Chapter 1 Concepts of compiler

It is necessary for professional people in the field of computer science who understand the principles, techniques and writing of a simple compiler. Compiler writing includes programming languages, machine architecture, language theory, algorithms, as well as software engineering. In this chapter, we introduce the compiler components, the environment and concepts of a simple compiler.

1.1 The concept

Generally, there are two kinds of languages, one is high level language, such as: FORTRAN, Pascal, C, ASL, C++ etc., discussed in this book, it is called source language; the other is low level language, it includes machine language and assembly language, we call it target language.

Conceptually, source program is the program written in a source (high level) language. A target program is the program written in a target (low level) language. Compiler is a program which translates source program into an equivalent target program. A compiler is shown in Fig. 1.1.

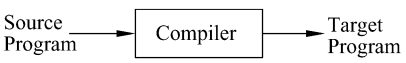

Fig. 1.1 A compiler

We shall use a simple example to explain the compiler. The source program of a compiler is as follows.

$$I := I_0 + L * 2 \tag{1.1}$$

Target program: the output of compiler is an equivalent machine code of its source program. It looks like these.

```
MOVF id3, R2          (1.2)
MULF #2.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

第 1 章 编译程序概述

作为计算机专业人员来说，了解编译程序的原理、结构以及编写简单的编译程序是十分必要的。编译程序涵盖程序设计语言、机器代码、计算机编程语言的理论知识以及算法和软件工程的知识。本章介绍编译程序的组成部分、运行环境和简单的编译程序。

1.1 概念

编译程序(编译器)是语言的一种实现系统，它将用高级语言或汇编语言编写的程序转换成等价的机器语言程序，因此编译程序是高级语言程序到某种低级语言程序的转换器。解释程序(解释器)也是语言的一种实现系统，解释程序的工作结果是得到源程序的执行结果，因此，解释器是源程序的一个执行系统，而编译程序是程序的一个转换系统。

图 1.1 一个编译器

[重点词汇]

high level language：高级语言
source language：源语言
low level language：低级语言
machine language：机器语言
assembly language：汇编语言
source program：源程序
target program ：目标程序
compiler：编译器
Interpreter：解释程序

1.2 Analysis of the source program

Generally, the analysis of compiler includes six phases, they are Lexical Analyzer, Syntax Analyzer, Semantic Analyzer, Intermediate Code Generator, Code Optimizer and Code Generator. Every phase transforms the source program from one form presentation to another, and during the transformation it communicates with error handlers and symbol table.

In lexical analysis, characters are grouped into tokens; syntax analyzer changes a group of token into grammatical phrases, it is often called parsing tree; in semantic analyzer, grammatical phrases are checked by semantic errors and type information are added; the intermediate code generator is a program which is easy to produce program from semantic analyzer and is easy to translate into the target program; code optimization phase attempts to improve the intermediate code; a intermediate code instructions are each translated into a sequence of machine instructions, it is the task of code generator. The phases of compiler are shown by Fig. 1.2.

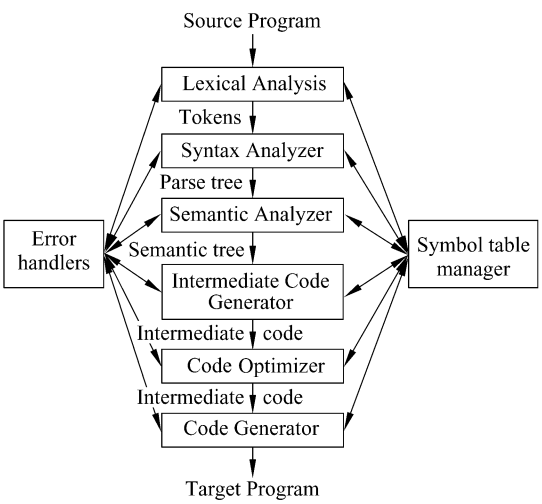

Fig. 1.2 Phases of Compiler

1.2.1 Lexical analysis

Lexical analysis is also called scanner. It is a program which recognizes patterns in text. Scanners may be hand

1.2 源程序的分析

源程序是以字符串形式存在的文本文件，目标程序是根据目标机特点从中间代码产生高质量机器代码。即词法分析部分的主要任务是检查词法错误并把源程序中的字符转换成一种内部形式(数据形式——单词)储存在符号表中；语法分析的任务是检查源程序的语法错误，当发现错误时输出一些信息，并尽可能地继续检查；语义分析是检查数据类型是否正确等语义错误；中间代码是源程序的一种便于优化和便于产生目标代码的内部表示；中间代码优化任务是进行不依赖于目标机的优化，以产生高质量目标代码。所有编译过程都需要符号表和错误表的参与来完成其相应功能。

[重点词汇]

Lexical Analysis：词法分析
Syntax Analyzer：语法分析
Semantic Analyzer：语义分析
Intermediate Code Generator：中间代码生成
Code Optimizer：中间代码优化
Code Generator：目标代码生成
Phase：阶段
Tokens：字符
Parsing tree：语法分析树

图 1.2 编译器的组成

1.2.1 词法分析器

[重点词汇]

Scanner：扫描器

written or be automatically generated by a lexical analysis generator from descriptions of the patterns to be recognized.

The example is an assignment statement:

$$I := I_0 + L * 2$$

Lexical analyzer can analysis it into a group of tokens:

"I", " :=","I_0", "+","L", " * ","2".

Namely, they are the identifier, the assignment symbol, the identifier, the plus, the identifier, the multiplication and the digital.

Note: During the lexical analysis, the blanks which separate the characters of these tokens would normally be eliminated.

1.2.2 Syntax Analyzer

Syntax Analyzer is also called parser. It groups tokens of the source program into grammatical phrases; what's more, it is also a program which determines if its input is syntactically valid and determines its structure. Parsers may be hand written or may be automatically generated by a parser generator from descriptions of valid syntactical structures.

Usually, the grammatical phrases of the source program are represented by a parsing tree. For example:

$$I := I_0 + L * 2$$

the parsing tree of it is shown by Fig. 1.3.

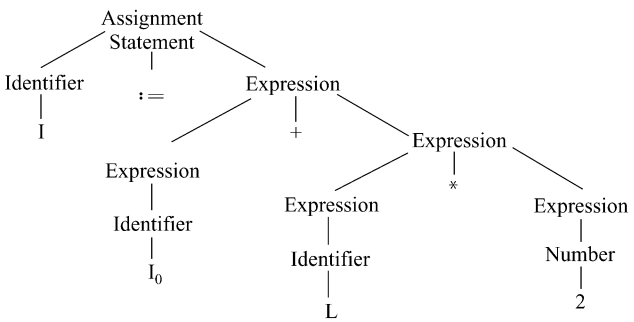

Fig. 1.3 Parsing tree for $I := I_0 + L * 2$

From this parsing tree, we can see the grammatical phrases include expressions and statements. The definition of expression is as follows:

Assignment statement: 赋值语句
Identifier: 标识符，变量名
Plus: 加号
Multiplication: 乘号
Digital: 数字
Blank: 空格

词法分析器又可称作扫描器。它的主要作用是将输入序列符号区分为不同的单词。

1.2.2 语法分析器

语法分析器是将词法分析器识别出的单词划分成不同的语法成分，最终形成一棵树，我们称之为语法树。

例如表达式 $I := I_0 + L * 2$，通过词法分析被分成一个个单词，然后在语法分析器中确定其语法成分，最后形成如图 1.3 所示的语法树。其中，I，I_0，L 为标识符，2 为数字，"+"，" * "连接两个表达式，" :="形成语句。

从图 1.3 的语法树中，我们能看出语法树包括两种语法成分：表达式和句子。

图 1.3 表达式 $I := I_0 + L * 2$ 的语法树

1. Any an identifier is an expression, such as I, L, I_0

2. Any a digital is an expression, such as 2

3. If expression1 and expression2 are expressions, then

expression1 * expression2 is an expression, such as L * 2

expression1 + expression2 is an expression, such as I_0 + L * 2

表达式的定义有3条。1. 标识符是表达式。2. 数字是表达式。3. 表达式的算术运算是表达式。

On the other hand, we can similarly define statement recursively. That is:

1. If identifier1 is an identifier, and expression2 is an expression, then

identifier1 := expression2

2. If expression1 is an expression and expression2 is an expression, so

While (expression1) do statement2

If (expression1), then statement2

句子的定义：1. 通过赋值号相连的表达式和标识符是句子。2. 循环语句和条件语句都是句子。

Sometimes, we can compress the parsing tree into syntax tree, where the operators appear as the interior nodes, and the operands of an operator are the operator's children, shown in Fig. 1.4. We shall discuss this in more detail in Chapter 4.

有时语法树又可以表示成如图1.4所示的形式。运算符是双亲节点，运算对象是孩子节点。

[重点词汇]

Operator：运算符

Operand：运算对象

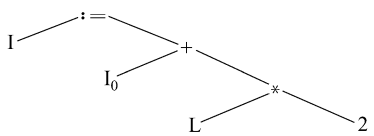

Fig. 1.4 Syntax tree for I := I_0 + L * 2

图1.4 另一种形式的语法树 I := I_0 + L * 2

1.2.3 Semantic Analyzer

The semantic analysis analyzes the parsing tree for semantic errors, gathers information type and checks information type. The syntax tree is used to identify if the operators and operands are correct. In addition, many program languages need to report and correct an error type, this need to store the name and type of variables and other objects in a symbol table. The information type can be checked by means of the symbol table. The output of the semantic analysis phase is an annotated parsing tree, i. e. adding the type of

1.2.3 语义分析

语义分析是在已形成的语法树的叶子上加数据的属性类型，从而形成一棵语义树。

objects based on parsing tree, we name it as semantic tree. This phase of semantic analysis is often combined with the parser. Attribute grammars are used to describe the static semantics of a program. For example, suppose we have declared all identifiers as real type shown in Fig. 1.5, and digital 2 is an integer. Firstly, semantic analyzer gathers and stores the type information of all identifiers and digitals in a symbol table; then checks the type of them, it reveals that * multiplies a real(L) by an integer(2); so the semantic analyzer creates a new node "real"; finally, it converts the digital 2 into a real type and builds a semantic tree.

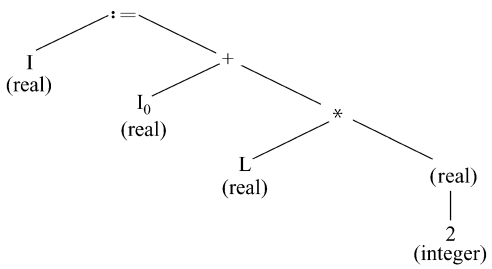

Fig. 1.5 Semantic tree

语义分析的功能是检查语义树上的叶子节点的属性类型是否正确,并将它们存储在符号表中。例如图 1.5 中叶子节点 2 是整数类型,而与之进行"*"运算的运算对象是实数类型,二者不相同;所以语义分析就将节点 2 的整数类型转成实数类型,从而形成一棵语义正确的语义树。

图 1.5 语义树

[重点词汇]

Annotate: 注释
Semantic tree: 语义树

1.2.4 Intermediate Code Generator

When compiler analysis reaches the phase of Intermediate Code Generator, compiler has analyzed source language into a series of tokens, and built a parse or syntax tree and semantic tree, stored information type in symbol table and generated the error information. In order to obtain machine code of source program, some compilers generate an intermediate representation, it's a program for an abstract machine. The phase of creating the intermediate representation is called intermediate code generator. The function of intermediate code generator is for easy producing and translating the source program into the target program.

There are several forms of intermediate code, the typical one is "three-address code", which is similar to the assembly language. Three-address code is a sequence of instructions;

1.2.4 中间代码生成器

中间代码是在语义分析的基础上为便于生成目标代码而生成的一种代码表示形式。

中间代码有多种表示形式,代表性的中间代码是三元组码。这种代码近似于汇编语言。三元组码由三个部分组成。下面的三元组码是三元组码多种表示中的一种,赋值号右边由一个运算符和两个运算对象组成。

each having at most three parts. We shall explain three-address code and other intermediate exact representation in Chapter 7.

The source program in (1.1) can be written in three-address code:

$$\begin{aligned} &\text{temp1} := \text{real}(2) \\ &\text{temp2} := \text{L} * \text{temp1} \\ &\text{temp3} := \text{I}_0 + \text{temp2} \\ &\text{I} := \text{temp3} \end{aligned} \tag{1.3}$$

1.2.5 Code Optimizer

Intermediate code is not a faster-running code, so a code optimizer can improve it. For example we can change the Intermediate code (1.3) into code optimization (1.4).

$$\begin{aligned} &\text{temp1} := \text{L} * 2 \\ &\text{I} := \text{I}_0 + \text{temp1} \end{aligned} \tag{1.4}$$

This means that code optimizer can decrease the number of instructions and increase the running speed of the target program. There are many type of optimizers, these are covered in Chapter 8.

1.2.6 Code Generator

The function of this phase is to create target code. Target code means machine code or assembly code. The feature of machine code is that it needs memory location for each of the variables used by the program, and register of the assignment of variables. The code of (1.4) might be translated into a series of machine instructions, such as

```
MOVF   L,    R2
MULF  #2.0,  R2
MOVF   I0,   R1
ADDF   R2,   R1
MOVF   R1,   I
```

(1.5)

The machine instructions mean

(1) The F in above instructions (1.5) means that the digital is the type of floating point number.

(2) The function of MOV is to put the contents of the address L into register 2.

(3) MUL signifies multiplication of R2 by 2.0, and then sends the result to R2. ADD presents adding R2 and R1

[重点词汇]

Assembly language：汇编语言
Three-address code：三元组

1.2.5 代码优化

代码优化是为了生成更简化的中间代码。

例如式1.3可优化成式1.4的简化形式。

1.2.6 代码生成

代码生成是在中间代码的基础上转化出目标代码。

如式1.4转化成机器代码如式1.5所示。

(1) F表示式1.5中的数字类型为浮点型。

(2) MOV是将地址L中的内容存储在寄存器2中。

(3) MUL是将寄存器2中的值乘以2再送回寄存器2中。

(4) #表示2是一个常量。

together, and then stores the result in R1.

(4) # means 2.0 is a constant.

Chapter 9 gives a detailed discussion of code generation. In the above phase of compiler, there are two important parts in compiler, they are symbol table and error table, the details are as follows.

1.2.7 Error handlers

There are many error information found and need to be corrected in the phase of compiler. For example, in lexical phase some characters can't be formed into any token of a language. During syntax phase, there are some errors that do not abide by any syntax structure rules of a language. In semantic phase, some errors appear in incorrect type on both sides of assignment. For example, on the one hand, the type of variable in assignment is integer and on the other hand, the constant is the type of float. It's right in syntactic structure, but it is incorrect in semantic meaning. So we need the error handlers in every phase of compiler.

1.2.8 Symbol table

After the analysis of lexical of compiler, source program is turned into tokens and is prepared for being analyzed by next phase. The point is that these tokens should be stored in some place for use at any time. Where to store these tokens?

Symbol table is a data structure or a database. It has two functions, the first one is the storage function, it stores the information of token, such as the name, type and other character of identifier; the second function is to check or retrieve this information, for example, in the phase of semantic analysis and intermediate code generation, it needs to check the type of identifier and to generate proper operations.

In addition, the actions of storing some information in symbol table and checking the information in some phase exist in all the phases of compiler. It is covered in detail in Chapter 5.

1.3 Conclusion

This section further discusses some compiler concepts. It is not necessary for all the compilers to consist of all six

1.2.7 出错表

出错表始终伴随在编译分析的各个阶段,记录每个阶段所产生的出错信息。

例如在词法分析阶段,出错表记录一些不能形成单词的字符。语法分析中,出错表记录不符合语法规则的错误。语义分析中,出错表记录运算中数据类型不一致的问题。

1.2.8 符号表

符号表同样伴随编译分析的各个阶段,记录存储每段分析的数据,以备下一步分析使用。

符号表是一种数据存储结构或者是一种数据库。它有两个功能,一个是存储数据信息,例如名字,类型等;另一个是检查和检索信息的功能。例如在语义分析阶段,检查两种数据类型是否一致。

1.3 总结

这部分主要总结编译分析的各个阶段,以及每一阶段在编译器中所起的

phase, some compilers only have three phases. In this book, we mainly introduce the common compiler structure (six phases) shown in Fig. 1.6. Every phase of a compiler will be discussed in detail in the following chapters.

Compilers are not particularly difficult programs to understand once you are familiar with the structure of a compiler in a general way. The point is that not any of the phases of a compiler is hard to understand; but, there are so many phases that you need to absorb and make sense of them. Table 1.1 is the description of each of the compiler phases.

Table 1.1 Description of compiler phases

Phase	Description
Lexical analysis	Turn the source file into a sequence of tokens
Syntax Analyzer	Analyze the phrase structure of the program and build a syntax tree
Semantic Analyzer	Analyze the type of each expression and build a semantic tree
Intermediate Code Generator	A tie of source program and target program, produce the intermediate code
Code Optimizer	Optimize the intermediate code
Code Generator	Produce the target code

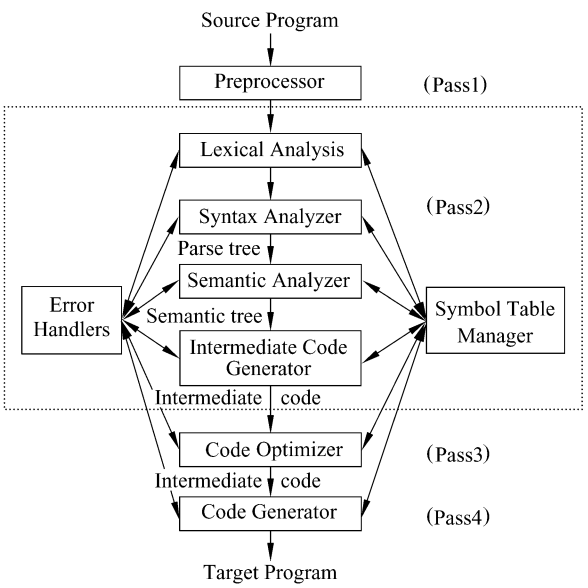

Fig. 1.6 Phases of a compiler

作用。

图 1.6 表示的是编译过程的六个阶段。每个编译阶段的详细内容将在后面各章节进行介绍。表 1.1 是对各编译阶段的功能进行的简单说明。

表 1.1 编译器组成的说明

词法分析的作用是将源程序分成一系列的单词串。

语法分析是用来分析各单词串的语法结构进而形成一棵语法树。

语义分析是在语法树上加上数据类型信息。

中间代码是介于源程序和目标程序之间的一种表示形式。

代码优化是对中间代码化简的过程。

代码生成是用来产生目标代码的阶段。

图 1.6 四遍编译器的组成

1.4 The pass of compiler

Compiler is a complex program. When a source program is compiled, it often needs several passes to finish all phases of compiler. So we name the distinct chunk pass. It is a part of the compilation process and it communicates with one another via temporary file. The typical structure is a four-pass compiler, it is shown in Fig. 1.6.

The first pass is preprocessor. Its first task is to strip comments from the source code, such as {,} or begin, end. Second task is to handle various housekeeping tasks with which you don't want to burden the compiler proper, for example, the housekeeping is # include < global. h > in the source program language C. The second pass is the heart of compiler, it consist of lexical analysis, parser, semantic analyzer, and intermediate code generator. The input is source language, and output is intermediate language. The third pass is the optimizer, which improves the quality of the intermediate code. Finally, the fourth pass, it translates the optimizer code into real assembly language or some other form of binary, executable code.

Note: There are many different passes of a compiler. Not all compilers have four passes. Some have two passes, others generate assembly language in the third pass, and some compiler only has one pass. Many compilers do not use preprocessors, or have intermediate language, but generate the machine code directly.

1.5 Compiler example 1

So far we have described all the compiler phases and some theory knowledge about it. But, what is a compiler program? How to build a compiler from a simple expression? We give some parts of a typical compiler program and explain the compiler phase that consists of lexical analysis (Lex), parser analyzer (Yacc) and code generation (ANSI C code generator).

Lex and Yacc can generate program fragments that solve the task of reading the source program and discovering its

1.4 编译器的遍

前面介绍的编译器结构是按功能结构方式组织起来的功能结构。一个具体的编译器体系还可能是按形式来组织起来，例如按编译器扫描的遍数，把编译器分为一遍扫描和多遍扫描的编译器(包括两遍、三遍和四遍扫描编译器)。一遍的编译器是通过一遍扫描直接从源程序生成目标代码，而多遍的编译器则通过多遍扫描完成各阶段的编译任务产生目标代码。例如四遍编译器，第一遍扫描进行预处理，第二遍进行词法分析、语法分析、语义分析和中间代码生成，第三遍是中间代码优化，而第四遍扫描则从中间代码产生目标代码。当然这些过程也可以是两遍或三遍的编译器来完成。

典型四遍编译器参见图 1.6。

1.5 实例 1——编译器程序

本节通过 Jeremy Bennett 提供的一个编译程序实例，具体描述词法分析器、语法分析器以及代码生成器，使读者对编译器有一个全面的认识。

structure.

Lex is well suited for segmenting input in preparation for a parsing routine and helps write programs whose control flow is directed by instances of regular expressions in the input stream. Lex table made up of regular expressions and corresponding program fragments is translated to a program that reads an input stream, copy it to an output stream and partition the input into strings that match the given expressions. What's more, Lex can generate the recognition of the expressions and strings to corresponding program fragments that are executed.

The Yacc specifies the structures of his input and recognize each structure. Yacc changes such a specification into a subroutine that handles the input process; usually, it uses the subroutine to make it convenient and appropriate to handle the flow of control.

Note: The following compiler code is provided by Jeremy Bennett and he has permitted to add his code to this book. We sincerely thank him for his kind support. If you wish to read the complete compiler source code, please access the Jeremy Bennett's website at http://www.jeremybennett.com/.

1.5.1 The lexical analysis

Let us first discuss the concept of token. Token is an input symbol, which is used both for digitals and identifiers; so the tokens are made up of a sequence of characters in the range '0-9', 'a-z', 'A-Z'. The following lexical program is only a part of the whole lexical analysis. It starts from input system, getting characters from standard input, and then isolating tokens. The detailed lexical analysis is as follows.

It starts from the definition of syntax analyzer, which is to obtain the definition of type ctype.h and of the routine parse program.

scanner.c (LEX scanner for the vc compiler)

```
#include <stdio.h>
#include <ctype.h>
#include "vc.h"
#include "parser.h"
```

词法分析器－Lex，能将输入信息分割成一个个单词。词法分析器的符号表能将单词存储起来。

语法分析器－Yacc，是用来分析单词的语法结构，找出它们间的语法关系，为进一步的分析做准备。

1.5.1 词法分析器

该词法分析主要用来识别0～9数字和26个大小写字母，是词法分析器的一个部分。

一般来说，源程序主要由字母和数字组成，因此，如果字母和数字能够被识别出来，绝大部分的源程序就能够被识别。

所以，该词法分析器Lex，程序名：scanner.c是较为实用的词法分析器。

```
/* Routines are defined here. These are in the code section below
and build symbol table entries for variable names, integer con-
stants and text strings respectively. */

void mkname(char *name);
void mkval(void);
void mktext(void);

do_action:                    /* this label is used only to access
                                 EOF actions */

switch (yy_act)
{
  case 0:                     /* must backtrack */
                              /* undo the effects of */
                              /* YY_DO_BEFORE_ACTION */
    *yy_cp = yy_hold_char;
    yy_cp = yy_last_accepting_cpos;
    yy_current_state = yy_last_accepting_state;
    goto yy_find_action;
  case 1:
    #line 91 "scanner.l"
    {                         /* Ignore comments */ }
    YY_BREAK
  case 2:
    #line 92 "scanner.l"
    {                         /* Ignore whitespace */ }
    YY_BREAK
  case 3:
    #line 93 "scanner.l"
    { mkname(yytext);   /* Save the variable name */
    return VARIABLE;    /* A variable name */ }
    YY_BREAK
  case 4:
    #line 95 "scanner.l"
    { mkval();                /* Save the integer value */
    return INTEGER;           /* A number */ }
    YY_BREAK
  case 5:
    #line 97 "scanner.l"
    { mktext();               /* Save the text string */
    return TEXT;              /* A string */ }
    YY_BREAK
```

该词法分析器的功能主要是将变量名、常量和字符串区分出来，并分别用 mkname，mkval 和 mktext 存到符号表中。整个词法分析器通过情况判断语句(条件判断语句)来实现该功能。

情况 0：回溯，撤销指令。

情况 1：去掉注释语句。

情况 2：去掉空格。

情况 3：存储变量名。

情况 4：存储数值。

情况 5：存储文本字符串。

```
case 6:
    # line 99 "scanner.l"
    {return ASSIGN_SYMBOL;/* ':=' */}
    YY_BREAK
case 7:
    # line 100 "scanner.l"
    {return FUNC;          /* 'FUNC' */}
    YY_BREAK
case 8:
    # line 101 "scanner.l"
    {return PRINT;         /* 'PRINT' */}
    YY_BREAK
case 9:
    # line 102 "scanner.l"
    {return RETURN;        /* 'RETURN' */}
    YY_BREAK
case 10:
    # line 103 "scanner.l"
    {return CONTINUE;      /* 'CONTINUE' */}
    YY_BREAK
case 11:
    # line 104 "scanner.l"
    {return IF;            /* 'IF' */}
    YY_BREAK
case 12:
    # line 105 "scanner.l"
    {return THEN;          /* 'THEN' */}
    YY_BREAK
case 13:
    # line 106 "scanner.l"
    {return ELSE;          /* 'ELSE' */}
    YY_BREAK
case 14:
    # line 107 "scanner.l"
    {return FI;            /* 'FI' */}
    YY_BREAK
case 15:
    # line 108 "scanner.l"
    {return WHILE;         /* 'WHILE' */}
    YY_BREAK
case 16:
    # line 109 "scanner.l"
    {return DO;            /* 'DO' */}
```

情况 6：判断出赋值语句。

情况 7：判断是否是函数。

情况 8：判断是否是打印语句。

情况 9：判断返回语句。

情况 10：判断继续进行语句。

情况 11：判断条件语句 IF。

情况 12：判断条件语句 THEN。

情况 13：判断条件语句 ELSE。

情况 14：判断 FI。

情况 15：判断循环语句 WHILE。

情况 16：判断循环语句 DO。

```
    YY_BREAK
  case 17:
    # line 110 "scanner.l"
    { return DONE ;      /* 'DONE' */ }
    YY_BREAK
  case 18:
    # line 111 "scanner.l"
    { return VAR ;       /* 'VAR' */ }
    YY_BREAK
  case 19:
    # line 112 "scanner.l"
    { return yytext[0] ;    /* Single character operator */ }
    YY_BREAK
  case 20:
    # line 114 "scanner.l"
    ECHO;
    YY_BREAK
case YY_STATE_EOF(INITIAL):
    yyterminate();
      case YY_END_OF_BUFFER:
      {       /* amount of text matched not including the EOB
                 char */
        int yy_amount_of_matched_text = yy_cp - yytext - 1;
                /* undo the effects of YY_DO_BEFORE_AC-
                   TION */
        *yy_cp = yy_hold_char;
/* note that here we test for yy_c_buf_p "<=" to the
position of the first EOB in the buffer, since yy_c_buf_p will
already have been incremented past the NUL character
(since all states make transitions on EOB to the end-of-buffer
state).  Contrast this with the test in yyinput().  */
            if (yy_c_buf_p <= &yy_current_buffer
                                  ->yy_ch_buf[yy_n_chars])
                /* this was really a NUL  */
      {
        yy_state_type= yy_next_state;
        yy_c_buf_p = yytext + yy_amount_of_matched_text;
        yy_current_state = yy_get_previous_state();

/* okay, we're now positioned to make the NUL transition.  */
/* We couldn't have yy_get_previous_state() go ahead and do it
for us because it doesn't know how to deal with the possibility of
jamming (and we don't want to build jamming into it because then
```

情况 17：判断 DONE。

情况 18：判断 VAR 变量。

情况 19：判断单个符号。

情况 20：结束。

判断是否所有输入信息都被识别了。

识别后将相应的信息存储起来。

当出现识别问题时返回到上一个识别状态，再继续进行。

```
it will run more slowly) */
            yy_next_state = yy_try_NUL_trans(yy_current_state);
            yy_bp = yytext + YY_MORE_ADJ;
            case EOB_ACT_CONTINUE_SCAN:
            yy_c_buf_p = yytext + yy_amount_of_matched_text;
            yy_current_state = yy_get_previous_state();
            yy_cp = yy_c_buf_p;
            yy_bp = yytext + YY_MORE_ADJ;
            goto yy_match;
            case EOB_ACT_LAST_MATCH:
            yy_c_buf_p =
                &yy_current_buffer->yy_ch_buf[yy_n_chars];
            yy_current_state = yy_get_previous_state();
            yy_cp = yy_c_buf_p;
            yy_bp = yytext + YY_MORE_ADJ;
            goto yy_find_action;
            }
        break;
        }
        default:
          #ifdef FLEX_DEBUG
          printf("action # %d\n", yy_act);
          #endif
          YY_FATAL_ERROR(
          "fatal flex scanner internal error--no action
                        found");
        }
    }
}
```

除上述各种情况外，再出现问题被认为无效。

1.5.2 The Parser

In the parser phase no code is generated, it just analyze the input tokens, i. e. parse the input. Each subroutine corresponds to the left of one sentence and in the original grammar that bears the same name, and it must match the grammar exactly. The following parser part includes the routine to make an IF statement, a WHILE loop and expression node.

parser. c (YACC parser for the vc compiler)

1.5.2 语法分析器

语法分析器主要用来分析条件语句、循环语句和表达式的语法结构。

```
#include <stdio.h>
#include <ctype.h>
#include "yc.h"
TAC *do_if(ENODE *expr,          /* Condition */
           TAC   *stmt)          /* Statement to execute */
/* For convenience we have two routines to handle IF statements, "do_if()" where there is no ELSE part and "do_test()" where there is. We always allocate TAC_LABEL instructions, so that the destinations of all branches will appear as labels in the resulting TAC code. */
{
  TAC *label = mktac(TAC_LABEL, mklabel(next_label++),
                     NULL, NULL);
  TAC *code  = mktac(TAC_IFZ, (SYMB *)label, expr->
                     res, NULL);
  code->prev = expr->tac;
  code       = join_tac(code, stmt);
  label->prev= code;
  free_enode(expr);          /* Expression finished with */
  return label;
}                            /* TAC *do_if(ENODE *ex-
                                pr, TAC *stmt) */
TAC *do_test(ENODE *expr,          /* Condition */
             TAC   *stmt1,         /* THEN part */
             TAC   *stmt2)         /* ELSE part */
/* Construct code for an if statement with else part */
{
  TAC *label1 = mktac(TAC_LABEL, mklabel(next_label++),
                      NULL, NULL);
  TAC *label2 = mktac(TAC_LABEL, mklabel(next_label++),
                      NULL, NULL);
  TAC *code1= mktac(TAC_IFZ, (SYMB *)label1, expr->
                    res, NULL);
  TAC *code2= mktac(TAC_GOTO, (SYMB *)label2,
                    NULL, NULL);
  code1->prev= expr->tac;                /* Join the code */
  code1      = join_tac(code1, stmt1);
  code2->prev = code1;
  label1->prev = code2;
  label1     = join_tac(label1, stmt2);
  label2->prev = label1;
  free_enode(expr);    /* Free the expression */
```

这里分析的条件语句有两种：没有ELSE的条件语句“do_if()”；另一种是有ELSE的条件语句“do_test()”。TAC_LABEL是分支入口的指令。

左边是“do_if()”的条件语句的语法分析程序。

左边是“do_test()”的条件语句的语法分析程序。

```
    return label2 ;
}                                   /* TAC *do_test(ENODE
                                       *expr, TAC stmt1, TAC
                                       *stmt2) */

TAC *do_while(ENODE *expr,      /* Condition */
              TAC   *stmt)      /* Body of loop */

/* Do a WHILE loop. This is the same as an IF statement with a
jump back at the end. We bolt a goto on the end of the statement,
call do_if to construct the code and join the start label right at the
beginning */
{
    TAC *label = mktac(TAC_LABEL, mklabel(next_label++),
                       NULL, NULL);
    TAC *code= mktac(TAC_GOTO, (SYMB *)label,
                       NULL, NULL);
    code->prev = stmt ;         /* Bolt on the goto */
    return join_tac(label, do_if(expr, code)) ;
}                                   /* TAC *do_while(ENODE
                                       *expr,TAC stmt) */

ENODE *mkenode(ENODE *next,SYMB *res, TAC
               *code)

/* The routine to make an expression node. We put this here
rather than with the other utilities in "main.c", since it is only
used in the parser. */
{
    ENODE *expr = get_enode() ;
    expr->next = next ;
    expr->res  = res ;
    expr->tac  = code ;
    return expr ;
}                                   /* ENODE *mkenode(ENODE
                                       *next, SYMB res, TAC
                                       *code) */

void  yyerror(char *str)
                                    /* The Yacc default error han-
                                       dler. This just calls our er-
                                       ror handler */
```

左边是循环语句“do_while”的语法分析程序。

左边是表达式的语法分析程序。

左边是程序出错时进行错误处理的程序。

```
{
    error(str);
}                                    /* void yyerror(char *str) */
```

1.5.3 Code generation

Moving on to the code generation, the goal of this phase is to build a compiler given that the programs in the phase of parser have been strictly recognized without any error, here the error means that the input is an illegal sentence in the grammar. However, before generating code, first you need to build the bar-bones recognizer.

For example:

a := b op c

We need create some temporary variables and maintain internally by the compiler. The step of generating Code is as follows. First, create a mechanism for allocating temporaries. In fact, the temporaries should be recycled, that is they should be reused after they are no longer needed in the current subexpression. Second, determine the name of the temporary that uses arguments and return values. So, this section of code generator consists of generating code for a binary operator, generating code for a copy instruction, generating code for an instruction that loads the argument into a register, and then write it onto the new stack frame, including the standard call sequence and the standard return sequence.

cg.c (ANSI C code generator for the vc compiler)

```
#include <stdio.h>
#include <ctype.h>
#include "vc.h"
void  cg_bin(char *op,                  /* Opcode to use */
             SYMB *a,                   /* Result */
             SYMB *b,                   /* Operands */
             SYMB *c)
                                        /* Generate code for a
                                           binary operator */
```

1.5.3 代码生成器

该代码生成器主要演示了表达式a:=b op c代码生成的全过程。

代码生成的步骤：1. 设置一个临时的缓存区，且该缓存区可重复使用。2. 给缓存区命名，名字包括变量参数和运算结果的值。

因此，总的来说，代码生成器包括二进制运算，生成命令代码和生成控制命令（变量存储，使用变量和变量值的返回）。

```
/* a := b op c  VAM has 2 address opcodes with the result going into the second operand This is a typical code generation functions. We find and load a separate register for each argument, the second argument also being used for the result. We then generate the code for binary operator, updating the register descriptor appropriately. */
{
    int cr = get_rreg(c) ;      /* Result register */
    int br = get_areg(b, cr) ;/* Second argument register */
    printf("        %s  R%u,R%u\n", op, br, cr) ;
                                /* Delete c from the descriptors
                                   and insert a */
    clear_desc(cr) ;
    insert_desc(cr, a, MODIFIED) ;
}
void  cg_copy(SYMB *a,SYMB *b)
/* Generate code for a copy instruction  a := b
   We load b into an register, then update the descriptors to indicate that a is also in that register. We need not do the store until the register is spilled or flushed. */
{
    int  br = get_rreg(b) ;         /* Load b into a register */
    insert_desc(br, a, MODIFIED) ;/* Indicate a is there */

}                                   /* void cg_copy(SYMB *
                                       a,SYMB *b) */

void  cg_cond(char *op,
              SYMB *a,          /* Condition */
              int  l)           /* Branch destination */

/* Generate for "goto", "ifz" or "ifnz". We must spill registers before the branch. In the case of unconditional goto we have no condition, and so "b" is NULL. We set the condition flags if necessary by explicitly loading "a" into a register to ensure the zero flag is set. A better approach would be to keep track of what is in the status register, so saving this load. */
{
    spill_all() ;
    if(a != NULL)
{
        int  r ;
```

为了生成表达式 a := b op c 的代码，首先设置两个寄存器。第二个寄存器又可用来存储运算结果。其次，生成运算符的代码。

生成赋值语句的代码(a := b)。首先将 b 存入寄存器中，然后更新标志，指示 a 也在该寄存器。

生成无条件转移语句的代码，从而确定使用哪个寄存器。

```
    for(r = R_GEN ; r < R_MAX ; r++)
                                /* Is it in reg? */
        if(rdesc[r].name == a)
            break ;
                                /* Bug fix 3/5/91 to reload into
                                   the existing register correct-
                                   ly. */
        if(r < R_MAX)
                                /* Reload into existing reg.
                                   Don't use load_reg, since it
                                   updates rdesc */

            printf("  LDR   R%u,R%u\n", r, r) ;
        else
            (void)get_rreg(a);  /* Load into new register
                                   */
    }
    printf("%s   L%u\n", op, l);/* Branch */
}                               /* void cg_cond(char *op,
                                   SYMB *a, int l) */
void  cg_arg(SYMB *a)
/* Generate for an ARG instruction. We load the argument into a
register, and then write it onto the new stack frame, which is 2
words past the current top of stack. We keep track of which arg
this is in the global variable "next_arg". */

/* We assume that ARG instructions are always followed by
other ARG instructions or CALL instructions. */
{
    int  r  = get_rreg(a) ;
    printf("ST1   R%u,%u(R%u)\n", r, tos + VAR_OFF +
        next_arg,R_P) ;
    next_arg += 4 ;
}                               /* void cg_arg(SYMB *a) */

void  cg_call(int f,SYMB *res)

/* The standard call sequence is
      LDA   f(R0),R2
      ST1   R1,tos(R1)
      LDA   tos(R1),R1
      BAL   R2, R3
    (ST1   R4,res)
```

生成 ARG 变量的代码。首先将变量名存入寄存器中，然后将之放入栈的顶部，此时栈底存入以前的变量。

生成调用代码。

```
    We flush out the registers prior to a call and then execute the
standard CALL sequence. Flushing involves spilling modified reg-
isters, and then clearing the register descriptors. We use BAL to
do the call, which means R_RET will hold the return address on
entry to the function which must be saved on the stack. After the
call if there is a result it will be in R_RES so enter this in the de-
scriptors. We reset "next_arg" before the call, since we know we
have finished all the arguments now. */
{
    flush_all() ;
    next_arg = 0 ;
    printf(" LDA   L%u,R%u\n", l, R_CALL) ;
    printf(" STI   R%u,%u(R%u)\n", R_P, tos, R_P) ;
    printf(" LDA   %u(R%u),R%u\n", tos, R_P, R_P) ;
    printf(" BAL   R%u,R%u\n", R_CALL, R_RET) ;
    if(res != NULL)          /* Do a result if there is one */
        insert_desc(R_RES, res, MODIFIED) ;
}                            /* void cg_call(int l, */
                             /*  SYMB *res) */
void   cg_return(SYMB *a)
/* The standard return sequence is
    (LDI   a,R4)
      LDI   4(R1),R2      return program counter
      LDI   0(R1),R1      return stack pointer
      BAL   R2,R3
    If "a" is NULL we don't load anything into the result
register. */
{
    if(a != NULL)
{
        spill_one(R_RES) ;
        load_reg(R_RES, a) ;
    }
     printf(" LDI   %u(R%u),R%u\n", PC_OFF, R_P, R_
            CALL) ;
     printf(" LDI   %u(R%u),R%u\n", P_OFF, R_P, R_P);
     printf(" BAL   R%u,R%u\n", R_CALL, R_RET) ;
}                            /* void cg_return(SYMB *a) */
```

在生成调用代码之前，先清空寄存器（包括清空修改寄存器和寄存器指定标志）。用 R_RET 来保存调用返回地址。调用返回的值用 R_RES 来保存。在调用前要重新设置“next_arg”变量。

设置生成返回语句代码。

1.6 Compiler example 2 for using Flex tool

The steps of running the Flex tool are:

1. Download the source code of Flex tool from the

1.6 实例2——使用工具 Flex

运行编译器工具软件 Flex 包括以下8个步骤。