(write_stmt_node_t *node);
static syment_t *gen_expr(expr_node_t *node);
static syment_t *gen_term(term_node_t *node);
static syment_t *gen_factor(factor_node_t *node);
static syment_t *gen_fcall_stmt(fcall_stmt_node_t *node);
static void gen_cond(cond_node_t *node, syment_t *dest);
static void gen_arg_list(arg_list_node_t *node);

void genir();
```

gen_xxx 函数家族是一些 gen_ 开头的分析函数，这些函数对特定的语法树结构进行分析，并调用 emit() 发射函数进行翻译生成四元式序列，最后，这些函数的入口点是 genir() 函数。

5.4.2 gen_xxx 生成函数实现

gen_xxx 函数家族实现在文件 source/gen.c 中，这部分代码不复杂，需要注意的是一些控制流和表达式求值的具体实现。同时，函数调用也是通过发射 ENT_OP 和 FIN_OP 进行包裹。

```
static void gen_pgm(pgm_node_t *node)
{
        block_node_t *b = node->bp;
```

```
4           gen_pf_dec_list(b->pfdlp);
5
6           // main function
7           syment_t *entry = node->entry->symbol;
8           emit1(ENT_OP, entry);
9           gen_comp_stmt(b->csp);
10          emit1(FIN_OP, entry);
11  }
12
13  static void gen_pf_dec_list(pf_dec_list_node_t *node)
14  {
15          pf_dec_list_node_t *t;
16          for (t = node; t; t = t->next) {
17                  switch (t->kind) {
18                  case PROC_PFDEC:
19                          gen_proc_decf(t->pdp);
20                          break;
21                  case FUN_PFDEC:
22                          gen_fun_decf(t->fdp);
23                          break;
24                  default:
25                          unlikely();
26                  }
27          }
28  }
29
30  static void gen_proc_decf(proc_dec_node_t *node)
31  {
32          proc_dec_node_t *t;
33          for (t = node; t; t = t->next) {
34                  block_node_t *b = t->pdp->bp;
35                  gen_pf_dec_list(b->pfdlp);
36
37                  emit1(ENT_OP, t->pdp->php->idp->symbol);
38                  gen_comp_stmt(b->csp);
39                  emit1(FIN_OP, t->pdp->php->idp->symbol);
40          }
41  }
42
43  static void gen_fun_decf(fun_dec_node_t *node)
44  {
45          fun_dec_node_t *t;
```

```
46          for (t = node; t; t = t->next) {
47                  block_node_t *b = t->fdp->bp;
48
49                  gen_pf_dec_list(b->pfdlp);
50                  emit1(ENT_OP, t->fdp->fhp->idp->symbol);
51                  gen_comp_stmt(b->csp);
52                  emit1(FIN_OP, t->fdp->fhp->idp->symbol);
53          }
54  }
55
56  static void gen_comp_stmt(comp_stmt_node_t *node)
57  {
58          comp_stmt_node_t *t;
59          for (t = node; t; t = t->next) {
60                  gen_stmt(t->sp);
61          }
62  }
63
64  static void gen_stmt(stmt_node_t *node)
65  {
66          switch (node->kind) {
67          case ASSGIN_STMT:
68                  gen_assign_stmt(node->asp);
69                  break;
70          case IF_STMT:
71                  gen_if_stmt(node->ifp);
72                  break;
73          case REPEAT_STMT:
74                  gen_repe_stmt(node->rpp);
75                  break;
76          case FOR_STMT:
77                  gen_for_stmt(node->frp);
78                  break;
79          case PCALL_STMT:
80                  gen_pcall_stmt(node->pcp);
81                  break;
82          case COMP_STMT:
83                  gen_comp_stmt(node->cpp);
84                  break;
85          case READ_STMT:
86                  gen_read_stmt(node->rdp);
87                  break;
```

```
88          case WRITE_STMT:
89                  gen_write_stmt(node->wtp);
90                  break;
91          case NULL_STMT:
92                  break;
93          default:
94                  unlikely();
95          }
96  }
97
98  static void gen_assign_stmt(assign_stmt_node_t *node)
99  {
100         syment_t *r, *s, *d;
101         d = node->idp->symbol;
102         switch (node->kind) {
103         case NORM_ASSGIN:
104                 r = gen_expr(node->rep);
105                 emit2(ASS_OP, d, r);
106                 break;
107         case FUN_ASSGIN:
108                 r = gen_expr(node->rep);
109                 emit2(ASS_OP, d, r);
110                 break;
111         case ARRAY_ASSGIN:
112                 s = gen_expr(node->lep);
113                 r = gen_expr(node->rep);
114                 emit3(ASA_OP, d, r, s);
115                 break;
116         default:
117                 unlikely();
118         }
119 }
120
121 static void gen_if_stmt(if_stmt_node_t *node)
122 {
123         syment_t *ifthen, *ifdone;
124         ifthen = symalloc(node->stab, "@ifthen", LABEL_OBJ, VOID_TYPE);
125         ifdone = symalloc(node->stab, "@ifdone", LABEL_OBJ, VOID_TYPE);
126
127         gen_cond(node->cp, ifthen);
128         if (node->ep) {
129                 gen_stmt(node->ep);
```

```
130         }
131         emit1(JMP_OP, ifdone);
132         emit1(LAB_OP, ifthen);
133         gen_stmt(node->tp);
134         emit1(LAB_OP, ifdone);
135 }
136
137 static void gen_repe_stmt(repe_stmt_node_t *node)
138 {
139         syment_t *loopstart, *loopdone;
140         loopstart = symalloc(node->stab, "@loopstart", LABEL_OBJ, VOID_TYPE);
141         loopdone = symalloc(node->stab, "@loopdone", LABEL_OBJ, VOID_TYPE);
142
143         emit1(LAB_OP, loopstart);
144         gen_stmt(node->sp);
145         gen_cond(node->cp, loopdone);
146         emit1(JMP_OP, loopstart);
147         emit1(LAB_OP, loopdone);
148 }
149
150 static void gen_for_stmt(for_stmt_node_t *node)
151 {
152         syment_t *beg, *end;
153         beg = gen_expr(node->lep);
154         end = gen_expr(node->rep);
155
156         syment_t *forstart, *fordone;
157         forstart = symalloc(node->stab, "@forstart", LABEL_OBJ, VOID_TYPE);
158         fordone = symalloc(node->stab, "@fordone", LABEL_OBJ, VOID_TYPE);
159
160         syment_t *d;
161         d = node->idp->symbol;
162         emit2(ASS_OP, d, beg);
163         emit1(LAB_OP, forstart);
164         switch (node->kind) {
165         case TO_FOR:
166                 emit3(GTT_OP, fordone, d, end);
167                 gen_stmt(node->sp);
168                 emit1(INC_OP, d);
169                 emit1(JMP_OP, forstart);
170                 emit1(LAB_OP, fordone);
171                 emit1(DEC_OP, d);
```

```
172                 break;
173         case DOWNTO_FOR:
174                 emit3(LST_OP, fordone, d, end);
175                 gen_stmt(node->sp);
176                 emit1(DEC_OP, d);
177                 emit1(JMP_OP, forstart);
178                 emit1(LAB_OP, fordone);
179                 emit1(INC_OP, d);
180                 break;
181         default:
182                 unlikely();
183         }
184 }
185
186 static void gen_pcall_stmt(pcall_stmt_node_t *node)
187 {
188         gen_arg_list(node->alp);
189         emit2(CALL_OP, NULL, node->idp->symbol);
190         arg_list_node_t *t;
191         for (t = node->alp; t; t = t->next) {
192                 emit1(POP_OP, NULL);
193         }
194 }
195
196 static void gen_read_stmt(read_stmt_node_t *node)
197 {
198         read_stmt_node_t *t;
199         syment_t *d = NULL;
200         for (t = node; t; t = t->next) {
201                 d = t->idp->symbol;
202                 switch (d->type) {
203                 case CHAR_TYPE:
204                         emit1(RDC_OP, d);
205                         break;
206                 case INT_TYPE:
207                         emit1(RDI_OP, d);
208                         break;
209                 }
210         }
211 }
212
213 static void gen_write_stmt(write_stmt_node_t *node)
```

```
214 {
215         syment_t *d = NULL;
216         switch (node->type) {
217         case STR_WRITE:
218                 d = symalloc(node->stab, "@write/str", STR_OBJ, STR_TYPE);
219                 strcopy(d->str, node->sp);
220                 emit1(WRS_OP, d);
221                 break;
222         case ID_WRITE:
223                 d = gen_expr(node->ep);
224                 switch (d->type) {
225                 case CHAR_TYPE:
226                         emit1(WRC_OP, d);
227                         break;
228                 case INT_TYPE:
229                         emit1(WRI_OP, d);
230                         break;
231                 default:
232                         unlikely();
233                 }
234                 break;
235         case STRID_WRITE:
236                 d = symalloc(node->stab, "@write/str", STR_OBJ, STR_TYPE);
237                 strcopy(d->str, node->sp);
238                 emit1(WRS_OP, d);
239                 d = gen_expr(node->ep);
240                 switch (d->type) {
241                 case CHAR_TYPE:
242                         emit1(WRC_OP, d);
243                         break;
244                 case INT_TYPE:
245                         emit1(WRI_OP, d);
246                         break;
247                 default:
248                         unlikely();
249                 }
250                 break;
251         default:
252                 unlikely();
253         }
254 }
255
```

```
256 static syment_t *gen_expr(expr_node_t *node)
257 {
258         expr_node_t *t;
259         syment_t *d, *r, *e;
260         d = r = e = NULL;
261         for (t = node; t; t = t->next) {
262                 r = gen_term(t->tp);
263                 if (!d) {
264                         switch (t->kind) {
265                         case NEG_ADDOP:
266                                 d = symalloc(node->stab, "@expr/neg", TMP_OBJ,
267                                              r->type);
268                                 emit2(NEG_OP, d, r);
269                                 break;
270                         case NOP_ADDOP:
271                                 d = r;
272                                 break;
273                         default:
274                                 unlikely();
275                         }
276                         continue;
277                 }
278                 switch (t->kind) {
279                 case NOP_ADDOP:
280                 case ADD_ADDOP:
281                         e = d;
282                         d = symalloc(node->stab, "@expr/add", TMP_OBJ, e->type);
283                         emit3(ADD_OP, d, e, r);
284                         break;
285                 case MINUS_ADDOP:
286                 case NEG_ADDOP:
287                         e = d;
288                         d = symalloc(node->stab, "@expr/sub", TMP_OBJ, e->type);
289                         emit3(SUB_OP, d, e, r);
290                         break;
291                 default:
292                         unlikely();
293                 }
294         }
295         return d;
296 }
297
```

```
298 static syment_t *gen_term(term_node_t *node)
299 {
300         term_node_t *t;
301         syment_t *d, *r, *e;
302         d = r = e = NULL;
303         for (t = node; t; t = t->next) {
304                 r = gen_factor(t->fp);
305                 if (!d) {
306                         if (t->kind != NOP_MULTOP) {
307                                 unlikely();
308                         }
309                         d = r;
310                         continue;
311                 }
312                 switch (t->kind) {
313                 case NOP_MULTOP:
314                 case MULT_MULTOP:
315                         e = d;
316                         d = symalloc(node->stab, "@term/mul", TMP_OBJ, e->type);
317                         emit3(MUL_OP, d, e, r);
318                         break;
319                 case DIV_MULTOP:
320                         e = d;
321                         d = symalloc(node->stab, "@term/div", TMP_OBJ, e->type);
322                         emit3(DIV_OP, d, e, r);
323                         break;
324                 default:
325                         unlikely();
326                 }
327         }
328         return d;
329 }
330
331 static syment_t *gen_factor(factor_node_t *node)
332 {
333         syment_t *d, *r, *e;
334         d = r = e = NULL;
335         switch (node->kind) {
336         case ID_FACTOR:
337                 d = node->idp->symbol;
338                 break;
339         case ARRAY_FACTOR:
```

```
340                     r = node->idp->symbol;
341                     e = gen_expr(node->ep);
342                     d = symalloc(node->stab, "@factor/array", TMP_OBJ, r->type);
343                     emit3(LOAD_OP, d, r, e);
344                     break;
345             case UNSIGN_FACTOR:
346                     d = symalloc(node->stab, "@factor/usi", NUM_OBJ, INT_TYPE);
347                     d->initval = node->value;
348                     break;
349             case CHAR_FACTOR:
350                     d = symalloc(node->stab, "@factor/char", NUM_OBJ, CHAR_TYPE);
351                     d->initval = node->value;
352                     break;
353             case EXPR_FACTOR:
354                     d = gen_expr(node->ep);
355                     break;
356             case FUNCALL_FACTOR:
357                     d = gen_fcall_stmt(node->fcsp);
358                     break;
359             default:
360                     unlikely();
361             }
362             return d;
363     }
364
365     static syment_t *gen_fcall_stmt(fcall_stmt_node_t *node)
366     {
367             syment_t *d, *e;
368             e = node->idp->symbol;
369             d = symalloc(node->stab, "@fcall/ret", TMP_OBJ, e->type);
370             gen_arg_list(node->alp);
371             emit2(CALL_OP, d, e);
372             arg_list_node_t *t;
373             for (t = node->alp; t; t = t->next) {
374                     emit1(POP_OP, NULL);
375             }
376             return d;
377     }
378
379     static void gen_cond(cond_node_t *node, syment_t *label)
380     {
381             syment_t *r, *s;
```

```
382         r = gen_expr(node->lep);
383         s = gen_expr(node->rep);
384         switch (node->kind) {
385         case EQU_RELA:
386                 emit3(EQU_OP, label, r, s);
387                 break;
388         case NEQ_RELA:
389                 emit3(NEQ_OP, label, r, s);
390                 break;
391         case GTT_RELA:
392                 emit3(GTT_OP, label, r, s);
393                 break;
394         case GEQ_RELA:
395                 emit3(GEQ_OP, label, r, s);
396                 break;
397         case LST_RELA:
398                 emit3(LST_OP, label, r, s);
399                 break;
400         case LEQ_RELA:
401                 emit3(LEQ_OP, label, r, s);
402                 break;
403         }
404 }
405
406 static void gen_arg_list(arg_list_node_t *node)
407 {
408         if (!node) {
409                 return;
410         }
411         arg_list_node_t *t = node;
412
413         // Push arguments in reverse order
414         gen_arg_list(t->next);
415
416         syment_t *d = NULL, *r = NULL;
417         switch (t->refsym->cate) {
418         case BYVAL_OBJ:
419                 d = gen_expr(t->ep);
420                 emit1(PUSH_OP, d);
421                 break;
422         case BYREF_OBJ:
423                 d = t->argsym;
```

```
424                     switch (t->argsym->cate) {
425                     case VAR_OBJ:
426                             emit2(PADR_OP, d, NULL);
427                             break;
428                     case ARRAY_OBJ:
429                             r = gen_expr(t->idx);
430                             emit2(PADR_OP, d, r);
431                             break;
432                     default:
433                             unlikely();
434                     }
435                     break;
436             default:
437                     unlikely();
438             }
439     }
440
441     void genir()
442     {
443             gen_pgm(pgm);
444             chkerr("generate fail and exit.");
445             phase = CODE_GEN;
446     }
```


5.4.3 四元式生成的注意点

gen_xxx 生成函数的实现可以参考上小节中的代码，直接通过代码阅读收获会更大。这里对通过语法树节点生成四元式过程的一些注意点进行补充说明。

在语义分析的时候，我们会往符号表中插入一些的符号项，这些符号项是在代码中可以找到具体定义的符号，例如：常量、变量、函数名和过程名等。然而，在四元式生成函数中需要生成一些临时的符号项，这些符号项的 cate 主要包括如下几类：

1. cate=NUM 表示数字，通常是数字字面量，例如：0，-23 等。
2. cate=TMP 表示临时变量，这是类型的变量和 cate=VAR 的逻辑结构类似，都需要分配内存，但是临时变量是编译器生成的中间变量，它与源代码中的变量没有对应关系。
3. cate=LABEL 表示标号，它的主要功能是为汇编的 label 提供参照。通常控制流相关的语义都需要包含 label，例如：if 语句、for 语句等。

调用函数的入参需要放入栈中传递，我们这里对这些参数进行逆序入栈，后续需要根据这样的顺序来读取参数。这部分具体实现在 gen_arg_list() 函数中，gen_arg_list() 通过递归调用自身来逆序压入入参，请读者注意 gen_arg_list() 递归调用位置，必须放在 emit() 发射函数调用之前。

表达式求值相关生成函数是 gen_factor() 、gen_term() 和 gen_expr() 。这些函数都有返回值，返回值其实就是记录表达式的值的符号项，所以表达式求值会相互递归调用来完成表达式的求值。

5.5 prtir 调试工具

为了调试方便，我们还实现了一个打印四元式解析结构的命令行工具，具体见文件 tool/prtir.c ，这里演示它的使用方法。

```
$ ./bin/prtir ./example/ir01.pas
compiler pcc start, version v0.18.4
reading file ./example/ir01.pas

DUMP SYMBOLS:
label=L001 type=1 cate=FUN name=_start off=1 stab=1 depth=1 initval=0 arrlen=0 str=
label=L002 type=1 cate=VAR name=a off=2 stab=1 depth=1 initval=0 arrlen=0 str=
label=L003 type=1 cate=VAR name=b off=3 stab=1 depth=1 initval=0 arrlen=0 str=
label=L004 type=1 cate=VAR name=c off=4 stab=1 depth=1 initval=0 arrlen=0 str=
label=L005 type=1 cate=VAR name=d off=5 stab=1 depth=1 initval=0 arrlen=0 str=
label=L006 type=1 cate=VAR name=x off=6 stab=1 depth=1 initval=0 arrlen=0 str=
label=T007 type=1 cate=TMP name=@term/mul off=7 stab=1 depth=1 initval=0 arrlen=0 str=
label=T008 type=1 cate=TMP name=@term/div off=8 stab=1 depth=1 initval=0 arrlen=0 str=
label=T009 type=1 cate=TMP name=@expr/sub off=9 stab=1 depth=1 initval=0 arrlen=0 str=

DUMP INTERMEDIATE REPRESENTATION:
#001: ENT       d=L001
#002: MUL       d=T007  r=L002  s=L003
#003: DIV       d=T008  r=L004  s=L005
#004: SUB       d=T009  r=T007  s=T008
#005: ASS       d=L006  r=T009
#006: FIN       d=L001

$
```

5.6 本章总结

本章主要介绍中间代码的概念，具体介绍了三地址码和四元式，并完成了 $PL/0\epsilon$ 的四元式的设计与实现。在四元式设计完成的情况下，介绍了语法树往四元式转换的案例。代码实现部分给出了 gen_xxx 家族函数，并介绍了调试工具。

第六章　目标代码

6.1　目标代码

目标代码生成（Target Code Generation）是指将源代码转换成目标代码的过程。目标代码通常是机器语言或汇编语言代码，可以直接被计算机执行。

在编译器中，目标代码生成是编译过程的最后一个阶段。在这个阶段，编译器将源代码转换成机器语言或汇编语言代码，以便计算机能够直接执行。目标代码生成是编译过程的核心环节之一，其生成的代码质量和执行效率对于整个编译器的性能和用户体验具有重要意义。

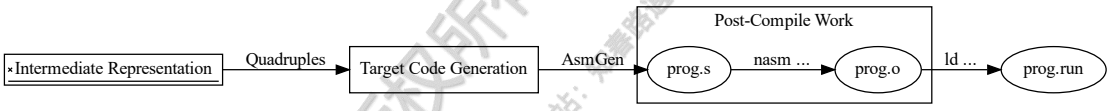


图 6.1: 目标代码及后置工作示意图

图 6.1 展示 pcc 工具将中间代码编译成目标代码的最后过程。目标代码生成器负责将四元式生成汇编文件 `prog.s` ，接着是编译的两个后置工作，使用 nasm 汇编器将汇编文件 `prog.s` 转换成二进制的目标文件 `prog.o` 。使用 ld 工具将 `prog.o` 链接成可执行文件 `prog.run` 。这样，我们最终得到了在操作系统中直接运行的编译产物，整个编译工作就圆满完成了。

6.2　x86 体系结构

X86 体系结构是一种微处理器指令集架构，由 Intel 公司开发。它是基于 “段加偏移” 的寻址模式，指令系统相对简单，易于扩展，因此被广泛应用于各种计算机和处理器中。X86 体系结构最早的微处理器是 Intel 为 PC-MRI（PC-Mint）公司生产的 8086，随后的发展包括 8 位、16 位、32 位和 64 位的处理器，从实模式到保护模式，从单核到多核，逐渐成为现代计算机的重要组成部分。

为了更好地理解编译器后端的工作原理，pcc 编译器将采用 X86 作为一个具体的后端汇编语言进行汇编翻译。由于主题的限制，关于 X86 的 32 位汇编语言我们这里只做一些基本知识的介

绍，不做深入讨论。

X86 体系结构中的寄存器是用于存储数据和指令的内部存储单元。这些寄存器在 CPU 内部，通过总线与内存和其他硬件设备进行通信。表 6.1 列举了 X86 体系结构中常用的寄存器，可以分成：通用寄存器、指针寄存器、变地址寄存器、控制寄存器和段寄存器这几个类别。

表 6.1: X86 体系结构的常用寄存器

分类	英文全称	16 位	32 位	64 位
通用寄存器	Accumulator	ax	eax	rax
	Base	bx	ebx	rbx
	Counter	cx	ecx	rcx
	Data	dx	edx	rdx
指针寄存器	Stack Pointer	sp	esp	rsp
	Base Pointer	bp	ebp	rbp
变地址寄存器	Source Index	si	esi	rsi
	Destination Index	di	edi	rdi
控制寄存器	Instruction Pointer	ip	eip	rip
	Flag	flag	eflag	eflag
段寄存器	Code Segment	cs	cs	cs
	Data Segment	ds	ds	ds
	Stack Segment	ss	ss	ss
	Extra Segment	es	es	es

关于寄存器的定义代码见文件 include/x86.h ，其中定义了寄存器的结构体 reg_t 类型，还有一些寄存器的宏和操作函数的定义。

```
// register
typedef struct _reg_struct {
        char name[MAXREGNAME];
        int refcnt;
} reg_t;

// Pointer register
#define REG_BP "ebp"
#define REG_SP "esp"
#define REG_DI "edi"
#define REG_SI "esi"
#define REG_RA "eax"
#define REG_RB "ebx"
#define REG_RC "ecx"
#define REG_RD "edx"
#define REG_CL "cl"
#define REG_DL "dl"
#define BTP_SI "byte[esi]"
```

```
#define SYSCAL "0x80"

#define ALIGN 4

// General register operations
reg_t *allocreg();
reg_t *lockreg(char *name);
void freereg(reg_t *r);

```

X86 指令集可以分为四类：通用指令、浮点数运算指令、SIMD 指令和特殊指令。通用指令用于执行算术运算、逻辑运算、传送数据等基本操作。浮点数运算指令用于执行浮点数运算，包括加、减、乘、除等操作。SIMD 指令用于执行向量运算和图像处理等操作，可以同时处理多个数据。特殊指令用于执行特殊操作，如控制转移和系统管理。

X86 指令集的优点包括简单性、易扩展性和广泛应用性。由于其简单性，X86 指令集易于实现和维护，同时由于其易扩展性，可以方便地添加新的指令和功能。此外，由于其广泛应用性，X86 指令集已经成为计算机和处理器领域中的主流指令集之一。由于 X86 指令集的数量比较多，一般是通过查找 Intel 官网 [1] 中指令集的手册来了解其功能，图 6.2 是官网的截图。这里对指令集不作过多展开，后续在使用中在根据具体场景介绍。

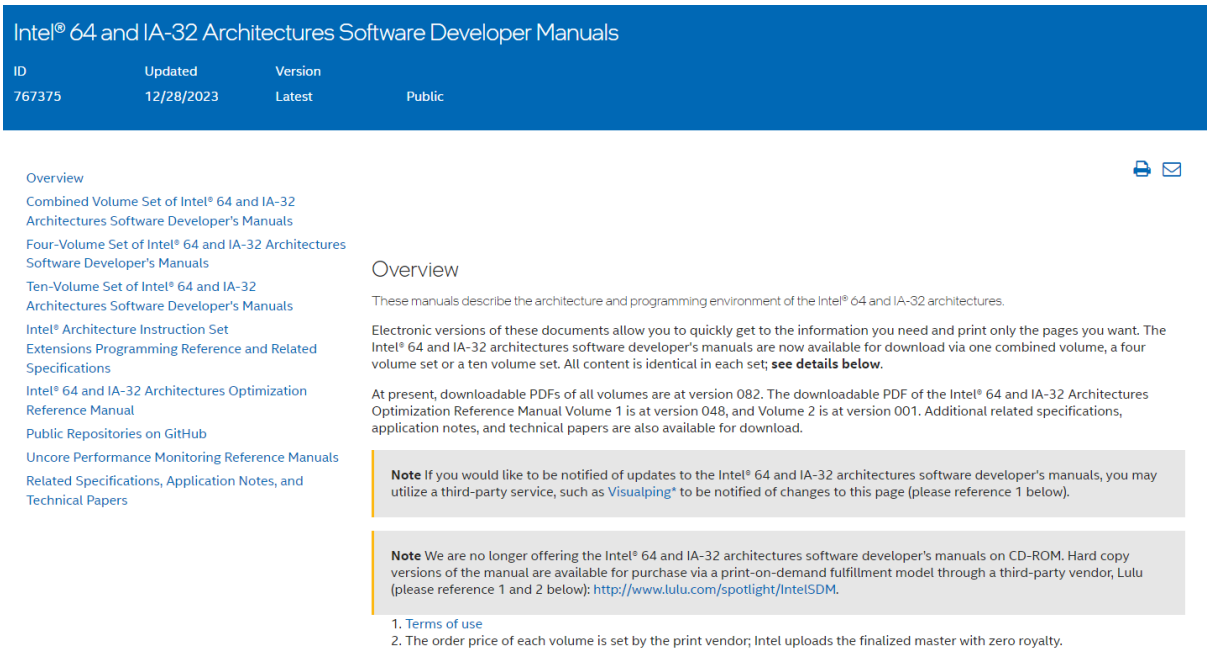
pcc 使用到的 X86 指令也在文件 `include/x86.h` 中进行了定义，具体包括 X86 指令结构体 `x86i_t` 类型，还有一系列以 `x86_xxx()` 开头的 X86 的指令生成函数的声明。

```
// x86 instructions
typedef struct _x86_inst_struct {
        bool islab; // if instruction is a label
        char op[MAXFIELDLEN]; // operator or label
        char fa[MAXFIELDLEN]; // operand field a
        char fb[MAXFIELDLEN]; // operand field b
        char et[MAXFIELDLEN]; // extra: comment, label etc.
} x86i_t;

typedef struct _program_code_struct {
        int idata;
        x86i_t data[MAXDATASEC];
        int itext;
        x86i_t text[MAXTEXTSEC];
} progcode_t;

typedef enum _rwmemmode_enum {
        READ_MEM_VAL,
        READ_MEM_REF,
```

[1] https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html



Intel® 64 and IA-32 Architectures Software Developer Manuals

ID 767375 | Updated 12/28/2023 | Version Latest | Public

Overview
Combined Volume Set of Intel® 64 and IA-32 Architectures Software Developer's Manuals
Four-Volume Set of Intel® 64 and IA-32 Architectures Software Developer's Manuals
Ten-Volume Set of Intel® 64 and IA-32 Architectures Software Developer's Manuals
Intel® Architecture Instruction Set Extensions Programming Reference and Related Specifications
Intel® 64 and IA-32 Architectures Optimization Reference Manual
Public Repositories on GitHub
Uncore Performance Monitoring Reference Manuals
Related Specifications, Application Notes, and Technical Papers

Overview

These manuals describe the architecture and programming environment of the Intel® 64 and IA-32 architectures.

Electronic versions of these documents allow you to quickly get to the information you need and print only the pages you want. The Intel® 64 and IA-32 architectures software developer's manuals are now available for download via one combined volume, a four volume set or a ten volume set. All content is identical in each set; **see details below**.

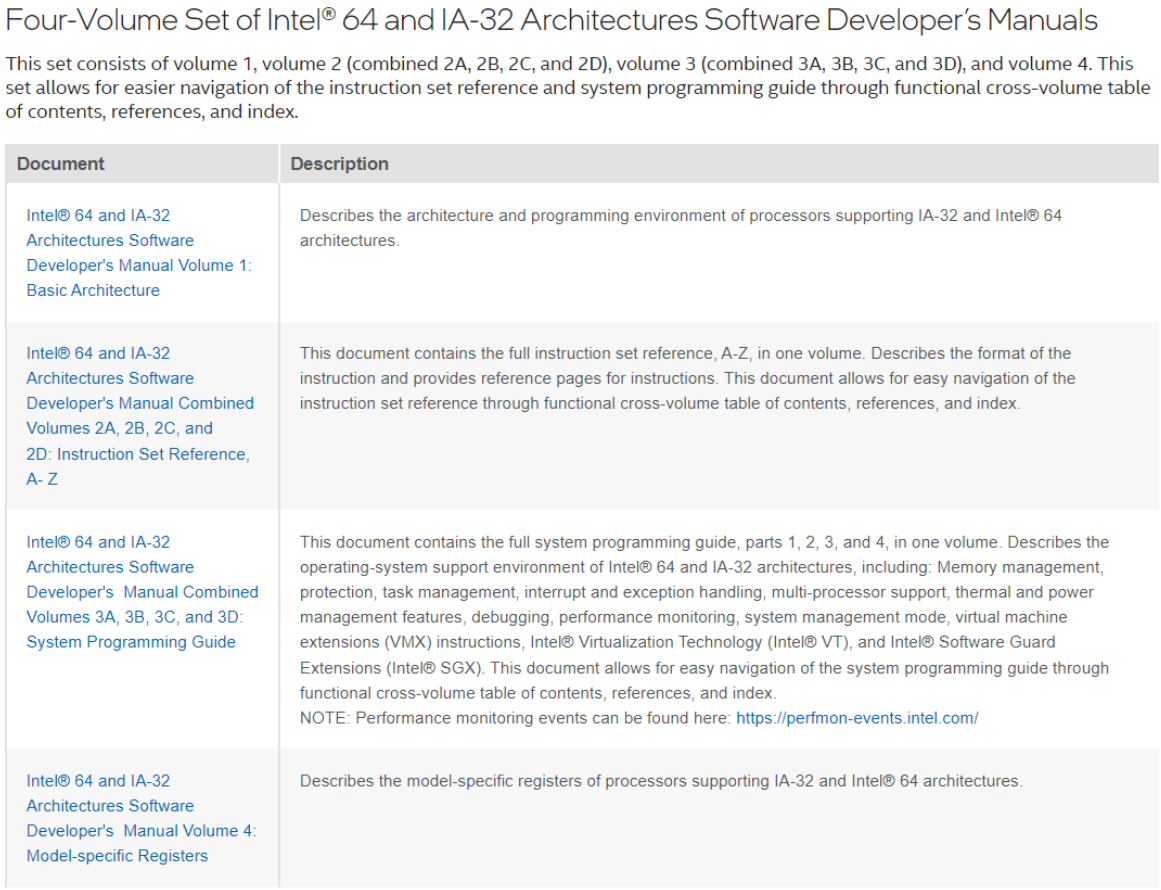
At present, downloadable PDFs of all volumes are at version 082. The downloadable PDF of the Intel® 64 and IA-32 Architectures Optimization Reference Manual Volume 1 is at version 048, and Volume 2 is at version 001. Additional related specifications, application notes, and technical papers are also available for download.

Note If you would like to be notified of updates to the Intel® 64 and IA-32 architectures software developer's manuals, you may utilize a third-party service, such as Visualping* to be notified of changes to this page (please reference 1 below).

Note We are no longer offering the Intel® 64 and IA-32 architectures software developer's manuals on CD-ROM. Hard copy versions of the manual are available for purchase via a print-on-demand fulfillment model through a third-party vendor, Lulu (please reference 1 and 2 below): http://www.lulu.com/spotlight/IntelSDM.

1. Terms of use
2. The order price of each volume is set by the print vendor; Intel uploads the finalized master with zero royalty.

(a) Intel 开发者官网截图



Four-Volume Set of Intel® 64 and IA-32 Architectures Software Developer's Manuals

This set consists of volume 1, volume 2 (combined 2A, 2B, 2C, and 2D), volume 3 (combined 3A, 3B, 3C, and 3D), and volume 4. This set allows for easier navigation of the instruction set reference and system programming guide through functional cross-volume table of contents, references, and index.

Document	Description
Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture	Describes the architecture and programming environment of processors supporting IA-32 and Intel® 64 architectures.
Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 2A, 2B, 2C, and 2D: Instruction Set Reference, A- Z	This document contains the full instruction set reference, A-Z, in one volume. Describes the format of the instruction and provides reference pages for instructions. This document allows for easy navigation of the instruction set reference through functional cross-volume table of contents, references, and index.
Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 3A, 3B, 3C, and 3D: System Programming Guide	This document contains the full system programming guide, parts 1, 2, 3, and 4, in one volume. Describes the operating-system support environment of Intel® 64 and IA-32 architectures, including: Memory management, protection, task management, interrupt and exception handling, multi-processor support, thermal and power management features, debugging, performance monitoring, system management mode, virtual machine extensions (VMX) instructions, Intel® Virtualization Technology (Intel® VT), and Intel® Software Guard Extensions (Intel® SGX). This document allows for easy navigation of the system programming guide through functional cross-volume table of contents, references, and index. NOTE: Performance monitoring events can be found here: https://perfmon-events.intel.com/
Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 4: Model-specific Registers	Describes the model-specific registers of processors supporting IA-32 and Intel® 64 architectures.

(b) Intel 开发者手册列表【共四卷】

图 6.2: Intel 开发者官网

```
20          SAVE_REG_VAL,
21          SAVE_MEM_REF,
22          LOAD_MEM_ADDR
23  } rwmode_t;
24
25  // asm instructions
26  void x86_init();
27  void x86_mov(reg_t *reg, syment_t *var);
28  void x86_mov2(syment_t *var, reg_t *reg);
29  void x86_mov3(reg_t *reg, syment_t *arr, reg_t *idx);
30  void x86_mov4(syment_t *arr, reg_t *idx, reg_t *reg);
31  void x86_mov5(reg_t *r1, reg_t *r2);
32  void x86_mov6(reg_t *reg, int num);
33  void x86_mov7(reg_t *reg, char *strconst);
34  void x86_lea(reg_t *reg, syment_t *var);
35  void x86_lea2(reg_t *reg, syment_t *arr, reg_t *idx);
36  void x86_add(reg_t *r1, reg_t *r2);
37  void x86_sub(reg_t *r1, reg_t *r2);
38  void x86_mul(reg_t *r1, reg_t *r2);
39  reg_t *x86_div(reg_t *r1, reg_t *eax, reg_t *edx);
40  void x86_neg(reg_t *r1);
41  void x86_inc(reg_t *r1);
42  void x86_dec(reg_t *r1);
43  void x86_xor(reg_t *r1, reg_t *r2);
44  void x86_cls(reg_t *r1);
45  void x86_pop(reg_t *reg);
46  void x86_push(reg_t *reg);
47  void x86_push2(syment_t *var);
48  void x86_enter(syment_t *func);
49  void x86_leave(syment_t *func);
50  void x86_call(syment_t *func);
51  void x86_ret();
52  reg_t *x86_syscall(char *func, reg_t *eax);
53  void x86_label(syment_t *lab);
54  void x86_jmp(syment_t *lab);
55  void x86_cmp(reg_t *r1, reg_t *r2);
56  void x86_jz(syment_t *lab);
57  void x86_jnz(syment_t *lab);
58  void x86_jg(syment_t *lab);
59  void x86_jng(syment_t *lab);
60  void x86_jl(syment_t *lab);
61  void x86_jnl(syment_t *lab);
```

```
62 void x86_stralloc(char *name, char *initval);
```

6.3 库函数实现

6.3.1 库函数和链接过程

库函数是指将常用的功能实现封装起来，提供给用户直接调用的函数。在 C 语言中，标准库提供了许多常用的库函数，用于执行各种操作，如输入输出、字符串处理、数学运算等。

通常程序不能直接进行输入输出操作，输入输出操作的具体实现是通过系统调用来和操作系统进行交互来实现的。我们以 C 语言经典的”Hello World” 程序来介绍，其实现文件 hello.c 具体代码如下：

```
1 #include<stdio.h>
2
3 int main (int argc, char *argv[])
4 {
5         printf("Hello world\n");
6         return 0;
7 }
```

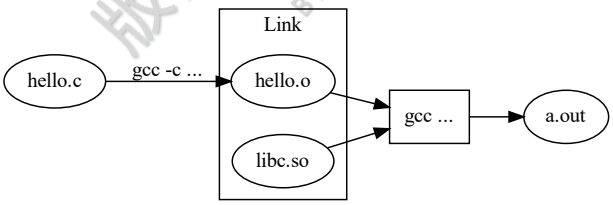



图 6.3: hello.c 链接 glibc 库函数示意图

如图 6.3 所示，在 hello.c 代码中的 printf() 函数调用的声明在头文件 stdio.h 中，gcc 编译器负责将生成的目标对象文件 hello.o 和系统库链接成最终可执行文件 a.out ，这样当程序运行到 printf() 函数时就会调用系统的库函数实现。另外，操作系统中都有动态链接库的实现，这个实现一般就是 glibc 库，可以通过 ldd 命令查看已经编译成功的二进制文件的链接库，比如下面示例中，ls 的链接库就包含 libc.so.6 库。

```
$ ldd /bin/ls
        linux-vdso.so.1 (0x00007fffa3cf3000)
        libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007f823d875000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f823d64d000)
        libpcre2-8.so.0 => /lib/x86_64-linux-gnu/libpcre2-8.so.0 (0x00007f823d5b6000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f823d8d7000)
```

```
$
```

6.3.2 I/O 库函数的实现

通常库函数包含功能较多，实现起来也比较复杂。$PL/0\epsilon$ 中对于输入输出操作只包含几个简单的指令，即 RDI、RDC、WRI、WRC 和 WRS 这五个读写指令。为了简化链接过程，我们这里不链接一个外置的库，而是直接使用汇编实现五个对应功能的函数，通过代码注入的方式来实现库函数功能，这样生成的汇编文件只需要调用实现的库函数即可。

6.3.3 输出库函数

输出函数通过 nasm 风格汇编实现，分别输出字符，字符串和整型。主要有以下几个库函数：

1. `libwchr` 功能是输出一个字符，`eax` 传入待输出的字符数值
2. `libwstr` 功能是输出一个字符串，`eax` 传入待输出的字符串的地址
3. `libwint` 功能是输出一个整型数字，`eax` 传入待输出的数值

`libwchr` 的实现文件为 `libwchr.s` ，其核心逻辑就是调用 Linux 系统调用 `int 0x80` 来输出一个长度为一的字符串，具体的代码如下：

```
1  section .text
2          global  _start
3
4  _libwchr:
5          push    ebp
6          mov     ebp, esp
7          push    esi
8          push    edi
9          push    ebx
10         mov     [_chrbuf], eax
11         mov     eax, 4
12         mov     ebx, 1
13         mov     ecx, _chrbuf
14         mov     edx, 1
15         int     0x80
16         pop     ebx
17         pop     edi
18         pop     esi
19         mov     esp, ebp
20         pop     ebp
21         ret
22
23 _start:
```

```
        mov     eax, 'a'
        call    _libwchr
        mov     eax, 1
        xor     ebx, ebx
        int     0x80

section .data
        _chrbuf db 'x', 0
```

libwstr 的实现文件为 libwstr.s ，其实现和 libwchr.s 类似，只不过需要计算待输出的字符串长度，具体的代码如下：

```
section .text
        global  _start

_libwstr:
        push    ebp
        mov     ebp, esp
        push    esi
        push    edi
        push    ebx
        mov     esi, eax
        xor     ecx, ecx
_nextchar@wstr:
        mov     cl, byte[esi]
        test    ecx, ecx
        jz      _syswrite@wstr
        inc     esi
        jmp     _nextchar@wstr
_syswrite@wstr:
        sub     esi, eax
        mov     ecx, eax
        mov     eax, 4
        mov     ebx, 1
        mov     edx, esi
        int     0x80
        pop     ebx
        pop     edi
        pop     esi
        mov     esp, ebp
        pop     ebp
        ret

```

```
32 _start:
33         lea     eax, msg
34         call    _libwstr
35         mov     eax, 1
36         xor     ebx, ebx
37         int     0x80
38
39 section .data
40         msg db 'hello', 0
```

libwint 的实现文件为 libwint.s ，libwint 实现比较复杂，其大致思路是先将 eax 中的数值转换成字符串，字符串结果存到 _intbuf 中，然后调用 Linux 系统调用来输出该字符串。具体的代码如下：

```
1  section .text
2          global  _start
3
4  _libwint:
5          push    ebp
6          mov     ebp, esp
7          push    esi
8          push    edi
9          push    ebx
10         xor     edi, edi
11         cmp     eax, 0
12         jnl     _noneneg@wint
13         inc     edi
14         neg     eax
15 _noneneg@wint:
16         mov     ebx, 10
17         xor     ecx, ecx
18         mov     esi, _intbuf+15
19 _loopdigit@wint:
20         xor     edx, edx
21         div     ebx
22         add     edx, '0'
23         mov     byte[esi], dl
24         dec     esi
25         inc     ecx
26         test    eax, eax
27         jnz     _loopdigit@wint
28         test    edi, edi
29         jnz     _negsign@wint
```

```
        inc     esi
        jmp     _syswrite@wint
_negsign@wint:
        mov     byte[esi], '-'
        inc     ecx
_syswrite@wint:
        mov     edx, ecx
        mov     eax, 4
        mov     ebx, 1
        mov     ecx, esi
        int     0x80
        pop     ebx
        pop     edi
        pop     esi
        mov     esp, ebp
        pop     ebp
        ret

_start:
        mov     eax, 123
        call    _libwint
        mov     eax, 1
        xor     ebx, ebx
        int     0x80

section .data
        _intbuf db '????????????????', 0
```


6.3.4 输入库函数

输入函数也是通过 nasm 汇编实现，主要有读取字符和读取整数的两个函数，具体实现函数如下：

1. librchr 功能是读取一个字符，eax 返回读取到的字符数值
2. librint 功能是读取一个整数，eax 返回读取到的整数

librchr 的实现文件为 librchr.s ，其核心逻辑就是调用 Linux 系统调用 int 0x80 来读取字符串，然后将第一个字符写入 eax 寄存器。具体的代码如下：

```
section .text
        global  _start

_librchr:
        push    ebp
```

```
        mov     ebp, esp
        push    esi
        push    edi
        push    ebx
_sysread@rchr:
        mov     eax, 3
        mov     ebx, 0
        mov     ecx, _scanbuf
        mov     edx, 1
        int     0x80
        xor     ecx, ecx
        mov     cl, [_scanbuf]
        cmp     cl, 10
        jz      _sysread@rchr
        mov     eax, ecx
        pop     ebx
        pop     edi
        pop     esi
        mov     esp, ebp
        pop     ebp
        ret

_start:
        call    _librchr
        mov     ebx, eax
        mov     eax, 1
        int     0x80

section .data
        _scanbuf db '????????????????', 0
```


librint 的实现文件为 librint.s ，其核心逻辑就是调用 Linux 系统调用 int 0x80 来读取字符串，将读取的字符串放入 _scanint 缓冲区中，然后解析整数后将第一个字符写入 eax 寄存器。具体的代码如下：

```
section .text
        global  _start

_librint:
        push    ebp
        mov     ebp, esp
        push    esi
        push    edi
```

```
9           push    ebx
10  _sysread@rint:
11          mov     eax, 3
12          mov     ebx, 0
13          mov     ecx, _scanint
14          mov     edx, 16
15          int     0x80
16  _init@rint:
17          xor     eax, eax
18          xor     ecx, ecx
19          mov     ebx, 1
20          mov     esi, _scanint
21  _begchar@rint:
22          mov     cl, byte[esi]
23          cmp     ecx, '-'
24          jz      _negnum@rint
25          cmp     ecx, '0'
26          jl      _skipchar@rint
27          cmp     ecx, '9'
28          jg      _skipchar@rint
29          jmp     _numchar@rint
30  _skipchar@rint:
31          inc     esi
32          jmp     _begchar@rint
33  _negnum@rint:
34          mov     ebx, -1
35          inc     esi
36  _numchar@rint:
37          mov     cl, byte[esi]
38          cmp     ecx, '0'
39          jl      _notdigit@rint
40          cmp     ecx, '9'
41          jg      _notdigit@rint
42          sub     ecx, '0'
43          imul    eax, 10
44          add     eax, ecx
45          inc     esi
46          jmp     _numchar@rint
47  _notdigit@rint:
48          imul    eax, ebx
49          pop     ebx
50          pop     edi
```

```
51          pop     esi
52          mov     esp, ebp
53          pop     ebp
54          ret
55
56  _start:
57          call    _librint
58          mov     ebx, eax
59          mov     eax, 1
60          int     0x80
61
62  section .data
63          _scanint db '????????????????', 0
```

6.4 函数调用

6.4.1 函数调用运行栈

在 x86 架构中，运行栈（Run-Time Stack）是一个重要的概念，用于支持函数调用和局部变量存储。运行栈的工作原理如下：

1. 局部变量存储：当一个函数被调用时，它的参数、局部变量等数据被推入运行栈中。这样，函数可以在栈上分配存储空间来保存这些数据。
2. 函数调用：函数调用时，返回地址被压入栈中，以便函数执行完毕后可以返回到调用者。同时，函数的局部变量和参数也被推入栈中。
3. 异常处理：运行栈也用于异常处理。当异常发生时，异常处理程序的相关信息被压入栈中，以便处理异常。
4. 动态内存分配：在某些情况下，运行栈还用于动态内存分配。例如，C 语言的 `malloc` 函数会从运行栈上分配内存。

在 x86 架构中，运行栈通常以向下增长的方式工作，即新的数据被推入栈的顶部。这意味着栈的顶部是最后一个入栈的数据项。这种设计简化了内存管理，因为栈的顶部始终是可用的最大内存区域。

当发生函数调用时，pcc 编译器会为函数中的实参分配运行栈结构，图 6.4 是 pcc 编译的函数调用运行栈的布局图，根据 x86 的函数调用规范，运行栈的布局大致可以分成以下几个区域：

1. argmuments 参数区：`arg(1)` 到 `arg(N)` 部分。
2. access link 区：`ebp(2)` 到 `ebp(N)` 部分。
   - 这里不需要记录当前的 ebp，所以从 `ebp(2)` 开始编号。
3. x86 的 call 指令函数调用时的参数：
   - 返回地址：`retaddr` 。
   - 上一帧 ebp 值：`prevebp` 。

- 函数调用返回值：retval 。

4. variable 变量区：var(1) 到 var(N) 部分。
5. temporary variable 临时变量区：tmp(1) 到 tmp(N) 部分。

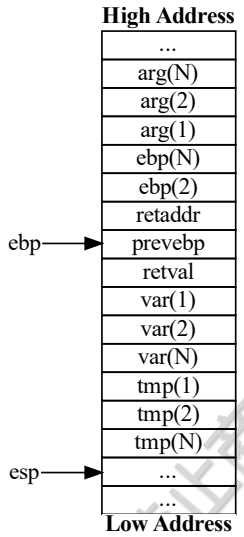


图 6.4：pcc 编译的函数调用运行栈的布局图

6.4.2 函数调用帧

函数调用帧（Function Call Frame）是计算机程序中用于保存函数调用状态的一种数据结构。在函数调用过程中，程序需要保存当前的执行环境，以便在函数返回时能够恢复执行。函数调用帧就是用于保存这些执行环境信息的结构。具体的函数调用帧实现方式可能因编程语言、编译器和硬件平台的不同而有所差异。在 pcc 编译器中，实现函数调用帧如图 6.4 所示。

为了更加直观的理解函数调用帧的逻辑，我们这边编写了一个文件 frame.pas 来说明函数调用帧的状态，具体代码如下：

```
1  var u, v, ans: integer;
2  function p1(x, y : integer): integer;
3    function p2(x, y : integer): integer;
4      begin p2 := x + y end;
5    begin
6       p1 := p2(x, y)
7    end;
8  begin
9     ans := p1(u, v)
10 end.
```

图 6.5 表示 frame.pas 进入 p2 函数时的调用帧布局，此时 ebp 是指向 p2 函数的运行栈，

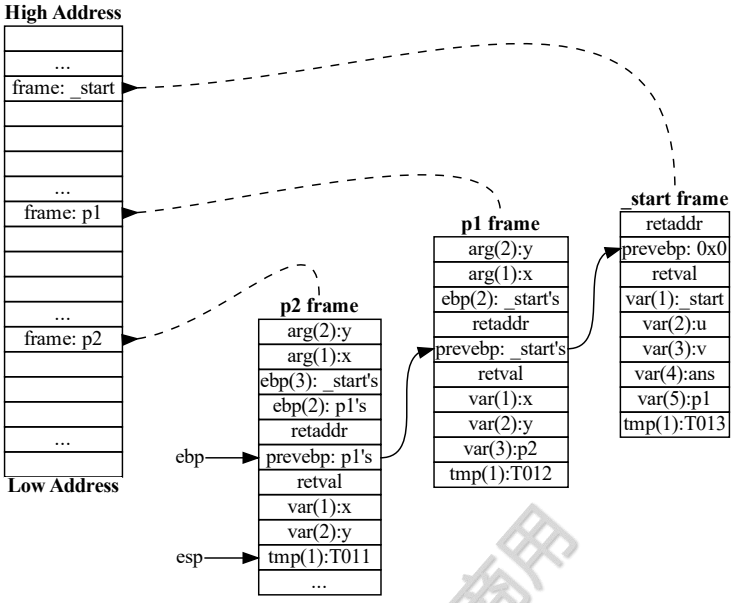


图 6.5: frame.pas 进入 p2 函数时的调用帧布局图

除了 p2 的运行栈以外，还有 p1 和 _start 两个运行栈，每个运行栈就是一个函数调用帧。图的右侧将每个函数调用帧的内部存储细节都标记出来。

1. _start 是主函数，它不包含入参，
   - 它的 access link 区没有数据。
   - 它有 _start u v 等全局变量定义，这些变量保存在 var(1) 到 var(5) 中。
   - 它有一个临时变量的定义：保存在 tmp(1) 中。
2. p1 是调用函数是 _start ，
   - 它的 access link 区保存着 _start 的 ebp 指针，即 ebp(2) 指针。
   - p1 调用还有入参 x 和 y ，分别保存在 arg(1) 和 arg(2) 中。
   - 变量和临时变量保存在对应的区域。
3. p2 是调用函数是 p1 ，
   - 它的 access link 区保存着 p1 和 _start 的指针，分别保存在 ebp(2) 和 ebp(3) 。
   - 它的调用入参 x 和 y 分别保存在 arg(1) 和 arg(2) 中。
   - 变量和临时变量保存在对应的区域。

图中其他的的具体变量和临时变量的分配需要结合符号表进行说明，下面使用 prtir 将代码的符号表打印出来，结果如下：

```
DUMP SYMBOLS:
label=L001 type=1 cate=FUN name=_start off=1 stab=1 depth=1 initval=0 arrlen=0 str=
label=L002 type=1 cate=VAR name=u off=2 stab=1 depth=1 initval=0 arrlen=0 str=
label=L003 type=1 cate=VAR name=v off=3 stab=1 depth=1 initval=0 arrlen=0 str=
label=L004 type=1 cate=VAR name=ans off=4 stab=1 depth=1 initval=0 arrlen=0 str=
label=L005 type=1 cate=FUN name=p1 off=5 stab=1 depth=1 initval=0 arrlen=0 str=
```

```
label=L006 type=1 cate=BYVAL name=x off=0 stab=2 depth=2 initval=0 arrlen=0 str=
label=L007 type=1 cate=BYVAL name=y off=1 stab=2 depth=2 initval=0 arrlen=0 str=
label=L008 type=1 cate=FUN name=p2 off=1 stab=2 depth=2 initval=0 arrlen=0 str=
label=L009 type=1 cate=BYVAL name=x off=0 stab=3 depth=3 initval=0 arrlen=0 str=
label=L010 type=1 cate=BYVAL name=y off=1 stab=3 depth=3 initval=0 arrlen=0 str=
label=T011 type=1 cate=TMP name=@expr/add off=1 stab=3 depth=3 initval=0 arrlen=0 str=
label=T012 type=1 cate=TMP name=@fcall/ret off=2 stab=2 depth=2 initval=0 arrlen=0 str=
label=T013 type=1 cate=TMP name=@fcall/ret off=6 stab=1 depth=1 initval=0 arrlen=0 str=

DUMP INTERMEDIATE REPRESENTATION:
#001: ENT       d=L008
#002: ADD       d=T011  r=L009  s=L010
#003: ASS       d=L008  r=T011
#004: FIN       d=L008
#005: ENT       d=L005
#006: PUSH      d=L007
#007: PUSH      d=L006
#008: CALL      d=T012  r=L008
#009: POP
#010: POP
#011: ASS       d=L005  r=T012
#012: FIN       d=L005
#013: ENT       d=L001
#014: PUSH      d=L003
#015: PUSH      d=L002
#016: CALL      d=T013  r=L005
#017: POP
#018: POP
#019: ASS       d=L004  r=T013
#020: FIN       d=L001
```


6.4.3 传值和传引用

之前的章节已经说明了传值和传引用的区别，在目标代码生成时也需要考虑这两种参数传递方式的不同所带来的影响，下面通过 `byval.pas` 和 `byref.pas` 这两个具体代码实例来说明，下面是具体代码。

`byval.pas` 的代码和对应的四元式序列如下：

```
1  var u, v, ans: integer;
2
3  function adder(x, y : integer):integer;
4  begin
```

```
5     adder := x + y
6  end;
7
8  begin
9     u := 1;
10    v := 2;
11    ans := adder(u, v)
12 end.
```

```
DUMP INTERMEDIATE REPRESENTATION:
#001: ENT       d=L005
#002: ADD       d=T008  r=L006  s=L007
#003: ASS       d=L005  r=T008
#004: FIN       d=L005
#005: ENT       d=L001
#006: ASS       d=L002  r=T009
#007: ASS       d=L003  r=T010
#008: PUSH      d=L003
#009: PUSH      d=L002
#010: CALL      d=T011  r=L005
#011: POP
#012: POP
#013: ASS       d=L004  r=T011
#014: FIN       d=L001
```


byref.pas 的代码和对应的四元式序列如下：

```
1  var u, v, ans: integer;
2
3  function adder(var x, y : integer):integer;
4  begin
5     adder := x + y
6  end;
7
8  begin
9     u := 1;
10    v := 2;
11    ans := adder(u, v)
12 end.
```

```
DUMP INTERMEDIATE REPRESENTATION:
#001: ENT       d=L005
#002: ADD       d=T008  r=L006  s=L007
#003: ASS       d=L005  r=T008
```

```
#004: FIN       d=L005
#005: ENT       d=L001
#006: ASS       d=L002  r=T009
#007: ASS       d=L003  r=T010
#008: PADR      d=L003
#009: PADR      d=L002
#010: CALL      d=T011  r=L005
#011: POP
#012: POP
#013: ASS       d=L004  r=T011
#014: FIN       d=L001
```

通过对比可以发现，在调用函数 CALL 指令前需要将参数压入栈传递。传值是通过调用 PUSH 指令入栈，传引用是通过调用 PADR 指令入栈，这样就可以根据语义分析的语法树来的生成对应的传值或传引用的代码。

6.5 目标代码生成

6.5.1 汇编代码生成

汇编代码生成的入口函数 genasm() 声明位于文件 include/asm.h 中，该文件还包括一些之前汇编语言实现的如果函数名称，其具体代码如下：

```
#define LIBEXIT "_libexit"
#define LIBWCHR "_libwchr"
#define LIBWSTR "_libwstr"
#define LIBWINT "_libwint"
#define LIBRCHR "_librchr"
#define LIBRINT "_librint"

void genasm();
```

对于具体代码实现在 source/asm.c 中，这个部分代码功能是实现将四元式生成汇编代码的功能，主要有以下几个注意点：

1. 代码的入口函数是 genasm() ，它是通过一个 switch 语句将实际生成函数路由到对应实现的子函数。
2. 代码实现了一些通用存取功能的函数，具体如下：
   - loadvar(reg_t *reg, syment_t *var) 将变量 var 的值读取到寄存器 reg 中。
   - savevar(syment_t *var, reg_t *reg) 将寄存器 reg 写入到变量 var 中。
   - loadptr(reg_t *reg, syment_t *var) 将变量 var 的地址读取到寄存器 reg 中。
   - loadptr2(reg_t *reg, syment_t *arr, reg_t *off) 将数组 arr[off] 的地址读取到寄存器 reg 中。

- loadarr(reg_t *reg, syment_t *arr, reg_t *off) 将数组 arr[off] 的值读取到寄存器 reg 中。
- savearr(syment_t *arr, reg_t *off, reg_t *reg) 将寄存器 reg 写入到数组 arr[off] 中。

3. 以 asmbl_xxx 开头的函数都是生成对应四元式的汇编函数，实现可以通过具体代码查看。

文件 source/asm.c 中的代码如下：

```
1  static void loadvar(reg_t *reg, syment_t *var)
2  {
3          switch (var->cate) {
4          case CONST_OBJ:
5          case NUM_OBJ:
6                  x86_mov6(reg, var->initval);
7                  break;
8          case VAR_OBJ:
9          case TMP_OBJ:
10         case BYVAL_OBJ:
11         case BYREF_OBJ:
12         case FUN_OBJ:
13                 x86_mov(reg, var);
14                 break;
15         default:
16                 unlikely();
17         }
18 }
19
20 static void savevar(syment_t *var, reg_t *reg)
21 {
22         switch (var->cate) {
23         case VAR_OBJ:
24         case TMP_OBJ:
25         case BYVAL_OBJ:
26         case BYREF_OBJ:
27         case FUN_OBJ:
28                 x86_mov2(var, reg);
29                 break;
30         default:
31                 unlikely();
32         }
33 }
34
35 static void loadptr(reg_t *reg, syment_t *var)
```

```
36  {
37          switch (var->cate) {
38          case VAR_OBJ:
39          case TMP_OBJ:
40                  x86_lea(reg, var);
41                  break;
42          default:
43                  unlikely();
44          }
45  }
46
47  static void loadptr2(reg_t *reg, syment_t *arr, reg_t *off)
48  {
49          switch (arr->cate) {
50          case ARRAY_OBJ:
51                  x86_lea2(reg, arr, off);
52                  break;
53          default:
54                  unlikely();
55          }
56  }
57
58  static void loadarr(reg_t *reg, syment_t *arr, reg_t *off)
59  {
60          switch (arr->cate) {
61          case ARRAY_OBJ:
62                  x86_mov3(reg, arr, off);
63                  break;
64          default:
65                  unlikely();
66          }
67  }
68
69  static void savearr(syment_t *arr, reg_t *off, reg_t *reg)
70  {
71          switch (arr->cate) {
72          case ARRAY_OBJ:
73                  x86_mov4(arr, off, reg);
74                  break;
75          default:
76                  unlikely();
77          }
```

```
78  }
79
80  void asmbl_add_op(inst_t *x)
81  {
82          reg_t *r1 = allocreg();
83          reg_t *r2 = allocreg();
84
85          loadvar(r1, x->r);
86          loadvar(r2, x->s);
87          x86_add(r1, r2);
88          savevar(x->d, r1);
89
90          freereg(r1);
91          freereg(r2);
92  }
93
94  void asmbl_sub_op(inst_t *x)
95  {
96          reg_t *r1 = allocreg();
97          reg_t *r2 = allocreg();
98
99          loadvar(r1, x->r);
100         loadvar(r2, x->s);
101         x86_sub(r1, r2);
102         savevar(x->d, r1);
103
104         freereg(r1);
105         freereg(r2);
106 }
107
108 void asmbl_mul_op(inst_t *x)
109 {
110         reg_t *r1 = allocreg();
111         reg_t *r2 = allocreg();
112
113         loadvar(r1, x->r);
114         loadvar(r2, x->s);
115         x86_mul(r1, r2);
116         savevar(x->d, r1);
117
118         freereg(r1);
119         freereg(r2);
```

```
120 }
121
122 void asmbl_div_op(inst_t *x)
123 {
124         // idiv use specific registers
125         reg_t *ra = lockreg(REG_RA);
126         reg_t *rd = lockreg(REG_RD);
127         reg_t *r = allocreg();
128
129         loadvar(ra, x->r);
130         loadvar(r, x->s);
131         ra = x86_div(r, ra, rd);
132         savevar(x->d, ra);
133
134         freereg(ra);
135         freereg(rd);
136         freereg(r);
137 }
138
139 void asmbl_inc_op(inst_t *x)
140 {
141         reg_t *r = allocreg();
142
143         loadvar(r, x->d);
144         x86_inc(r);
145         savevar(x->d, r);
146
147         freereg(r);
148 }
149
150 void asmbl_dec_op(inst_t *x)
151 {
152         reg_t *r = allocreg();
153
154         loadvar(r, x->d);
155         x86_dec(r);
156         savevar(x->d, r);
157
158         freereg(r);
159 }
160
161 void asmbl_neg_op(inst_t *x)
```

```
{
        reg_t *r = allocreg();

        loadvar(r, x->r);
        x86_neg(r);
        savevar(x->d, r);

        freereg(r);
}

void asmbl_load_op(inst_t *x)
{
        reg_t *r1 = allocreg();

        if (x->s) { // load array
                reg_t *offreg = allocreg();
                loadvar(offreg, x->s);
                loadarr(r1, x->r, offreg);
                freereg(offreg);
        } else {
                loadptr(r1, x->r);
        }

        savevar(x->d, r1);
        freereg(r1);
}

void asmbl_ass_op(inst_t *x)
{
        reg_t *r1 = allocreg();

        loadvar(r1, x->r);
        savevar(x->d, r1);

        freereg(r1);
}

void asmbl_asa_op(inst_t *x)
{
        reg_t *r1 = allocreg();
        reg_t *r2 = allocreg();

```

```
204         loadvar(r1, x->s); // r1 = offset
205         loadvar(r2, x->r);
206         savearr(x->d, r1, r2);
207
208         freereg(r1);
209         freereg(r2);
210 }
211
212 void asmbl_equ_op(inst_t *x)
213 {
214         reg_t *r1 = allocreg();
215         reg_t *r2 = allocreg();
216
217         loadvar(r1, x->r);
218         loadvar(r2, x->s);
219         x86_cmp(r1, r2);
220         x86_jz(x->d);
221
222         freereg(r1);
223         freereg(r2);
224 }
225
226 void asmbl_neq_op(inst_t *x)
227 {
228         reg_t *r1 = allocreg();
229         reg_t *r2 = allocreg();
230
231         loadvar(r1, x->r);
232         loadvar(r2, x->s);
233         x86_cmp(r1, r2);
234         x86_jnz(x->d);
235
236         freereg(r1);
237         freereg(r2);
238 }
239
240 void asmbl_gtt_op(inst_t *x)
241 {
242         reg_t *r1 = allocreg();
243         reg_t *r2 = allocreg();
244
245         loadvar(r1, x->r);
```

```
246         loadvar(r2, x->s);
247         x86_cmp(r1, r2);
248         x86_jg(x->d);
249 
250         freereg(r1);
251         freereg(r2);
252 }
253 
254 void asmbl_geq_op(inst_t *x)
255 {
256         reg_t *r1 = allocreg();
257         reg_t *r2 = allocreg();
258 
259         loadvar(r1, x->r);
260         loadvar(r2, x->s);
261         x86_cmp(r1, r2);
262         x86_jnl(x->d);
263 
264         freereg(r1);
265         freereg(r2);
266 }
267 
268 void asmbl_lst_op(inst_t *x)
269 {
270         reg_t *r1 = allocreg();
271         reg_t *r2 = allocreg();
272 
273         loadvar(r1, x->r);
274         loadvar(r2, x->s);
275         x86_cmp(r1, r2);
276         x86_jl(x->d);
277 
278         freereg(r1);
279         freereg(r2);
280 }
281 
282 void asmbl_leq_op(inst_t *x)
283 {
284         reg_t *r1 = allocreg();
285         reg_t *r2 = allocreg();
286 
287         loadvar(r1, x->r);
```

```
288         loadvar(r2, x->s);
289         x86_cmp(r1, r2);
290         x86_jng(x->d);
291
292         freereg(r1);
293         freereg(r2);
294 }
295
296 void asmbl_jmp_op(inst_t *x)
297 {
298         x86_jmp(x->d);
299 }
300
301 void asmbl_push_op(inst_t *x)
302 {
303         reg_t *r = allocreg();
304         loadvar(r, x->d);
305         x86_push(r);
306         freereg(r);
307 }
308
309 void asmbl_padr_op(inst_t *x)
310 {
311         reg_t *r1 = allocreg();
312         if (x->r) {
313                 reg_t *offreg = allocreg();
314                 loadvar(offreg, x->r);
315                 loadptr2(r1, x->d, offreg);
316                 freereg(offreg);
317         } else {
318                 loadptr(r1, x->d);
319         }
320         x86_push(r1);
321         freereg(r1);
322 }
323
324 void asmbl_pop_op(inst_t *x)
325 {
326         reg_t *r = allocreg();
327         x86_pop(r);
328         if (x->d) {
329                 savevar(x->d, r);
```

```
330         }
331         freereg(r);
332 }
333
334 void asmbl_call_op(inst_t *x)
335 {
336         x86_call(x->r);
337         if (x->d) {
338                 reg_t *r = allocreg();
339                 loadvar(r, x->r);
340                 savevar(x->d, r);
341                 freereg(r);
342         }
343 }
344
345 void asmbl_ent_op(inst_t *x)
346 {
347         x86_enter(x->d);
348 }
349
350 void asmbl_fin_op(inst_t *x)
351 {
352         x86_leave(x->d);
353 }
354
355 void asmbl_rdi_op(inst_t *x)
356 {
357         reg_t *ra = lockreg(REG_RA);
358         x86_syscall(LIBRINT, ra);
359         savevar(x->d, ra);
360         freereg(ra);
361 }
362
363 void asmbl_rdc_op(inst_t *x)
364 {
365         reg_t *ra = lockreg(REG_RA);
366         x86_syscall(LIBRCHR, ra);
367         savevar(x->d, ra);
368         freereg(ra);
369 }
370
371 void asmbl_wrs_op(inst_t *x)
```

```
372 {
373         x86_stralloc(x->d->label, x->d->str);
374         reg_t *ra = lockreg(REG_RA);
375         x86_mov7(ra, x->d->label);
376         x86_syscall(LIBWSTR, ra);
377         freereg(ra);
378 }
379
380 void asmbl_wri_op(inst_t *x)
381 {
382         reg_t *ra = lockreg(REG_RA);
383         loadvar(ra, x->d);
384         x86_syscall(LIBWINT, ra);
385         freereg(ra);
386 }
387
388 void asmbl_wrc_op(inst_t *x)
389 {
390         reg_t *ra = lockreg(REG_RA);
391         x86_cls(ra);
392         loadvar(ra, x->d);
393         x86_syscall(LIBWCHR, ra);
394         freereg(ra);
395 }
396
397 void asmbl_lab_op(inst_t *x)
398 {
399         x86_label(x->d);
400 }
401
402 void genasm()
403 {
404         x86_init();
405         inst_t *x;
406         for (x = xhead; x; x = x->next) {
407                 dbg("xid=%d, op=%d\n", x->xid, x->op);
408                 switch (x->op) {
409                 case ADD_OP:
410                         asmbl_add_op(x);
411                         break;
412                 case SUB_OP:
413                         asmbl_sub_op(x);
```

```
414                         break;
415                 case MUL_OP:
416                         asmbl_mul_op(x);
417                         break;
418                 case DIV_OP:
419                         asmbl_div_op(x);
420                         break;
421                 case INC_OP:
422                         asmbl_inc_op(x);
423                         break;
424                 case DEC_OP:
425                         asmbl_dec_op(x);
426                         break;
427                 case NEG_OP:
428                         asmbl_neg_op(x);
429                         break;
430                 case LOAD_OP:
431                         asmbl_load_op(x);
432                         break;
433                 case ASS_OP:
434                         asmbl_ass_op(x);
435                         break;
436                 case ASA_OP:
437                         asmbl_asa_op(x);
438                         break;
439                 case EQU_OP:
440                         asmbl_equ_op(x);
441                         break;
442                 case NEQ_OP:
443                         asmbl_neq_op(x);
444                         break;
445                 case GTT_OP:
446                         asmbl_gtt_op(x);
447                         break;
448                 case GEQ_OP:
449                         asmbl_geq_op(x);
450                         break;
451                 case LST_OP:
452                         asmbl_lst_op(x);
453                         break;
454                 case LEQ_OP:
455                         asmbl_leq_op(x);
```

```
456                 break;
457             case JMP_OP:
458                 asmbl_jmp_op(x);
459                 break;
460             case PUSH_OP:
461                 asmbl_push_op(x);
462                 break;
463             case PADR_OP:
464                 asmbl_padr_op(x);
465                 break;
466             case POP_OP:
467                 asmbl_pop_op(x);
468                 break;
469             case CALL_OP:
470                 asmbl_call_op(x);
471                 break;
472             case ENT_OP:
473                 asmbl_ent_op(x);
474                 break;
475             case FIN_OP:
476                 asmbl_fin_op(x);
477                 break;
478             case RDI_OP:
479                 asmbl_rdi_op(x);
480                 break;
481             case RDC_OP:
482                 asmbl_rdc_op(x);
483                 break;
484             case WRS_OP:
485                 asmbl_wrs_op(x);
486                 break;
487             case WRI_OP:
488                 asmbl_wri_op(x);
489                 break;
490             case WRC_OP:
491                 asmbl_wrc_op(x);
492                 break;
493             case LAB_OP:
494                 asmbl_lab_op(x);
495                 break;
496             default:
497                 unlikely();
```

```
498                 }
499         }
500
501         chkerr("assemble fail and exit.");
502         phase = ASSEMBLE;
503
504         writeasm();
505 }
```

6.5.2 X86 代码生成

X86 汇编代码生成实现位于文件 source/x86.c ，这部分逻辑包含以下几种功能：

1. 寄存器管理：
   - reg_t *allocreg() 函数申请使用寄存器
   - reg_t *lockreg(char *name) 函数申请锁定一个特定名字 name 的寄存器
   - void freereg(reg_t *r) 释放一个寄存器
2. 汇编代码生成管理：
   - addlabel(...) 函数用于添加一个标号。
   - adddata2(...) 函数用于添加一个数据定义，其中 name 为数据名称，initval 为初始值。
   - addcode1(...) 到 addcode4(...) 用于添加一个 X86 指令。
3. 汇编文件生成：
   - writeasm() 用于写入 nasm 汇编文件。
   - 调用 savetext() 函数写入 section .data 数据段的汇编指令。
   - 调用 savedata() 函数写入 section .text 代码段的汇编指令。

这些代码实现如下：

```
1  // x86 general purpose registers
2  reg_t X86_GP_REGS[4] = {
3          [0] = { REG_RA, 0 },
4          [1] = { REG_RC, 0 },
5          [2] = { REG_RD, 0 },
6          [3] = { REG_RB, 0 },
7  };
8
9  #define MAXREGS (sizeof(X86_GP_REGS) / sizeof(reg_t))
10
11 // Alloc a register
12 reg_t *allocreg()
13 {
14         int i;
15         for (i = 0; i < MAXREGS; ++i) {
```

```
16                  reg_t *r = &X86_GP_REGS[i];
17                  if (r->refcnt == 0) {
18                          r->refcnt++;
19                          return r;
20                  }
21          }
22
23          panic("NO_REGISTER_LEFT");
24          return 0;
25  }
26
27  // Lock specific register
28  reg_t *lockreg(char *name)
29  {
30          int i;
31          for (i = 0; i < MAXREGS; ++i) {
32                  reg_t *r = &X86_GP_REGS[i];
33                  if (!strcmp(r->name, name) && r->refcnt == 0) {
34                          r->refcnt++;
35                          return r;
36                  }
37          }
38
39          panic("NO_REGISTER_LEFT");
40          return 0;
41  }
42
43  // Free a register
44  void freereg(reg_t *r)
45  {
46          r->refcnt--;
47  }
48
49  // hold x86 program code and data
50  progcode_t prog;
51
52  // send label
53  void addlabel(char *label)
54  {
55          if (prog.itext >= MAXTEXTSEC) {
56                  panic("TEXT_SECTION_TOO_BIG");
57          }
```

```
58          dbg("ipos=%d, label=%s\n", prog.itext, label);
59
60          x86i_t *i = &prog.text[prog.itext++];
61          i->islab = TRUE;
62          strcopy(i->op, label);
63  }
64
65  // send data
66  void adddata2(char *name, char *initval)
67  {
68          if (prog.idata >= MAXDATASEC) {
69                  panic("DATA_SECTION_TOO_BIG");
70          }
71          dbg("dpos=%d, name=%s\n", prog.idata, name);
72
73          x86i_t *d = &prog.data[prog.idata++];
74          d->islab = FALSE;
75          strcopy(d->op, name);
76          strcopy(d->fa, initval);
77  }
78
79  // send text/code
80  void addcode4(char *op, char *fa, char *fb, char *extra)
81  {
82          if (prog.itext >= MAXTEXTSEC) {
83                  panic("TEXT_SECTION_TOO_BIG");
84          }
85          dbg("ipos=%d, op=%s\n", prog.itext, op);
86
87          x86i_t *i = &prog.text[prog.itext++];
88          i->islab = FALSE;
89          strcopy(i->op, op);
90          strcopy(i->fa, fa);
91          strcopy(i->fb, fb);
92          strcopy(i->et, extra);
93  }
94
95  void addcode3(char *op, char *fa, char *fb)
96  {
97          addcode4(op, fa, fb, "");
98  }
99
```

```
100 void addcode2(char *op, char *fa)
101 {
102         addcode4(op, fa, "", "");
103 }
104
105 void addcode1(char *op)
106 {
107         addcode4(op, "", "", "");
108 }
109
110 void savetext(x86i_t *i)
111 {
112         // label
113         if (i->islab) {
114                 fprintf(asmble, "%s:\n", i->op);
115                 return;
116         }
117
118         // text
119         fprintf(asmble, "\t");
120         if (strlen(i->op)) {
121                 fprintf(asmble, "%s", i->op);
122         } else {
123                 unlikely();
124         }
125
126         if (strlen(i->fb)) {
127                 fprintf(asmble, "\t%s, %s", i->fa, i->fb);
128         } else if (strlen(i->fa)) {
129                 fprintf(asmble, "\t%s", i->fa);
130         }
131
132         if (strlen(i->et)) {
133                 fprintf(asmble, " ; %s", i->et);
134         }
135         fprintf(asmble, "\n");
136 }
137
138 void savedata(x86i_t *d)
139 {
140         fprintf(asmble, "\t%s db '%s', 0\n", d->op, d->fa);
141 }
```

```
142
143 void writeasm()
144 {
145         // open target nasm file
146         asmble = fopen(PLOE_ASSEM, "w");
147         if (!asmble) {
148                 panic("target file not found!");
149         }
150         msg("compile assemble file %s\n", PLOE_ASSEM);
151
152         // write content
153         fprintf(asmble, "section .text\n");
154         int k;
155         for (k = 0; k < prog.itext; ++k) {
156                 savetext(&prog.text[k]);
157         }
158
159         if (!prog.idata) {
160                 goto dofree;
161         }
162
163         fprintf(asmble, "section .data\n");
164         for (k = 0; k < prog.idata; ++k) {
165                 savedata(&prog.data[k]);
166         }
167
168 dofree:
169         fclose(asmble);
170 }
171
172 // hold current scope, especially for get current function depth
173 symtab_t *scope = NULL;
174
175 // i386 instructions
176 static char addrbuf[MAXSTRBUF];
177
178 // conv offset to base pointer
179 static char *ptr(char *reg, int offset)
180 {
181         if (offset > 0) {
182                 sprintf(addrbuf, "[%s+%d]", reg, offset * ALIGN);
183         } else if (offset < 0) {
```

```
184                 sprintf(addrbuf, "[%s-%d]", reg, -offset * ALIGN);
185         } else {
186                 sprintf(addrbuf, "[%s]", reg);
187         }
188         return addrbuf;
189 }
190
191 // read/write memory from register, depends on mode
192 static void rwmem(rwmode_t mode, reg_t *reg, syment_t *var, reg_t *idx)
193 {
194         char *mem;
195         int off, gap;
196
197         char extra[MAXSTRBUF];
198         sprintf(extra, "%s %s", var->label, var->name);
199
200         dbg("current scope= %p, depth=%d\n", scope, scope->depth);
201         if (!scope) {
202                 panic("CURR_SCOPE_IS_NULL");
203         }
204
205         symtab_t *tab = var->stab;
206         switch (var->cate) {
207         case BYVAL_OBJ:
208         case BYREF_OBJ:
209                 off = 1 + scope->depth + var->off;
210                 mem = REG_BP;
211                 goto doit;
212         case TMP_OBJ:
213                 off = -var->off;
214                 mem = REG_BP;
215                 goto doit;
216         case VAR_OBJ:
217         case ARRAY_OBJ:
218         case FUN_OBJ:
219         case PROC_OBJ:
220                 off = -var->off;
221                 goto findaddr;
222         default:
223                 unlikely();
224         }
225
```

```
findaddr:
        gap = scope->depth - tab->depth;
        if (gap == 0) {
                mem = REG_BP;
        } else if (gap > 0) {
                addcode3("mov", REG_SI, ptr(REG_BP, gap + 1));
                mem = REG_SI;
        } else {
                unlikely();
        }

        if (var->cate == ARRAY_OBJ) {
                nevernil(idx);
                if (!strcmp(mem, REG_BP)) {
                        addcode3("mov", REG_SI, mem);
                        mem = REG_SI;
                }
                addcode3("imul", idx->name, itoa(ALIGN));
                addcode3("sub", mem, idx->name);
        }

doit:
        switch (mode) {
        case READ_MEM_VAL:
                addcode4("mov", reg->name, ptr(mem, off), extra);
                break;
        case READ_MEM_REF:
                addcode4("mov", REG_DI, ptr(mem, off), extra);
                addcode4("mov", reg->name, ptr(REG_DI, 0), extra);
                break;
        case SAVE_REG_VAL:
                addcode4("mov", ptr(mem, off), reg->name, extra);
                break;
        case SAVE_MEM_REF:
                addcode4("mov", REG_DI, ptr(mem, off), extra);
                addcode4("mov", ptr(REG_DI, 0), reg->name, extra);
                break;
        case LOAD_MEM_ADDR:
                addcode4("lea", reg->name, ptr(mem, off), extra);
                break;
        default:
                unlikely();
```

```
268         }
269 }
270
271 // duplicate current ebp, construct access link area
272 static void dupebp(syment_t *func)
273 {
274         int caller = scope->depth; // caller depth
275         int callee = func->stab->depth; // callee depth
276         dbg("%s=%d %s=%d\n", scope->nspace, caller, func->name, callee);
277
278         int off, i;
279
280         // prepare total `callee-1' saved ebps for access link
281         for (i = 0; i < callee; i++) {
282                 off = caller - i;
283
284                 // if off == 1, reach return value boundary,
285                 // we take current ebp as the last access link, and break
286                 if (off == 1) {
287                         addcode4("mov", REG_DI, REG_BP, "dup fresh ebp");
288                         addcode2("push", REG_DI);
289                         break;
290                 }
291
292                 addcode4("mov", REG_DI, ptr(REG_BP, off), "dup old ebp");
293                 addcode2("push", REG_DI);
294         }
295 }
296
297 void x86_lib_enter()
298 {
299         addcode2("push", REG_BP);
300         addcode3("mov", REG_BP, REG_SP);
301         addcode2("push", REG_SI);
302         addcode2("push", REG_DI);
303         addcode2("push", REG_RB);
304 }
305
306 void x86_lib_leave()
307 {
308         addcode2("pop", REG_RB);
309         addcode2("pop", REG_DI);
```

```
310         addcode2("pop", REG_SI);
311         addcode3("mov", REG_SP, REG_BP);
312         addcode2("pop", REG_BP);
313         addcode1("ret");
314 }
315
316 void x86_syslib_exit()
317 {
318         addlabel(LIBEXIT);
319         addcode3("mov", REG_RA, "1"); // syscall number
320         addcode3("xor", REG_RB, REG_RB); // return value
321         addcode2("int", SYSCAL);
322 }
323
324 void x86_iolib_wrtchr()
325 {
326         adddata2("_chrbuf", "?");
327
328         addlabel(LIBWCHR);
329         x86_lib_enter();
330
331         addcode3("mov", "[_chrbuf]", REG_RA);
332         addcode3("mov", REG_RA, "4");
333         addcode3("mov", REG_RB, "1");
334         addcode3("mov", REG_RC, "_chrbuf");
335         addcode3("mov", REG_RD, "1");
336         addcode2("int", SYSCAL);
337
338         x86_lib_leave();
339 }
340
341 void x86_iolib_wrtstr()
342 {
343         addlabel(LIBWSTR);
344         x86_lib_enter();
345
346         addcode3("mov", REG_SI, REG_RA);
347         addcode3("xor", REG_RC, REG_RC);
348         addlabel("_nextchar@wstr");
349         addcode3("mov", REG_CL, BTP_SI);
350         addcode3("test", REG_RC, REG_RC);
351         addcode2("jz", "_syswrite@wstr");
```

```
352         addcode2("inc", REG_SI);
353         addcode2("jmp", "_nextchar@wstr");
354         addlabel("_syswrite@wstr");
355         addcode3("sub", REG_SI, REG_RA); // string length
356         addcode3("mov", REG_RC, REG_RA);
357         addcode3("mov", REG_RA, "4");
358         addcode3("mov", REG_RB, "1");
359         addcode3("mov", REG_RD, REG_SI);
360         addcode2("int", SYSCAL);
361
362         x86_lib_leave();
363 }
364
365 void x86_iolib_wrtint()
366 {
367         adddata2("_intbuf", "??????????????");
368
369         addlabel(LIBWINT);
370         x86_lib_enter();
371
372         addcode3("xor", REG_DI, REG_DI); // negtive flag
373         addcode3("cmp", REG_RA, "0");
374         addcode2("jnl", "_noneneg@wint");
375         addcode2("inc", REG_DI);
376         addcode2("neg", REG_RA);
377         addlabel("_noneneg@wint");
378         addcode3("mov", REG_RB, "10"); // number base
379         addcode3("xor", REG_RC, REG_RC); // number string length
380         addcode3("mov", REG_SI, "_intbuf+15"); // number string pointer
381         addlabel("_loopdigit@wint");
382         addcode3("xor", REG_RD, REG_RD);
383         addcode2("div", REG_RB);
384         addcode3("add", REG_RD, "'0'");
385         addcode3("mov", BTP_SI, REG_DL);
386         addcode2("dec", REG_SI);
387         addcode2("inc", REG_RC);
388         addcode3("test", REG_RA, REG_RA);
389         addcode2("jnz", "_loopdigit@wint");
390         addcode3("test", REG_DI, REG_DI);
391         addcode2("jnz", "_negsign@wint");
392         addcode2("inc", REG_SI);
393         addcode2("jmp", "_syswrite@wint");
```

```
394         addlabel("_negsign@wint");
395         addcode3("mov", BTP_SI, "'-'");
396         addcode2("inc", REG_RC);
397         addlabel("_syswrite@wint");
398         addcode3("mov", REG_RD, REG_RC); // string length
399         addcode3("mov", REG_RA, "4"); // syscall number, NR
400         addcode3("mov", REG_RB, "1"); // fd: 1=stdout
401         addcode3("mov", REG_RC, REG_SI); // ptr to string buffer
402         addcode2("int", SYSCAL);
403
404         x86_lib_leave();
405 }
406
407 void x86_iolib_readchr()
408 {
409         adddata2("_scanbuf", "????????????????");
410
411         addlabel(LIBRCHR);
412         x86_lib_enter();
413
414         addlabel("_sysread@rchr");
415         addcode3("mov", REG_RA, "3"); // syscall number, NR
416         addcode3("mov", REG_RB, "0"); // fd: 0=stdin
417         addcode3("mov", REG_RC, "_scanbuf"); // ptr to scan buffer
418         addcode3("mov", REG_RD, "1"); // buffer size
419         addcode2("int", SYSCAL);
420         addcode3("xor", REG_RC, REG_RC);
421         addcode3("mov", REG_CL, "[_scanbuf]");
422         addcode3("cmp", REG_CL, "10"); // if ra == 'nl'(10), retry
423         addcode2("jz", "_sysread@rchr");
424         addcode3("mov", REG_RA, REG_RC); // save result to eax
425
426         x86_lib_leave();
427 }
428
429 void x86_iolib_readint()
430 {
431         adddata2("_scanint", "????????????????");
432
433         addlabel(LIBRINT);
434         x86_lib_enter();
435
```

```
436         addlabel("_sysread@rint");
437         addcode3("mov", REG_RA, "3"); // syscall number, NR
438         addcode3("mov", REG_RB, "0"); // fd: 0=stdin
439         addcode3("mov", REG_RC, "_scanint"); // ptr to scan buffer
440         addcode3("mov", REG_RD, "16"); // buffer size
441         addcode2("int", SYSCAL);
442         addlabel("_init@rint");
443         addcode3("xor", REG_RA, REG_RA);
444         addcode3("xor", REG_RC, REG_RC);
445         addcode3("mov", REG_RB, "1");
446         addcode3("mov", REG_SI, "_scanint");
447         addlabel("_begchar@rint");
448         addcode3("mov", REG_CL, BTP_SI);
449         addcode3("cmp", REG_RC, "'-'");
450         addcode2("jz", "_negnum@rint");
451         addcode3("cmp", REG_RC, "'0'");
452         addcode2("jl", "_skipchar@rint");
453         addcode3("cmp", REG_RC, "'9'");
454         addcode2("jg", "_skipchar@rint");
455         addcode2("jmp", "_numchar@rint");
456         addlabel("_skipchar@rint");
457         addcode2("inc", REG_SI);
458         addcode2("jmp", "_begchar@rint");
459         addlabel("_negnum@rint");
460         addcode3("mov", REG_RB, "-1");
461         addcode2("inc", REG_SI);
462         addlabel("_numchar@rint");
463         addcode3("mov", REG_CL, BTP_SI);
464         addcode3("cmp", REG_RC, "'0'");
465         addcode2("jl", "_notdigit@rint");
466         addcode3("cmp", REG_RC, "'9'");
467         addcode2("jg", "_notdigit@rint");
468         addcode3("sub", REG_RC, "'0'");
469         addcode3("imul", REG_RA, "10");
470         addcode3("add", REG_RA, REG_RC);
471         addcode2("inc", REG_SI);
472         addcode2("jmp", "_numchar@rint");
473         addlabel("_notdigit@rint");
474         addcode3("imul", REG_RA, REG_RB);
475
476         x86_lib_leave();
477 }
```

```
478
479 void x86_init()
480 {
481         addcode2("global", MAINFUNC);
482
483         x86_iolib_readchr();
484         x86_iolib_readint();
485         x86_iolib_wrtchr();
486         x86_iolib_wrtstr();
487         x86_iolib_wrtint();
488         x86_syslib_exit();
489 }
490
491 void x86_mov(reg_t *reg, syment_t *var)
492 {
493         switch (var->cate) {
494         case BYREF_OBJ:
495                 rwmem(READ_MEM_REF, reg, var, NULL);
496                 break;
497         default:
498                 rwmem(READ_MEM_VAL, reg, var, NULL);
499                 break;
500         }
501 }
502
503 void x86_mov2(syment_t *var, reg_t *reg)
504 {
505         switch (var->cate) {
506         case BYREF_OBJ:
507                 rwmem(SAVE_MEM_REF, reg, var, NULL);
508                 break;
509         default:
510                 rwmem(SAVE_REG_VAL, reg, var, NULL);
511                 break;
512         }
513 }
514
515 void x86_mov3(reg_t *reg, syment_t *arr, reg_t *idx)
516 {
517         rwmem(READ_MEM_VAL, reg, arr, idx);
518 }
519
```

```
520 void x86_mov4(syment_t *arr, reg_t *idx, reg_t *reg)
521 {
522         rwmem(SAVE_REG_VAL, reg, arr, idx);
523 }
524
525 void x86_mov5(reg_t *r1, reg_t *r2)
526 {
527         addcode3("mov", r1->name, r2->name);
528 }
529
530 void x86_mov6(reg_t *reg, int num)
531 {
532         addcode3("mov", reg->name, itoa(num));
533 }
534
535 void x86_mov7(reg_t *reg, char *strconst)
536 {
537         addcode3("mov", reg->name, strconst);
538 }
539
540 void x86_lea(reg_t *reg, syment_t *var)
541 {
542         rwmem(LOAD_MEM_ADDR, reg, var, NULL);
543 }
544
545 void x86_lea2(reg_t *reg, syment_t *arr, reg_t *idx)
546 {
547         rwmem(LOAD_MEM_ADDR, reg, arr, idx);
548 }
549
550 void x86_add(reg_t *r1, reg_t *r2)
551 {
552         addcode3("add", r1->name, r2->name);
553 }
554
555 void x86_sub(reg_t *r1, reg_t *r2)
556 {
557         addcode3("sub", r1->name, r2->name);
558 }
559
560 void x86_mul(reg_t *r1, reg_t *r2)
561 {
```

```
562         addcode3("imul", r1->name, r2->name);
563 }
564
565 // idiv (r/imm32)
566 //     edx:eax / (r/imm32)
567 // result:
568 //     eax <- quotient
569 //     edx <- remainder
570 reg_t *x86_div(reg_t *r1, reg_t *eax, reg_t *edx)
571 {
572         addcode3("xor", edx->name, edx->name);
573         addcode2("div", r1->name);
574         return eax;
575 }
576
577 void x86_neg(reg_t *r1)
578 {
579         addcode2("neg", r1->name);
580 }
581
582 void x86_inc(reg_t *r1)
583 {
584         addcode2("inc", r1->name);
585 }
586
587 void x86_dec(reg_t *r1)
588 {
589         addcode2("dec", r1->name);
590 }
591
592 void x86_xor(reg_t *r1, reg_t *r2)
593 {
594         addcode3("xor", r1->name, r2->name);
595 }
596
597 void x86_cls(reg_t *r1)
598 {
599         addcode3("xor", r1->name, r1->name);
600 }
601
602 void x86_pop(reg_t *reg)
603 {
```

```
604         addcode2("pop", reg->name);
605 }
606
607 void x86_push(reg_t *reg)
608 {
609         addcode2("push", reg->name);
610 }
611
612 void x86_push2(syment_t *var)
613 {
614         addcode2("push", var->label);
615 }
616
617 void x86_enter(syment_t *func)
618 {
619         char buf[MAXSTRBUF];
620         if (!strcmp(func->name, MAINFUNC)) {
621                 addlabel(MAINFUNC);
622         } else {
623                 sprintf(buf, "%s$%s", func->label, func->name);
624                 addlabel(buf);
625         }
626         scope = func->scope;
627         dbg("setscope depth=%d, nspace=%s\n", scope->depth, scope->nspace);
628
629         addcode2("push", REG_BP);
630         addcode3("mov", REG_BP, REG_SP);
631
632         int off = ALIGN * (func->scope->varoff + func->scope->tmpoff);
633         sprintf(buf, "reserve %d bytes for %s", off, func->name);
634         addcode4("sub", REG_SP, itoa(off), buf);
635         addcode2("push", REG_SI);
636         addcode2("push", REG_DI);
637         addcode2("push", REG_RB);
638 }
639
640 void x86_leave(syment_t *func)
641 {
642         addcode2("pop", REG_RB);
643         addcode2("pop", REG_DI);
644         addcode2("pop", REG_SI);
645         addcode3("mov", REG_SP, REG_BP);
```

```
646         addcode2("pop", REG_BP);
647         if (!strcmp(func->name, MAINFUNC)) {
648                 x86_syscall(LIBEXIT, NULL);
649         } else {
650                 x86_ret();
651         }
652 }
653
654 void x86_call(syment_t *func)
655 {
656         dupebp(func);
657
658         char buf[MAXSTRBUF];
659         sprintf(buf, "%s$%s", func->label, func->name);
660         addcode2("call", buf);
661
662         int i, npop;
663         npop = func->scope->depth - 1;
664         for (i = 0; i < npop; ++i) {
665                 addcode2("pop", REG_DI);
666         }
667         dbg("npop=%d\n", npop);
668 }
669
670 void x86_ret()
671 {
672         addcode1("ret");
673 }
674
675 reg_t *x86_syscall(char *func, reg_t *eax)
676 {
677         addcode2("call", func);
678         return eax;
679 }
680
681 void x86_label(syment_t *lab)
682 {
683         addlabel(lab->label);
684 }
685
686 void x86_jmp(syment_t *lab)
687 {
```

```
688         addcode2("jmp", lab->label);
689 }
690
691 void x86_cmp(reg_t *r1, reg_t *r2)
692 {
693         addcode3("cmp", r1->name, r2->name);
694 }
695
696 void x86_jz(syment_t *lab)
697 {
698         addcode2("jz", lab->label);
699 }
700
701 void x86_jnz(syment_t *lab)
702 {
703         addcode2("jnz", lab->label);
704 }
705
706 void x86_jg(syment_t *lab)
707 {
708         addcode2("jg", lab->label);
709 }
710
711 void x86_jng(syment_t *lab)
712 {
713         addcode2("jng", lab->label);
714 }
715
716 void x86_jl(syment_t *lab)
717 {
718         addcode2("jl", lab->label);
719 }
720
721 void x86_jnl(syment_t *lab)
722 {
723         addcode2("jnl", lab->label);
724 }
725
726 void x86_stralloc(char *name, char *initval)
727 {
728         adddata2(name, initval);
729 }
```

6.5.3 access link 区

access link 区是实现函数作用域的关键数据结构，当在某个特定作用域的变量时需要根据 access link 中保存的 ebp 来进行访问。图 6.6 是 frame.pas 进入 p2 函数时的 access link 区快照，从图中可以看出在每个 access link 区都保存了所以上级作用域的 ebp 指针，在访问非本作用域的变量时，需要使用 access link 区的 ebp 指针作为跳板来获取变量实际地址。

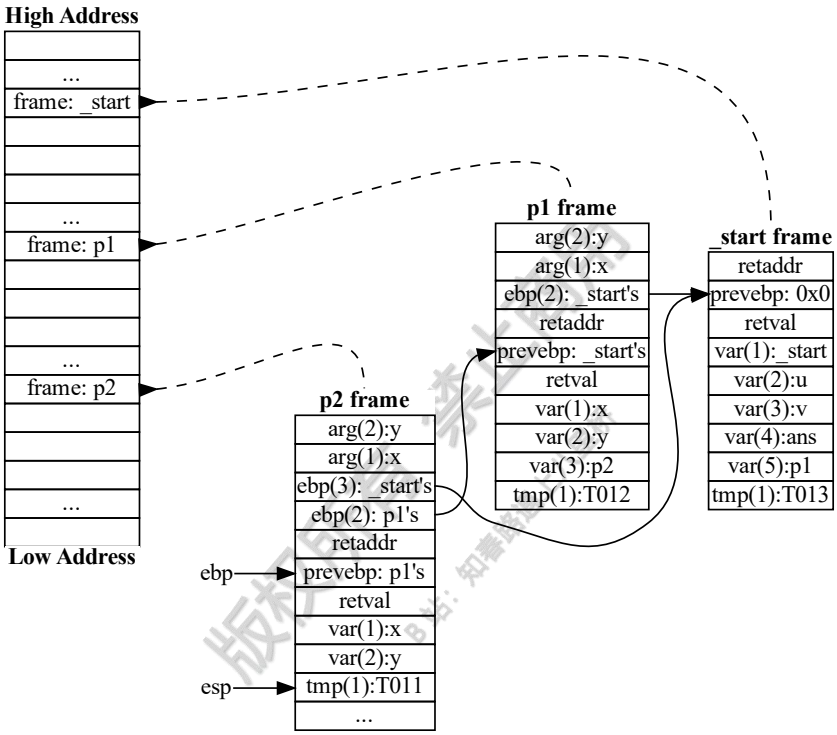


图 6.6: frame.pas 进入 p2 函数时的 access link 区快照

构建 access link 区是 pcc 生成汇编时的重要工作，具体实现是通过 `dupebp()` 函数实现的，该函数的实现大致逻辑如下：

1. 首先获取调用者和被调用者所在的作用域深度 `depth` ，分别记在 `caller` 和 `callee` 变量中。
2. 由于被调用者的 access link 区和 `callee` 数量相同，实现中通过 for 循环来准备 `callee` 个 ebp 指针。
3. `off` 记录需要复制的 ebp 指针与当前函数 ebp 的偏移值，
   - 如果 `off` 等于 1 的话，需要将当前 ebp 放入 access link 区
   - 否则将之前已经存在的 ebp 复制到被调用者的 access link 区

为了直观理解，这里将 `frame.pas` 生成的 nasm 汇编代码截取出来进行说明，具体 nasm 代码如下：

1. 第 63 到 64 行是 `_start` 函数调用 `p1` 函数时准备 access link 区的过程，这部分仅仅将

当前 ebp 入栈。

2. 第 32 到 35 行是 p1 函数调用 p2 函数时准备 access link 区的过程，这部分先将当前的 access link 中的历史 ebp 入栈，然后再将当前 ebp 入栈。
3. 在进入 p2 函数时，对应的 access link 就如图 6.6 所示。

```
1  L008$p2:
2          push    ebp
3          mov     ebp, esp
4          sub     esp, 8 ; reserve 8 bytes for p2
5          push    esi
6          push    edi
7          push    ebx
8          mov     eax, [ebp+16] ; L009 x
9          mov     ecx, [ebp+20] ; L010 y
10         add     eax, ecx
11         mov     [ebp-4], eax ; T011 @expr/add
12         mov     eax, [ebp-4] ; T011 @expr/add
13         mov     esi, [ebp+8]
14         mov     [esi-4], eax ; L008 p2
15         pop     ebx
16         pop     edi
17         pop     esi
18         mov     esp, ebp
19         pop     ebp
20         ret
21 L005$p1:
22         push    ebp
23         mov     ebp, esp
24         sub     esp, 12 ; reserve 12 bytes for p1
25         push    esi
26         push    edi
27         push    ebx
28         mov     eax, [ebp+16] ; L007 y
29         push    eax
30         mov     eax, [ebp+12] ; L006 x
31         push    eax
32         mov     edi, [ebp+8] ; dup old ebp
33         push    edi
34         mov     edi, ebp ; dup fresh ebp
35         push    edi
36         call    L008$p2
37         pop     edi
38         pop     edi
```

```
39          mov     eax, [ebp-4] ; L008 p2
40          mov     [ebp-8], eax ; T012 @fcall/ret
41          pop     eax
42          pop     eax
43          mov     eax, [ebp-8] ; T012 @fcall/ret
44          mov     esi, [ebp+8]
45          mov     [esi-20], eax ; L005 p1
46          pop     ebx
47          pop     edi
48          pop     esi
49          mov     esp, ebp
50          pop     ebp
51          ret
52  _start:
53          push    ebp
54          mov     ebp, esp
55          sub     esp, 28 ; reserve 28 bytes for _start
56          push    esi
57          push    edi
58          push    ebx
59          mov     eax, [ebp-12] ; L003 v
60          push    eax
61          mov     eax, [ebp-8] ; L002 u
62          push    eax
63          mov     edi, ebp ; dup fresh ebp
64          push    edi
65          call    L005$p1
66          pop     edi
67          mov     eax, [ebp-20] ; L005 p1
68          mov     [ebp-24], eax ; T013 @fcall/ret
69          pop     eax
70          pop     eax
71          mov     eax, [ebp-24] ; T013 @fcall/ret
72          mov     [ebp-16], eax ; L004 ans
73          pop     ebx
74          pop     edi
75          pop     esi
76          mov     esp, ebp
77          pop     ebp
78          call    _libexit
```

6.5.4 X86 内存寻址

通过之前的章节可以知道 X86 汇编的函数调用是通过运行栈来进行参数和变量传递的，图 6.7 是这个运行栈变量寻址的示意图。

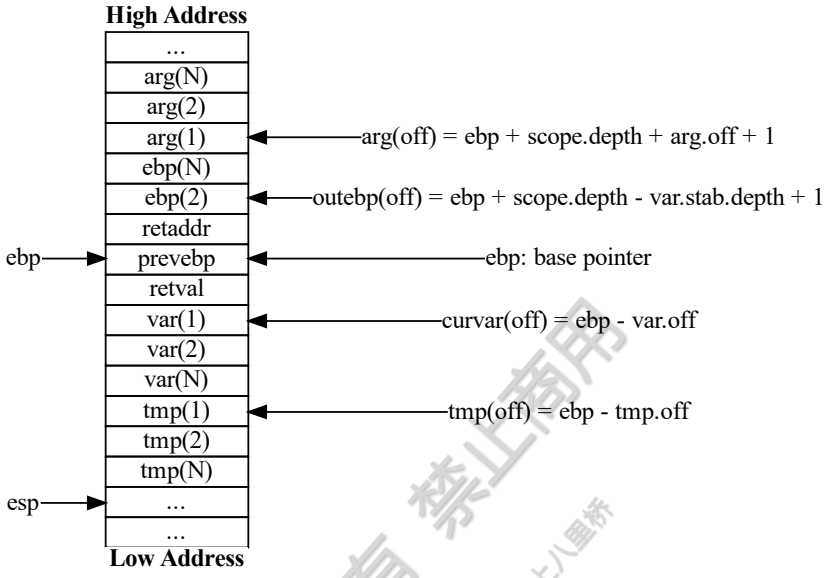


图 6.7: X86 运行栈变量寻址示意图

这里把变量地址方式的计算过程具体说明一下:

1. 基地址指针和寻址对齐。
   - 在 X86 中使用 ebp 寄存器作为基地址指针 (base pointer)，所有变量寻址都是基于 ebp 来计算偏移的。
   - 在 X86 中，变量都是 32 位的，而每个字节是 8 位，所以生成的汇编对齐值 `ALIGN=4` ，所有的寻址后得到地址都会乘以这个对齐值。
2. 实参的寻址方式:
   - 在 X86 的函数调用中是通过 `push` 入栈进行传递，并且入栈顺序和函数签名顺序刚好相反。
   - 这部分在 ebp 的高地址处，中间需要跳过 retval 和 access link 区，最终的计算公式为: arg(off) = ebp + scope.depth + arg.off + 1 ，其中: scope.depth 是当前作用域的深度，arg.off 是参数在符号表中分配的偏移值。
3. 临时变量的寻址方式:
   - 临时变量只会出现在当前作用域中。
   - 临时变量在 ebp 的低地址处，在符号表分配是已经考虑了变量的偏移，所以临时变量不需要二次跳过变量，最终临时变量的计算公式为: tmp(off) = ebp − tmp.off 。其中 tmp.off 是临时变量在符号表中分配的偏移值。
4. 变量的寻址方式: 函数作用域可以嵌套使得变量寻址方式分成了本作用域变量和外层作用域变量两种。

- 首先需要计算变量的 gap 值，gap 计算公式为 $gap = scope.depth - var.stab.depth$ ，其中：scope.depth 为当前作用域的深度，var.stab.depth 为目标变量作用域的深度。
- 如果 $gap = 0$ 说明目标变量位于当前作用域，这种变量寻址方式和临时变量一致，具体公式为：$curvar(off) = ebp - var.off$ ，其中 var.off 是变量在符号表中分配的偏移值。
- 否则 $gap > 0$ ，这时目标变量位于外层作用域，需要先找到目标作用域的 ebp 指针值，然后再加上偏移值来获取外层变量的实际地址。
  - 外层作用域的 ebp 值计算公式：$outebp = ebp + gap + 1$ 。
  - 外层变量的地址为 $outvar(off) = *outebp - var.off$ ，其中 * 符号表示读取对应地址的值。
  - 最终合并计算公式为：$outvar(off) = *(ebp + scope.depth - var.stab.depth + 1) - var.off$ 。

5. 数组变量的寻址方式和变量类似，只不过它是先获取到数组变量的首地址，然后通过偏移来定位数组中的具体元素。

变量寻址的实现位于 `rwmem(...)` 函数中，这个函数比较复杂，除了处理了变量寻址以外，该函数还计算了函数参数传值和传引用的不同场景。它大致分成以下几个计算阶段，通过 c 语言的标号来实现：

1. 初始阶段，`rwmem(...)` 函数的第一个 switch 语句中实现了初始阶段逻辑。
   - 首先，该部分处理了参数和临时变量的寻址，由于参数和临时变量不涉及跨作用域寻址，所以直接计算好地址后跳转到 `doit` 标号。
   - 其他的跨作用域寻址，需要跳转到 `findaddr` 标号来进一步寻址。
2. 查找地址阶段，标号 `findaddr:` 中实现了查找地址的逻辑。
   - 首先计算 gap 值来判断是否涉及到获取外部作用域的变量。
   - 然后通过 `var->cate == ARRAY_OBJ` 判断是否为数组，如果是数组需要计算数组下标，数组下标保存在寄存器 `reg_t *idx` 中，计算好下标后就可以获得数组元素的实际地址。
3. 执行阶段，标号 `doit:` 中实现的执行逻辑逻辑，这部分通过若干种模式用来控制执行行为，具体如下：
   - READ_MEM_VAL 表示读取内存中的值到寄存器中。
   - READ_MEM_REF 表示读取内存的引用值（地址）到寄存器中。
   - SAVE_REG_VAL 表示将寄存器的值存入到内存中。
   - SAVE_MEM_REF 表示将寄存器的引用值存入到内存中。
   - LOAD_MEM_ADDR 表示加载内存的地址到寄存器中。

6.6 编译后置工作

当得到 nasm 汇编文件后，pcc 还实现了通过调用 `system()` 系统调用来直接编译并链接成可执行文件的功能：

1. `post_nams()` 通过调用 nasm 来将 `.s` 文件编译成 `.o` 目标文件。
2. `post_link()` 通过调用 ld 来将 `.o` 目标文件编译成 `.run` 可执行文件。

这部分的实现代码见 `source/post.c` 文件，具体代码如下：

```
void post_nasm()
{
        if (!chkcmd("nasm")) {
                panic("nasm not installed.\n");
        }

        char cmd[MAXSTRBUF];
        sprintf(cmd, "nasm -f elf -o %s %s", PLOE_OBJECT, PLOE_ASSEM);
        dbg("%s\n", cmd);

        errnum = system(cmd);
        chkerr("post_nasm fail and exit.");
        phase = LINK;

        msg("assemble object file %s\n", PLOE_OBJECT);
}

void post_link()
{
        if (!chkcmd("ld")) {
                panic("ld not installed.\n");
        }

        char cmd[MAXSTRBUF];
        sprintf(cmd, "ld -m elf_i386 -o %s %s", PLOE_TARGET, PLOE_OBJECT);
        dbg("%s\n", cmd);

        errnum = system(cmd);
        chkerr("post_link fail and exit.");
        phase = SUCCESS;

        msg("link target file %s\n", PLOE_TARGET);
}

void post_clean()
{
        if (!PLOE_OPT_KEEP_NASM_FILE) {
                remove(PLOE_ASSEM);
                msg("remove file %s\n", PLOE_ASSEM);
        }

        if (!PLOE_OPT_KEEP_OBJECT_FILE) {
```

```
43                  remove(PL0E_OBJECT);
44                  msg("remove file %s\n", PL0E_OBJECT);
45          }
46  }
```

6.7 调试 pcc 生成的汇编程序

有时候需要查看 pcc 编译出来的 nasm 文件，可以使用 -s 选项来保存 nasm 的 *.s 文件，具体命令行如下：

```
pcc -s input.pas
```

使用 pcc 编译出来的 *.run 文件可以直接通过 gdb 进行调试，为了调试方便可以预先在 .gdbinit 文件中设置好下面选项。

```
# 开启 intel 反汇编风格
set disassembly-flavor intel
# 开启自动打印下一条汇编
set disassemble-next-line on
```

开启选项后可以通过 starti 来启动调试程序，然后使用 si 来单步调试，另外还有一些 ni 等 gdb 指令这里不做过多介绍，下面是一个具体调试过程。

```
$ gdb input.run
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
...
Reading symbols from input.run...
(No debugging symbols found in input.run)
(gdb) starti  <= 开启调试，并在第一条汇编处停止
Starting program: /cc/pcc/input.run

Program stopped.
0x080491e2 in _start ()
(gdb) si      <= 单条指令调试
0x080491e3 in _start ()
(gdb) set disassembly-flavor intel
(gdb) set disassemble-next-line on
(gdb) si
0x080491e5 in _start ()
=> 0x080491e5 <_start+3>:       83 ec 2c        sub    esp,0x2c
(gdb)
0x080491e8 in _start ()
=> 0x080491e8 <_start+6>:       56      push   esi
(gdb)
```

```
0x080491e9 in _start ()
=> 0x080491e9 <_start+7>:       57      push   edi
(gdb)
0x080491ea in _start ()
=> 0x080491ea <_start+8>:       53      push   ebx
(gdb)
0x080491eb in _start ()
=> 0x080491eb <_start+9>:       b8 01 00 00 00  mov    eax,0x1
(gdb)
```

6.8 本章总结

本章介绍了目标代码生成的底层实现逻辑，首先引入 X86 体系结构，然后使用 nasm 汇编实现了库函数，接着介绍了函数调用中的常见概念：运行栈和调用帧，最后通过介绍汇编代码生成函数来说明目标代码的具体细节。

第七章　代码优化

7.1　代码优化

编译器的代码优化是编译器设计中的一个重要阶段，其主要目标是提高目标代码的执行效率，减少资源消耗，或者改善其他相关的性能指标。代码优化通常发生在编译器生成最终的目标代码之前，即在中间代码生成之后。

在代码优化阶段，编译器会尝试应用一系列转换技术来改善代码的性能。这些技术可以大致分为两类：局部优化和全局优化。局部优化主要关注于单个基本块或单个函数的内部代码，而全局优化则考虑整个程序的行为。

下面是一些常见的编译器代码优化技术：

1. 常量折叠和常量传播：
    - 常量折叠是在编译时计算常量表达式的值，并用该值替换该表达式。
    - 常量传播是将一个已知的值从一个点传播到所有可以到达的点上，从而简化代码。
2. 无用代码删除：删除那些永远不会被执行到的代码，如不可达的基本块、未使用的变量和函数等。
3. 循环优化：针对于循环语句进行特定的优化。
    - 循环展开：通过增加循环体内的代码量来减少循环次数。
    - 循环合并：将多个小循环合并成一个更大的循环。
    - 循环不变量提取：将循环中不变化的计算移出循环。
4. 控制流优化：
    - 消除不必要的分支，如已知条件为真的 `if` 语句。
    - 将条件分支转换为无条件分支，如通过常量传播。
5. 内存优化：
    - 减少内存访问次数，如使用寄存器来存储频繁访问的变量。
    - 调整数据结构的布局以减少缓存未命中的可能性。
6. 指令调度：重新排列指令以优化流水线执行，减少指令间的依赖关系。
7. 函数内联：将函数调用替换为函数体的复制品，以减少函数调用的开销。
8. 寄存器分配：有效地将变量分配到处理器的寄存器中，以减少内存访问的开销。

在进行代码优化时，编译器需要遵循一些规则，以确保优化不会改变程序的语义。这通常意味着优化后的代码必须产生与原始代码相同的结果。同时，优化本身也应该是高效的，不应该显著增加编译时间。

编译器通常包含多种优化技术，并且会根据目标平台的特点和性能要求来选择和组合这些技术。此外，编译器也可能提供优化级别选项，允许用户选择不同程度的优化。

7.2 基本块和流图

7.2.1 基本块

在代码优化阶段，基本块 (Basic Block) 是一个重要的概念。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。换句话说，它是程序中连续执行且不会被打断的指令序列。入口就是该序列的第一个语句，而出口则是该序列的最后一个语句。对于基本块来说，执行时只能从其入口进入，并从其出口退出。

基本块的特点是其内部不存在跳转指令，控制流在块的第一个语句开始，并在最后一个语句结束，不会停止也不会在内部进行分支。在编译器中，基本块通常被用作优化的基本单位，因为它们的简单结构和明确的控制流使得优化变得更为容易和高效。

基本块 (Basic Block) 是编译器优化中的一个重要概念，尤其在程序分析和代码优化阶段。基本块是一种特殊的代码段，它具有以下特性：

1. **单一入口和单一出口**：基本块只有一个入口语句和一个出口语句。这意味着控制流只能从一个点进入基本块，并从另一个点离开。
2. **顺序执行**：基本块内的语句按照它们在源代码中出现的顺序执行，没有分支或跳转指令。
3. **不包含函数调用**：基本块通常不包含函数调用，因为函数调用可能会改变程序的执行上下文，从而引入额外的复杂性。

基本块在编译器中的应用场景主要是代码优化。由于基本块具有明确的控制流和数据依赖关系，编译器可以更容易地分析这些块，并应用一系列优化策略，如常量折叠、无用代码删除、循环展开等。

7.2.2 流图

基本块还用于构建流图 (Flow Graph) [1]。流图是一个有向图，其中节点代表基本块，边代表控制流的方向。流图为编译器提供了程序结构的紧凑表示，有助于进行各种程序分析和优化。

考虑以下简单的 $PL/0\epsilon$ 语言代码段：这段代码可以划分为一个基本块，因为它具有单一入口 (第一行) 和单一出口 (第三行)，并且语句按顺序执行，没有分支或函数调用。

```
1 a := 5;
2 b := 10;
3 c := a + b;
```

如果代码包含条件语句，如：在这个例子中，if 语句的条件分支创建了两个不同的基本块。第一个基本块是第一行的 `a := 5;` ，而第二个基本块是第四行的 `b := 10;` 和第五行的 `c := a +`

[1]后文中 NAC 指的是 Not A Constant (不是常量)

b; 两个语句。这两个基本块通过 if 语句的控制流连接。

```
a := 5;
if a > 3 then
begin
   b := 10;
   c := a + b
end
```

总之，基本块和流图是编译器优化中的一个关键概念，它简化了程序分析的过程，并使得各种优化策略更加有效。

7.2.3 构建流图流程

为了直观理解流图的明细，下面通过实际例子 opt01.pas 代码中来说明基本块和流图的实际结构。

```
var a, b, c : integer;
begin
   a := 1;
   if a > 3 then
   begin
      b := 10;
      c := a + b
   end
end
.
```


首先使用 prtir 工具将代码转换成四元式序列，其结果如下：

```
$ ./bin/prtir ./example/opt01.pas
compiler pcc start, version v1.0.2
reading file ./example/opt01.pas

DUMP SYMBOLS:
label=L001 type=1 cate=FUN name=_start off=1 stab=1 depth=1 initval=0 arrlen=0 str=
label=L002 type=1 cate=VAR name=a off=2 stab=1 depth=1 initval=0 arrlen=0 str=
label=L003 type=1 cate=VAR name=b off=3 stab=1 depth=1 initval=0 arrlen=0 str=
label=L004 type=1 cate=VAR name=c off=4 stab=1 depth=1 initval=0 arrlen=0 str=
label=T005 type=1 cate=NUM name=@factor/usi off=0 stab=1 depth=1 initval=1 arrlen=0 str=
label=T006 type=0 cate=LABEL name=@ifthen off=0 stab=1 depth=1 initval=0 arrlen=0 str=
label=T007 type=0 cate=LABEL name=@ifdone off=0 stab=1 depth=1 initval=0 arrlen=0 str=
label=T008 type=1 cate=NUM name=@factor/usi off=0 stab=1 depth=1 initval=3 arrlen=0 str=
label=T009 type=1 cate=NUM name=@factor/usi off=0 stab=1 depth=1 initval=10 arrlen=0 str=
label=T010 type=1 cate=TMP name=@expr/add off=5 stab=1 depth=1 initval=0 arrlen=0 str=
```

```
DUMP INTERMEDIATE REPRESENTATION:
#001: ENT       d=L001
#002: ASS       d=L002  r=T005
#003: GTT       d=T006  r=L002  s=T008
#004: JMP       d=T007
#005: LAB       d=T006
#006: ASS       d=L003  r=T009
#007: ADD       d=T010  r=L002  s=L003
#008: ASS       d=L004  r=T010
#009: LAB       d=T007
#010: FIN       d=L001

$
```

算法 1 描述了将四元式序列划分成基本块集合。首先算法的输入是四元式序列，输出是划分号的基本块集合。它的大体思路是：

1. 选好每个基本块的第一条指令放入 *leader* 变量中，
2. 如果没遇到跳转类型（条件跳转，无条件跳转，函数入口/出口，标号）的指令，则持续将当前指令放入 *bb* ，这样最终 *bb* 会是保存了一系列指令的基本块，
3. 最后将 *bb* 放入 *bbset* 基本块集合中，直到 *leader* 为空算法终止。

算法 1：划分基本块算法

输入：四元式序列 *quads*

输出：基本块集合 *bbset*

```
1  leader ← quads.first() ;
2  while leader 非空 do
3  |  bb ← ∅ ;
4  |  curr ← leader ;
5  |  leader ← leader.next() ;
6  |  while curr 非空 do
7  |  |  往 bb 中添加 curr 指令;
8  |  |  if leader 是跳转类型的指令 then
9  |  |  |  break ;
10 |  |  end
11 |  |  if curr 是跳转类型的指令 then
12 |  |  |  break ;
13 |  |  end
14 |  |  curr ← curr.next()
15 |  end
16 |  往 bbset 中添加 bb 基本块;
17 end
```

通过算法 1 可以将上述四元式序列分解成下面五个基本块，分别为基本块 B1、B2 、B3、B4 和 B5 （输出指令序列中忽略了 ENT 和 FIN 指令)。每个基本块内部的指令都是连续执行的序列，下面是基本块的输出结果：

```
block(B1 pred= succ=B2)
  #002: ASS     d=L002  r=T005
block(B2 pred=B1 succ=B3,B4)
  #003: GTT     d=T006  r=L002  s=T008
block(B3 pred=B2 succ=B4,B5)
  #004: JMP     d=T007
block(B4 pred=B3,B2 succ=B5)
  #005: LAB     d=T006
  #006: ASS     d=L003  r=T009
  #007: ADD     d=T010  r=L002  s=L003
  #008: ASS     d=L004  r=T010
block(B5 pred=B4,B3 succ=)
  #009: LAB     d=T007
```

在上述基本块划分的基础上，我们可以对通过跳转指令中的标号来将基本块之间的边连接起来，最终形成图 7.1 所示的流图。图中包含的元素解释如下：

1. 图中一定包含两个特殊的椭圆节点。
   - 其中一个是入口节点，即 entry 节点，
   - 另一个是出口节点，即 exit 节点。
2. 图中每个矩形的节点都是一个基本块。
   - 基本块通过字母 B 后添加阿拉伯数字来命名基本块，
   - 每个基本块内部的指令包含至少一个四元式指令。
3. 如果基本块之间存在跳转关系，则通过有向边连接。
   - 例如：图中有一条 B1 $\rightarrow$ B2 的有向边，
   - 这时我们会称 B1 是 B2 的前驱 (predecessor)，
   - 也称 B2 是 B1 的后继 (successor)。

至此，我们就完成将输入的源代码转化的流图的功能，为后续的代码优化提供了基础数据准备。

7.2.4 带循环的流图结构

在 $PL/0\epsilon$ 语言中包含一些循环结构的控制流，例如：for 循环，loop-until 循环等。这些循环结构也是可以通过流图进行表示，下面使用一个带 for 循环的代码 `opt02.pas` ，我们使用这个作为示例来说明带循环的流图

```
1 var x, y, z, i :integer;
2 begin
3    x := 1;
4    y := x;
5
```

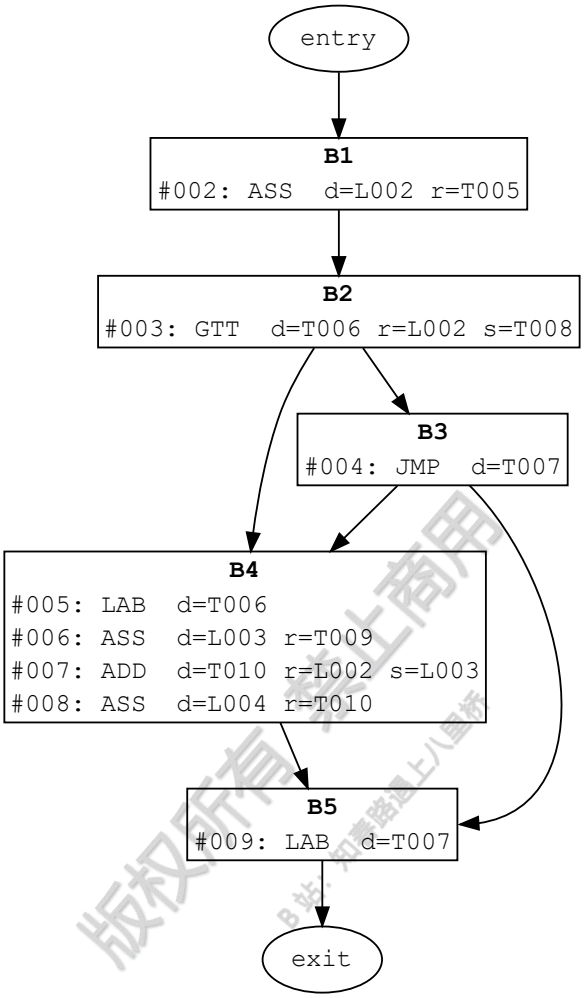


图 7.1: opt01.pas 对应的基本块划分和流图

```
6     for i := 1 to 10 do
7     begin
8        z := z * i + x + y
9     end;
10
11    z := z / 2
12  end
13  .
```

首先对代码进行四元式指令的解析，解析的结构如下：

```
$ ./bin/prtir ./example/opt02.pas
compiler pcc start, version v1.0.4
reading file ./example/opt02.pas
```

```
DUMP SYMBOLS:
label=L001 type=1 cate=FUN name=_start off=1 stab=1 depth=1 initval=0 arrlen=0 str=
label=L002 type=1 cate=VAR name=x off=2 stab=1 depth=1 initval=0 arrlen=0 str=
label=L003 type=1 cate=VAR name=y off=3 stab=1 depth=1 initval=0 arrlen=0 str=
label=L004 type=1 cate=VAR name=z off=4 stab=1 depth=1 initval=0 arrlen=0 str=
label=L005 type=1 cate=VAR name=i off=5 stab=1 depth=1 initval=0 arrlen=0 str=
label=T006 type=1 cate=NUM name=@factor/usi off=0 stab=1 depth=1 initval=1 arrlen=0 s
tr=
label=T007 type=1 cate=NUM name=@factor/usi off=0 stab=1 depth=1 initval=1 arrlen=0 s
tr=
label=T008 type=1 cate=NUM name=@factor/usi off=0 stab=1 depth=1 initval=10 arrlen=0
str=
label=T009 type=0 cate=LABEL name=@forstart off=0 stab=1 depth=1 initval=0 arrlen=0 s
tr=
label=T010 type=0 cate=LABEL name=@fordone off=0 stab=1 depth=1 initval=0 arrlen=0 st
r=
label=T011 type=1 cate=TMP name=@term/mul off=6 stab=1 depth=1 initval=0 arrlen=0 str
=
label=T012 type=1 cate=TMP name=@expr/add off=7 stab=1 depth=1 initval=0 arrlen=0 str
=
label=T013 type=1 cate=TMP name=@expr/add off=8 stab=1 depth=1 initval=0 arrlen=0 str
=
label=T014 type=1 cate=NUM name=@factor/usi off=0 stab=1 depth=1 initval=2 arrlen=0 s
tr=
label=T015 type=1 cate=TMP name=@term/div off=9 stab=1 depth=1 initval=0 arrlen=0 str
=

DUMP INTERMEDIATE REPRESENTATION:
#001: ENT       d=L001
#002: ASS       d=L002  r=T006
#003: ASS       d=L003  r=L002
#004: ASS       d=L005  r=T007
#005: LAB       d=T009
#006: GTT       d=T010  r=L005  s=T008
#007: MUL       d=T011  r=L004  s=L005
#008: ADD       d=T012  r=T011  s=L002
#009: ADD       d=T013  r=T012  s=L003
#010: ASS       d=L004  r=T013
#011: INC       d=L005
#012: JMP       d=T009
#013: LAB       d=T010
#014: DEC       d=L005
```

```
#015: DIV       d=T015  r=L004  s=T014
#016: ASS       d=L004  r=T015
#017: FIN       d=L001

$
```

图 7.2 中是代码 opt02.pas 的流图，解析过后四元式三点被分成了 6 个基本块，相比较直接查看四元式序列，通过流图可以非常直观的观察出存在循环控制流，即存在 B5 → B2 这个包含回溯的有向边。

7.3 基本块内优化

7.3.1 构建 DAG 图

DAG （Directed Acyclic Graph）图，中文翻译为有向无环图，尤其在程序优化和代码生成阶段。DAG 图是由节点和它们之间的有向边组成的图，每个节点代表一个操作或任务，有向边表示依赖关系，也就是某个节点的输出作为另一个节点的输入。

DAG 图通常用于表示程序的基本块。基本块的 DAG 图可以用于多种优化，例如常量折叠（将常量表达式在编译时求值，而不是在运行时）、公共子表达式消除（避免重复计算相同的表达式）等。

DAG 图的构建通常从四元式开始，四元式由操作符、两个操作数和一个结果组成。在构建 DAG 图时，每个四元式都被转换为一个节点，其中操作符是节点的标记，操作数是节点的输入，结果是节点的输出。此外，还有一些特殊的节点，如叶节点，它们用标识符（变量名）或常量作为标记。算法 2 描述了 DAG 图构建算法它的大体思路如下：

1. **节点创建**：为每个变量创建一个节点。节点可以表示变量的值或运算结果。
2. **边创建**：分析中间代码中的语句，为每个语句创建一个有向边。边的起点是产生值的节点（或常量节点），终点是使用该值的节点。
3. **合并节点**：如果两个节点表示相同的值，可以将它们合并为一个节点。这有助于减少 DAG 图的大小和复杂性。

为了直观理解 DAG 图的构建流程，我们使用 opt03.pas 中的代码作为示例来介绍 DAG 图，其代码如下：

```
1 var a, b, c, d, x :integer;
2 begin
3    x := a + a*(b-c) + (b-c)*d
4 end
5 .
```

代码 opt03.pas 的核心就是第 3 行计算的表达式的值，我们通过 prtir 将代码转化成四元式序列进行分析。转换的结果如下：

```
$ ./bin/prtir ./example/opt03.pas
```

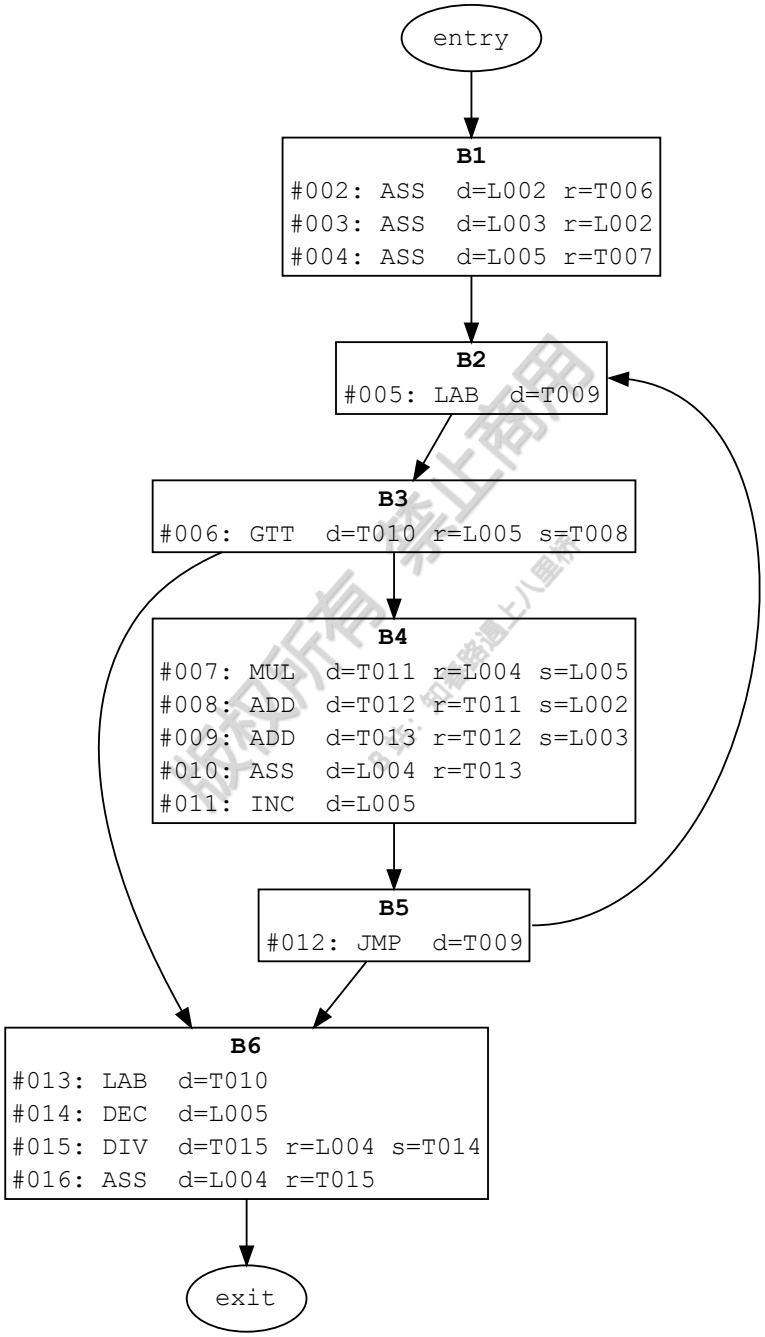


图 7.2: 带循环结构代码 opt02.pas 的流图

算法 2: 构建 DAG 图算法

输入: 单个基本块中的四元式序列 *quads*

输出: 四元式构成的 DAG 图 *graph*

```
// 四元式 quad 包含三个属性: 操作符 op, 左操作数 lhs 和右操作数 rhs
for quad ← quads do
    // Step1: 准备左操作符
    lhs ← 查找 graph 中包含 quad.lhs 符号的节点;
    if lhs 为空 then
        lhs ← 往 graph 中新建 quad.lhs 符号的节点;
    end
    // Step2: 准备右操作符
    rhs ← 查找 graph 中包含 quad.rhs 符号的节点;
    if rhs 为空 then
        rhs ← 往 graph 中新建 quad.rhs 符号的节点;
    end
    // Step3: 准备输出节点
    for node ←graph 中的非叶子节点 do
        if node.op ≠quad.op then
            continue ;
        end
        if node.lhs ≠quad.lhs then
            continue ;
        end
        if  node.rhs ≠quad.rhs then
            continue ;
        end
        out ← node ;
    end
    if out 为空 then
        out ← 根据 quad 的 op, lhs, rhs 新建的节点;
    end
    // Step4: 插入新节点
    将 out 节点插入 graph 中;
end
```

```
compiler pcc start, version v1.0.4
reading file ./example/opt03.pas

DUMP SYMBOLS:
label=L001 type=1 cate=FUN name=_start off=1 stab=1 depth=1 initval=0 arrlen=0 str=
label=L002 type=1 cate=VAR name=a off=2 stab=1 depth=1 initval=0 arrlen=0 str=
label=L003 type=1 cate=VAR name=b off=3 stab=1 depth=1 initval=0 arrlen=0 str=
label=L004 type=1 cate=VAR name=c off=4 stab=1 depth=1 initval=0 arrlen=0 str=
label=L005 type=1 cate=VAR name=d off=5 stab=1 depth=1 initval=0 arrlen=0 str=
label=L006 type=1 cate=VAR name=x off=6 stab=1 depth=1 initval=0 arrlen=0 str=
label=T007 type=1 cate=TMP name=@expr/sub off=7 stab=1 depth=1 initval=0 arrlen=0 str=
label=T008 type=1 cate=TMP name=@term/mul off=8 stab=1 depth=1 initval=0 arrlen=0 str=
label=T009 type=1 cate=TMP name=@expr/add off=9 stab=1 depth=1 initval=0 arrlen=0 str=
label=T010 type=1 cate=TMP name=@expr/sub off=10 stab=1 depth=1 initval=0 arrlen=0 str=
label=T011 type=1 cate=TMP name=@term/mul off=11 stab=1 depth=1 initval=0 arrlen=0 str=
label=T012 type=1 cate=TMP name=@expr/add off=12 stab=1 depth=1 initval=0 arrlen=0 str=

DUMP INTERMEDIATE REPRESENTATION:
#001: ENT       d=L001
#002: SUB       d=T007  r=L003  s=L004
#003: MUL       d=T008  r=L002  s=T007
#004: ADD       d=T009  r=L002  s=T008
#005: SUB       d=T010  r=L003  s=L004
#006: MUL       d=T011  r=T010  s=L005
#007: ADD       d=T012  r=T009  s=T011
#008: ASS       d=L006  r=T012
#009: FIN       d=L001

$
```


通过分析生成的四元式可知：上述代码没有分支跳转语句，所以最终 **只包含一个基本块**。这个基本块是表达式 `x := a + a*(b-c) + (b-c)*d` 拆分成的四元式序列，为了清晰地看出这个等价关系，我们把转化后的三地址码写出来。结果如下：

$$T_7 \leftarrow L_3(b) - L_4(c) \tag{7.1}$$
$$T_8 \leftarrow L_2(a) * T_7 \tag{7.2}$$
$$T_9 \leftarrow L_2(a) + T_8 \tag{7.3}$$
$$T_{10} \leftarrow L_3(b) - L_4(c) \tag{7.4}$$
$$T_{11} \leftarrow T_{10} * L_5(d) \tag{7.5}$$
$$T_{12} \leftarrow T_9 + T_{11} \tag{7.6}$$
$$L_6(x) \leftarrow T_{12} \tag{7.7}$$

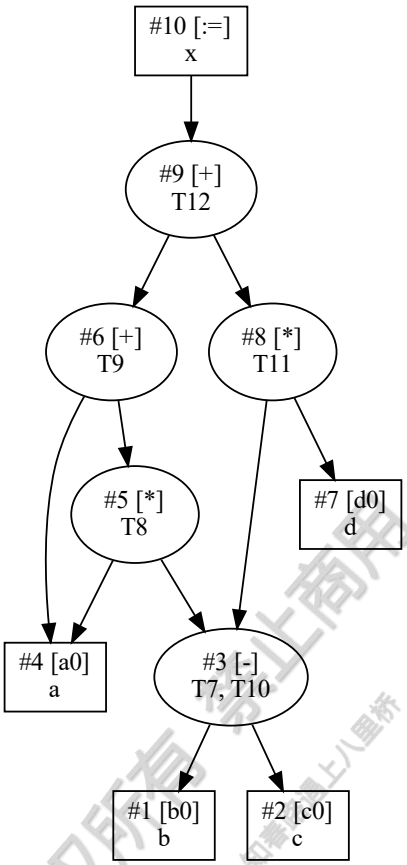


图 7.3: opt03.pas 代码构建的 DAG 图

通过上述代码可以构建出图 7.3 的 DAG 图，图中包括两类节点：

1. 矩形形状的是数值节点，用于表示初值
2. 椭圆形状的时计算节点，用于表示常见的计算操作
3. 每个节点第一行方括号中表示节点的属性
   - 如果是叶子节点，b0 和 c0 表示初始值
   - 如果是非叶子节点，+ 和 * 表示加法和乘法等运算符
4. 每个节点第二行时该节点对应的变量的值

在 DAG 图构建完成后，可以对其进行各种优化。例如，通过重新排列节点的顺序，可以改进代码的执行效率。此外，还可以通过合并已知量、删除无用赋值等方式进一步简化 DAG 图。

最后，将优化后的 DAG 图转换为可执行代码是编译过程的最后一步。这通常涉及到将 DAG 图中的节点和边转换为相应的指令序列，并进行进一步的优化和调度，以生成高效的机器代码。

7.3.2 SSA 静态单赋值

Static Single Assignment (SSA) 对于中文指的是静态单赋值，它是是一种程序属性，用于描述程序中的变量只被赋值一次，并且在赋值后不再改变。这个属性通常用于编译器优化和程序分析。

在 SSA 形式中，每个变量都有一个唯一的定义点，这意味着一旦变量被赋值，它就不会再被重新赋值。这有助于简化某些编译器优化和程序分析任务，因为它减少了需要跟踪的变量状态的数量。

要实现 SSA，编译器可能需要执行一系列转换，例如复制传播 (Copy Propagation)、死代码消除 (Dead Code Elimination) 等，以确保每个变量只被赋值一次。

虽然 SSA 是一种有用的属性，但它并不总是必需的。在某些情况下，编译器可能会选择不保持 SSA 形式，以便进行其他类型的优化或简化代码生成。

需要注意的是，Static 在这里指的是变量的赋值在编译时就可以确定，而不是在运行时。这与 Dynamic 相对，后者指的是变量的状态可能在运行时发生变化。

当然，我可以通过一个简单的例子来解释 SSA。

假设我们有以下的伪代码:

```
y := 1;
y := 2;
x := y;
```

在普通的程序中，变量 y 被赋值了两次，而 x 的值取决于 y 的最后一个赋值。在 SSA 形式中，每个变量只能被赋值一次。因此，上述代码将被转换为:

```
y1 := 1;
y2 := 2;
x  := y2;
```

在这个 SSA 形式的代码中，y 被拆分成了两个不同的变量 y1 和 y2。y1 被赋值为 1，而 y2 被赋值为 2。这样，每个变量都只被赋值一次。最后，x 被赋值为 y2，即 y 的最后一个值。

这种转换对于编译器优化非常有用，因为它简化了变量的使用-定义关系。例如，现在我们可以清楚地看到 x 的值来自于 y2，而不是之前的 y1。这有助于编译器进行诸如常量传播 (constant propagation)、值范围传播 (value range propagation) 等优化。

此外，SSA 还有一个重要的概念是 ϕ 函数 (Phi Function)。在存在分支的代码中，一个变量可能在多个分支中被赋值。为了保持 SSA 属性，编译器会引入一个 ϕ 节点来合并这些分支中的值。

例如，在下面分支代码中，如果 y 在一个条件分支中被赋值，

```
if flag then y := -1 else y := 1;
x := y * a;
```

那么会有一个 ϕ 节点来根据条件选择正确的 y 值。

```
if flag then y1 := -1 else y2 := 1;
y3 := phi(y1, y2);
x := y3 * a;
```

在修改成 SSA 形式的代码中，第 2 行就是使用 ϕ 函数的例子，这里的 y3 值可能是 y1 ，也可能是 y2 。ϕ 函数的作用是根据程序执行时实际选择的路径，选择相应的变量值。具体来说，ϕ 函数接受多个输入值（通常来自不同分支的赋值），并产生一个输出值，该输出值是根据程序执行路径选择的输入值。这样，在 SSA 形式中，每个变量都只有一个定义点，而 ϕ 函数则用于处理由于分支导致的不确定性

7.3.3 消除公共表达式

消除公共表达式（Common Subexpression Elimination，CSE）是编译器优化技术中的一种，它的目的是减少程序中重复计算的表达式的数量。通过识别并存储已经计算过的表达式的值，当这些表达式再次出现时，可以直接使用存储的值，而不是重新计算。

在进行公共表达式消除之前，编译器首先会对程序进行数据流分析，以识别出哪些表达式是公共的，即它们在程序中出现了多次。然后，编译器会创建一个存储这些表达式值的表（通常称为值号表或值映射表），并在程序中每次遇到这些表达式时，先检查值号表，看是否已经计算过这个表达式的值。

这里是一个简单的例子来说明公共表达式消除的过程：

```
x := a + a*(b-c) + (b-c)*d
```

通过之前关于 DAG 图构建的分析，上述表达式会得到原始四元式序列如下：

```
#002: SUB     d=T007  r=L003  s=L004          t7  := b - c
#003: MUL     d=T008  r=L002  s=T007          t8  := a * t7
#004: ADD     d=T009  r=L002  s=T008          t9  := a + t8
#005: SUB     d=T010  r=L003  s=L004          t10 := b - c
#006: MUL     d=T011  r=T010  s=L005          t11 := t10 * d
#007: ADD     d=T012  r=T009  s=T011          t12 := t9 * t11
#008: ASS     d=L006  r=T012                  x   := t12
```

通过 DAG 图沟通分析可以知道，子表达式 (b-c) 被重复计算了两次，所以 DAG 图重新推导四元式序列会将两次计算合并，实际只计算一次，最终生成的结果如下：

```
#000: SUB     d=T007  r=L003  s=L004          t7  := b - c
#000: MUL     d=T008  r=L002  s=T007          t8  := a * t7
#000: ADD     d=T009  r=L002  s=T008          t9  := a * t8
#000: MUL     d=T011  r=T007  s=L005          t11 := t7 * d
#000: ADD     d=T012  r=T009  s=T011          t12 := t9 * t11
#000: ASS     d=L006  r=T012                  x   := t12
```

公共表达式消除可以显著提高程序的执行效率，特别是在计算密集型的程序中。然而，它也可能增加程序的代码大小，因为需要额外的存储空间来保存中间结果。因此，编译器在决定是否进行

公共表达式消除时，通常会权衡这两个因素。

7.4 全局优化

7.4.1 数据流分析

数据流分析（Data Flow Analysis）是一种用于程序优化的经典技术，它主要关注程序中数据的流动情况。数据流分析能够帮助编译器推断程序在运行时的数据流信息，进而进行各种优化操作。

数据流分析的主要目的是收集程序的语义信息，并通过代数方法在编译时确定变量的定义和使用情况。它通常涉及对程序中的控制流进行分析，以确定数据在不同程序点之间的流动情况。

数据流分析可以包括多种类型，如到达定值分析、活跃变量分析和可用表达式分析等。到达定值分析关注变量值的流动情况，确定每个变量在使用前的最后一个定义位置。活跃变量分析则关注哪些变量在当前程序点是活跃的（即可能被使用），这有助于编译器进行寄存器分配等优化操作。可用表达式分析则旨在找到程序中可以重复使用的表达式，从而避免重复计算。

7.4.2 传递函数

如图 7.4 所示，我们把每个语句 s：$a \leftarrow b \oplus c$ 之前和之后的数据流值分别记为 $\mathsf{IN}[s]$ 和 $\mathsf{OUT}[s]$ [2] 。数据流问题就是对一组约束求解，得到所有 $\mathsf{IN}[s]$ 和 $\mathsf{OUT}[s]$ 的结果。每个语句 s 都约束了 IN 和 OUT 这两个数据流值之间的关系，这种约束关系叫做传递函数（transfer function），由 f_s 来表示，其中下标 s 表示是关于语句 s 的传递函数。

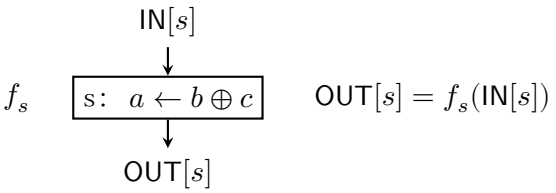


图 7.4：单个语句的数据流转移函数关系

传递函数也有两种不同的风格：数据流信息可能沿着执行路径向前传播，或者沿着程序路径逆向流动，相应的就有前向（Forward）数据流问题和后向（Backward）数据流问题。

对于前向数据流问题，一个程序语句 s 的传递函数以语句前程序点的数据流值作为输入，并产生出语句之后程序点对应的新数据流值，这种情况对应传递函数见公式（7.8）。例如到达定值就是前向数据流问题。

$$\mathsf{OUT}[s] = f_s(\mathsf{IN}[s]) \tag{7.8}$$

[2]注意语句 s 中的 $\oplus$ 表示通用二元操作符，可以是 $+$ 、$-$ 、$\times$ 和 $\div$ 等。

对于后向数据流问题，一个程序语句 s 的传递函数以语句后的程序点的数据流值作为输入，转变成语句之前程序点的新数据流值，这种情况对应传递函数见公式（7.9）。例如活变量分析就是后向数据流问题。

$$\mathsf{IN}[s] = f_s(\mathsf{OUT}[s]) \tag{7.9}$$

所有的数据流分析的结果都具有相同的形式：对于程序中的每个指令，它们描述了该指令每次执行时必然成立的一些性质。

7.4.3 基本块的传递函数

我们首先定义一个基本块 B ，它包含 n 条语句 $s_1, s_2, \cdots, s_n$ 。这个基本块 B 的形式如公式（7.10）所示。

$$\begin{aligned} s_1 &: a_1 \leftarrow b_1 \oplus_1 c_1 \\ s_2 &: a_2 \leftarrow b_2 \oplus_2 c_2 \\ &\cdots \\ s_n &: a_n \leftarrow b_n \oplus_n c_n \end{aligned} \tag{7.10}$$

如图 7.5 所示，我们把基本块 B 之前和之后的数据流值分别记为 $\mathsf{IN}[B]$ 和 $\mathsf{OUT}[B]$ ，则基本块的传递函数就可以定义成 f_B 。

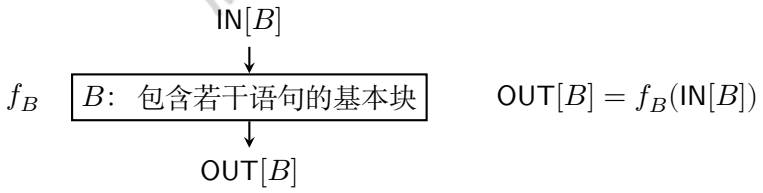


图 7.5: 单个基本块的数据流转移函数关系

由于基本块的每条语句都是顺序执行的，并且没有分支跳转。所以基本块内的控制流比较简单。为了讨论方便，我们把基本块每条语句的 IN 集合、OUT 集合和传递函数都通过图示画出来，其结果如图 7.6 所示。

我们来分析这个基本块 B 内的传递函数，可以得出以下性质：

1. 基本块的 IN 和 OUT 集合存在以下等价关系：
   - $\mathsf{IN}[B] = \mathsf{IN}[s_1]$
   - $\mathsf{OUT}[B] = \mathsf{OUT}[s_n]$
2. 语句间的 IN 和 OUT 存在传递关系，即满足公式（7.11）

$$\mathsf{IN}[s_{i+1}] = \mathsf{OUT}[s_i], \qquad \text{其中 } i = 1, 2, \cdots, n-1 \tag{7.11}$$

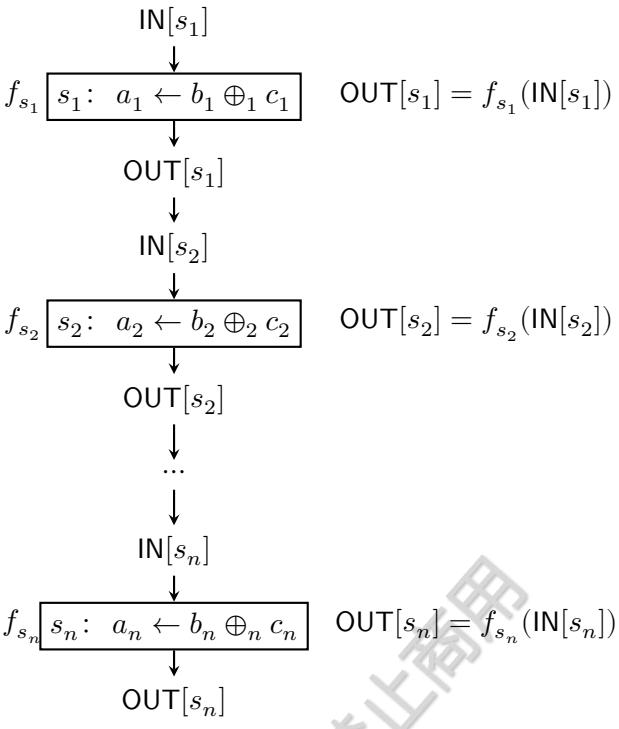


图 7.6: 基本块内语句的数据流转移函数关系

3. 基本块的传递函数是其中的各个语句传递函数的复合[3]，最终基本块 B 的传递函数 f_B 的求解满足公式（7.12）

$$f_B = f_{s_n} \circ \cdots \circ f_{s_2} \circ f_{s_1} \tag{7.12}$$

7.4.4 基本块的前驱和后继

我们之前介绍控制流图时简单提及到前驱和后继的概念，在控制流图中，前驱和后继表示的时基本块之间的之间跳转关系。

前驱（predecessor）是指那些能够直接跳转到当前基本块 B 的块，换句话说，前驱基本块中的指令执行完成后，控制流会直接转移到当前基本块。基本块 B 的所有前驱基本块构成前驱基本块集合，记做 pred_B 。通过控制流图可知前驱还满足公式（7.13）。

$$\mathsf{IN}[B] = \bigcup_{P \in \mathsf{pred}_B} \mathsf{OUT}[P] \tag{7.13}$$

后继（successor）则是指那些可以从当前基本块 B 直接跳转到的块，换句话说，当前基本块中的指令执行完成后，控制流可能会直接转移到后继基本块。基本块 B 的所有后继基本块构成后继基本块集合，记做 succ_B 。通过控制流图可知前驱还满足公式（7.14）。

[3]函数的复合运算记为 $g \circ f$ 形式，它是将一个函数的输出作为另一个函数的输入，进行连续运算的过程。其含义是：设有函数 $f : A \to B$ 和函数 $g : B \to C$ ，则 f 和 g 的复合函数满足：$\forall x \in A,\ g \circ f(x) = g(f(x))$ 。

$$\mathsf{OUT}[B] = \bigcup_{S \in \mathsf{succ}_B} \mathsf{IN}[S] \tag{7.14}$$

基本块的前驱和后继也经常被用于中间代码的优化和变换。我们通过具体示例来说明前驱基本块集合和后继基本块集合的求解方式，如图 7.7 所示，可以直接通过控制流图的跳转关系计算出：

- $\mathsf{pred}_B = \{B_1, B_2, B_3\}$
- $\mathsf{succ}_B = \{B_4, B_5\}$

基本块的前驱和后继的 $\mathsf{IN}[B]$ 和 $\mathsf{OUT}[B]$ 还满足下面性质：

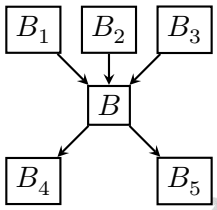


图 7.7：基本块之间的前驱和后继关系

7.4.5 数据流应用示例

这里通过例子来说明数据流分析的是如何在真实场景中应用的。有时候我们需要知道程序中各个变量在某个程序点上是否为常量[4]，这种需求是可以使用数据流来分析，下面是一个分析的片段。

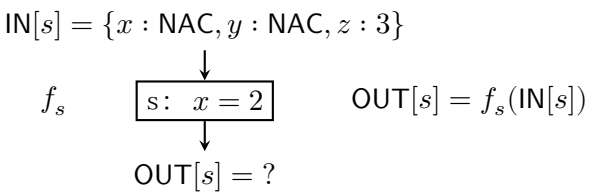


图 7.8：单语句的程序点是否为常量分析的例子

如图 7.8 所示，语句 s 为 $x = 2$，图中满足以下条件，分析 $\mathsf{OUT}[s]$ 是通过传递函数 f_s 作用后会是什么结果？

1. 考虑变量 x，y 和 z 的状态
2. 已知 $\mathsf{IN}[s] = \{x : \mathsf{NAC}, y : \mathsf{NAC}, z : 3\}$

这里 $\mathsf{OUT}[s]$ 的计算过程如下：

- 对于变量 x，由于语句 s 将其赋值成 2，所以有 $x : 2$
- 对于变量 y，语句 s 没有修改，$\mathsf{IN}[s]$ 中 $y : \mathsf{NAC}$，所以有 $y : \mathsf{NAC}$
- 对于变量 z，语句 s 没有修改，$\mathsf{IN}[s]$ 中 $z : 3$，所以有 $z : 3$
- 所以最终 $\mathsf{OUT}[s] = \{x : 2, y : \mathsf{NAC}, z : 3\}$

[4]后文中 NAC 指的是 Not A Constant（不是常量）

这样，我们就知道了通过传递函数对单个语句中 OUT 集合的计算流程。

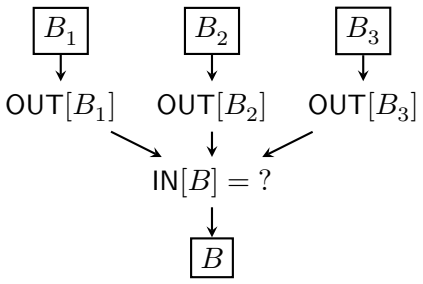


图 7.9: 基本块之间 IN 和 OUT 合并的例子

我们再看一个基本块合并的例子。图 7.9 满足下列条件，计算出 IN$[B]$ 的结果?

1. 基本块 B 的前驱基本块集合 $\text{pred}_B = \{B_1, B_2, B_3\}$
2. 考虑变量 x，y 和 z 的状态
3. 已知前驱基本块的 OUT 集合的值如下:
   - OUT$[B_1] = \{x : 1, y : 2, z : \text{NAC}\}$
   - OUT$[B_2] = \{x : 1, y : 5, z : 4\}$
   - OUT$[B_3] = \{x : 1, y : 7, z : 4\}$

这个例子中需要考虑基本块 B 的所有前驱基本块的 OUT 值，具体计算过程如下:

- 对于变量 x ，在 OUT$[B_1]$，OUT$[B_2]$ 和 OUT$[B_3]$ 中都有 $x : 1$ ，所以可以得出 IN$[B]$ 中也有 $x : 1$
- 对于变量 y ，在 OUT$[B_1]$ ，OUT$[B_2]$ 和 OUT$[B_3]$ 中的值不一样，所以可以得出 IN$[B]$ 中也有 $y : \text{NAC}$
- 对于变量 z ，在 OUT$[B_3]$ 中有 $z : \text{NAC}$ ，所以可以得出 IN$[B]$ 中也有 $z : \text{NAC}$
- 所以最终 IN$[B] = \{x : 1, y : \text{NAC}, z : \text{NAC}\}$

7.4.6 到达定值分析

到达定值（Reaching Definition）是一个数据流分析的一个常见的应用场景。它描述了在程序执行过程中，某个变量的某个定值（即对该变量的赋值）是否能够在特定的程序点被访问到。

如果一个定值 d 在程序执行过程中能够到达某个程序点 p，并且在这个路径上 d 没有被其他定值所 “杀死”（即变量没有被重新赋值为其他值），那么我们就说这个定值 d 能够到达程序点 p。这里的 “杀死” 意味着在该路径上，变量被赋予了新的值，从而使得之前的定值不再有效。

如图 7.10 所示，可以通过对图中的定值流动情形分析得出:

- B_1 全部定值到达 B_2 的开头
- d_5 到达 B_2 的开头（循环）
- d_2 被 d_5 杀死，不能到达 B_3 、B_4 的开头
- d_4 不能到达 B_2 的开头，因为被 d_7 杀死
- d_6 到达 B_2 的开头

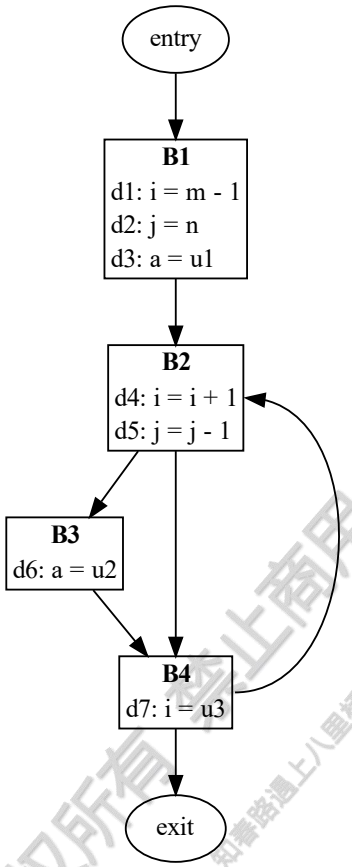


图 7.10：到达定值示例流图

我们把到达定值问题进行形式化建模，首先我们给出单个语句的定义。如公式（7.15）所示，它定义了变量 u 的值，值通过表达式 $v \oplus w$ 给出。换一句话说，它“生成”了变量 u 的定值，这个记做定值 d 。

$$d : u \leftarrow v \oplus w \tag{7.15}$$

我们以图 7.10 中的定值 d_1 进行举例，它定义并赋值了变量 i ，所有可以求出对应的 $\mathsf{gen}_{d_1} = \{d_1\}$ ，$\mathsf{kill}_{d_1} = \{d_4, d_7\}$ 。其它语句的 gen 和 kill 可以通过同样的规则求出，最终所有语句的“生成”集和“杀死”集结果如下：

- $\mathsf{gen}_{d_1} = \{d_1\}$，$\mathsf{kill}_{d_1} = \{d_4, d_7\}$
- $\mathsf{gen}_{d_2} = \{d_2\}$，$\mathsf{kill}_{d_2} = \{d_5\}$
- $\mathsf{gen}_{d_3} = \{d_3\}$，$\mathsf{kill}_{d_3} = \{d_6\}$
- $\mathsf{gen}_{d_4} = \{d_4\}$，$\mathsf{kill}_{d_4} = \{d_1, d_7\}$
- $\mathsf{gen}_{d_5} = \{d_5\}$，$\mathsf{kill}_{d_5} = \{d_2\}$
- $\mathsf{gen}_{d_6} = \{d_6\}$，$\mathsf{kill}_{d_6} = \{d_3\}$
- $\mathsf{gen}_{d_7} = \{d_7\}$，$\mathsf{kill}_{d_7} = \{d_1, d_4\}$

由于 d “生成” 了变量 u 的定值，在执行该语句过后，它还会 “杀死” 代码中其它的关于变量 u 的定值。所以它对应的传递函数可以写成公式（7.16）。

$$f_d(x) = \mathsf{gen}_d \cup (x - \mathsf{kill}_d) \tag{7.16}$$

这里 $\mathsf{gen}_d = \{d\}$ ，kill_d 是代码中其它关于变量 u 的定值构成的集合。

根据前文的公式（7.12）的结论，我们讨论基本块的定值的传递函数，我们假设基本块 B 中包含两个连续的定值语句 d_1 和 d_2 ，它们对应的传递函数分别为：$f_{d_1}(x) = \mathsf{gen}_{d_1} \cup (x - \mathsf{kill}_{d_1})$ 和 $f_{d_2}(x) = \mathsf{gen}_{d_2} \cup (x - \mathsf{kill}_{d_2})$ 。

$$\begin{aligned}
f_{d_2} \circ f_{d_1}(x) &= f_{d_2}(f_{d_1}(x)) \\
&= \mathsf{gen}_{d_2} \cup \big(f_{d_1}(x) - \mathsf{kill}_{d_2}\big) \\
&= \mathsf{gen}_{d_2} \cup \big(\mathsf{gen}_{d_1} \cup (x - \mathsf{kill}_{d_1}) - \mathsf{kill}_{d_2}\big) \\
&= \mathsf{gen}_{d_2} \cup \Big(\big(\mathsf{gen}_{d_1} - \mathsf{kill}_{d_2}\big) \cup \big((x - \mathsf{kill}_{d_1}) - \mathsf{kill}_{d_2}\big)\Big) \\
&= \mathsf{gen}_{d_2} \cup \big(\mathsf{gen}_{d_1} - \mathsf{kill}_{d_2}\big) \cup \big((x - \mathsf{kill}_{d_1}) - \mathsf{kill}_{d_2}\big) \\
&= \mathsf{gen}_{d_2} \cup \big(\mathsf{gen}_{d_1} - \mathsf{kill}_{d_2}\big) \cup \big(x - (\mathsf{kill}_{d_1} \cup \mathsf{kill}_{d_2})\big) \\
&= \big(\mathsf{gen}_{d_2} \cup (\mathsf{gen}_{d_1} - \mathsf{kill}_{d_2})\big) \cup \big(x - (\mathsf{kill}_{d_1} \cup \mathsf{kill}_{d_2})\big)
\end{aligned} \tag{7.17}$$

将公式（7.17）推广到包含 n 个语句的基本块 B 中。假设基本块 B 的第 i 个语句 d_i 的传递函数为：$f_{d_i}(x) = \mathsf{gen}_{d_i} \cup (x - \mathsf{kill}_{d_i})$ ，其中 $i = 1, 2, \cdots, n$ 。那么，这个基本块 B 的传递函数 f_B 可以写成：

$$f_B(x) = \mathsf{gen}_B \cup (x - \mathsf{kill}_B) \tag{7.18}$$

其中 gen_B 和 kill_B 满足公式（7.19）- （7.20）：

$$\begin{aligned}
\mathsf{gen}_B = \mathsf{gen}_{d_n} &\cup (\mathsf{gen}_{d_{n-1}} - \mathsf{kill}_{d_n}) \cup (\mathsf{gen}_{d_{n-2}} - \mathsf{kill}_{d_{n-1}} - \mathsf{kill}_{d_n}) \cup \\
&\cdots \cup (\mathsf{gen}_{d_1} - \mathsf{kill}_{d_2} - \mathsf{kill}_{d_3} - \cdots - \mathsf{kill}_{d_n})
\end{aligned} \tag{7.19}$$

$$\mathsf{kill}_B = \mathsf{kill}_{d_1} \cup \mathsf{kill}_{d_2} \cup \cdots \cup \mathsf{kill}_{d_n} \tag{7.20}$$

- 公式（7.19）表示 gen_B 是被第 i 个语句生成，且没有被其后的句子 “杀死” 的定值的集合。
- 公式（7.20）表示 kill_B 为被基本块 B 中各个语句 “杀死” 的定值的并集。

我们来通过一个例子来说明基本块的 gen 集合和 kill 集合的计算过程。假设有一个基本块包含下面两条语句：

$$d_1 : a \leftarrow 3$$

$$d_2 : a \leftarrow 4$$

根据公式（7.18）的结论可以计算出 gen_B 和 kill_B 如下：

- $\mathsf{gen}_B = \{d_2\}$
- $\mathsf{kill}_B = \{$流图中所有针对 a 的定值$\}$

7.4.7 到达定值迭代求解算法

算法 3 描述了求解每个基本块的 $\mathsf{IN}[B]$ 和 $\mathsf{OUT}[B]$ 集合的方法。我们假设每个流图都包含两个空的入口基本块 entry 和出口基本块 exit。

1. 首先根据之前的介绍的算法准备输入数据：
   - 将四元式序列划分基本块，通过基本块的跳转关系生成流图，
   - 通过公式（7.19）-（7.20）可以求出各基本块的 gen_B 和 kill_B 。
2. 接着初始化算法中的 OUT 集合：
   - 第 1 行设置 $\mathsf{OUT}[entry]$ 为空集，
   - 第 2 至 4 行令所有的 $\mathsf{OUT}[B]$ 都是空集。
3. 最后就是通过流图中基本块的前驱关系不停迭代，得到最小不动点的解。
   - 算法的退出条件是：所有 OUT 集合的值不发生变化，
   - 迭代算法每个循环更新除 entry 以外的基本块得 IN 和 OUT ，
   - 第 7 行根据流图的前驱公式（7.13）更新 $\mathsf{IN}[B]$ ，
   - 第 8 行根据公式（7.19）的合并公式更新 $\mathsf{OUT}[B]$ 。

算法 3：到达定值问题的迭代求解算法

输入：流图和各个基本块的 gen 和 kill 集合
输出：每个基本块的 $\mathsf{IN}[B]$ 和 $\mathsf{OUT}[B]$ 集合

```
1  OUT[entry] ← ∅ ;
2  for B ← 除 entry 以外的基本块 do
3  |  OUT[B] ← ∅ ;
4  end
5  while 某个 OUT 集合值发生改变 do
6  |  for B ← 除 entry 以外的基本块 do
7  |  |  IN[B] ← ⋃_{P∈pred_B} OUT[P] ;
8  |  |  OUT[B] ← gen_B ∪ (IN[B] − kill_B) ;
9  |  end
10 end
```

迭代算法的严谨性的证明需要利用到 Lattice [5] 和不动点理论等数学理论，证明过程比较繁琐并且也超出了本书介绍的范围，我们不妨使用不太严谨的直观方法说明算法的正确性和终止性。

首先证明算法 3 的正确性，即算法得到的 IN 和 OUT 包含的结果一定是正确的：

[5]Lattice 的中文翻译是格，一般使用英文术语表述能避免歧义。

1. **初始状态**：在算法开始时，所有变量的到达定值集合都是空的，这是正确的初始状态。
2. **保持性质**：我们需要证明在每次迭代中，算法 3 都保持了以下性质：
   - 对于每个基本块和每个变量，其基本块的 IN 包含了所有能够到达该基本块开头的定值，
   - OUT 包含了所有在该基本块内产生的新定值以及从 IN 中传递过来的定值。
3. **归纳证明**：假设在第 k 次迭代后，算法保持了上述性质。我们需要证明在第 k+1 次迭代后，算法仍然保持该性质。
   - IN 的正确性：对于每个基本块，其 IN 是由其所有前驱基本块的 OUT 合并得到的。由于在第 k 次迭代后，每个前驱基本块的 OUT 都是正确的，因此第 k+1 次迭代计算得到的 IN 也是正确的。
   - OUT 的正确性：对于每个基本块，其 OUT 是由该基本块内的所有新定值和 IN 合并得到的。由于在第 k 次迭代后，IN 是正确的，并且基本块内的新定值也是正确计算的，因此第 k+1 次迭代计算得到的 OUT 也是正确的。

通过归纳证明，我们可以得出结论：在算法收敛（即 IN 和 OUT 不再变化）时，每个基本块的输入和 OUT 都是正确的，因此算法正确地计算了每个变量的到达定值。

然后证明算法的终止性，即 IN 和 OUT 最终会收敛不再变化。这通过分析算法的性质和程序的结构来证明。例如，可以证明在每次迭代中，至少有一个基本块的 IN 或 OUT 会发生变化，并且 IN 和 OUT 的范围只会扩大，当 IN 和 OUT 的范围最终无法扩大，也就是到达传递函数的不动点的情景，迭代过程最终会终止。

综上所述，我们可以证明到达定值的迭代算法的正确性和终止性，算法 3 可以产出到达定值的 IN 和 OUT 。

7.4.8 到达定值求解示例

我们以前文中的图 7.10 作为示例流图来说明到达定值的求解过程，之前已经求了每个定值的 gen 和 kill ，通过公式 (7.19) - (7.20) 可以求出各基本块的 gen_B 和 kill_B ，结果如下：

- $\mathsf{gen}_{B_1} = \{d_1, d_2, d_3\}$，$\mathsf{kill}_{B_1} = \{d_4, d_5, d_6, d_7\}$
- $\mathsf{gen}_{B_2} = \{d_4, d_5\}$，$\mathsf{kill}_{B_2} = \{d_1, d_2, d_7\}$
- $\mathsf{gen}_{B_3} = \{d_6\}$，$\mathsf{kill}_{B_3} = \{d_3\}$
- $\mathsf{gen}_{B_4} = \{d_7\}$，$\mathsf{kill}_{B_4} = \{d_1, d_4\}$

由于算法涉及到集合的交和差集运算，为了后面说明的方便，我们使用 7 位的二进制数字 bitmap 的将定值进行编码，从左到右依次表示 d_1，d_2，…，d_7 ，其中 0 表示集合中定值存在，1 表示集合中定值不存在。编码过后上面的 gen 和 kill 可以表示成下面的形式：

- $\mathsf{gen}_{B_1} = 1110000$，$\mathsf{kill}_{B_1} = 0001111$
- $\mathsf{gen}_{B_2} = 0001100$，$\mathsf{kill}_{B_2} = 1100001$
- $\mathsf{gen}_{B_3} = 0000010$，$\mathsf{kill}_{B_3} = 0010000$
- $\mathsf{gen}_{B_4} = 0000001$，$\mathsf{kill}_{B_4} = 1001000$

然后我们求出流图中每个基本块的 pred 集，结果如下：

- $\mathsf{pred}_{B_1} = \emptyset$

- $\text{pred}_{B_2} = \{B_1, B_4\}$
- $\text{pred}_{B_3} = \{B_2\}$
- $\text{pred}_{B_4} = \{B_2, B_3\}$

最后，我们通过 python 实现到达定值求解的迭代算法。代码中使用了二进制的位操作来高效实现集合运算，具体代码如下：

```
# CONSTANT: number of definition, basic block
n_def, n_block = 7, 5

# INPUT: pred, gen, kill
pred = [
    [],         # pred_B1: empty
    [1, 4],     # pred_B2: B1, B4
    [2],        # pred_B3: B2
    [2, 3],     # pred_B4: B2, B3
    [4],        # pred_exit: B4
]
gen = [
    0b1110000,  # B1
    0b0001100,  # B2
    0b0000010,  # B3
    0b0000001,  # B4
    0b0000000,  # exit
]
kill = [
    0b0001111,  # B1
    0b1100001,  # B2
    0b0010000,  # B3
    0b1001000,  # B4
    0b0000000,  # exit
]

# OUTPUT: din, dout
din, dout = [0] * n_block, [0] * n_block


# Dump all set
def dumpset(var, data):
    for i in range(0, n_block):
        b = bin(data[i])[2:]
        print(
            "%s[%d]\t= %s"
```

```
37              % (var, i + 1, b[:n_def] if len(b) >= n_def else b.zfill(n_def))
38          )
39
40
41  # Iterative Solver
42  def solve():
43      changed, epoch = True, 1
44      while changed:
45          print("======== epoch=%d ========" % epoch)
46          dumpset("IN", din)
47          dumpset("OUT", dout)
48
49          changed = False
50          for b in range(0, n_block):
51              # update IN[B]
52              for p in pred[b]:
53                  din[b] |= dout[p - 1]
54
55              # update OUT[B]
56              old = dout[b]
57              dout[b] = gen[b] | (din[b] & ~kill[b])
58
59              # check exit condition
60              if dout[b] != old:
61                  changed = True
62
63          epoch += 1
64
65
66  if __name__ == "__main__":
67      solve()
68      print("======== RDA DONE ========")
69      dumpset("IN", din)
70      dumpset("OUT", dout)
```


代码的第 1 至 29 行用于初始化输入，包括：pred 前驱集、gen 生成集、OUT 杀死集，并且将 IN 和 OUT 都设置成空集。第 31 至 38 实现的 dumpset() 函数用于打印集合的输出。核心的迭代求解器是第 41 至 63 行实现的 solve() 函数，solve() 函数每个轮次首先打印当前的 IN 和 OUT 集，变量 changed 标记当前 OUT 集合是否发生变化，如果发生变化则继续迭代，其中第 52 至 53 行更新 IN 集合，第 57 行更新 OUT 集合。

运行 python 代码并获得下面的输出结果 [6]。可以看出，经过 3 轮次的迭代，最终算法会收敛

[6]这里的 B_5 基本块表示 exit 出口

到不动点。

```
======== epoch=1 ========
IN[1]   = 0000000
IN[2]   = 0000000
IN[3]   = 0000000
IN[4]   = 0000000
IN[5]   = 0000000
OUT[1]  = 0000000
OUT[2]  = 0000000
OUT[3]  = 0000000
OUT[4]  = 0000000
OUT[5]  = 0000000
======== epoch=2 ========
IN[1]   = 0000000
IN[2]   = 1110000
IN[3]   = 0011100
IN[4]   = 0011110
IN[5]   = 0010111
OUT[1]  = 1110000
OUT[2]  = 0011100
OUT[3]  = 0001110
OUT[4]  = 0010111
OUT[5]  = 0010111
======== epoch=3 ========
IN[1]   = 0000000
IN[2]   = 1110111
IN[3]   = 0011110
IN[4]   = 0011110
IN[5]   = 0010111
OUT[1]  = 1110000
OUT[2]  = 0011110
OUT[3]  = 0001110
OUT[4]  = 0010111
OUT[5]  = 0010111
======== RDA DONE ========
IN[1]   = 0000000
IN[2]   = 1110111
IN[3]   = 0011110
IN[4]   = 0011110
IN[5]   = 0010111
OUT[1]  = 1110000
OUT[2]  = 0011110
```

```
OUT[3]  = 0001110
OUT[4]  = 0010111
OUT[5]  = 0010111
```

最终到达定值迭代求解算法计算结果整理如表 7.1 所示，表中包含 3 个轮次的计算，其中每一列表示一次计算。

表 7.1：到达定值迭代求解算法计算结果

基本块 \ 轮次	第 1 轮	第 2 轮		第 3 轮	
	$\mathsf{OUT}[B]^0$	$\mathsf{IN}[B]^1$	$\mathsf{OUT}[B]^1$	$\mathsf{IN}[B]^2$	$\mathsf{OUT}[B]^2$
B_1	0000000	0000000	1110000	0000000	1110000
B_2	0000000	1110000	0011100	1110111	0011110
B_3	0000000	0011100	0001110	0011110	0001110
B_4	0000000	0011110	0010111	0011110	0010111
exit	0000000	0010111	0010111	0010111	0010111

表 7.1 中最后的计算轮次结果汇总如下：

- $\mathsf{IN}[B_1] = \emptyset$ ， $\mathsf{OUT}[B_1] = \{d_1, d_2, d_3\}$
- $\mathsf{IN}[B_2] = \{d_1, d_2, d_3, d_5, d_6, d_7\}$ ， $\mathsf{OUT}[B_2] = \{d_3, d_4, d_5, d_6\}$
- $\mathsf{IN}[B_3] = \{d_3, d_4, d_5, d_6\}$ ， $\mathsf{OUT}[B_3] = \{d_4, d_5, d_6\}$
- $\mathsf{IN}[B_4] = \{d_3, d_4, d_5, d_6\}$ ， $\mathsf{OUT}[B_4] = \{d_3, d_5, d_6, d_7\}$
- $\mathsf{IN}[\mathrm{exit}] = \{d_3, d_5, d_6, d_7\}$ ， $\mathsf{OUT}[\mathrm{exit}] = \{d_3, d_5, d_6, d_7\}$

7.4.9 活跃变量分析

活跃变量分析（Live Variable Analysis）用于确定在程序的特定点（或程序点）上，某个变量是否可能在未来的某条路径上被使用。如果变量在程序点的值可能在未来的某条路径上被使用，那么这个变量在该程序点被认为是 “活跃” 的；否则，它被认为是 “死的”。

这种分析的主要用途之一是用于基本块的存储器分配。如果一个值在计算后被保存到一个寄存器中，并且这个值在基本块中可能被使用，那么该值就是活跃的。如果这个值在基本块中是死的，那么在基本块的结尾处就不必保存这个值。当所有寄存器都被占用，但还需要申请新的寄存器时，应该考虑使用已经存储了死亡值的寄存器，因为这个值不需要再保存到内存中。

对于语句 s：$x \leftarrow y \oplus z$ ，我们先说明定义集 def 和使用集 use 所表示的含义：

- def_s 表示语句 s 中定义出的变量集合，上述语句 s 中 $\mathsf{def}_s = \{x\}$ ，
- use_s 表示语句 s 中使用到的变量集合，上述语句 s 中 $\mathsf{use}_s = \{y, z\}$ 。

活跃变量分析是通过程序控制流的反向进行的，它的传递函数 f_s 可以写成：$\mathsf{IN}[s] = f_s(\mathsf{OUT}[s])$ 。通过到达定值分析类似的推导过程，我们组合基本块中的所有语句的传递函数以得到基本块的传递函数。假设基本块 B 包含 n 个语句 $s_1, s_2, \cdots, s_n$ 。则它的传递函数 f_B 满足公式（7.21）：

$$f_B(x) = \mathsf{use}_B \cup (x - \mathsf{def}_B) \tag{7.21}$$

其中 use_B 满足公式（7.22），def_B 满足公式（7.22）。

$$\begin{aligned}\mathsf{use}_B = {} & \mathsf{use}_{s_1} \cup (\mathsf{use}_{s_2} - \mathsf{def}_{s_1}) \cup (\mathsf{use}_{s_3} - \mathsf{def}_{s_1} - \mathsf{def}_{s_2}) \cup \\ & \cdots \cup (\mathsf{use}_{s_n} - \mathsf{def}_{s_1} - \mathsf{def}_{s_2} - \cdots - \mathsf{def}_{s_{n-1}}) \end{aligned} \tag{7.22}$$

$$\mathsf{def}_B = \mathsf{def}_{s_1} \cup \mathsf{def}_{s_2} \cup \cdots \cup \mathsf{def}_{s_n} \tag{7.23}$$

最后，使用这些数据流方程来计算各个程序点的活跃变量集合。我们可以给出描述活跃变量问题的迭代求解算法，具体描述见算法 4，以下几点需要注意：

1. 基本块的转换函数仍然是生成-杀死形式，但是计算方向是反向的，
   - 先计算出 OUT 然后再计算 IN，
   - 计算 OUT 时算法取的是后继集合 succ。
2. 任何变量在程序出口 exit 处不再活跃，所以初始化 $\mathsf{OUT}[\text{exit}] \leftarrow \emptyset$。
3. 对于所有非 exit 基本块迭代的合并函数满足公式（7.21）。

算法 4：活跃变量问题的迭代求解算法

输入：流图和各个基本块的 def 和 use 集合
输出：每个基本块的 $\mathsf{IN}[B]$ 和 $\mathsf{OUT}[B]$ 集合

```
1  OUT[exit] ← ∅ ;
2  for B ← 除 exit 以外的基本块 do
3  |  IN[B] ← ∅ ;
4  end
5  while 某个 IN 集合值发生改变 do
6  |  for B ← 除 exit 以外的基本块 do
7  |  |  OUT[B] ← ⋃_{S∈succ_B} IN[S] ;
8  |  |  IN[B] ← use_B ∪ (OUT[B] − def_B) ;
9  |  end
10 end
```

7.4.10 活跃变量分析求解举例

首先我们编写一个 $PL/0\epsilon$ 的代码文件 opt04.pas，其代码如下：

```
1  const a = 3;
2  var x, y, z :integer;
3  begin
4     x := 1;
5     y := 2;
```

```
6
7     if x > a then
8     begin
9        y := y + 1;
10       z := a;
11       write(x);
12    end
13    else y := y + 2;
14
15    y := 2 * y;
16    write(y);
17 end.
```

通过 prtir 工具可以将文件 opt04.pas 中代码生成的四元式打印出来，其结果如下：

```
$ ./bin/prtir ./example/opt04.pas
compiler pcc start, version v1.1.4
reading file ./example/opt04.pas

DUMP SYMBOLS:
label=L001 type=1 cate=FUN name=_start off=1 stab=1 depth=1 initval=0 arrlen=0 str=
label=L002 type=1 cate=CONST name=a off=0 stab=1 depth=1 initval=3 arrlen=0 str=
label=L003 type=1 cate=VAR name=x off=2 stab=1 depth=1 initval=0 arrlen=0 str=
label=L004 type=1 cate=VAR name=y off=3 stab=1 depth=1 initval=0 arrlen=0 str=
label=L005 type=1 cate=VAR name=z off=4 stab=1 depth=1 initval=0 arrlen=0 str=
label=T006 type=1 cate=NUM name=@factor/usi off=0 stab=1 depth=1 initval=1 arrlen=0 str=
label=T007 type=1 cate=NUM name=@factor/usi off=0 stab=1 depth=1 initval=2 arrlen=0 str=
label=T008 type=0 cate=LABEL name=@ifthen off=0 stab=1 depth=1 initval=0 arrlen=0 str=
label=T009 type=0 cate=LABEL name=@ifdone off=0 stab=1 depth=1 initval=0 arrlen=0 str=
label=T010 type=1 cate=NUM name=@factor/usi off=0 stab=1 depth=1 initval=2 arrlen=0 str=
label=T011 type=1 cate=TMP name=@expr/add off=5 stab=1 depth=1 initval=0 arrlen=0 str=
label=T012 type=1 cate=NUM name=@factor/usi off=0 stab=1 depth=1 initval=1 arrlen=0 str=
label=T013 type=1 cate=TMP name=@expr/add off=6 stab=1 depth=1 initval=0 arrlen=0 str=
label=T014 type=1 cate=NUM name=@factor/usi off=0 stab=1 depth=1 initval=2 arrlen=0 str=
label=T015 type=1 cate=TMP name=@term/mul off=7 stab=1 depth=1 initval=0 arrlen=0 str=

DUMP INTERMEDIATE REPRESENTATION:
#001: ENT       d=L001
#002: ASS       d=L003  r=T006                  x   := 1
#003: ASS       d=L004  r=T007                  y   := 2
#004: GTT       d=T008  r=L003  s=L002          goto t8 if x > a
#005: ADD       d=T011  r=L004  s=T010          t11 := y + 2
#006: ASS       d=L004  r=T011                  y   := t11
```

```
#007: JMP       d=T009                          goto t9
#008: LAB       d=T008                    t8:
#009: ADD       d=T013  r=L004  s=T012           t13 := y + 1
#010: ASS       d=L004  r=T013                   y   := t13
#011: ASS       d=L005  r=L002                   z   := a
#012: WRI       d=L003                           write(x)
#013: LAB       d=T009                    t9:
#014: MUL       d=T015  r=T014  s=L004           t15 := 2 * y
#015: ASS       d=L004  r=T015                   y   := t15
#016: WRI       d=L004                           write(y)
#017: FIN       d=L001

$
```

将上述四元式序列转换成流图的结果见图 7.11 ，它包含了 B1 至 B6 这 6 个基本块。为了方便查看，图中还翻译了四元式对应的伪代码序列，分号右边的伪代码注释。

结合图 7.11 中的流图分析，我们首先需要将四元式包含的所有变量找到，其实就是集合 $\{x, y, z, t_{11}, t_{13}, t_{15}\}$ 这些变量。通过公式（7.22）和公式（7.22）可以求出每个基本块的 def 和 use 集，结果如下：

- $\mathsf{def}_{B_1} = \{x, y\}$， $\mathsf{use}_{B_1} = \emptyset$
- $\mathsf{def}_{B_2} = \emptyset$， $\mathsf{use}_{B_2} = \{x\}$
- $\mathsf{def}_{B_3} = \{t_{11}\}$， $\mathsf{use}_{B_3} = \{y\}$
- $\mathsf{def}_{B_4} = \emptyset$， $\mathsf{use}_{B_4} = \emptyset$
- $\mathsf{def}_{B_5} = \{x, z, t_{13}\}$， $\mathsf{use}_{B_5} = \{y\}$
- $\mathsf{def}_{B_6} = \{t_{15}\}$， $\mathsf{use}_{B_6} = \{y\}$

如果以变量 $x, y, z, t_{11}, t_{13}, t_{15}$ 从左往右依次使用二进制编码，其中 0 表示变量是 “活跃”，1 表示变量是 “死亡” 的。可以得出以下结果：

- $\mathsf{def}_{B_1} = 110000$， $\mathsf{use}_{B_1} = 000000$
- $\mathsf{def}_{B_2} = 000000$， $\mathsf{use}_{B_2} = 100000$
- $\mathsf{def}_{B_3} = 000100$， $\mathsf{use}_{B_3} = 010000$
- $\mathsf{def}_{B_4} = 000000$， $\mathsf{use}_{B_4} = 000000$
- $\mathsf{def}_{B_5} = 101010$， $\mathsf{use}_{B_5} = 010000$
- $\mathsf{def}_{B_6} = 000001$， $\mathsf{use}_{B_6} = 010000$

如表 7.2 所示，通过数据流分析的迭代算法求解，经过 4 个轮次的迭代，最终结果稳定在不动点处。

我将最后一轮的 IN 和 OUT 集合中的值进行整理可以得到下面的结论：

- $\mathsf{IN}_{B_1} = \emptyset$， $\mathsf{OUT}_{B_1} = \{x, y\}$
- $\mathsf{IN}_{B_2} = \{x, y\}$， $\mathsf{OUT}_{B_2} = \{y\}$
- $\mathsf{IN}_{B_3} = \{y\}$， $\mathsf{OUT}_{B_3} = \{y\}$

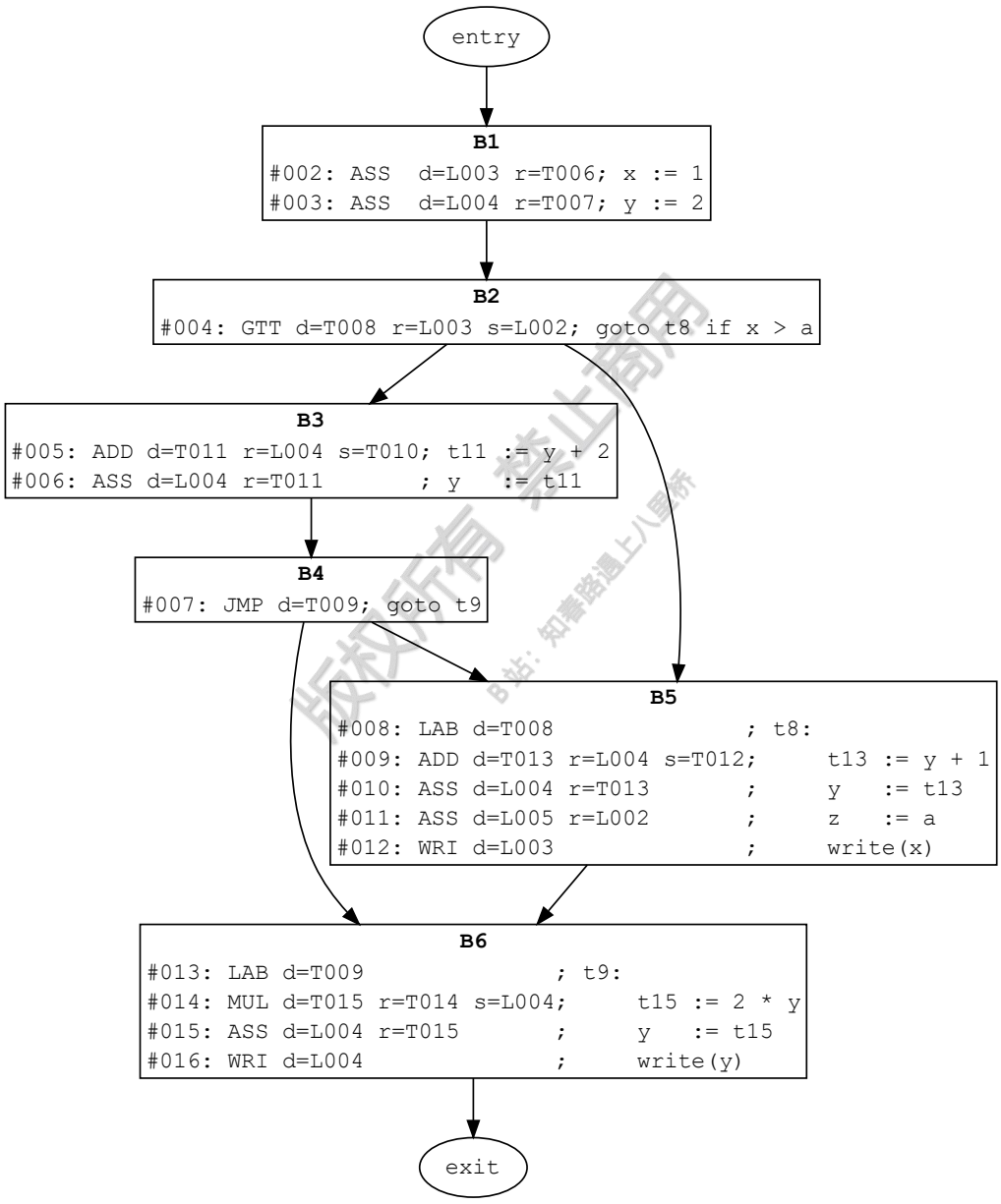


图 7.11: 代码 opt04.pas 构成的流图

表 7.2: 活跃变量迭代求解算法计算结果

基本块 \ 轮次	第 1 轮	第 2 轮		第 3 轮		第 4 轮	
	$\mathsf{IN}[B]^0$	$\mathsf{OUT}[B]^1$	$\mathsf{IN}[B]^1$	$\mathsf{OUT}[B]^2$	$\mathsf{IN}[B]^2$	$\mathsf{OUT}[B]^3$	$\mathsf{IN}[B]^3$
B_1	000000	000000	000000	100000	000000	110000	000000
B_2	000000	000000	100000	010000	110000	010000	110000
B_3	000000	000000	010000	000000	010000	010000	010000
B_4	000000	000000	000000	010000	010000	010000	010000
B_5	000000	000000	010000	010000	010000	010000	010000
B_6	000000	000000	010000	000000	010000	000000	010000

- $\mathsf{IN}_{B_4} = \{y\}$, $\mathsf{OUT}_{B_4} = \{y\}$
- $\mathsf{IN}_{B_5} = \{y\}$, $\mathsf{OUT}_{B_5} = \{y\}$
- $\mathsf{IN}_{B_6} = \{y\}$, $\mathsf{OUT}_{B_6} = \emptyset$

这里观察基本块 B_5 中的 IN_{B_5} 和 OUT_{B_5} 都是 $\{y\}$，说明该基本块中只有变量 y 是 “活跃” 的，所以四元式 `#011: ASS d=L005 r=L002; z := a` 其实是一个无效赋值，最终可以进行无效代码移除。

7.5 思考题

1. 对比下列三地址码 (7.24) - (7.28) 序列和修改后的 SSA 形式 (7.29) - (7.33) 序列，通过构造 DAG 图来体会 SSA 的优点。

   普通赋值三地址码序列

$$p \leftarrow a + b \tag{7.24}$$
$$q \leftarrow p - c \tag{7.25}$$
$$p \leftarrow q * d \tag{7.26}$$
$$p \leftarrow e - p \tag{7.27}$$
$$q \leftarrow p + q \tag{7.28}$$

   SSA 形式赋值三地址码序列

$$p_1 \leftarrow a + b \tag{7.29}$$
$$q_1 \leftarrow p_1 - c \tag{7.30}$$
$$p_2 \leftarrow q_1 * d \tag{7.31}$$
$$p_3 \leftarrow e - p_2 \tag{7.32}$$
$$q_2 \leftarrow p_3 + q_1 \tag{7.33}$$

7.6 本章总结

本章介绍了编译器代码优化的理论，具体涉及到基本块划分，构建流图。然后介绍了基本块内的优化：DAG 图消除公共表达式等。最后介绍了全局优化：数据流分析中的到达定值分析和活跃变量分析的细节。

第八章　附录

8.1　GCC 编译套件

GCC [1] 是 GNU [2] 编译器套件，是由 GNU 自由软件基金会开发的编程语言编译器。GCC 的初衷是为 GNU 操作系统专门编写的一款编译器。最初支持 C 语言的编译，后续它陆续支持 C++、Objective-C、Fortran、Java、Ada 和 Go 等编程语言，也包括了这些语言的库（如 libstdc++, libgcj 等）。

cpp、cc1 [3]、as 和 ld 是完成编译的重要工具。cpp 和 cc1 是 GCC 编译器套件中的组件，as 和 ld 属于 binutils [4] 工具集的组件。它们各自具有特定的功能，协同完成编译任务。

1. cpp：cpp 是预处理器，它的主要任务是在编译过程的开始阶段对源代码进行预处理。预处理器负责处理源代码中的预处理指令，例如宏定义、条件编译等。在预处理之后，源代码会被转换为纯 C 代码，然后传递给后续的编译阶段。
2. cc1：cc1 是 GCC 编译器集合中的一个主要编译程序，它负责将预处理后的 C 源代码编译成汇编代码。cc1 将 C 代码翻译成汇编语言，然后生成一个汇编语言文件。这个过程涉及将源代码转换为机器语言的中间表示形式，以便后续的链接器能够将它们链接成可执行文件。
3. as：as 是汇编器，它的任务是将汇编语言代码转换为机器语言代码。as 将汇编语言文件作为输入，并生成一个目标文件（.o 文件）。这个目标文件包含了机器语言的二进制代码，但还没有链接到其他目标文件或库文件。
4. ld：ld 是链接器，它的任务是将多个目标文件和库文件链接成一个可执行文件。ld 将目标文件和库文件作为输入，并生成一个可执行文件。这个可执行文件包含了所有必要的代码和数据，可以在特定的操作系统和硬件上运行。

这些工具在编译过程中协同工作，从源代码开始，经过预处理、编译、汇编和链接等步骤，最终生成可执行文件。每个工具都有其特定的功能和输入输出格式，以确保编译过程的正确性和高效性。

图 8.1 展示了 gcc 构建 hello.c 源文件的过程，由于实验环境之间的差异性可能导致部分结果有偏差，这里笔者的实验环境中 gcc 的版本信息如下：

```
$ gcc --version
gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
```

[1] GCC 官网 https://gcc.gnu.org/

[2] GNU 软件集 https://www.gnu.org/software/software.html

[3] 实际操作系统中 cc1 是编译器的前端，这里先使用它代指编译器

[4] binutils https://www.gnu.org/software/binutils/

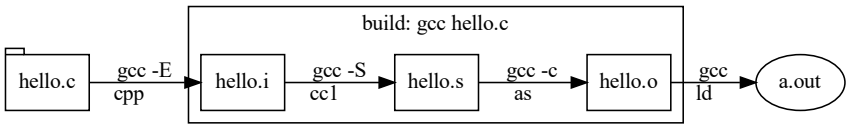


图 8.1: GCC 构建场景流程图

```
Copyright (C) 2021 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

$
```

有了实验环境，我们在结合具体构建实例来说明 gcc 处理细节。首先我们编写一个 hello.c 文件，其内容如下：

```
#include<stdio.h>

#define INIT_VAL 123

int main (int argc, char *argv[])
{
        int val = INIT_VAL;
        printf("Hello, val = %d\n", val);
        return 0;
}
```

预处理部分由 cpp 工具支撑，这里可以使用 gcc -E 命令来唤醒 cpp 做预处理，并输出中间文件。其命令行如下：

```
# 预处理 cpp -E -o hello.i
gcc -E -o hello.i hello.c
```

得到的预处理的输出文件为 hello.i，可以看出在预处理后的文件中，#define 的宏定义 INIT_VAL 会被替换成数字 123 ，这就是预处理的核心操作。

```
# 0 "hello.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 0 "<command-line>" 2
# 1 "hello.c"
# 1 "/usr/include/stdio.h" 1 3 4
```

```
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/libc-header-start.h" 1 3 4

typedef unsigned char __u_char;
typedef unsigned short int __u_short;

... 此处删除几百行

# 885 "/usr/include/stdio.h" 3 4
extern int __uflow (FILE *);
extern int __overflow (FILE *, int);
# 902 "/usr/include/stdio.h" 3 4

# 2 "hello.c" 2

# 5 "hello.c"
int main (int argc, char *argv[])
{
 int val = 123;
 printf("Hello, val = %d\n", val);
 return 0;
}
```


接下来可以调用 cc1 编译器工具来对 hello.i 进行编译，这里使用 gcc -S 命令来唤醒 cc1 编译器进行编译，-S 选项指定只生成汇编文件，这样我们可以得到 hello.s 文件

```
# 编译 cc1 -fpreprocessed -o hello.s hello.i
gcc -S -o hello.s hello.i
```

我们可以打开 hello.s 文件查看其内容如下。在笔者实验机器上生成了 x86 的汇编指令，它包含数据段和代码段，这就是编译器的主要工作。

```
        .file   "hello.c"
        .text
        .section        .rodata
.LC0:
        .string "Hello, val = %d\n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        endbr64
```

```
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $32, %rsp
        movl    %edi, -20(%rbp)
        movq    %rsi, -32(%rbp)
        movl    $123, -4(%rbp)
        movl    -4(%rbp), %eax
        movl    %eax, %esi
        leaq    .LC0(%rip), %rax
        movq    %rax, %rdi
        movl    $0, %eax
        call    printf@PLT
        movl    $0, %eax
        leave
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0"
        .section    .note.GNU-stack,"",@progbits
        .section    .note.gnu.property,"a"
        .align 8
        .long   1f - 0f
        .long   4f - 1f
        .long   5
0:
        .string  "GNU"
1:
        .align 8
        .long   0xc0000002
        .long   3f - 2f
2:
        .long   0x3
3:
        .align 8
4:
```


接下来是汇编器的处理流程，可以通过 gcc -c 来唤醒 as 汇编器将 hello.s 文件汇编成二进制目标文件 hello.o，其中 -c 表示只进行编译，不进行链接操作。这里使用的命令行如下：

```
# 汇编器 as -o hello.o hello.s
gcc -c -o hello.o hello.s
```

hello.o 文件是对 hello.s 的编译的产物，在实际的源文件 hello.c 还调用了 printf(...) 函数，但是 printf(...) 在 hello.c 中并没有直接定义。根据 c 语言知识我们可以知道 printf(...) 函数作为库函数是定义在 glibc 中，通过 #include <stdio.h> 导入到 hello.c 中，所以在实际的编译最后一步还需要对 hello.o 文件和 glibc 库进行链接。具体的链接工作是由连接器 ld 实现的，这里调用 gcc 来唤醒 ld 进行链接操作，具体命令行如下：

```
# 链接 ld -o a.out hello.o
gcc -o a.out hello.o
```

最后得到的产物是 a.out，通过命令运行得到最终输出结果如下。

```
$ ./a.out
Hello, val = 123
$
```

8.2 FPC 编译器

FPC [5] 编译器是一种免费的 Pascal 编译器，全称为 Free Pascal Compiler。它具有高度的兼容性和跨平台能力，支持 Windows、Linux、Mac OS X 等多种操作系统，并可编译多种架构的 CPU，包括 x86、x86-64、ARM 等。FPC 编译器采用了 GNU GPL 协议，是一款免费的 Pascal 编译器。

FPC 编译器主要用于编译和执行使用 Pascal 编程语言编写的程序。Pascal 语言是一种结构化编程语言，具有简洁明了的语法和丰富的数据类型和控制结构，适用于各种应用领域的程序开发。FPC 编译器的目标是生成高效、可靠和可移植的目标代码，使得使用 Pascal 语言编写的程序能够在不同的平台上运行，并且具有较好的性能和可扩展性。

在 Ubuntu LTS 22.04 操作系统中可以通过 apt 包管理器安装 FPC 编译器，安装命令如下

```
apt install fpc
```

编写一个简单的 hello.pas 源代码文件，测试 FPC 编译器是否可以正确工作

```
1 { simple hello world program }
2 begin
3     writeln('Hello from Pascal!');
4 end
5 .
```

上述代码的功能是向标准输出打印一个字符串，其中第 1 行是对程序功能的注释，第 3 行是调用函数向标准输出打印字符串，需要注意，第 5 行的 **点符号** . 标记程序结束。

使用 FPC 编译器编译命令如下

[5] free pascal 官网 https://www.freepascal.org/

```
fpc hello.pas
```

通过动手编译并实际编译并运行的例子

```
$ fpc hello.pas
Free Pascal Compiler version 3.2.2+dfsg-9ubuntu1 [2022/04/11] for x86_64
Copyright (c) 1993-2021 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling hello.pas
Linking hello
4 lines compiled, 0.1 sec
$ ./hello
Hello from Pascal!
$
```


8.3 NASM 汇编器

NASM [6] 是一个很好用的汇编器，全称 Netwide Assembler，是一款基于 x86 架构的汇编与反汇编软件。它支持 16 位、32 位和 64 位的程序编写，并能输出多种二进制格式，包括通用对象文件格式 (COFF)、OMF (Relocatable Object Module Format，用于 80x86 系列处理器上)、a.out、可执行与可链接格式 (ELF)、Mach-O、二进制文件 (.bin，二进制磁盘映像，用于编译操作系统) 等。

NASM 也有自己的二进制格式，称为 RDOFF (Relocatable Dynamic Object File Format)。此外，NASM 还可以作为交叉汇编程序在非 x86 架构 (如 PowerPC 和 SPARC) 上运行，尽管它不能生成这些架构的处理器可用的程序。在语法格式上，NASM 风格的汇编语言与其他风格的汇编语言 (如 AT&T 风格和 Windows 风格) 会有所不同。它的安装方式如下

```
apt-get install nasm
```

我们可以使用它编写简单的汇编代码，创建 hello.s 文件，并编写如下内容的汇编代码

```
1  section .data
2          hello db 'Hello, World!', 10, 0
3
4  section .text
5          global _start
6
7  _start:
8          ; write the string to stdout
9          mov eax, 4          ; system call number for sys_write
10         mov ebx, 1          ; file descriptor 1 is stdout
11         mov ecx, hello      ; pointer to the message
12         mov edx, 14         ; length of the message
```

[6] nasm 官网 https://nasm.us/

```
13          int 0x80            ; interrupt to invoke system call
14
15          ; exit the program
16          mov eax, 1          ; system call number for sys_exit
17          xor ebx, ebx        ; exit code 0
18          int 0x80            ; interrupt to invoke system call
```

上述代码包含 .data 数据段和 .text 代码段，第 7 行的 _start 标记程序入口。第 9 到 13 行先调用 write 系统调用打印 Hello, World! 字符串，然后在 16 到 18 行调用 exit 系统调用退出程序。

nasm 支持命令行编译汇编文件，具体编译方式如下所示：

```
# 编译 compile
nasm -f elf -o hello.o hello.s
# 链接 link
ld -m elf_i386 -o hello.out hello.o
```

相关编译选项解释如下：

1. nasm 选项 -f elf 指定输出为 ELF 格式
2. nasm 选项 -o hello.o 指定编译输出文件名 hello.o
3. ld 选项 -m elf_i386 制定输入文件格式是 32 位 ELF 目标文件
4. ld 选项 -o hello.out 制定输出文件名 hello.out

当得到 hello.out 可执行文件后，可以直接在 Shell 中运行，其运行结果如下

```
$ ./hello.out
Hello, World!
```

第九章　读者列表

本章列举本书的读者或捐赠者，按照接收时间顺序排列。

- 烟 ** 寂寞 11*29@qq.com
- code**jun code**jun@126.com
- void**ptr lr***0@163.com
- m56**7 m56**7@gmail.com
- 安 ** 瓜 57**94@qq.com
- 芒 ** 鬼 14**77@qq.com