

# 武汉理工大学毕业设计（论文）

## 程序语言 C 的编译器设计与实现

学院（系）： 计算机科学与技术学院

专业班级： 计算机 1401 班

学生姓名： 李开心

指导教师： 林 泓

# 学位论文原创性声明

本人郑重声明：所呈交的论文是本人在导师的指导下独立进行研究所取得的研究成果。除了文中特别加以标注引用的内容外，本论文不包括任何其他个人或集体已经发表或撰写的成果作品。本人完全意识到本声明的法律后果由本人承担。

作者签名：

年 月 日

# 学位论文版权使用授权书

本学位论文作者完全了解学校有关保障、使用学位论文的规定，同意学校保留并向有关学位论文管理部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权省级优秀学士论文评选机构将本学位论文的全部或部分内容编入有关数据进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本学位论文属于 1、保密口，在 年解密后适用本授权书  
2、不保密口。

（请在以上相应方框内打“√”）

作者签名： 年 月 日

导师签名： 年 月 日

# 摘 要

编译器是计算机系统软件中的重要组成部分，为计算机的发展提供了重要的基础。编译器分为多趟式编译器和单趟式编译器，多趟式编译器实现为多个相对独立的阶段，每个阶段都对源代码或上一阶段生成的表示进行扫描，生成的目标代码更加高效，但是实现过程更加复杂。单趟式编译器将编译的四个阶段组合成一遍，只对源代码扫描一遍就能完成编译过程，实现更加简单，但是生成的目标代码没有多趟式编译器生成的目标代码高效。本文设计了一个将 C--源代码翻译成目标虚拟机代码的单趟式编译器。

主要的内容总结如下：

（1）采用形式语言中的文法，定义了支持变量、函数、各种语句的 C--语言。设计一个虚拟机，将其指令作为目标代码，并能执行目标代码得到源程序运行结果。

（2）本文研究了编译器的相关技术，并采用 C++语言实现了一个单趟式编译器。词法分析部分采用硬编码方式完成词法单元的识别。解析部分采用语法制导的翻译技术，其中语法分析部分基于自顶向下的递归下降方法完成语句的解析，并在语法分析过程中插入语义分析、代码生成的语句完成翻译。

（3）为了验证编译器的正确性和健壮性，设计了一些测试用例，对编译器进行详细地测试，并分析了生成的目标代码。

**关键词：**编译器；虚拟机；单趟式

# Abstract

Compiler is an important part of computer system software, which provides an important basis for the development of computer. The compiler is divided into multi-pass compiler and single-pass compiler. Multi-pass compilers are often implemented as multiple relatively independent stages, each of which will scan the source code or the representation generated by the previous stage, the generated target code is more efficient, but the implementation is more complex. The single-pass compiler compiles the four stages of compilation into one pass, only scanning the source code once completes the compiling process. The implementation is simpler, but the generated target code is not as efficient as the target code generated by the multi-pass compiler. This thesis designed a single-pass compiler that translates C++ source code into the target code.

The main contents are summarized as follows:

(1) C++ language that supports variables, functions, and statements is defined using grammar in formal languages. Design a virtual machine whose instructions are used as target code, and can execute the target code to get the result.

(2) This thesis studies the compiler technology and uses C++ language to implement a single-pass compiler. In the lexical analysis part, the recognition of lexical units is completed by hard coding. The parsing part uses grammar guided translation techniques, in which the part of syntax analysis based on the top-down recursive descent method to complete the statement analysis, and the semantic analysis and code generation statements are inserted in the syntax analysis process to complete the translation.

(3) To verify the correctness and robustness of the compiler, some test cases are designed, the compiler is tested in detail, and the generated target code is analyzed.

**Keywords:** compiler; virtual machine; single-pass

# 目 录

第 1 章 绪论.....	1
1.1 研究背景和意义.....	1
1.2 编译器技术的研究现状.....	1
1.3 主要研究内容.....	2
第 2 章 高级语言 C--与虚拟机 .....	3
2.1 高级语言 C--文法定义 .....	3
2.2 虚拟机.....	6
2.2.1 指令集设计.....	7
2.2.2 数据与指令存储.....	10
第 3 章 编译器设计.....	11
3.1 编译器的整体结构.....	11
3.2 词法分析模块设计.....	11
3.2.1 符号的表示.....	12
3.2.2 符号表的设计.....	15
3.2.3 词法分析算法设计.....	16
3.3 解析模块的设计.....	17
3.3.1 语法分析算法设计.....	18
3.3.2 语义分析算法设计.....	22
3.3.3 代码生成算法设计.....	23
3.4 错误处理.....	27
第 4 章 编译器实现与测试.....	29
4.1 词法分析模块的实现.....	29
4.1.1 内建符号的处理.....	29
4.1.2 词法单元的识别.....	31
4.2 解析模块的实现.....	32
4.2.1 全局变量的解析.....	32
4.2.2 全局数组的解析.....	33
4.2.3 全局 enum 的解析.....	35
4.2.4 函数的解析与目标代码生成.....	36
4.2.5 语句的解析与目标代码生成.....	37
4.3 编译器测试.....	41

4.3.1 编译选项.....	41
4.3.2 对目标代码进行分析.....	42
第 5 章 总结与展望.....	52
5.1 本文总结.....	52
5.2 未来展望.....	52
参考文献.....	54
致 谢.....	55

# 第 1 章 绪论

## 1.1 研究背景和意义

编译器是现代计算机系统的基本组成部分之一，是把用一种语言书写的程序翻译成另一种语言书写的等价程序。编译器使得多数计算机用户不必考虑机器相关的繁琐细节，使程序员与程序设计专家独立于机器。

编译器的实现涉及各种编译原理知识的学习和对程序运行逻辑的了解。例如变量如何存储、函数调用时参数传递规则、各种语句的运行方式等。在实现编译器的过程中，需要对程序进行模块划分从而让程序具有可扩展性，需要仔细地设计符号表以便能够为编译过程提供完整的符号信息，需要设计栈帧结构以便函数能够准确地进行调用，需要设计优良的目标代码让程序能够更加高效地运行。这非常考验系统设计能力，同时也能有效地提升编程能力。

## 1.2 编译器技术的研究现状

20 世纪 50 年代编译器领域刚刚起步，研究焦点仅在从高级语言到机器码的转换以及优化程序对时间和空间的需求。此后，该领域产生了大量的有关程序分析与转换、代码自动生成以及运行时服务等方面的新知识。同时，编译算法也被用于便利软件和硬件开发、提高应用程序性能、检测或避免软件缺陷和恶意软件等方面。编译领域与其它方向越来越多地相互交叉渗透，这些方向包括计算机体系结构、程序设计语言、形式化方法、软件工程以及计算机安全等。

随着编译技术的发展和人们对编译程序需求的不断增长，20 世纪 50 年代末有人开始研究编译程序的自动生成工具，提出并研制编译程序的编译程序。目前自动生成工具已广泛使用，如词法分析程序的生成系统 Lex，语法分析程序的生产系统 Yacc 等。

到目前为止，编译器领域最为突出的成就是高级语言的广泛使用。从银行、企业的管理软件，到高性能计算和各种万维网（Web）应用，今天的绝大多数软件都是用高级语言编写并经过静态或动态编译而来。伴随着 20 世纪 90 年代中期 Java 的出现，易于管理的运行时系统，包括垃圾回收和即时编译技术，通过根除内存泄露进一步提高了程序员的开发效率。

我国编译器研发工作起步并不算晚，早在 60 年代初期，董韪美院士和杨芙清院士就分别在中科院和北大领导研究组开发编译器。90 年代以来中科院计算所张兆庆教授研究组先后开发了共享内存多处理机的并行识别器，分布式内存多处理机的并行识别器，SIMD 芯片和 VLIW 芯片的并行优化 C 编译器。此外，计算机专家朱传琪教授研究组研制的面向共

享存储并行机的并行优化编译器 AFT 达到世界领先水平；汤志忠教授研究组在软流水优化技术上做了很多优秀的研究工作。

### 1.3 本文主要内容

全文共五个章节，各个章节的内容介绍如下：

第 1 章，绪论。首先详细论述了论文的研究背景和意义，并对编译器的技术研究现状进行了详细的介绍。最后对本文的主要内容进行了总结。

第 2 章，高级语言 C--与虚拟机。本章采用 EBNF 文法对 C--语言的文法进行了详细的定义，并设计了一个基于栈的虚拟机作为 C--语言的目标机，其指令集作为编译器生成的目标代码。

第 3 章，编译器设计。本章对编译器进行了整体的设计。编译器采用单趟式实现，分为词法分析模块和解析模块。词法分析模块设计了符号的表示方式，符号表中符号的存储方式，以及采用硬编码方式的词法分析算法。解析模块设计了采用自顶向下的递归下降法的语法分析方法，确定了实现语义分析的方法，并对函数及各种语句的代码生成方法进行了详细的设计。

第 4 章，编译器实现与测试。本章首先对编译器各个模块的实现细节进行了详细的介绍。然后采用包含多种 C--语言功能的源文件对编译器进行测试，并对生成的目标代码进行了详细的分析。

第 5 章，总结与展望。本章对本文实现的编译器进行了总结，提出了存在的不足，并指出了改进方向。



## 第 2 章 高级语言 C--与虚拟机

本章对源语言及目标机进行了介绍，采用 EBNF 形式对 C--语言进行文法定义，设计了一个基于栈的虚拟机，定义了其指令集。虚拟机的指令作为 C--编译器的目标代码。

### 2.1 高级语言 C--文法定义

语法是语言的一组规则，它可以形成和产生一个合适的程序。通常使用文法作为程序设计语言语法的描述工具。文法是一种用有限的规则定义无限集合的方法<sup>[1]</sup>。目前广泛使用的是上下文无关文法，最著名的文法描述形式是 Backus-Naur 范式（BNF），而其扩展形式 EBNF 则更加简洁、灵活。因此本语言的文法使用 EBNF 文法来进行描述。本高级语言 C--是 C 语言的子集，参考了 C 语言的语法结构<sup>[2]</sup>。

（1）程序可由任意多个全局定义组成。其文法定义为：

`<program> ::= { <global_decl> }*`

（2）全局定义包含 enum 定义、变量定义、数组定义，以及函数定义，因变量定义、数组定义和函数定义都有相同的首部，避免在识别时出现歧义，因此将它们合并为 `<other_decl>`，再进行区分。其文法定义为：

`<global_decl> ::= <enum_decl> | <other_decl>`

`<other_decl> ::= <type> { '*' }* <id> <decl_tail>`

`<decl_tail> ::= <var_decl> | <arr_decl> | <func_decl>`

enum 变量可由数字进行初始化。其文法定义为：

`<enum_decl> ::= 'enum' '{' <id> [ '=' <number> ] { ',' <id> [ '=' <number> ] }* '}'`

全局变量只允许用常量进行初始化，并且允许在一条定义语句中定义多个变量。其文法定义为：

`<var_decl> ::= [ '=' <number> ] { ';' { '*' }+ <id> [ '=' <number> ] }* ';' ;`

数组允许用常量列表进行初始化。其文法定义为：

`<arr_decl> ::= '[' <number> ']' [ '=' '{' [ <number> , { ',' <number> }* ] '}' ] ';' ;`

函数定义分为函数参数定义和函数体定义，函数体则可以由多条语句组成。其文法定义为：

`<func_decl> ::= '(' <func_param> ')' '{' <func_body> '}'`

`<func_param> ::= <null> | <type> { '*' }* <id> { ',' <type> { '*' }* <id> }*`

`<func_body> ::= { <statement> }*`

（3）语句包含局部变量定义语句、if 语句、while 语句、return 语句、块语句以及表达式语句。其文法定义为：

`<statement> ::= <local_var> | <if_stat> | <while_stat> | 'return' <expr> ';' | '{' <statement> '}'`

|<expr> ';'

<local\_var> ::= <type> { '\*' } \* <id> [ '=' <expr> ] { ';' { '\*' } \* <id> [ '=' <expr> ] } \* ';'

<if\_stat> ::= 'if' '(' <expr> ')' <statement> [ 'else' <statement> ]

<while\_stat> ::= 'while' '(' <expr> ')' <statement>

（4）表达式用于执行实际功能。在进行表达式文法定义前，需要明确表达式中各种运算符的优先级。表 2-1 对 C--语言中运算符的优先级进行了定义，优先级数值越小表示优先级越高。

表 2-1 运算符优先级

运算符	描述	优先级
=	赋值	10
	逻辑或	9
&&	逻辑与	8
> < >= <= == !=	比较	7
+ -	加减	6
* /	乘除模	5
! - & * ++ --	前置运算	4
++ --	后置运算	3
()	括号运算	2
[] ()	数组索引、函数调用	1

表达式文法定义为：

<expr> ::= <assign\_expr>

赋值语句优先级低于或语句，其包含两个逻辑或表达式操作数。赋值语句的左操作数只能是左值，但是在文法定义阶段无法描述左值，因此这个问题将在语义分析中处理。其文法定义为：

<assign\_expr> ::= <or\_expr> <assign\_tail>

<assign\_tail> ::= '=' <or\_expr> <assign\_tail> | <null>

逻辑“或”表达式包含两个逻辑“与”表达式操作数。其文法定义为：

<or\_expr> ::= <and\_expr> <or\_tail>

<or\_tail> ::= '||' <and\_expr> <or\_tail> | <null>

逻辑“与”表达式包含两个关系运算表达式操作数。其文法定义为：

<and\_expr> ::= <cmp\_expr> <and\_tail>

<and\_tail> ::= '&&' <cmp\_expr> <and\_tail> | <null>

关系表达式包含两个算术表达式操作数。其文法定义为：

```

<cmp_expr> ::= <alo_expr> <cmp_tail>
<cmp_tail> ::= <cmps> <alo_expr> <cmp_tail>
<cmps> ::= '<' | '>' | '<=' | '>=' | '==' | '!='
    
```

算术运算表达式包含两个乘除表达式操作数。其文法定义为：

```

<alo_expr> ::= <mul_expr> <alo_tail>
<alo_tail> ::= <adds> <mul_expr> | <null>
<adds> ::= '+' | '-'
    
```

乘除表达式包含两个因子表达式操作数。其文法定义为：

```

<mul_expr> ::= <factor_expr> <mul_tail>
<mul_tail> ::= <muls> <factor_expr> | <null>
<muls> ::= '*' | '/' | '%'
    
```

因子表达式可以是值表达式，也可以是包含一个前置运算符及一个因子表达式的表达式。其文法定义为：

```

<factor_expr> ::= <left_op> <factor_expr> | <val_expr>
<left_op> ::= '!' | '-' | '&' | '*' | '++' | '--'
    
```

值表达式包含一个元素表达式，以及可选的后置运算符。其文法定义为：

```

<val_expr> ::= <elem_expr> <rop>
<rop> ::= '++' | '--'
    
```

元素表达式不包含任何运算符，是基本的操作数单元，如变量、数组、函数调用、括号表达式、常量等，因为这些内容都是以<id>开始，因此需要将<id>提取，并在<elem\_tail>中进行区分。其文法定义为：

```

<elem_expr> ::= <id> <elem_tail>
<elem_tail> ::= '[' <expr> ']' | '(' <args> ')' | '(' <expr> ')' | <literal> | <null>
<args> ::= <expr> { ',' <expr> } * | <null>
<literal> ::= <number> | <char> | <string>
    
```

（5）常量包括数字、字符和字符串。数字包括 0-9 共 10 个数字字符组成的串，以及描述其正负属性的符号。文法定义为：

```

<number> ::= <sign> { <num> } +
<sign> ::= [ '+' ] | [ '-' ]
<num> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
    
```

字符包括各种可能的字符，以及转义字符，字符串则是由双引号括起来的字符的集合。其文法定义如下：

```

<string> ::= '"' { <char> } * '"'
    
```

空串不包含任何内容，其文法定义为：

`<null> ::=`

通过对文法定义，可以看出表达式为程序提供真正的计算，语句为程序提供控制流程，函数为程序提供功能封装，全局变量为程序提供信息共享。

## 2.2 虚拟机

本节设计了一个基于栈的虚拟机作为 C--语言的目标机，虚拟机指令作为编译器生成的目标代码。

### 2.2.1 虚拟机概述

虚拟机是以软件的方式模拟的一台计算机，为了能够执行指令，需要模拟出物理机器的相关内容，如内存、寄存器等。编译器在代码生成阶段生成的代码就是虚拟机的指令。使用虚拟机指令集作为目标语言，可以通过虚拟机执行指令查看程序运行结果，从而检验编译器的正确性。并且自行设计虚拟机，可以自定义精简的指令集，内存可以是十分简单的数组，寄存器可以是普通的变量，通过程序模拟，可以省去与编译器无关的很多机器内部的复杂细节。

虚拟机通常分为两类，一种是基于寄存器的虚拟机，另一种是基于栈的虚拟机。两者都需要实现取指令、译码、执行、存储结果等功能，但是分别适用于不同的场景。

基于寄存器的虚拟机，更加贴近真实机器，其拥有多个虚拟寄存器，指令集架构更加复杂，因为很多指令都能够指定源寄存器和目的寄存器，如指令：`add reg1, reg2, reg3`，一条指令就能够实现将存储在 `reg2` 和 `reg3` 寄存器中的数相加并存储于 `reg1` 寄存器中，十分高效。因为结构与真实 CPU 类似，因此将虚拟寄存器映射到 CPU 寄存器上也十分方便，这正是基于寄存器的虚拟机执行效率高的要点。

基于栈的虚拟机，所有操作都是对栈顶元素进行，指令架构比较简单，如指令 `add`，没有指定操作数，其约定两个操作数分别在栈顶和次栈顶，计算结果放于栈顶中。因为操作数隐含在栈上，因此指令不用指定操作数，编译器生成的指令相对于基于寄存器的虚拟机更小。不使用寄存器进行缓存，也意味着取数据和存数据的操作相对于基于寄存器的虚拟机更多，内存访问次数更多，运行效率也更低。

基于寄存器的虚拟机在编译器生成代码阶段需要考虑寄存器分配的问题，并且指令集也更加复杂，因此虚拟机的实现也更加复杂。基于栈的虚拟机则无需关注寄存器分配的问题，同时采用递归下降法进行语法制导翻译的编译器能够很容易地生成虚拟机的指令，因此虚拟机设计部分采用基于栈的虚拟机。

### 2.2.2 指令集设计

指令可分为存取指令、跳转指令、函数调用指令、算术运算指令和内建函数指令。存取指令即对数据进行存取操作的指令。跳转指令控制程序的跳转。函数调用指令实现函数调用时参数及局部变量的存放、栈帧的创建，以及函数退出时调用现场的恢复。算术运算指令用于执行各种运算，如四则运算、取反、位移等。内建函数指令用于提供 C 语言部分库函数的功能。

存取指令包括 I\_IMM、I\_LEA、I\_LI、I\_LC、I\_SI、I\_SC、I\_PUSH。存取指令及其功能如表 2-2 所示。

表 2-2 存取指令

指令	描述
I_IMM <num>	取立即数 num
I_LEA <offset>	取 bp+offset 地址处的值
I_LI	获取一个 int 型数据
I_LC	获取一个 char 型数据
I_SI	存储一个 int 型数据
I_SC	存储一个 char 型数据
I_PUSH	将 ax 寄存器中的值入栈

表 2-2 中，I\_IMM 指令用于获取立即数，其后方的<num>代表 I\_IMM 指令有一个数值型的操作数。I\_LEA 指令用于获取局部变量，局部变量的存放位置根据 bp 寄存器来确定，因此 I\_LEA 指令的操作数<offset>表示该变量相对于 bp 寄存器的偏移。I\_LI 指令和 I\_LC 指令分别用于获取 int 型数据和 char 型数据，其操作数隐含在 ax 寄存器中。I\_SI 指令和 I\_SC 指令分别用于存储 int 型数据和 char 型数据，其操作数隐含在 ax 寄存器中。I\_PUSH 指令将 ax 寄存器中的值放入栈顶。

跳转指令包括 I\_JMP、I\_JZ、I\_JNZ。跳转指令及其功能如表 2-3 所示。

表 2-3 跳转指令

指令	描述
I_JMP <addr>	无条件跳转到 addr
I_JZ <addr>	若 ax 寄存器中的值为 0，则跳转到 addr
I_JNZ <addr>	若 ax 寄存器中的值非 0，则跳转到 addr

表 2-3 中，三条跳转指令都有<addr>操作数，表示跳转的地址。I\_JMP 指令是无条件跳转指令，I\_JZ 指令在 ax 寄存器值为 0 时跳转，I\_JNZ 指令则在 ax 寄存器值为非 0 时跳转。

函数调用指令包括 I\_CALL、I\_ENT、I\_LEV、I\_ADJ。The calling convention specifies how values are passed to and from a function call<sup>[3]</sup>，即设计函数调用相关指令时需要设计好调用约定。在编译器识别到函数调用时，需要先将函数参数入栈，然后保存调用现场并调用函数。函数调用结束后，需要恢复调用现场，并将参数出栈。参数入栈以及参数出栈由编译器在函数调用处生成相关代码来完成，保存调用现场和恢复调用现场则由函数本身的代码完成。在解析函数定义时，需要为函数局部变量预留栈空间。函数调用指令及其功能如表 2-4 所示。

表 2-4 函数调用指令

指令	描述
I_CALL <addr>	保存调用现场并调用函数
I_ENT <num>	为局部变量预留栈空间，局部变量个数为<num>
I_LEV	结束函数调用并恢复调用现场
I_ADJ	将函数调用参数出栈

表 2-4 中，I\_CALL 指令保存调用现场，并将程序执行流程切换到函数内部。I\_ENT 指令为局部变量预留栈空间。I\_LEV 用于结束函数调用并恢复调用现场，I\_ADJ 指令则将函数调用参数出栈。

运算符指令包括 I\_OR、I\_XOR、I\_AND、I\_EQ、I\_NE、I\_LT、I\_GT、I\_LE、I\_GE、I\_SHL、I\_SHR、I\_ADD、I\_SUB、I\_MUL、I\_DIV、I\_MOD。运算符指令及其功能如表 2-5 所示。

表 2-5 运算符指令

指令	描述
I_OR	或运算， $a   b$
I_XOR	异或运算， $a \wedge b$
I_AND	与运算， $a \& b$
I_EQ	相等运算， $a == b$
I_NE	不等运算， $a != b$
I_LT	小于运算， $a < b$
I_GT	大于运算， $a > b$
I_LE	小于等于运算， $a \leq b$
I_GE	大于等于运算， $a \geq b$
I_SHL	左移运算， $a \ll b$
I_SHR	右移运算， $a \gg b$
I_ADD	加运算， $a + b$
I_SUB	减运算， $a - b$
I_MUL	乘运算， $a * b$
I_DIV	除运算， $a / b$
I_MOD	模运算， $a \% b$

表 2-5 中的算术运算指令都不带参数，第二列中的  $a$  表示存放在栈顶的第一个操作数， $b$  表示存放在  $ax$  寄存器中的第二个操作数。计算后栈顶元素出栈，并将结果存放于  $ax$  寄存器中。

内建函数指令包括  $I\_PRTF$ 、 $I\_MALC$ 、 $I\_EXIT$ 、 $I\_SCANF$ 、 $I\_GETC$ 、 $I\_PUTC$ ，用于提供常用的 C 库函数。内建函数指令及其功能如表 2-6 所示。

表 2-6 内建函数指令

指令	描述
I_PRTF	调用 C 标准库中的 <code>printf</code> 函数
I_MALC	调用 C 标准库中的 <code>malloc</code> 函数
I_EXIT	调用 C 标准库中的 <code>exit</code> 函数
I_SCANF	调用 C 标准库中的 <code>scanf</code> 函数
I_GETC	调用 C 标准库中的 <code>getc</code> 函数
I_PUTC	调用 C 标准库中的 <code>putc</code> 函数

调用表 2-6 中所示的 C 标准库函数前，参数已预先入栈，只需将各个参数依次填入参

数列表即可。

### 2.2.3 数据与指令存储

指令和数据是应用上的概念。在内存或磁盘上，指令和数据没有任何区别，都是二进制信息<sup>[4]</sup>。在虚拟机内部，存储数据的数据段使用字符型数组，因为数据的最小单位是字符，而存储指令及其操作数的代码段则使用整型数组。

在存储数据时，可能涉及字符型数据、字符串型数据以及整型数据。存储字符时，可以直接将字符放到前一个数据之后。存储字符串型数据时，因为字符串是由多个字符组成，因此与存储字符数据有相同的操作。

在存储整型数据时，需要先进行数据对齐操作。许多计算机系统对基本数据类型的合法地址做出了一些限制，要求某种类型对象的地址必须是某个值  $K$ （通常是 2、4 或 8）的倍数。这种对齐限制简化了形成处理器和内存系统之间接口的硬件设计<sup>[5]</sup>。因为整型数据大小在 32 位机器上是 4 字节，因此在存储整型数据时需要进行数据对齐，让整型数据的起始地址为 4 的倍数。

指令存储则简单很多，因为指令为人为定义，且指令操作数最大为 4 字节，因此所有生成的指令及操作数都将以整型的方式存储在代码段中。



## 第 3 章 编译器设计

本章基于 C-- 语言的文法设计了单趟式扫描的编译器。在词法分析模块中对源语言单词进行了类别定义及属性表示，完成了词法分析模块的详细设计。采用语法制导的翻译方法对解析模块的语法分析、语义分析及代码生成方法进行了详细的设计。

### 3.1 编译器的整体结构

本文设计的单趟式编译器整体结构如图 3.1 所示。

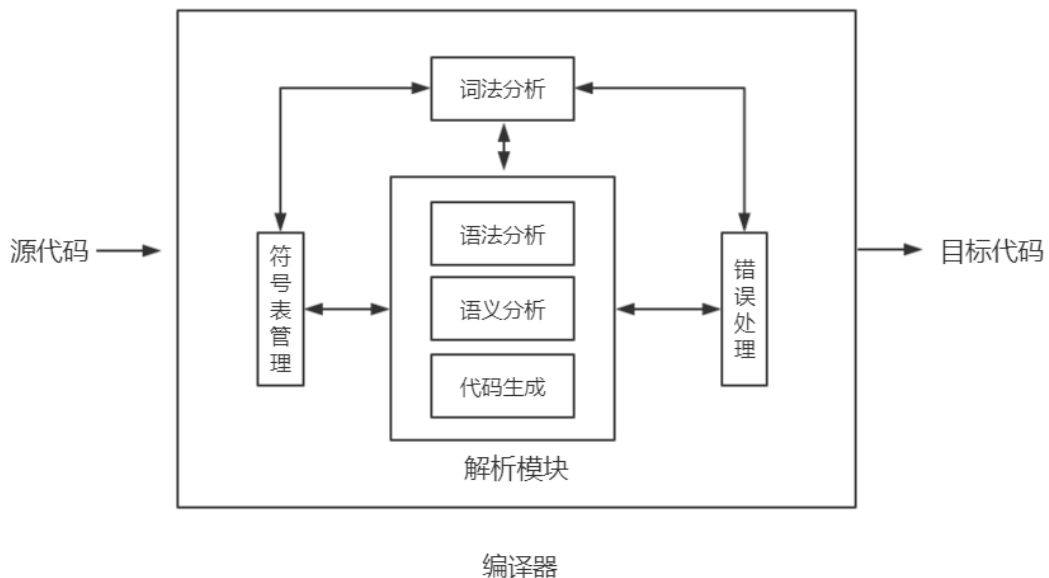


图 3.1 编译器整体结构

单趟式编译方式只需要对源代码扫描一遍，就能完成整个编译流程。词法分析模块完成源代码中词法单元的截取，向符号表中添加新的符号并完善信息。解析模块调用词法分析模块获得符号信息，完成语法分析、语义分析以及代码生成的工作，在此过程中还需要进行符号表管理以及错误处理。

### 3.2 词法分析模块设计

词法分析模块实现词法单元的截取，并将新的符号插入到符号表中。词法分析模块被设计成一个子程序，解析模块可以通过调用该子程序获得词法单元。词法分析模块的设

计包括对符号的设计、符号表的设计，以及词法分析算法的设计。

词法分析总控程序如图 3.2 所示。

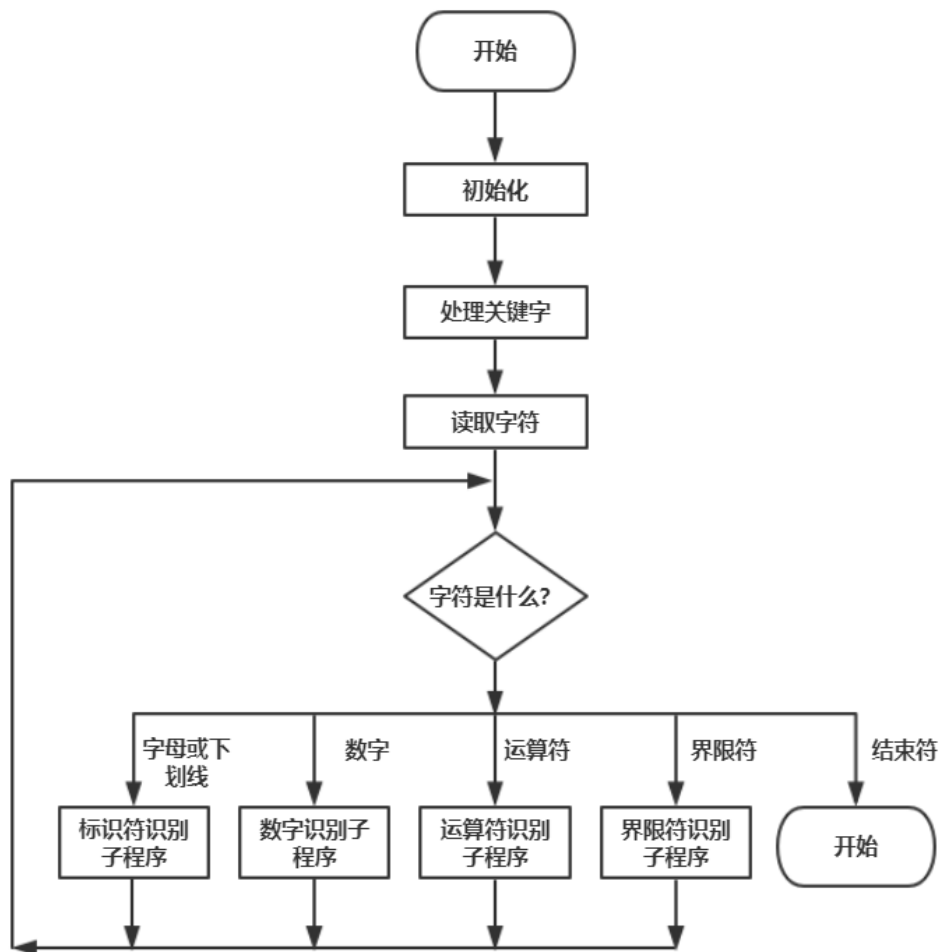


图 3.2 词法分析总控程序

图 3.2 中，词法分析器首先进行初始化，然后对关键字进行处理。当解析程序调用词法分析子程序时，词法分析子程序会读取字符进行词法单元识别，并将识别的词法单元返回给解析程序。

### 3.2.1 符号的表示

符号即词法分析过程中识别出的各种词法单元，有标识符、数字、关键字等类型，还有变量、函数、值等种类，变量和函数有数据类型，以及所在作用域等信息。下面是对符号的各种属性进行定义。

符号主要由标识符、数字、关键字、字符串、界限符和运算符组成。符号的类型定义及

其描述如表 3-1 所示。

表 3-1 符号类型定义

符号类型	描述
END	文件结束符
ERROR	无法识别的符号
LPAREN / RPAREN	()
LBRACK / RBRACK	[]
LBRACE / RBRACE	{ }
COMMA / COLON / SEMICON / TILDE	, : ; ~
ID	用户定义标识符
NUM_INT / NUM_CHAR	整型值 字符型值
CHAR / ELSE / ENUM / IF / INT / RETURN / WHILE	关键字
ASSIGN / COND / LOR / LAN / OR / XOR / AND	= ?    &&   ^ &
NOT / EQ / NE / LT / GT / LE / GE	! == != < > <= >=
SHL / SHR / AND / SUB / MUL / DIV / MOD	<< >> + - * / %
INC / DEC	++ --
STRING	字符串

符号类型无法区分全局变量、局部变量、函数等，因此需要使用种类来进行区分。表 3-2 展示了符号种类的定义及其描述。

表 3-2 符号种类定义

符号种类	描述
NUMBER	值（立即数）
SYS_FUNC	内建函数
FUNC	用户定义函数
GLOBAL_VARIABLE	全局变量
LOCAL_VARIABLE	局部变量

变量具有数据类型，函数具有返回类型，C--语言支持 int 类型和 char 类型，以及它们的多级指针。数据类型定义如表 3-3 所示。在实际定义中，CHAR\_TYPE 值为 0，INT\_TYPE 值为 1，PTR\_TYPE 值为 2。PTR\_TYPE 只是代表指针的级数，例如 char\*\*\*是 char 类型的三级指针，其值为 CHAR\_TYPE+PTR\_TYPE+PTR\_TYPE+PTR\_TYPE，即 6。int\*\*是 int 类型的二级指针，其值为 INT\_TYPE+PTR\_TYPE+PTR\_TYPE，即 5。使用这种方式，可以

通过 CHAR\_TYPE 或 INT\_TYPE 与 PTR\_TYPE 的多次组合，来实现多级指针的表示。

表 3-3 数据类型定义

数据类型	描述
CHAR_TYPE	char 类型
INT_TYPE	int 类型
PTR_TYPE	指针类型

符号类 Token 用于记录词法单元的各种信息，其各个属性的定义及其描述如表 3-4 所示。

表 3-4 Token 类的属性定义

属性名	描述	数据类型
type	类型	int
name	名字	string
hash	hash 值，用于名字的快速查找	int
klass	种类	int
dataType	数据类型	int
value	值	int
argsDataType	函数参数类型列表	vector<int>
scope	所在作用域	vector<int>

为了实现嵌套的作用域，每一个作用域都需要有独特的标识，可以使用一个自增的变量来记录，并用一个全局的列表来记录当前所在的作用域。每当进入一个作用域，则该变量自增 1，并添加到列表尾部，离开作用域时去掉尾部的值，就可完成不同作用域的标识。作用域示例如图 3.3 所示。

代码	当前作用域
<code>int a;</code>	0
<code>int main() {</code>	
<code>int b;</code>	0/1
<code>if(true) {</code>	
<code>//...</code>	0/1/2
<code>}</code>	
<code>while(true) {</code>	
<code>//...</code>	0/1/3
<code>}</code>	
<code>//...</code>	0/1
<code>}</code>	
<code>//...</code>	0

图 3.3 作用域的标识

由图 3.3 可知，全局作用域为 0。进入 main 函数后，作用域标记增加 1 变为 1，并添加到作用域列表尾部，当前作用域列表为 0/1。if 语句的语句体内，作用域标记自增 1 变为 2，添加到作用域列表尾部，当前作用域列表为 0/1/2。离开 if 语句后，作用域列表变为 0/1。进入 while 语句后，作用域标记自增 1 变为 3，添加到作用域列表尾部，当前作用域为 0/1/3。离开 while 语句后，回到 main 函数作用域，当前作用域为 0/1。离开 main 函数后，回到全局作用域，当前作用域列表为 0。通过对不同作用域进行标记，实现了嵌套的作用域。

### 3.2.2 符号表的设计

符号表是编译器保存信息的中心库，编译器的各部分通过符号表进行交互，并访问符号表中的数据<sup>[1]</sup>。

符号表使用一个符号类 Token 的列表来存储所有符号信息，并有一些辅助的变量来记录如作用域、main 函数位置等信息。符号表的各个属性定义及其描述如表 3-5 所示。

表 3-5 符号表的属性定义

属性名	描述	数据类型
table	符号列表	vector<Token>
current	当前符号在表中的索引	int
mainIndex	main 函数 Token 在表中的索引	size_t
scopeIndex	作用域标记	int
scope	当前作用域	vector<int>

表 3-5 中，table 属性用于存储所有的符号信息。current 属性记录当前处理的符号在符

号表中的索引，因为词法分析只会向符号中写入部分信息，如符号类型、符号名等，而符号是全局变量还是局部变量，符号的值是多少需要由解析模块写入，因此需要有 `current` 属性来记录当前处理符号的索引。`mainIndex` 用于记录 `main` 函数的 `Token` 在符号表中的位置，`main` 函数作为关键字被预先插入到符号表中，但是只写入了部分信息，`main` 函数的首地址需要在解析了 `main` 函数定义后才能确定，因此需要对 `main` 函数的 `Token` 在符号表中的位置进行标识，以便解析时能够找到该 `Token`。`scopeIndex` 用于作用域计数，`scope` 用于记录当前的作用域，进入新的作用域 `scopeIndex` 会自增 1 并添加到 `scope` 尾部，离开该作用域后 `scope` 会将尾部的数删除，通过这两个属性可以标识出不同的作用域。

符号表不仅存储符号信息，还提供 `has` 函数用于查询符号信息。词法分析模块获取到符号后，通过符号表的 `has` 函数来确定符号表中是否存在同名符号。若不存在，则向符号表中添加新的符号并写入部分信息，如符号类型、符号名、`hash` 值等，否则直接返回找到的符号。

`has` 函数查找符号时，首先会查找关键字作用域，关键字的作用域都被设置为 -1，通过先查找关键字作用域来确保关键字不被定义的同名变量隐藏。若不是关键字，会先从当前作用域内查找，若找到同名符号则返回 `true`，否则再进入更外层的作用域查找。若最终没有找到同名符号则返回 `false`。词法分析模块根据 `has` 函数的返回值来确定是否需要向符号表中插入信息。

### 3.2.3 词法分析算法设计

词法分析阶段的主要任务是从源代码中截取出词法单元，并将部分信息写入记录该词法单元信息的符号中，为后续阶段提供符号信息。

词法单元分为 5 种：

- (1) 关键字，如 `int`、`char`、`printf` 等。
- (2) 标识符，用于表示用户定义的名字，如变量名、函数名等。
- (3) 常量，包括数字、字符和字符串。
- (4) 运算符，用于进行各种计算，如 `+`、`*`、`>=` 等。
- (5) 界限符，用于分隔程序中的语句，如分号、逗号和括号等。

标识符的符号组成规则为：任意多个字母、数字和下划线组成的序列，第一个字符不

能是数字。

数字可以是八进制、十进制和十六进制。八进制由 0 开始，十六进制由 0x 开始。所有进制识别后都将转化成十进制值返回。

字符由单引号包裹，中间可以是任意的字符，如'a'，或者以反斜杠加字符的形式表示转义字符，如'\a'。本 C--语言只支持'\n'表示换行符，以及'\0'表示值为 0 的字符，其他转义字符形式都看做字符本身，如'\x'被视为'x'。

字符串则以双引号包裹，内部可以由多个字符组成，包括转义字符。字符串在识别过程中，会将各个字符按照顺序存入虚拟机数据段中，并且在最后添加字符'\0'，词法分析模块会返回字符串的首地址。

运算符和界限符识别中，对于仅由单个符号组成的词法单元，识别到则直接视为识别成功。对于部分运算符，如=和==，首字符相同，需要判断后一个字符来确定识别为=还是==。遇到换行符时，需要对行号加 1，便于进行错误处理时能够指明错误出现的行号。空白符直接跳过，不视作任何词法单元。

### 3.3 解析模块的设计

解析模块调用词法分析模块获得词法单元，完成语法分析、语义分析及目标代码生成的工作。在采用递归下降法的语法分析过程中，调用词法分析模块获取词法单元，并插入语义分析语句及目标代码生成语句，实现单趟式编译。解析流程如图 3.4 所示。

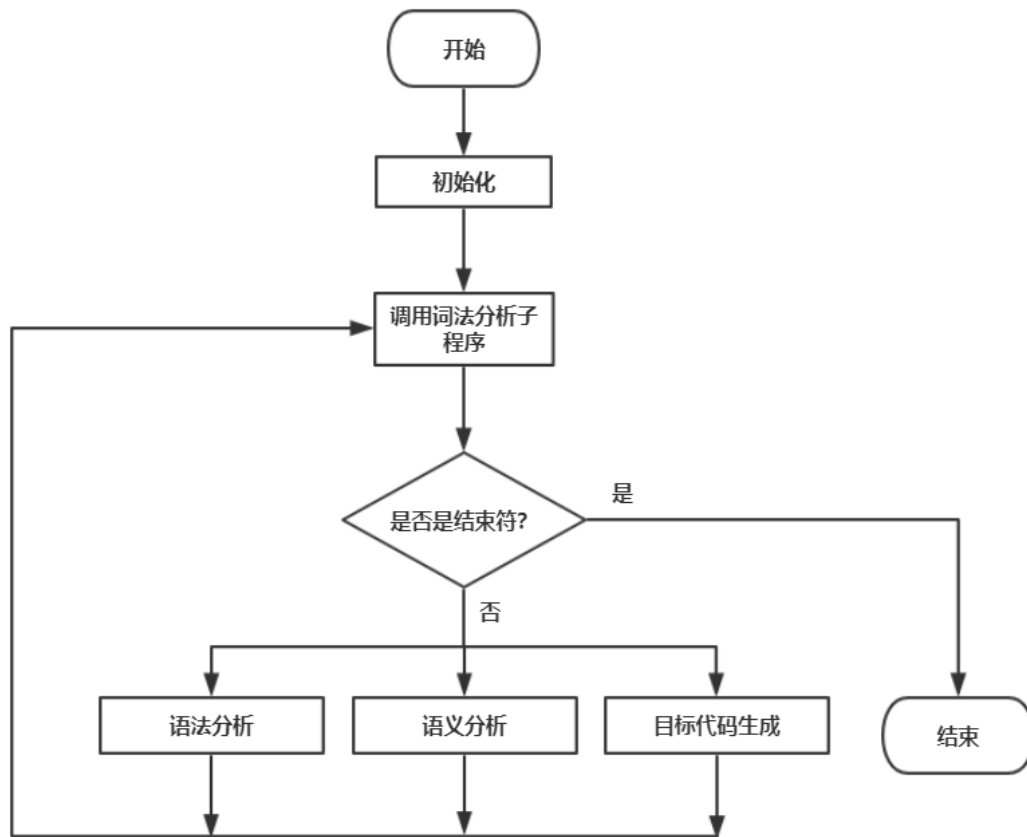


图 3.4 解析流程图

由图 3.4 可知，解析模块首先进行初始化，然后循环调用词法分析子程序获得词法单元，并同时进行语法分析、语义分析及目标代码生成完成解析。

### 3.3.1 语法分析算法设计

语法分析过程用于确认输入的语句是否符合语言的文法规则。目前常用的语法分析方法有自顶向下语法分析方法和自底向上语法分析方法。自顶向下语法分析方法中的递归下降方法是常用的语法分析方法，递归下降法为每个非终结符编写一个递归过程，每个过程的功能是识别由该非终结符推出的串，使用递归下降法可以很容易地根据文法定义写出对应的递归下降子程序。

下面是程序中各种文法对应的递归下降分析过程。

(1) 程序的文法定义为： $\langle \text{program} \rangle ::= \{ \langle \text{global\_decl} \rangle \}^*$ ，其递归下降语法分析过程如图 3.5 所示。



```
void program(){
    while(true){
        global_decl();
    }
}
```

图 3.5 program 的递归下降语法分析过程

由图 3.5 可知，program 可由任意多个 global\_decl 组成，通过多次调用 global\_decl 函数来实现。

（2）全局定义的文法为：<global\_decl> ::= <enum\_decl> | <other\_decl>，其递归下降分析过程如图 3.6 所示。

```
void global_decl(){
    if(currentToken == ENUM){
        enum_decl();
    }
    else{
        other_decl();
    }
}
```

图 3.6 global\_decl 的递归下降语法分析过程

由图 3.6 可知，global\_decl 会根据从词法分析中得到的符号类型来决定调用 enum\_decl() 还是 other\_decl()。

（3）函数的文法定义为：<func\_decl> ::= '(' <func\_param> ')' '{' <func\_body> '}'，其递归下降分析过程如图 3.7 所示。

```
void func_decl(){
    match('(');
    func_param();
    match(')');

    match('{');
    func_body();
    match('}');
}
```

图 3.7 函数的递归下降分析过程

图 3.7 中，match 函数用于匹配其参数，如果当前符号与参数相同，则会调用词法分析模块获取下一个词法单元，否则会进入错误处理，提示源程序中对应位置缺少需要的符号。

函数体的文法定义为：<func\_body> ::= { <statement> }<sup>\*</sup>，其递归下降分析过程如图 3.8 所示。

```
void func_body(){
    while(currentToken != '}{'){
        statement();
    }
}
```

图 3.8 函数体的递归下降分析过程

(4) 语句有多种类型，其文法定义为：<statement> ::= <local\_var> | <if\_stat> | <while\_stat> | return <expr> ';' | '{' <statement> '}' | <expr> ';'，语句的递归下降分析过程如图 3.9 所示。

```
void statement(){
    if(currentToken == INT || currentToken == CHAR){
        local_var();
    }
    else if(currentToken == IF){
        if_stat();
    }
    else if(currentToken == WHILE){
        while_stat();
    }
    else if(currentToken == RETURN){
        return_stat();
    }
    else if(currentToken == '{'){
        match('{');
        statement();
        match('}');
    }
    else {
        expr();
        match(';');
    }
}
```

图 3.9 语句的递归下降分析过程

由 3.9 可知，对语句进行解析时，会根据当前符号选择进入哪一个递归子程序。  
if 语句的递归下降分析过程如图 3.10 所示。

```
void if_stat(){
    match(IF);
    match('(');
    expr();
    match(')');
    statement();

    if(currentToken == ELSE){
        match(ELSE);
        statement();
    }
}
```

图 3.10 if 语句的递归下降分析过程

图 3.10 中，首先匹配 if 关键字，然后匹配左括号，并调用 `expr()` 匹配表达式，再匹配右括号，最后匹配表达式为真时执行的语句。若有 else 关键字，则匹配 else 关键字和表达式为假时执行的语句。

（5）表达式文法很多都类似，例如赋值表达式的文法定义为：

`<assign_expr> ::= <or_expr> <assign_tail>`

`<assign_tail> ::= '=' <or_expr> <assign_tail> | <null>`

其递归下降分析过程如图 3.11 所示：

```
void assign_expr(){
    or_expr();
    assign_tail();
}

void assign_tail(){
    if(currentToken == '='){
        match('=');
        or_expr();
        assign_tail();
    }
}
```

图 3.11 赋值表达式的递归下降分析过程

由图 3.11 可知，赋值语句解析时会先解析一个或表达式 `or_expr`，然后调用 `assign_tail` 函数。`assign_tail` 函数判断当前符号是否为等于来决定语句是否结束。

从各种语句的递归下降分析过程可知，对于文法中的非终结符，都会编写一个递归子程序来完成其解析，终结符则直接匹配。文法和对应的递归下降分析过程可以一一对应，十分直观且容易编写。

### 3.3.2 语义分析算法设计

语义分析阶段需要对每个语法结构进行语义检查，验证程序是否具有真正的意义。如果语义正确，则需要执行翻译生成目标代码，否则转到错误处理程序进行错误处理。

本编译器中，语义分析的任务是检查类型匹配，如赋值表达式中左右操作数是否具有相同的类型，函数调用时参数个数及类型是否同函数定义时一致等。

二元表达式通常需要两个操作数类型相同，如赋值表达式，若左操作数类型与右操作数类型不同，则有数据溢出的风险，因此需要进行提醒。对赋值表达式进行语义分析的方法如图 3.12 所示。

```
void assign_expr(){
    or_expr();
    ...    //记录左操作数类型
    match('=');
    or_expr();
    ...    //记录右操作数类型
    ...    //若两个操作数类型不匹配则报Warning
}
```

图 3.12 赋值表达式中的语义分析方法

由图 3.12 可知，赋值语句解析时，首先解析第一个表达式并记录其类型，然后匹配第二个操作数并记录其类型。最后对两个操作数的类型进行比较，若不相同则提示相应的信息。

函数调用时需要对函数参数进行匹配。函数参数个数及类型信息保存在记录函数信息的符号的 `argsDataType` 属性中，函数调用时需要与实际参数的个数及类型进行比较，若部分参数类型不同报 `Warning` 提示参数类型不匹配，参数个数不同报 `Error` 提示错误的函数调用。

### 3.3.3 代码生成算法设计

编译过程中涉及目标代码生成的只有函数及函数体内的语句。函数调用时虚拟机内部需要创建函数调用栈帧。有函数定义如图 3.13 所示。

```
void func(int arg1, int arg2, int arg3) {
    int i = 0;
    int j = 1;
    //...
}
```

图 3.13 函数定义

若有函数调用：func(1, 2, 3); 则创建的函数调用栈帧结构如图 3.14 所示。

...		
1	参数arg1	
2	参数arg2	
3	参数arg3	
pc+1	函数返回地址	
bp	bp寄存器旧值	<--bp
0	局部变量i	
1	局部变量j	<--sp

图 3.14 函数调用栈帧结构

由图 3.14 可知，参数顺序入栈，然后保存返回地址、bp 寄存器，为局部变量预留栈空间。函数调用栈帧中，参数入栈、保存返回地址与 bp 寄存器由函数调用处生成的 I\_CALL 指令来完成，而局部变量个数只有函数本身知道，因此为局部变量预留栈空间需要由函数本身的指令来完成。

函数定义生成的目标代码结构如图 3.15 所示：

```
I_ENT n    //函数有n个局部变量
...        //函数体的其他目标代码
...
...
I_LEV      //离开函数调用
```

图 3.15 函数定义生成的目标代码结构

函数定义中解析参数列表时会记录参数的个数和类型，解析函数体则会生成函数的目标代码。解析函数体前，会先生成 `I_ENT` 指令记录函数的局部变量个数，以便创建函数调用栈帧时能够为局部变量预留存储空间。函数体解析完成后生成 `I_LEV` 指令用于恢复函数调用现场。`I_ENT` 指令的操作数 `n` 需要在解析完函数体后进行回填，因为函数局部变量个数只有在函数体解析完后才能知道。

函数调用生成的目标代码结构如图 3.16 所示。

```

...           //其他代码
...           //函数参数入栈
I_CALL addr  //调用函数，其首条代码的地址为<addr>
I_ADJ n      //参数出栈
...           //其他代码
    
```

图 3.16 函数调用生成的目标代码结构

进行函数调用前，需要先将函数参数入栈，然后生成 `I_CALL` 指令调用函数。函数调用结束后，`I_ADJ` 指令将函数参数出栈。

函数体由各种语句组成，语句包括变量定义语句、`if` 语句、`while` 语句、`return` 语句、块语句、表达式语句。

（1）变量定义语句，如 `int a = b + c` 语句，会被拆分为 `int a` 语句和 `a=b+c` 语句进行解析，其生成的目标代码由表达式 `a=b+c` 生成，表达式的目标代码生成将在本节最后介绍，此处就不再单独介绍变量定义语句的目标代码生成。

（2）`if` 语句有两种格式：`if(E) S` 以及 `if(E) S1 else S2`。`S`（以及 `S1` 和 `S2`）可以是任意语句。对于第一种格式，其目标代码生成结构如图 3.17(a)所示。对于有 `else` 分支的 `if` 语句，会根据表达式 `E` 的运算结果选择 `S1` 或 `S2` 的代码执行。其目标代码生成结构如图 3.17(b)所示。

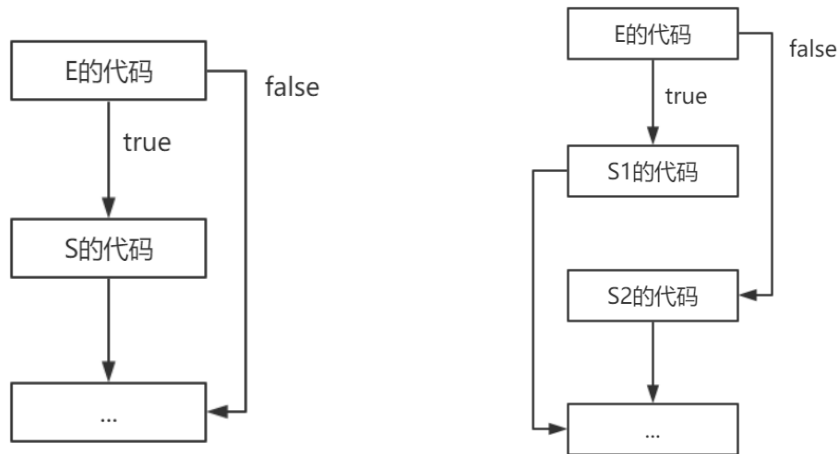


图 3.17(a) if 语句的目标代码结构    图 3.17(b) if-else 语句的目标代码结构

解析 if 语句的算法采用图 3.18 的伪代码表示。

```

void if_stat(){
    match(IF);
    match('(');
    expr();
    match(')');
    add_instruction(I_JZ, addr1);
    statement();

    if(currentToken == ELSE){
        match(ELSE);
        add_instruction(I_JMP, addr2);
        statement();
    }
}
    
```

图 3.18 解析 if 语句的伪代码

由图 3.18 可知，在对 if 语句进行语法分析的过程中，调用生成目标代码的函数 `add_instruction` 来完成目标代码生成。解析完表达式 E 后，先生成指令 `I_JZ addr1`，然后再解析语句 S。如果有 else 分支，则会生成 `I_JMP addr2` 指令，再解析语句 S2。addr1 和 addr2 的值需要在 if 语句解析完成后进行回填。

(3) while 语句与 if(E) S 语句类似，但是 while 语句在执行了 S 语句后会再次回到其目标代码开始，直到表达式 E 的运算结果为 false。其代码结构如图 3.19 所示。

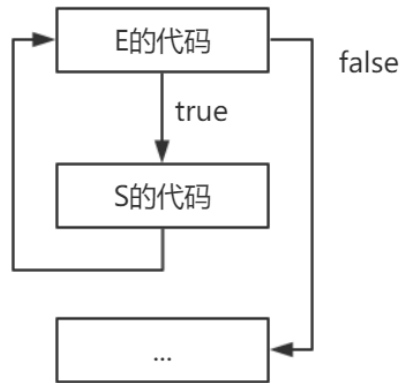


图 3.19 while 语句的代码结构

解析 while 语句的算法采用图 3.20 的伪代码表示。

```

void while_stat(){
    match(WHILE);
    match('(');
    expr();
    match(')');
    add_instruction(I_JZ, addr1);
    statement();
    add_instruction(I_JMP, addr2);
}
    
```

图 3.20 解析 while 语句的伪代码

由图 3.20 可知，解析完表达式 E 后，需要判断 E 的值，若为 0 则跳转到 addr1，即 while 语句生成的目标代码之后的位置，否则解析语句 S，并跳转到 addr2，即表达式 E 生成的首条目标代码的位置。

（4）return 语句的格式为 return E，会先计算表达式 E 的值，将其存储在虚拟机内 ax 寄存器中，然后生成 I\_LEV 指令，退出函数调用。解析 return 语句的算法采用图 3.21 的伪代码表示。

```

void return_stat(){
    match(RETURN);
    expr();
    add_instruction(I_LEV);
}
    
```

图 3.21 解析 return 语句的伪代码



（5）块语句的格式为 { S }，括号内是更深层的作用域，S 可以是任意语句。

（6）表达式语句分为一元表达式和二元表达式。一元表达式如取全局指针变量 `addr` 所指的地址的值时，会直接生成对应的取值指令。二元表达式形如 `expr1 op expr2`，会先生成表达式 `expr1` 的代码，并生成 `I_PUSH` 指令将结果入栈，然后生成表达式 `expr2` 的代码，再生成指令 `op` 对应的指令完成表达式的计算。解析一元表达式 `*addr` 的伪代码如图 3.22 所示，解析二元表达式 `expr1+expr2` 的算法采用图 3.23 的伪代码表示。

```
add_instruction(I_IMM, addr);
add_instruction(I_LI);
```

图 3.22 解析一元表达式 `*addr` 的伪代码

```
expr();
add_instruction(I_PUSH);
match('+');
expr();
add_instruction(I_ADD);
```

图 3.23 解析二元表达式 `expr1+expr2` 的伪代码

由图 3.23 可知，先解析第一个表达式，并使用指令 `I_PUSH` 将计算的值入栈，然后解析第二个表达式，最后生成 `I_ADD` 指令完成两个表达式的加运算。两个表达式的值分别在栈顶和 `ax` 寄存器中，满足 `I_ADD` 指令对其两个操作数的要求。

## 3.4 错误处理

编译器中设计有错误处理模块，用于在检测到源程序中的错误后为用户提供详细的错误信息。程序中定义 `Error` 类和 `Warning` 类用于记录错误和警告。它们会记录错误或警告的位置及描述。在遇到错误时，可根据词法分析器提供的行号，以及期待或错误的符号等信息来构建 `Error` 和 `Warning`。

在语法分析过程中，会根据语言文法对源程序进行匹配，若遇到错误的符号则会提示相应的错误。语义分析过程中会判断表达式的操作数是否具有相同的类型、函数调用参数是否同函数定义中参数列表匹配等，若类型不同会提示警告，个数不同会提示错误。

例如函数调用 `func(int a)`，函数调用参数中第一个符号是 `int`，为关键字，而函数调用参数的文法描述为：

$\langle \text{args} \rangle = \langle \text{expr} \rangle \{ ' \langle \text{expr} \rangle ] \}^* | \langle \text{null} \rangle$

变量定义语句 `int a` 不是表达式语句，不符合文法规范，编译器将记录出错行号，以及类似“函数调用参数错误”的描述并构建 `Error` 对象，将其作为异常抛出。

例如函数调用 `func(a, b)`，若函数声明为 `void func(int a)`，在语法分析阶段无法检测到错误，因为 `func(a, b)` 是一个语法正确的句子，但是可以在语义分析中检测到。在对函数进行调用时，先解析函数调用参数，然后将实际参数的类型与函数定义中参数列表进行比较，发现参数个数不匹配，因此检测到语义错误。

在解析过程中，编译器会记录各个表达式的类型，以便进行类型匹配。对于表达式 `a=b+c`，若 `a` 是 `char` 型，而 `b` 和 `c` 是 `int` 型，`b+c` 作为一个子表达式先进行计算，得到结果类型为 `int` 型，对 `a` 进行赋值时，检测到将 `int` 型值赋给 `char` 型值有溢出的风险，因此提示警告。

## 第 4 章 编译器实现与测试

本章对编译器进行实现及测试。在词法分析模块中完成对内建符号的处理，以及各种词法单元的识别。解析模块中采用了语法制导的翻译方法完成对 C++ 语言各种功能的解析。编译器测试部分首先测试一个覆盖大部分 C++ 语言功能的源程序，并对生成的目标代码进行详细分析，来验证编译器的正确性，然后编写了存在各种错误的 C++ 源程序来验证编译器的健壮性。

### 4.1 词法分析模块的实现

词法分析模块的主要任务是完成源代码词法单元的截取，识别到新的标识符会向符号表中插入符号并完善其信息，并向解析模块返回符号信息。

#### 4.1.1 内建符号的处理

在进行编译之前，需要先将内建符号如关键字、内建函数等插入到符号表，并完善相关的信息。内建符号的作用域设置为 KEY\_WORD\_SCOPE，其值为-1，不与任何其他作用域相同，对符号表进行查询时会先判断当前符号是否是内建符号，从而确保内建符号会被最先查找，避免用户定义的与内建符号同名的变量覆盖内建符号。

词法分析模块使用 next() 函数对源文件进行词法单元截取，对内建符号的处理也可以使用该函数来完成。插入关键字符的方式如图 4.1 所示。

```
source = "char else enum if int return sizeof while";
int type = CHAR;
while (type <= WHILE) {
    next();
    auto& tk = table->getCurrentToken();
    tk.type = type;
    tk.setScope({ KEY_WORD_SCOPE });
    ++type;
}
```

图 4.1 向符号表中插入关键字信息

图 4.1 中，首先设置 source 为多个关键字组合成的字符串，next 函数会将 source 中关

键字一个个截取出，因为符号表为空，每次取到的词法单元都是新的符号，因此 `next` 函数将会向符号表中插入对应关键字的符号，最后设置它的 `type`、`scope` 等信息。需要注意的是，符号类型定义中 `CHAR` 到 `WHILE` 的定义与 `source` 中的各个关键字一一对应，因此 `type` 可以在每次插入一个符号后进行简单的递增。

插入内建符号的方式与插入关键字符号的方式相同，如图 4.2 所示。

```
source = "printf malloc exit scanf getchar putchar";
type = I_PRTF;
while (type <= I_PUTC) {
    next();
    Token& tk = table->getCurrentToken();
    tk.klass = SYS_FUNC;
    tk.dataType = INT_TYPE;
    tk.value = type++;
    tk.setScope({ KEY_WORD_SCOPE });
}
```

图 4.2 向符号表中插入内建函数信息

图 4.2 中，`source` 被设置为由内建函数名组成的字符串，`next` 函数则将它们一个个插入到符号表中，最后设置它们的各种信息。内建函数的种类为 `SYS_FUNC`，返回值类型为 `INT_TYPE`。

`main` 符号也需要预先添加到符号表中，同时需要设置符号表中的 `mainIndex` 为当前符号所在位置，便于编译结束后，可以通过该索引获取到记录 `main` 函数的 `Token` 从而获取到 `main` 函数的首地址。向符号表中插入 `main` 函数符号的方式如图 4.3 所示。

```
source = "main";
next();
table->setMainToken();
table->getCurrentToken().setScope({ KEY_WORD_SCOPE });
```

图 4.3 向符号表中插入 `main` 函数信息

图 4.3 中，`table->setMainToken()` 会设置 `mainIndex` 为当前符号所在位置，从而记录了 `main` 函数所在的位置，`table` 是符号表对象。`main` 函数的值，即 `main` 函数的地址，需要在解析源文件中定义的 `main` 函数后才会写入到记录 `main` 函数的符号中。

### 4.1.2 词法单元的认识

标识符的符号组成规则为：以字母、数字和下划线组成的序列，第一个字符不能是数字。标识符的识别代码如图 4.4 所示。

```
if (isAlpha(curr) || curr == '_') {
    std::string name(1, curr);
    int hash = curr;
    curr = get();
    while (isAlpha(curr) || isNum(curr) || curr == '_') {
        name.push_back(curr);
        hash = hash * 147 + curr;
        curr = get();
    }

    //查符号表
    if (!table->has(hash, name)) {
        Token& tk = table->getCurrentToken();
        tk.type = ID;
        tk.name = name;
        tk.hash = hash;
    }
    return { ID, 0 };
}
```

图 4.4 标识符的识别

图 4.4 中，首先判断首字符是否满足要求，然后使用变量 `name` 来记录该标识符的名字，变量 `hash` 用于计算该标识符名字的 `hash` 值，优化符号表的查找速度。标识符识别完成后，若符号表中没有同名符号，则需要向符号表插入新的符号，并写入标识符的 `type`、`name`、`hash` 信息。

C++ 语言支持的数字格式有八进制、十进制和十六进制。八进制由 0 开始，十六进制由 0x 开始，识别后的值转化成十进制返回。其实现代码如图 4.5 所示：

```

    if (isNum(curr)) {
        value = curr - '0';
        curr = get();
        if (value != 0) { //十进制数
            while (isNum(curr)) {
                value = value * 10 + curr - '0';
                curr = get();
            }
        }
        else {
            if (curr == 'x' || curr == 'X') { //十六进制数
                curr = get();
                bool hasNum = false;
                while (isNum(curr) || isLowercase(curr) || isUppercase(curr)) {
                    hasNum = true;
                    value = value * 16 + (curr & 15) + (curr >= 'A' ? 9 : 0);
                    curr = get();
                }
                if (!hasNum) { throw Error(line, "Invalid hex type."); }
            }
            else { //八进制数
                while (curr >= '0' && curr <= '7') {
                    value = value * 8 + curr - '0';
                    curr = get();
                }
            }
        }
        return { NUM_INT, value };
    }
}
    
```

图 4.5 数字的识别

由图 4.5 可知支持三种数字，对于十六进制数，如果 0x 前缀后面没有跟数字，则会进行错误提示。最后返回值为{NUM\_INT, value}，表示数字的类型为 NUM\_INT，在识别过程中使用 value 记录其十进制表示的值，然后传递给解析模块。

其它符号通常由一个或两个字符组成，通过直接判断字符就可以完成识别。

## 4.2 解析模块的实现

解析模块调用词法分析模块获得词法单元，完成语法分析、语义分析及目标代码生成的工作。此外还需完善符号表信息，并进行错误处理。

### 4.2.1 全局变量的解析

全局变量的定义不在函数体内，不会生成目标代码，因此在编译时必须完善其所有信息，如存储位置、初始值等。若指定初始值，则初始值必须为常量，或已定义的全局变量。全局变量存储在虚拟机数据段中，其符号的 `value` 属性存放的是该变量的存储位置。变量的解析代码如图 4.6 所示。

```
void global_var_decl(int type) {
    Token& tk = table->getCurrentToken();
    tk.klass = GLOBAL_VARIABLE;
    tk.dataType = type;
    tk.value = reinterpret_cast<int>(vm->getNextDataPos(INT_TYPE));
    vm->addDataInt(0);

    //变量初始化
    if (tokenInfo.first == ASSIGN) {
        ... //将初始化值写入tk.value中
    }
}
```

图 4.6 全局变量的解析代码

由图 4.6 可知，`type` 参数为全局变量的类型，`global_var_decl` 完善全局变量的属性以及完成全局变量初始化的工作。变量 `Token` 的 `value` 属性记录的是变量的存储位置，需要通过 `vm->getNextDataPos(INT_TYPE)` 函数调用来申请一个存储单元，同时默认值为 0。若后续有赋值符号，说明有初始值，因此需要将全局变量的初始化值写入到 `tk.value` 所指的内存空间中，完成对全局变量的初始化。

### 4.2.2 全局数组的解析

数组是多个同类数据的集合，定义时需要指定长度，可以提供初始化列表。数组在存储时，会申请数组长度加 4 个字节的存储空间，第一个 4 字节的存储空间存放首元素的地址，每个元素在其后相邻存储。因为变量和数组的 `Token.value` 属性记录的是其位置，因此在使用它们时，都是获取 `Token.value` 属性所指内存的值，因此数组变量的 `Token.value` 属性需要是数组的指针的指针。例如有数组定义：`int arr[4] = {1, 2, 3, 4}`；则在虚拟机数据段中的存放格式如图 4.7 所示。

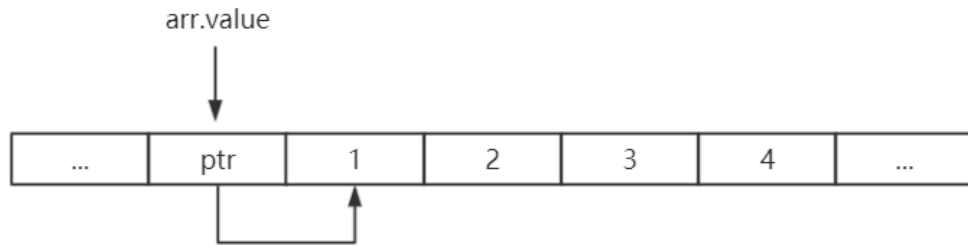


图 4.7 数组的存储

由图 4.7 可知，记录数组信息的 Token 其 value 属性指向 ptr，ptr 再指向数组的第一个元素，数组的元素在 ptr 后相邻存储。

全局数组的解析代码如图 4.8 所示。

```
//完善数组的Token信息
Token& tk = table->getCurrentToken();
tk.klass = GLOBAL_VARIABLE;
tk.dataType = type + PTR_TYPE;
tk.value = int(vm->getNextDataPos(type));

vm->addDataInt(0);
*(int*)(tk.value) = int(vm->getNextDataPos(type));

match(LBRACK);
if (tokenInfo.first != NUM_INT) {
    throw Error(lexer->getLine(), "wrong array declaration.");
}
int arraySize = std::min(MAX_INT, std::max(1, tokenInfo.second));
//申请arraySize个数据位置
if (type == CHAR_TYPE) vm->addDataDefaultChars(arraySize);
else vm->addDataDefaultInts(arraySize);
matchAny();
match(RBRACK);

//数组初始化
if (tokenInfo.first == ASSIGN) {
    ... //对数组元素进行初始化
}
```

图 4.8 全局数组的解析代码

由图 4.8 可知，首先需要完善全局数组的 Token 信息，如种类为 GLOBAL\_VARIABLE，数据类型为元素类型的指针，value 为指向数组首元素指针的指针。如果数组元素个数不是



数值型则报错，提示错误的数组定义。根据数组元素个数来向虚拟机申请数组的存储空间，并写入默认值 0。如果后续是赋值符号，则需要解析数组的初始化列表，并完成数组元素的初始化。

### 4.2.3 全局 enum 的解析

C++语言中，enum 的作用是定义多个值自增长的全局变量。第一个变量默认为 0。在解析过程中需要记录上一个变量的值，若变量定义没有提供初始值，则使用上一个变量的值加 1，否则，使用初始值。例如 enum 定义：

```
enum {a, b=2, c, d=1, e, f, g=10000};
```

a 默认值为 0。b 提供了初始值，因此值为 2。c 没有提供初始值，值为 b+1 即 3，d 的值为 1，e 的值为 2，f 的值为 3，g 的值为 10000。

enum 的解析代码如图 4.9 所示。

```
void enum_decl(){
    int varValue = 0; //enum常量的值
    match(ENUM);
    match(LBRACE);
    while (tokenInfo.first != RBRACE) {
        match(ID);
        if (tokenInfo.first == ASSIGN) {
            ... //记录变量的初始值，并存储在varValue中
        }
        Token& tk = table->getCurrentToken();
        tk.klass = NUMBER;
        tk.dataType = INT_TYPE;
        tk.value = varValue++;
        //后续还有变量定义
        if (tokenInfo.first == COMMA) match(COMMA);
    }
    match(RBRACE);
    match(SEMICON);
}
```

图 4.9 enum 的解析代码

由图 4.9 可知，varValue 变量用于记录当前定义的 enum 变量的值，默认为前一个 enum

变量的值加 1，如果当前变量有定义初始值，则使用该值。最后设置记录当前变量的 Token 属性。tk.klass 是 NUMBER，表明这是一个值类型，tk.value 记录的是变量的值。

#### 4.2.4 函数的解析与目标代码生成

函数包括返回值、函数名、参数列表和函数体。在进行参数列表的解析时进入函数的作用域，在完成函数体解析后离开函数的作用域。

函数参数列表中定义参数作为函数的局部变量，在解析参数时，需要记录参数的类型，并写入记录该函数信息的 Token 内 argsDataType 属性中，以便在函数调用时能够对参数数量及类型进行语义检查。

函数参数列表的解析代码如图 4.10 所示。

```
match(LPAREN);
int dataType = INT_TYPE, params = 0;
while (tokenInfo.first != RPAREN) {
    ... //解析变量的类型，并保存在dataType中
    match(ID, FORMAT(format));

    //记录函数参数类型
    table->getToken(currFuncIndex).addArgument(dataType);

    //填入局部变量信息
    Token& tk = table->getCurrentToken();
    tk.klass = LOCAL_VARIABLE;
    tk.dataType = dataType;
    tk.value = params++;
    if (tokenInfo.first == COMMA) match(COMMA);
}
match(RPAREN);
//当前变量相对于bp寄存器的位置
indexOfBP = params + 1;
```

图 4.10 函数参数列表的解析代码

由图 4.10 可知，解析函数参数列表时，使用 dataType 变量记录变量类型，params 记录参数个数，每解析完一个参数，就记录该参数的类型，便于函数调用时进行参数的匹配。indexOfBP 变量记录函数调用栈帧中 bp 寄存器的位置。

函数体由各种语句组成，包括局部变量定义语句、if 语句、while 语句、return 语句、块语句、表达式语句。在为函数生成目标代码时，需要先生成 I\_ENT 指令为函数局部变量

预留栈空间，但因为局部变量可以在函数体内部任意位置定义，因此需要在解析函数体时记录变量的个数，解析完函数体后对 I\_ENT 指令的操作数进行回填。

局部变量定义语句同全局变量定义语句类似，但是其 Token 的 value 属性存储的是相对于 bp 寄存器的位置。例如记录变量 a 的 Token，其 value 值为 3，则说明在函数调用时其存储在栈上 bp+3 的位置。

#### 4.2.5 语句的解析与目标代码生成

语句包括 if 语句、while 语句、return 语句、块语句和表达式语句。其中 if 语句、while 语句和块语句在进入其本身包含的语句体内部时需要更新作用域，并在离开时恢复作用域，以便可以在更深层的作用域内定义不影响外部的局部变量。

（1）if 语句的解析代码如图 4.11 所示。

```
void if_stat() {
    match(IF); match(LPAREN);
    expression(); //E
    match(RPAREN);

    int *branch; //记录分支跳转地址
    vm->addInst(I_JZ);
    branch = vm->getNextTextPos();
    vm->addInstData(0); //占据一个位置，用以写入 I_JZ 的跳转位置

    ENTER_SCOPE; //进入作用域
    statement(variableIndex); //S1
    LEAVE_SCOPE; //离开作用域

    if (tokenInfo.first == ELSE) {
        match(ELSE);
        *branch = int(vm->getNextTextPos() + 2);
        vm->addInst(I_JMP);
        branch = vm->getNextTextPos();
        vm->addInstData(0); //占据一个位置

        ENTER_SCOPE;
        statement(variableIndex); //S2
        LEAVE_SCOPE;
    }
    *branch = int(vm->getNextTextPos());
}
```

图 4.11 if 语句的解析代码

由图 4.11 可知，首先匹配 if 关键字，然后解析控制表达式，使用变量 `branch` 记录当控制表达式为 `false` 时需要跳转的地址，便于后面对该地址进行回填。`ENTER_SCOPE` 和 `LEAVE_SCOPE` 用于进入新的作用域和离开该作用域，将其放在 `statement()`调用前后可以确保 `statement()`内部是新的作用域。if 语句控制流跳转如图 4.12 所示。

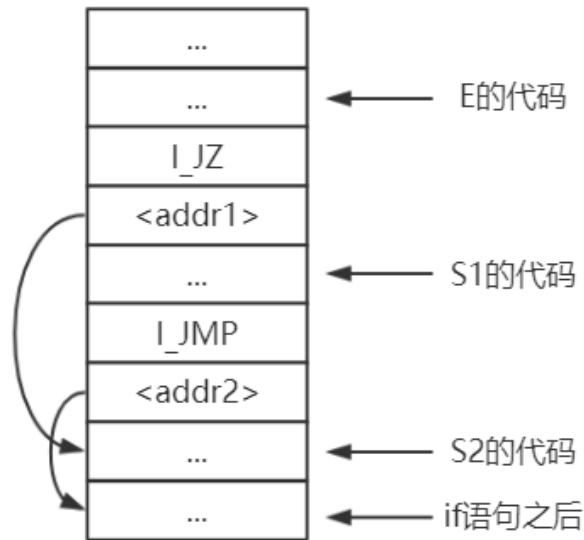


图 4.12 if 语句控制流跳转图

由图 4.12 可知，生成 E 的目标代码后，先生成指令 `I_JZ <addr1>`，此时 `<addr1>` 值未定。生成 S1 的目标代码后，生成指令 `I_JMP <addr2>`，此时 `<addr2>` 值未知，再将 `<addr1>` 置为 `<addr2>` 的后一个位置。生成 S2 的目标代码后，再更新 `<addr2>` 的值，即可完成整个 if 语句的控制流跳转。

(2) while 语句的解析代码如图 4.13 所示。

```

void while_stat() {
    int *branchTrue, *branchFalse; //记录分支跳转地址
    match(WHILE);
    branchTrue = vm->getNextTextPos();
    match(LPAREN);
    expression(); //E
    match(RPAREN);

    vm->addInst(I_JZ);
    branchFalse = vm->getNextTextPos();
    vm->addInstData(0);

    ENTER_SCOPE;
    statement(variableIndex); //S
    LEAVE_SCOPE;

    vm->addInst(I_JMP);
    vm->addInstData(int(branchTrue));
    *branchFalse = int(vm->getNextTextPos());
}
    
```

图 4.13 while 语句的解析代码

图 4-13 中，branchTrue 记录控制表达式 E 的首地址，便于语句体执行后跳转到该处。branchFalse 记录 while 语句生成的目标代码之后的地址，当控制表达式 E 为 false 时将跳转到该位置。将 statement() 用 ENTER\_SCOPE 和 LEAVE\_SCOPE 包裹，可确保 while 语句体内是新的作用域。while 语句的控制流跳转如图 4.14 所示。

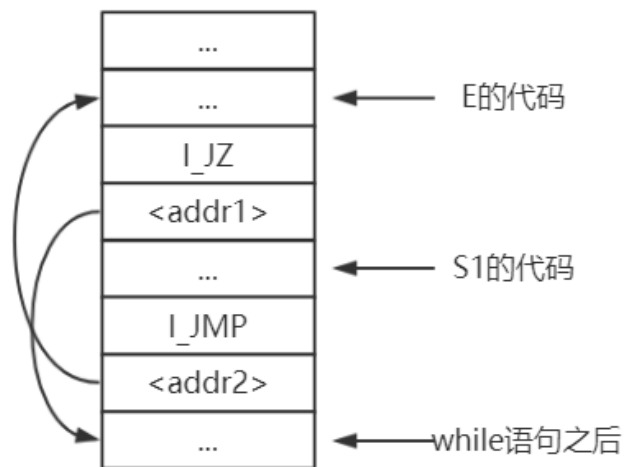


图 4.14 while 语句的控制流跳转图

由图 4.14 可知，生成 E 的目标代码后，I\_JZ <addr1>指令用于跳转到 while 语句之后。S1 的代码生成后，再生成 I\_JMP <addr2>指令跳转到 branchTrue 的位置，实现循环。branchTrue 记录<addr2>的值，branchFalse 记录<addr1>的值。

（3）return 语句的解析代码如图 4.15 所示。

```
void return_stat() {
    match(RETURN);
    if (tokenInfo.first != SEMICON) expression();
    match(SEMICON);
    vm->addInst(I_LEV);
}
```

图 4.15 return 语句的解析代码

由图 4.15 可知，首先匹配 return 关键字，如果后续字符不是分号，则解析表达式，然后匹配分号，最后生成 I\_LEV 指令用于退出函数调用。

（4）表达式语句有一元表达式和二元表达式，一元表达式形如 op expr1，如非语句!a，其解析代码如图 4.16 所示。

```
void not_expr {
    match(NOT);
    expression();

    //!expr 等价于 expr == 0
    vm->addInst(I_PUSH);
    vm->addInst(I_IMM);
    vm->addInstData(0);
    vm->addInst(I_EQ);

    exprType = INT_TYPE;
}
```

图 4.16 非语句的解析代码

由图 4.16 可知，首先匹配非操作符，然后解析表达式。因为!expr 等价于 expr==0，因此生成 I\_PUSH、I\_IMM 0 和 I\_EQ 三条指令完成 expr 的值与 0 的判断。因为结果值为 int 类型，因此需要设置表达式类型为 INT\_TYPE。

二元表达式形如 expr1 op expr2，例如赋值语句 a = b，其解析代码如图 4.17 所示。

```

void assign_expr(){
    or_expr();
    int tempType = exprType; //记录被赋值的表达式类型
    vm->deleteTopInst();      //删除I_LI或I_LC指令
    vm->addInst(I_PUSH);      //将左操作数入栈
    match(ASSIGN);
    or_expr();
    if(exprType != tempType) {
        WARNING->add(lexer->getLine(), "expression type not match.");
        exprType = tempType;
    }
    vm->addInst(exprType == CHAR_TYPE ? I_SC : I_SI);
}

```

图 4.17 赋值语句的解析代码

由图 4.17 可知，首先解析左操作数，然后用 `tempType` 记录该表达式的类型，因为左操作数是变量，因此 `or_expr()` 最后会先取出变量的地址，再生成 `I_LI` 或 `I_LC` 来取变量的值，但是赋值语句中，需要的是变量的地址，从而可以将右操作数的值写入该地址，因此需要使用 `vm->deleteTopInst()` 将最后一条指令删除。第二个 `or_expr()` 解析右操作数，此时 `exprType` 为右操作数的类型。两个操作数解析完成后，对其类型进行比较，如果类型不相同，说明有类型转换，故使用 `WARNING->add()` 函数添加警告。最后使用 `I_SC` 或 `I_SI` 指令将右操作数的值存入左操作数的存储空间中。

## 4.3 编译器测试

完成编译器的实现后，需要对编译器进行测试，以确保编译器能够正确地将源代码翻译成等价的目标代码。为了便于调试 C--源代码，增加了编译选项功能。

对编译器的测试包括正确性测试和健壮性测试。正确性测试中，编写一个包含大部分 C--语言功能的源文件，查看其运行结果是否正确，并对生成的目标代码进行详细的分析。健壮性测试中，编写多个存在各种错误的 C--源文件，查看编译器是否能够检测到错误并给出正确的错误信息。

### 4.3.1 编译选项

为了便于测试，增加编译选项功能。使用编译选项可以查看编译的中间过程、生成的

目标代码以及目标代码的运行情况等信息。各个编译选项的详细功能如表 4-1 所示。

表 4-1 编译选项

编译选项	描述
-test	运行所有测试文件
-gen	查看生成的目标代码
-gg	查看每个全局定义生成的目标代码
-e	查看执行的指令
-ev	查看执行的指令，以及各个寄存器的值
-nw	忽略编译警告
-dt1	显示少量编译过程
-dt2	显示适量编译过程
-dt3	显示所有编译过程
-df	将输出内容定向到 debug.txt

-test 选项会编译并运行项目中 test\_case 目录下所有的测试文件，可以方便地对编译器进行测试。

-gen 选项可以查看源文件编译生成的目标代码。

-gg 选项可以查看源文件生成的目标代码，并以全局定义进行划分。查看源代码中各个全局定义生成的目标代码。

-e 选项可以查看程序运行时执行的指令。

-ev 选项可以查看程序运行时执行的指令，以及指令执行过程中，各个寄存器的当前值。

-nw 选项可以忽略编译警告，编译结束后运行程序前不会输出任何编译警告。

-dt1 选项可以查看编译过程中的少量函数调用信息。-dt2 则会在-dt1 的基础上，输出函数调用时调用参数等信息。-dt3 在-dt2 的基础上，当符号表改变时输出所有符号表内容，以及函数调用过程中的分支执行情况等。通过这三个编译选项可以很方便地对编译器进行调试。

-df 选项可以将输出内容定向到 debug.txt 而非控制台中。

### 4.3.2 整体测试与目标代码分析

编译器的测试代码定义如图 4.18 所示。首先定义多个 enum 变量、4 个元素的数组以

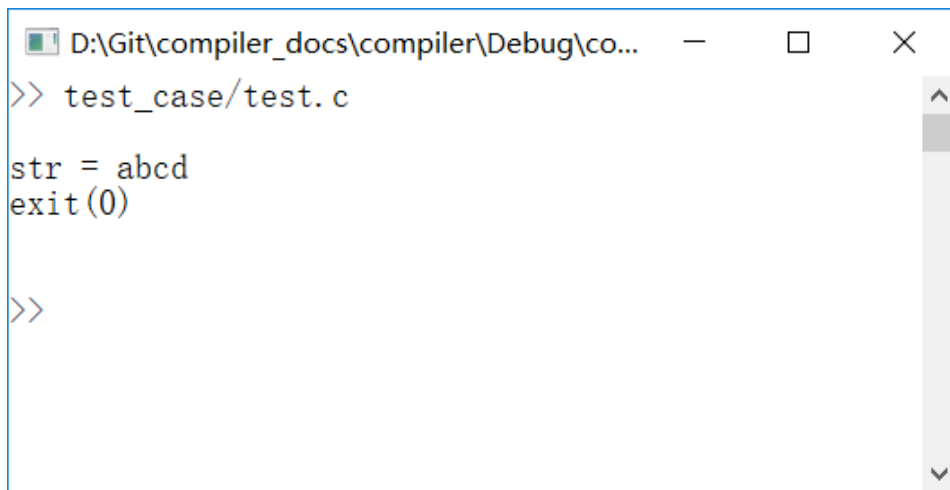


及一个字符串，然后定义了 4 个函数。测试代码覆盖了 C--语言的大部分功能。函数 sum 对数组进行求和，实现对数组、局部变量、while 语句、加法、自减、判断等功能的测试。函数 judgeEnum 用于判断 enum 变量是否有正确的值，实现对 enum、多语句嵌套、内建函数 printf 等功能的测试。judgeArray 函数则判断对数组的求和操作是否得到了正确的结果，测试了 if 语句以及函数调用等功能。main 函数是整个程序的入口，调用函数 judgeEnum 和 judgeArray，并测试了全局字符串变量 str 的值。其运行结果如图 4.19 所示。

```

C test.c  x
1
2  enum { e1, e2, e3 = 100, e4, e5 = -1};
3  int arr[4] = {1, 2, 3, 4};
4  char* str = "abcd";
5
6  int sum(int* arr, int len){
7      int result = 0;
8      while(--len >= 0) { result = result + arr[len]; }
9      return result;
10 }
11
12 void judgeEnum(){
13     if(e1 != 0 || e2 != 1 || e3 != 100 || e4 != 101 || e5 != -1){
14         printf("==== error! ====");
15     }
16 }
17
18 void judgeArray(){
19     if(sum(arr, 4) == 10) {}
20     else{ printf("==== error! ===="); }
21 }
22
23 int main(){
24     judgeEnum();
25     judgeArray();
26     printf("str = %s\n", str);
27     return 0;
28 }
    
```

图 4.18 测试代码



```
>> test_case/test.c  
str = abcd  
exit(0)  
  
>>
```

图 4.19 运行结果

由图 4.19 可知，测试文件中有对各种变量值、函数调用返回值等的判断，如果结果不与预期的相同则会输出“==== error! ====”，从运行结果中可以看出程序中各个部分都运行正确。顺利通过测试，说明编译器为测试代码生成了正确的目标代码。

下面通过-gg 编译选项来查看测试代码中每个全局定义生成的目标代码，并对生成的目标代码进行了详细地分析。

图 4.20 展示了 sum 函数定义生成的目标代码：

```

===== global declaration =====
int sum(int* arr, int len){
    int result = 0;
    while(--len >= 0) { result = result + arr[len]; }
    return result;
}

1      82006080      I_ENT      1
2      82006088      I_LEA      -1
3      82006096      I_PUSH
4      82006100      I_IMM      0
5      82006108      I_SI
6      82006112      I_LEA      2
7      82006120      I_PUSH
8      82006124      I_LI
9      82006128      I_PUSH
10     82006132      I_IMM      1
11     82006140      I_SUB
12     82006144      I_SI
13     82006148      I_PUSH
14     82006152      I_IMM      0
15     82006160      I_GE
16     82006164      I_JZ      82006268
17     82006172      I_LEA      -1
18     82006180      I_PUSH
19     82006184      I_LEA      -1
20     82006192      I_LI
21     82006196      I_PUSH
22     82006200      I_LEA      3
23     82006208      I_LI
24     82006212      I_PUSH
25     82006216      I_LEA      2
26     82006224      I_LI
27     82006228      I_PUSH
28     82006232      I_IMM      4
29     82006240      I_MUL
30     82006244      I_ADD
31     82006248      I_LI
32     82006252      I_ADD
33     82006256      I_SI
34     82006260      I_JMP      82006112
35     82006268      I_LEA      -1
36     82006276      I_LI
37     82006280      I_LEV
38     82006284      I_LEV
    
```

图 4.20 sum 函数定义生成的目标代码

图 4.20 中，首先是 sum 函数的定义，同测试文件中定义一致，紧接着是其生成的 38 条目标代码，其中第一列为代码序号，第二列为该代码在内存中的位置，第三列为生成的目标代码。指令 I\_ENT 1 表示该函数有 1 个局部变量，即变量 result。I\_LEA -1 和 I\_PUSH 指令实现将变量 result 的地址放入栈顶，而 I\_IMM 0 和 I\_SI 指令实现将立即数 0 存入变量 result 中，实现对变量 result 的赋值。编号为 6 到 13 的指令完成变量 len 的自减，并将其放

到栈顶。编号为 14 的指令 `I_IMM 0` 将参与  $\geq$  比较的 0 放入 `ax` 寄存器中。`I_GE` 指令实现  $\geq$  比较，并将结果存储在 `ax` 寄存器中。`I_JZ` 指令判断比较结果，若为 0 则跳转到 `while` 语句之后，在  $\geq$  操作为 `false` 时才会执行该条语句。编号为 17 到 33 的指令为 `while` 的语句体所生成的指令，`I_JMP` 指令实现跳转到 `while` 语句开始。`I_LEA -1` 和 `I_LI` 将变量 `result` 的值存储在 `ax` 寄存器中，因为函数调用约定中 `ax` 寄存器用于存放函数调用的返回值。最后的两条 `I_LEV` 指令中，第一条由 `return` 语句生成，第二条为 `sum` 函数定义结束时生成。

图 4.21 展示了 `judgeEnum` 函数定义生成的目标代码：

```

===== global declaration =====
void judgeEnum() {
    if(e1 != 0 || e2 != 1 || e3 != 100 || e4 != 101 || e5 != -1) {
        printf("==== error! ====");
    }
}

1      82006288      I_ENT      0
2      82006296      I_IMM      0
3      82006304      I_PUSH
4      82006308      I_IMM      0
5      82006316      I_NE
6      82006320      I_JNZ      82006352
7      82006328      I_IMM      1
8      82006336      I_PUSH
9      82006340      I_IMM      1
10     82006348      I_NE
11     82006352      I_JNZ      82006384
12     82006360      I_IMM      100
13     82006368      I_PUSH
14     82006372      I_IMM      100
15     82006380      I_NE
16     82006384      I_JNZ      82006416
17     82006392      I_IMM      101
18     82006400      I_PUSH
19     82006404      I_IMM      101
20     82006412      I_NE
21     82006416      I_JNZ      82006448
22     82006424      I_IMM      -1
23     82006432      I_PUSH
24     82006436      I_IMM      -1
25     82006444      I_NE
26     82006448      I_JZ       82006480
27     82006456      I_IMM      79335517
28     82006464      I_PUSH
29     82006468      I_PRTF
30     82006472      I_ADJ      1
31     82006480      I_LEV
    
```

图 4.21 `judgeEnum` 函数定义生成的目标代码

图 4.21 中，第一条指令 `I_ENT 0` 表示 `judgeEnum` 函数内没有局部变量。编号为 2 到 21 的指令共 4 组类似 `I_IMM a`、`I_PUSH`、`I_IMM b`、`I_JNZ addr` 形式的指令，它们完成 `a!=b`

的判断，若为 true，则跳转到 addr，否则继续执行，它们分别是 e1!=0、e2!=1、e3!=100、e4!=101 生成的目标代码。编号从 22 到 25 是 e5!=-1 生成的目标代码，此时 ax 寄存器中的值是整个 if 语句中条件判断表达式的结果。I\_JZ 82006480 的功能是若 ax 寄存器中值为 0，则跳转到地址 82006480，即最后 I\_LEV 指令的地址，否则不进行跳转，执行编号为 27 到 30 的指令，对 printf 函数进行调用。

图 4.22 展示了 judgeArray 函数定义生成的目标代码：

```

===== global declaration =====
void judgeArray() {
    if(sum(arr, 4) == 10) {}
    else{ printf("==== error! ===="); }
}

1      82006484      I_ENT    0
2      82006492      I_IMM    79335488
3      82006500      I_LI
4      82006504      I_PUSH
5      82006508      I_IMM    4
6      82006516      I_PUSH
7      82006520      I_CALL    82006080
8      82006528      I_ADJ    2
9      82006536      I_PUSH
10     82006540      I_IMM    10
11     82006548      I_EQ
12     82006552      I_JZ      82006568
13     82006560      I_JMP      82006592
14     82006568      I_IMM    79335534
15     82006576      I_PUSH
16     82006580      I_PRTF
17     82006584      I_ADJ    1
18     82006592      I_LEV

```

图 4.22 judgeArray 函数定义生成的目标代码

图 4.22 中，第一条指令 I\_ENT 0 表示 judgeArray 函数内部没有局部变量。编号为 2 到 6 的指令将 sum 函数调用的两个参数 arr 和 4 入栈，I\_CALL 53719104 指令则调用函数 sum，53719104 为函数 sum 的首地址。I\_ADJ 2 表示将 sum 函数调用的两个参数出栈。I\_PUSH 指令将 sum 函数调用的返回值放入栈顶，I\_IMM 10 指令将 sum(ar, 4) == 10 表达式中的 10 入栈，I\_EQ 指令执行等于比较，结果置于 ax 寄存器中。I\_JZ 82006568 表示若比较结果为 0 则跳转 else 分支的语句体内，否则跳转到控制表达式为 true 的分支语句体内。编号为 13 到 17 的指令为 else 分支的语句体产生的目标代码，即将 printf 调用的参数字符串地址入栈，然后生成 I\_PRTF 指令调用 printf 函数，最后将参数出栈。

图 4.23 展示了 main 函数定义生成的目标代码：

```

===== global declaration =====
int main() {
    judgeEnum();
    judgeArray();
    printf("str = %s\n", str);
    return 0;
}

1      82006596      I_ENT    0
2      82006604      I_CALL   82006288
3      82006612      I_CALL   82006484
4      82006620      I_IMM    79335551
5      82006628      I_PUSH
6      82006632      I_IMM    79335508
7      82006640      I_LI
8      82006644      I_PUSH
9      82006648      I_PRTF
10     82006652      I_ADJ    2
11     82006660      I_IMM    0
12     82006668      I_LEV
13     82006672      I_LEV
    
```

图 4.23 main 函数定义生成的目标代码

图 4.23 中，I\_ENT 0 指令表示 main 函数没有局部变量，I\_CALL 82006288 和 I\_CALL 82006484 分别是对 judgeEnum 和 judgeArray 的函数调用，编号为 4 到 8 的指令完成将字符串“str = %s\n”的地址和全局变量 str 的地址入栈，I\_PRTF 指令完成 printf 函数调用，I\_ADJ 2 指令将 printf 函数调用的两个参数出栈。编号为 11 的 I\_IMM 0 指令和编号为 12 的 I\_LEV 指令由语句 return 0 生成，最后的 I\_LEV 指令则是在 main 函数结束时生成。

通过对源代码生成的目标代码进行分析，可以看出包含函数调用、变量使用，复杂表达式的 C--源程序，都能够生成正确的目标代码，验证了编译器的正确性。

### 4.3.3 错误处理测试

对于源代码中的问题，编译器应该能够给出相应的提示，因此本节特别对错误的源代码进行测试，验证编译器的健壮性。

(1) 语法错误。对于语法错误的源代码，编译器需要对语法错误的位置，以及期待的符号给出正确的提示。测试代码如图 4.24 所示，运行结果如图 4.25 所示。

```
1  int main {    //main函数后方缺少括号
2      return 0;
3  }
```

图 4.24 语法错误测试代码

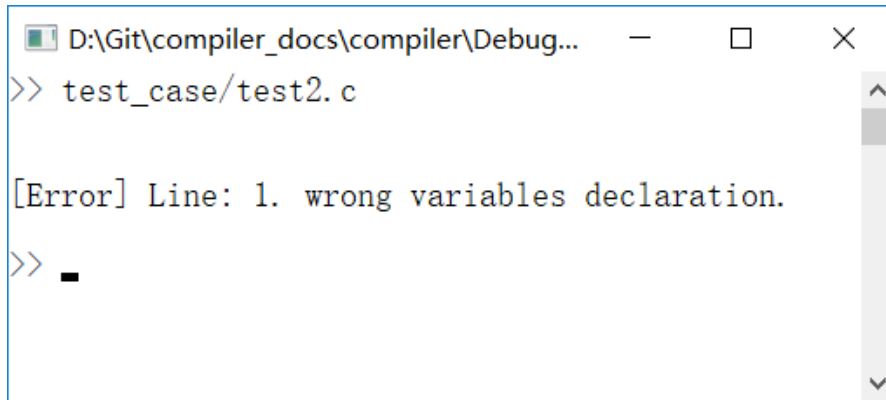


图 4.25 语法错误测试结果

图 4.24 中，main 函数定义缺少小括号，存在语义错误。图 4.25 运行结果提示源代码第一行存在错误的变量定义。因为标识符后不是小括号或方括号，则被识别为变量，而变量定义后方只能是逗号或分号，而源代码中 main 标识符后方为大括号，因此识别到语法错误。

（2）函数调用参数错误。函数调用中，编译器需要将函数调用参数与函数定义中的参数列表进行匹配，若不完全匹配则给出相应的提示。测试代码如图 4.26 所示，测试结果如图 4.27 所示。

```
1  void func(int a, int b) { }
2
3  int main(){
4      func(1); //参数太少
5      return 0;
6  }
```

图 4.26 函数调用参数错误测试代码

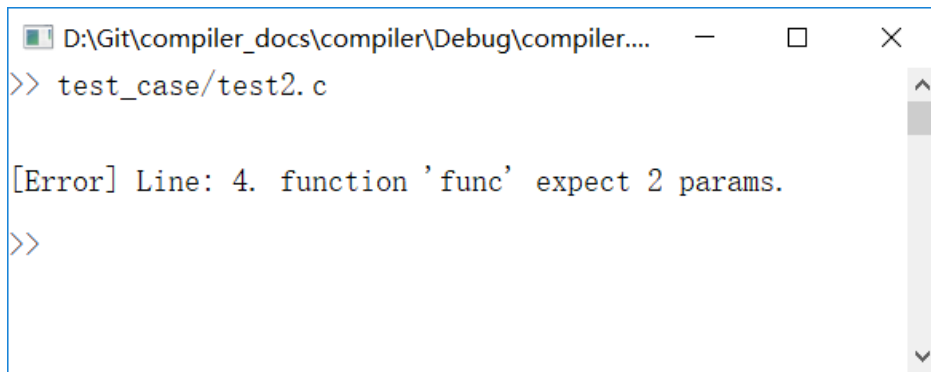


图 4.27 函数调用参数错误测试结果

图 4.26 中，fun 函数定义需要两个 int 型参数，而 main 函数中对 func 的调用只有一个参数，因此存在语义错误。由图 4.27 的运行结果可知，编译器对函数调用参数要求给出了详细的错误提示。

（3）符号未定义错误。使用未定义符号时，编译器应该提示符号未定义错误。测试代码如图 4.28 所示，测试结果如图 4.29 所示。

```

1  int main(){
2      int a = 1;
3      int b = 2;
4      c = a + b;
5      return 0;
6  }
```

图 4.28 符号未定义测试代码

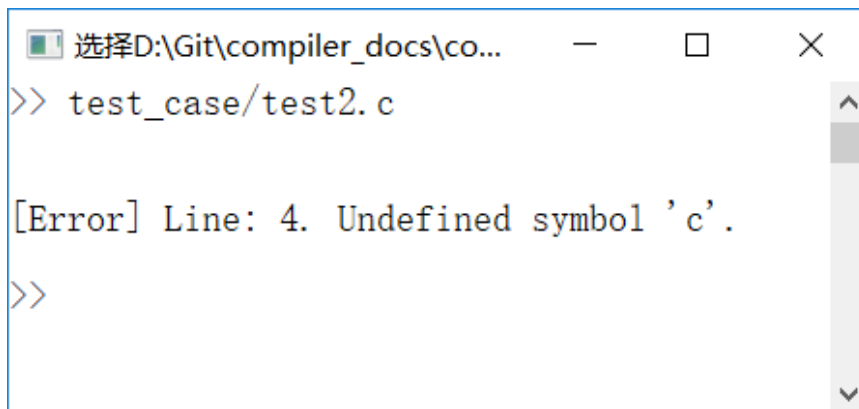


图 4.29 符号未定义测试结果

由图 4.28 可知，第 4 行代码使用了未定义的变量 c。图 4.29 的运行结果指出了变量 c 未定义，以及出错的位置为源代码中的第 4 行。

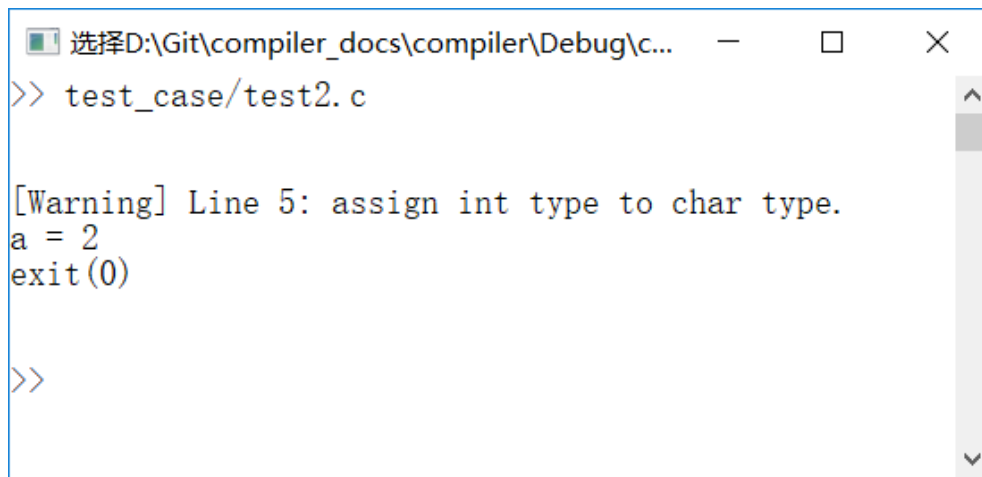


（4）运算符不匹配警告。测试代码如图 4.30 所示，测试结果如图 4.31 所示。

```

1  int main(){
2      char a = 'a';
3      int b = 20;
4      int c = 30;
5      a = b + c;
6      printf("a = %c\n", a);
7      return 0;
8  }
    
```

图 4.30 运算符不匹配测试代码



```

选择D:\Git\compiler_docs\compiler\Debug\c...
>> test_case/test2.c

[Warning] Line 5: assign int type to char type.
a = 2
exit(0)

>>
    
```

图 4.31 运算符不匹配测试结果

由图 4.30 可知，变量 `a` 是 `char` 型，`b+c` 的结果是 `int` 型，将 `int` 型赋值给 `char` 型有数据溢出的风险。图 4.31 中的运行结果指出，源代码中第 5 行的赋值语句存在将 `int` 型值赋予 `char` 型值的问题。Warning 不会终止程序的运行，目标代码运行后输出“`a = 2`”，与预期相同。

对存在错误的源代码，编译器都能指出错误位置并给出详细的错误描述，验证了编译器的健壮性。

## 第 5 章 总结与展望

### 5.1 本文总结

编译器是计算机系统软件的重要组成部分，为计算机的发展提供了重要的基础。编译器是一个复杂的程序，涉及多个阶段如词法分析、语法分析、语义分析、代码生成等，每个阶段也有适用于不同场景的多种算法。

主要的内容总结如下：

(1) 采用 EBNF 形式对 C--语言进行文法定义，设计了一个基于栈的虚拟机，定义了其指令集。虚拟机的指令作为 C--编译器的目标代码。

(2) 基于 C--语言的文法设计并实现了单趟式扫描的编译器。在词法分析模块中对源语言单词进行了类别定义及属性表示，完成了基于硬编码方式的词法分析算法。解析模块采用语法制导的翻译方法，在语法分析过程中插入语义分析、代码生成语句完成翻译。

(3) 采用使用不同 C--语言功能的源文件对编译器进行了测试，验证了编译器的正确性和健壮性，并对生成的目标代码进行了详细的分析。

通过实现一个语言编译器及其目标虚拟机，学习到了编译原理中的各种知识，并对程序运行原理及计算机内部结构有了更加深入的了解。

### 5.2 未来展望

本编译器采用的是单趟式编译方式，虽然实现了 C--语言的功能，但是相对于多趟式编译方式，编译过程中将语法分析、语义分析与代码生成组合在一起完成，结构不是特别清晰，部分功能没有参照标准做法来实现，存在一些性能上的不足，代码冗余比较多。整体存在很多需要改进的地方，主要包括：

(1) 采用多趟式编译方式，使整体结构更清晰。

由于采用单趟式编译，很多模块放在了一起实现，编译器的整体结构比较混乱，而使用多趟式编译，各个模块之间可以更加独立，从而可以分别针对不同模块的特点进行实现，提高性能与可读性。

(2) 增加生成中间代码阶段，以及代码优化阶段。

本编译器在代码生成阶段直接生成虚拟机指令，没有进行任何优化，因此可能存在多

余指令。可以增加生成中间代码阶段，以及代码优化阶段，先生成中间代码，然后进行代码优化，最后生成目标代码，从而提高程序的运行性能。

## 参考文献

- [1] Christopher W.Fraser, David R.Hanson. 王挺, 黄春, 等. 可变目标 C 编译器: 设计与实现[M]. 北京: 机械工业出版社, 2016.11.
- [2] Brian W.Kernighan, Dennis M.Ritchi. 徐宝文, 李志. C 程序设计语言[M]. 北京: 机械工业出版社, 2004.1.
- [3] Mayur Pandey. LLVM Cookbook[M]. Birmingham: Packet Publishing, 2015.5.
- [4] 王爽. 汇编语言[M]. 北京: 清华大学出版社, 2013.9.
- [5] Bryant,Radal E. Computer Systems: A Programmer's Perspective[M]. London: Pearson, 2015.6.
- [6] Alfred V.Aho, Monica S.Lam, Ravi Sethi, Jeffrey D.Ullman. 赵建华, 郑滔, 戴新宇. 编译原理[M]. 北京: 机械工业出版社, 2009.1.
- [7] Dick Grune, Criel J. H.Jacobs. Parsing Techniques: A Practicla Guide[M]. New York: Springer-Verlag, 2008.
- [8] 范志东, 张琼声. 自己动手构造编译系统: 编译、汇编与连接[M]. 北京: 机械工业出版社, 2016.8.
- [9] 王博俊, 张宇. 自己动手写编译器、链接器[M]. 北京: 清华大学出版社, 2015.2.
- [10] Stanley B.Lippman. 侯捷.深度探索 C++对象模型[M]. 北京: 电子工业出版社, 2012.1.
- [11] Scott Meyers. Effective Modern C++[M]. 南京: 东南大学出版社, 2015.9.
- [12] Scott Meyers. Effective C++[M]. 北京: 电子工业出版社, 2011.1.
- [13] Nicolai M.Josuttis. C++标准库[M]. 北京: 电子工业出版社, 2012.9.
- [14] Stanley B.Lippman, Josee Lajoie, Barbara E.Moo. C++ Primer[M]. 北京: 电子工业出版社, 2013.9.
- [15] Mark Allen Weiss. 风舜玺. 数据结构与算法: C 语言描述[M]. 北京: 机械工业出版社, 2004.1.
- [16] Thomas H.Cormen, Charles E.Leiserson, Ronald L.Rivest, Clifford Stein. 算法导论[M]. 北京: 机械工业出版社, 2013.1.

## 致 谢

在大学度过了四年紧张的学习时光，系统地学习了计算机的各方面知识，深深地佩服各位老师的学识，在此表示真挚的谢意。

毕业设计是检验大学四年学习成果最好的实践，在做毕业设计的过程中，指导老师不仅从撰写开题报告、项目实现等过程中提供了很多宝贵的意见，在写论文的过程中也从格式规范、各个章节内容的安排、遣词造句等方面也给我提出了十分详细的修改意见，让我能够顺利地完成论文的撰写，在此表示衷心的感谢。

最后要感谢我的家人，在这二十几年里，是家人无私地奉献，在成长的过程中教会了我做人的道理，并教导我要不断努力进取，我才能最终走进理想的大学并完成学业。