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、保密囗，在 年解密后适用本授权书

2、不保密囗 。

（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

编译器通常由一系列阶段组成：词法分析、语法分析、语义分析、代码生成等。编译器有多趟式编译器和单趟式编译器之分，多趟式编译器实现为多个相对独立的阶段，每个阶段都将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。单趟式编译器则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，实现过程更加简单，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。本文的目的是实现一个能够正确将源代码翻译成目标代码的编译器，主要内容集中在编译器前端部分，因此采用单趟方式实现，不生成中间代码，直接生成目标虚拟机指令。

主要的内容总结如下：

（1）定义一个支持变量、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，编译器将源代码翻译成虚拟机指令，并启动虚拟机实现目标代码的运行。

（2）采用硬编码方式实现词法分析程序，根据每个单词的符号组成，在词法分析程序中直接判断符号，完成词法单元的截取，为后续阶段提供词法单元。

（3）将语法分析、语义分析及代码生成进行组合，语法分析采用自顶向下的递归下降方法进行语句的解析，语义分析则只是简单的检查类型是否匹配，代码生成部分，则根据各种语句的代码生成方法直接生成虚拟机指令。

关键词：编译器；虚拟机；单趟式

**Abstract**

The compiler usually consists of a series of stages: the lexical analysis stage，the grammar analysis stage，the semantic analysis stage, the code generation stage. By partitioning the compiler, you can make the overall structure clearer and easier to write and maintain. 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 object 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 process is simpler, but the generated object code is not as efficient as the object code generated by the multi-pass compiler. The purpose of this article is to implement a compiler that can translate source code into object code correctly. The main content is concentrated in the front-end part of the compiler. Therefore, it is implemented in a single-pass way, virtual machine instructions are directly generated without generating intermediate code.

The main contents are summarized as follows:

(1) Define a C-- language that supports variables, functions, and various statements , and design a virtual machine whose instructions are used as object code. The compiler translates the source code into virtual machine instructions and starts the virtual machine to run the target code.

(2) A lexical analysis program is implemented in a hard-coded manner. According to the symbol composition of each word, symbols are directly judged in the lexical analysis program, and the lexical unit is intercepted, and a lexical unit is provided for the subsequent stage.

(3) A parsing program that combines parsing, semantic analysis, and code generation is performed. Parsing is performed using a top-down recursive descent method for parsing the statements. Semantic parsing is simply to check whether the types match, and the code generation part is based on the code generation method of various statements directly generates a virtual machine instruction.

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

目 录

[摘 要 I](#_Toc514398507)

[Abstract II](#_Toc514398508)

[第1章 绪论 1](#_Toc514398509)

[1.1 研究背景和意义 1](#_Toc514398510)

[1.2 编译器技术的研究现状 2](#_Toc514398511)

[1.3 主要研究内容 2](#_Toc514398512)

[第2章 高级语言C--与虚拟机 4](#_Toc514398513)

[2.1 高级语言C--文法定义 4](#_Toc514398514)

[2.2 虚拟机的设计 7](#_Toc514398515)

[2.2.1 指令集的设计 7](#_Toc514398516)

[2.2.2 数据与指令的存储 9](#_Toc514398517)

[第3章 编译器设计 10](#_Toc514398518)

[3.1 编译器的整体结构 10](#_Toc514398519)

[3.2 符号表设计 10](#_Toc514398520)

[3.2.1 符号的表示 10](#_Toc514398521)

[3.2.2 作用域的管理 12](#_Toc514398522)

[3.2.3 符号的查找与建立 13](#_Toc514398523)

[3.3 词法分析算法的设计 13](#_Toc514398524)

[3.4 语法分析算法的设计 14](#_Toc514398525)

[3.5 语义分析及代码生成算法设计 15](#_Toc514398526)

[3.5.1 语义分析方法 15](#_Toc514398527)

[3.5.2 代码生成算法设计 15](#_Toc514398528)

[第4章 编译器实现与测试 19](#_Toc514398529)

[4.1 编译器的实现 19](#_Toc514398530)

[4.1.1 处理内建符号 19](#_Toc514398531)

[4.1.2 符号的识别 19](#_Toc514398532)

[4.1.3 全局变量的解析 21](#_Toc514398533)

[4.1.4 全局数组的解析 21](#_Toc514398534)

[4.1.5 全局enum的解析 22](#_Toc514398535)

[4.1.6 函数的解析与代码生成 22](#_Toc514398536)

[4.1.7 语句的解析与代码生成 23](#_Toc514398537)

[4.2 错误处理 25](#_Toc514398538)

[4.3 编译选项 26](#_Toc514398539)

[4.4 编译器测试 27](#_Toc514398540)

[第5章 总结与展望 33](#_Toc514398541)

[5.1 本文总结 33](#_Toc514398542)

[5.2 未来展望 33](#_Toc514398543)

[致 谢 35](#_Toc514398544)

[参考文献 36](#_Toc514398545)

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。编译过程通常被划分成多个阶段，每个阶段都将源程序的一种表示形式转化为另一种表示形式。通过对阶段的划分，可以让编译器各个阶段能够相对独立，各自采用不同的算法来满足不同的要求。编译器的总体结构如图1.2所示。

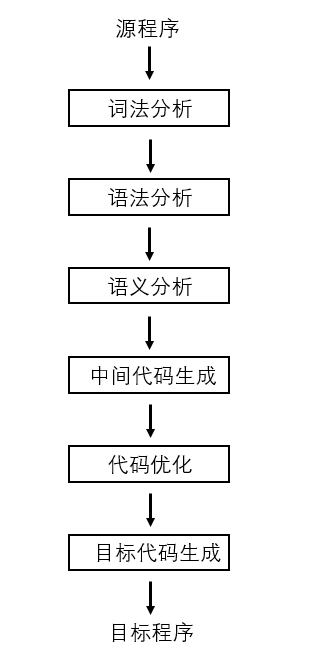


图1.2 编译的总体结构

编译器的实现涉及各种编译原理知识的学习和对程序的运行逻辑的了解。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

【写一下当前的技术研究现状】

【各个阶段当前的技术讲解】

因为编译器作为基础软件，历史十分悠久，很多人都在研究，所以编译器的很多技术已经相当成熟。例如主流的三大C/C++编译器：GCC、Clang、cl，词法分析器生成程序Lex及更加先进的Flex，语法分析器生成程序Yacc以及更加先进的Bison，编译器基础设施框架LLVM等。

GCC是一套由GNU开发的编译器，以GPL及LGPL许可证发布，并被很多操作系统视为标准的编译器。GCC 6.0提供了很多新的特性，如OpenMP 4.5、段寄存器支持、目标克隆、扩展存数规则等，完全支持C++14，并支持C++17的实验功能。

Lex是词法分析器生成器，yacc是语法分析器生成器，它们结合使用可以完成编译器前端的大部分工作。Lex采用确定的有穷自动机（DFA）的方式实现对正则表达式的识别，只需提供各种单词的正则表达式，就能使用Lex完成词法分析程序。Yacc采用LALR(1)语法分析算法实现，只需要提供语言文法描述，就可以通过Yacc实现一个语法分析器。这两个工具为编译器前端的实现提供了巨大帮助。

LLVM是Lower Level Virtual Machine的缩写，但它并不只是创建一个虚拟机，它是当今最流行的编译器基础框架。许多语言的编译器负责完成编译器前端的工作，生成的中间代码为LLVM IR，后端部分由LLVM进行程序优化和跨平台代码生成，可以很方便地实现一门语言。编译器领域的发展中LLVM起到了举足轻重的作用。

1.3 主要研究内容

编译器是一个语言翻译程序，能够把由源语言书写的程序翻译成等价的由目标语言书写的程序。整个翻译过程通常被划分为多个阶段，将编译过程划分为多个阶段可以降低实现复杂度，而且每个阶段都可以采用各自适当的方式进行实现而不影响其他阶段，降低各个阶段间的耦合。本编译器的目的是正确地将源代码翻译成目标代码，不考虑代码优化，因此不涉及中间代码的生成以及代码优化。

词法分析是编译过程的第一个阶段，任务是对源程序字符流进行扫描和分解，从而识别出一个个的单词。词法分析方法通常包括基于有限自动机的方式以及硬编码方式。

语法分析的作用是识别由词法分析器获取的单词符号序列是否是给定文法的正确句子。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。

语义分析则将变量定义与它们的使用联系起来，检查每一个表达式是否有正确的类型，并将抽象语法转换成更简单的、适合生成代码的表示。因为本编译器采用单趟式编译，没有语法树等内容，因此语义分析阶段的任务只是进行类型检查。

代码生成阶段是为各种语句生成目标代码。这个阶段中主要解决的问题是理解各种语句的代码生成规则并生成对应的目标代码。

此外，一个完整的编译器还应该包括符号表管理和错误处理。这两个内容贯穿整个编译流程，编译各个阶段都涉及到查找和更新各种符号数据，同时编译过程中可能发现源程序的各种错误，需要提供错误的性质以及错误发生位置，以便用户能够快速且方便地定位错误。

第2章 高级语言C--与虚拟机

2.1 高级语言C--文法定义

文法是一种用有限的规则定义无限集合的方法[1]。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[2]。

【【【】】】

（1）程序。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。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对数组进行初始化。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。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。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对高级语言中优先级进行了定义，其中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> ::= '++' | '--'

元素表达式则不包含任何运算符，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。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。通常数字字符的长度是有限的，但是文法中无法描述数字的长度，因此在词法分析中处理这个问题。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> }\* '"'

空串则不包含任何内容

<null> ::=

通过文法的定义，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 虚拟机的设计

【【【本语言采用自己设计的虚拟机作为目标机，虚拟机的指令作为编译器生成的目标代码。】】】【新增】

虚拟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看到运行结果，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等，如指令：add reg1, reg2, reg3，一条指令就能够实现将两个存在reg2和reg3中的数相加并存储于reg1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，结果放于栈顶中。因为其操作数隐含在栈上，因此指令通常不用指定操作数，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，内存访问次数增多，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。】】】【新增】

2.2.1 指令集的设计

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。跳转指令则控制程序的跳转。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是根据bp寄存器来确定的，因此I\_LEA指令的操作数<offset>表示该变量相对于bp寄存器的位置。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> | 取立即数num |
| I\_JZ <addr> | 取bp+num地址处的值 |
| I\_JNZ <addr> | 获取一个int型数据 |

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寄存器值为非零时跳转。

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[3]，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，并将参数出栈完成函数调用。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 |
| I\_ADJ | 将函数调用参数出栈 |

表2-4中，I\_CALL指令完成保存调用现场的功能，并将程序执行流程切换到函数内部。I\_ENT指令则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 ^ 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 <= b |
| I\_GE | 大于等于运算，a >= b |
| I\_SHL | 左移运算，a << b |
| I\_SHR | 右移运算，a >> 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printf函数 |
| I\_MALC | 调用C标准库中的malloc函数 |
| I\_EXIT | 调用C标准库中的exit函数 |
| I\_SCANF | 调用C标准库中的scanf函数 |
| I\_GETC | 调用C标准库中的getc函数 |
| I\_PUTC | 调用C标准库中的putc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。

2.2.2 数据与指令的存储

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[4]。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[5]。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，且指令操作数最大大小为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

3.1 编译器的整体结构

本编译器采用单趟式实现，即完成整个编译过程只需扫描一遍源代码，其整体结构如图3-1所示。在进行实现之前，需要对编译器进行详细的设计，完善各个模块需要完成的内容。

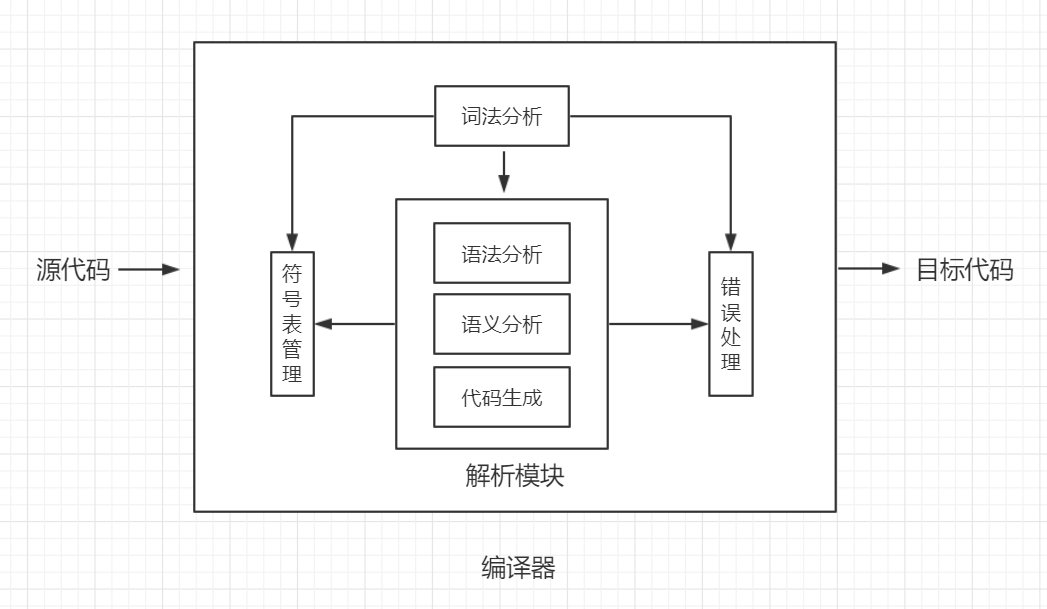


图3-1 编译器整体结构

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。词法分析器每次从源代码中获取一个词法单元，并提供给解析模块，解析模块进行语法分析、语义分析以及代码生成。在此过程中还需要进行符号表管理以及错误处理。

3.2 符号表设计

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—符号[1]。因为本编译器实现的高级语言语法较为简单，因此符号表设计为符号的数组。

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 |

仅仅是符号类型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展示了符号的种类及其描述：

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，本编译器支持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见表3-3：

表3-3 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> |

3.2.2 作用域的管理

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标识，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，离开作用域时则去掉尾部的值，就可完成不同作用域的标识。作用域示例如图3-1所示：



图3-1 作用域的标识

3.2.3 符号的查找与建立

在进行词法分析时，若遇到标识符，需要对符号表进行查找。先在当前作用域内查找，如果没有找到，则在更上一级的作用域内查找。如果该符号已存在，则直接返回对应的符号。否则需要新建符号，并将部分信息如符号名、hash值等写入。

对于内建的符号，如关键字、内建函数等，需要在词法分析之前预先将其插入到符号表中，避免词法分析时遇到这些符号从而识别成用户定义的标识符。

3.3 词法分析算法的设计

词法分析阶段的主要任务是从源代码中截取出单词，并将部分信息写入符号Token中，为后续阶段提供符号信息。常用的词法分析方法有两种，分别是基于有限自动机的词法分析算法，和硬编码方式的词法分析算法。

基于有限自动机的方式使用正则表达式来描述词法单元，使用有限自动机进行单词识别。对于一个词法单元，首先使用正则表达式对其进行表示，然后将该正则表达式转换成等价的NFA，进而再将NFA转换成DFA，则可以实现对满足该正则表达式的词法单元进行识别。通过配置正则表达式来实现词法分析程序的自动构造，具有很强的通用性。

硬编码的方式则根据给定词法单元描述直接在程序中识别词法单元所包含的符号，较基于有限自动机的方式更加简单直接，但是无法完成词法分析程序的自动构造，每一种词法单元都需要编写对应的识别代码。

由于硬编码方式的词法分析器更加直观且易于编写，因此本编译器的词法分析模块采用硬编码方式来实现。

3.4 语法分析算法的设计

语法分析过程确认输入是否符合语言的文法规则，并建立输入源程序的内部表示[1]。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，递归下降方法和预测分析法是常用的语法分析方法。自底向上分析又包括算符优先分析和LR分析。它们各有优缺点，分别适用于不同的场景。

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。对于图3-2所示的文法定义，编写的递归过程如图3-3所示。

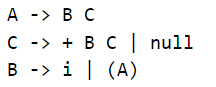


图3-2 文法定义

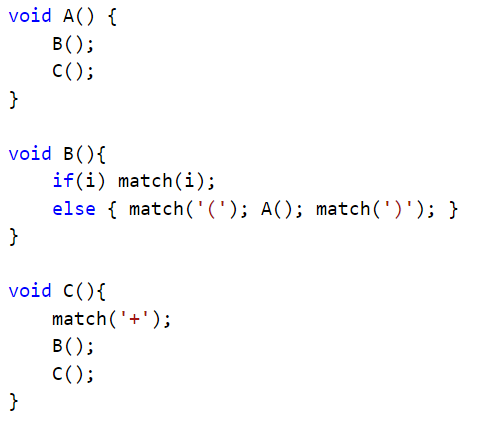


图3-3 递归过程

由图3-2可知文法定义有3个非终结符，因此需要有3个递归过程，每个递归过程根据对应的文法描述来识别其推出的串。递归下降法虽然递归调用多导致运行速度较慢，但是实现过程十分直观且容易编写，因此语法分析器模块采用递归下降方法进行实现。

3.5 语义分析及代码生成算法设计

3.5.1 语义分析方法

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是否匹配，如加法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是否相同，各个参数类型是否同函数定义一致等。表达式类型匹配可以通过一个全局的变量exprType来记录，而函数参数个数及类型Token类已设计有argsData属性来进行记录，只需在解析函数时将函数参数正确地记录，函数调用时就可以方便地进行参数匹配。

3.5.2 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定义。函数调用时虚拟机内部需要创建函数调用栈帧，对于图3-2的函数定义：

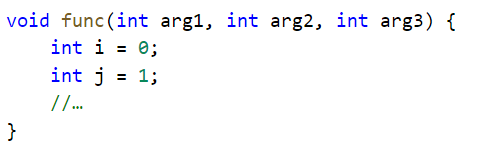


图3-2 函数定义

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



图3-3 函数调用栈帧结构

参数顺序入栈，I\_CALL指令完成保存返回地址、保存bp寄存器、更新bp寄存器与pc寄存器的功能，而I\_ENT指令完成更新sp寄存器，以便为局部变量预留空间。I\_CALL指令在函数调用语句中生成，而I\_ENT指令则需要在函数解析时生成在函数本身的代码首部。

在解析完函数参数列表，并解析函数体前首先生成I\_ENT指令用于为局部变量预留栈空间。完成函数体的解析并生成相应指令后，再生成一条I\_LEV指令，用于离开函数体。函数定义生成的目标代码结构如图3-4所示：

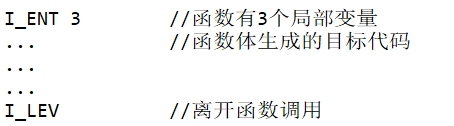


图3-4 函数定义生成的目标代码结构

函数体由各种语句组成，语句包括if语句、while语句、return语句、块语句、表达式语句。

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，其代码生成结构如图3-5所示：

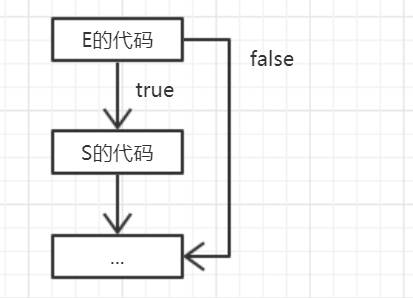


图3-5 if语句的代码结构

如果表达式E的运算结果为true，则执行语句S生成的代码，否则不执行语句S生成的代码。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。其代码生成结构如图3-6所示：

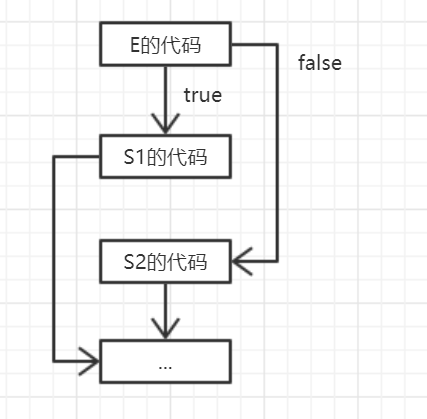


图3-6 if-else语句的代码结构

while语句与if(E) S语句类似，但是while语句在执行了S语句会再次回到其代码开始，直到表达式E的运算结果为false。其代码结构如图3-7所示：

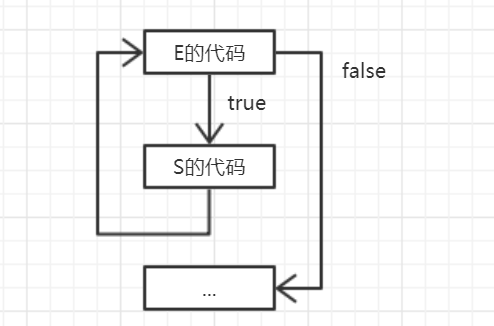


图3-7 while语句的代码结构

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。

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。

表达式语句分为一元表达式和二元表达式。一元表达式先求出子表达式的值，再生成其目标代码。如函数调用语句中，先生成各个参数表达式的目标代码，再将参数入栈，最后生成函数调用指令I\_CALL。再如取值表达式\*expr，先生成expr表达式的代码，再生成I\_LI或I\_LC获取expr表达式计算得到的地址指向的值。二元表达式形如expr1 op expr2，将先生成expr1的代码，然后生成expr2的代码，再生成指令op对应的指令完成表达式的计算。

第4章 编译器实现与测试

4.1 编译器的实现

4.1.1 处理内建符号

本语言包括一些关键字及内建函数，不允许用户对这些符号进行定义，否则将提示错误。在对源代码进行词法分析之前，预先对这些符号进行处理，分别写入各自的信息，并插入到符号表中。所有关键字在作用域KEY\_WORD\_SCOPE中，其值为-1，在对符号表进行查询时，会先判断该符号是否在KEY\_WORD\_SCOPE中有同名符号，若有则识别为关键字或内建函数，否则再根据作用域的查找规则，先从当前作用域查找，然后一层层向外查找。

4.1.2 符号的识别

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。因此其实现逻辑如图4-1所示：



图4-1 标识符的识别

本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，并且将识别后的值转化成十进制返回。其实现逻辑如图4-2所示：



图4-2 数字的识别

注释包含两种类型，行注释由//开始直到行结尾，块注释则由/\*和\*/包裹。注释的识别逻辑如图4-3所示：

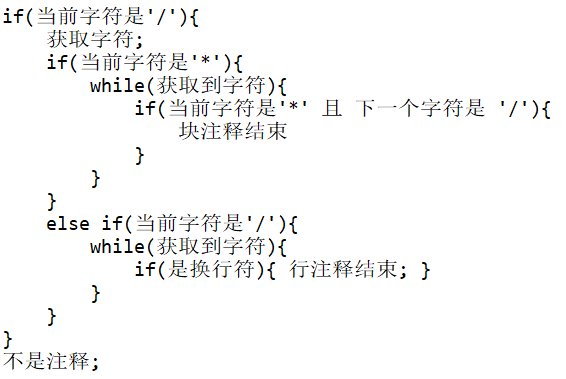


图4-3 注释的识别

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+字符的形式表示转义字符，如’\a’。本语言只支持’\n’表示换行，以及’\0’表示字符0，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。

其余的符号都比较简单，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，因则需要判断后一个字符来确定。遇到换行符时，需要对行号加1，便于后续阶段报错时能够指明错误出现的行号，空白符则直接跳过，不视作任何词法单元。

4.1.3 全局变量的解析

全局变量的定义不在任何函数体内，编译时必须完善其所有信息，如存放位置、初始值等。若是变量定义，则初始值必须为常量，或已定义的全局变量。全局变量的存储在虚拟机数据段中，其Token的value域存放的是其存储位置的地址。

对于全局变量定义语句：int a=1; 解析完a后，发现后续不是‘(’或者‘[’，则判定a不是函数定义或数组定义，而是全局变量声明或定义，这会在虚拟机数据段中请求一个4字节大小的空间作为该变量的存储空间。发现后续是‘=’，则继续进行变量初始化。

4.1.4 全局数组的解析

数组是多个同类数据的集合，定义时需要指定数量，可以提供初始化列表。数组在存储时，第一个空间存放其首元素的地址，每个元素在其后相邻存储，因此其占用的存储空间是所有元素占用的存储空间再加上4个字节。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4-4所示：

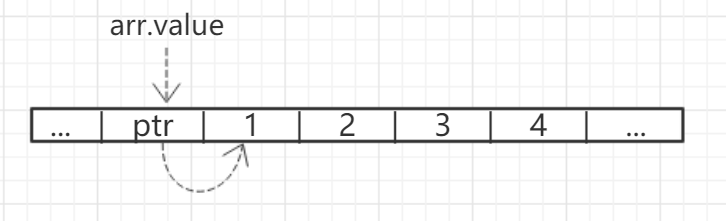


图4-4 数组的存储

对于数组定义语句：int arr[4] = {1, 2, 3, 4}; 解析完arr后，发现后续符号是左方括号‘[’，则判定为数组定义，解析完4后，会在虚拟机数据段中请求5（1个数组首元素指针加上4个元素）个4字节大小的连续空间。发现后续是等于符号‘=’，则继续解析数组的初始化列表，并将初始化列表中的值写入对应的数组元素中。

4.1.5 全局enum的解析

本语言中，enum的作用仅仅是定义多个值自增长的全局变量。第一个变量默认为0，若设定了初始值，则值为设定的值。在解析过程中需要记录上一个变量的值，若后一个变量没有提供初始值，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。

4.1.6 函数的解析与目标代码生成

函数包括参数列表和函数体。在进入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的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。函数生成的目标代码结构如图3-4所示。

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。

4.1.7 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。

对于没有else分支的if语句，其代码生成结构如图3-3所示，最终生成的代码结构如图4-5所示：

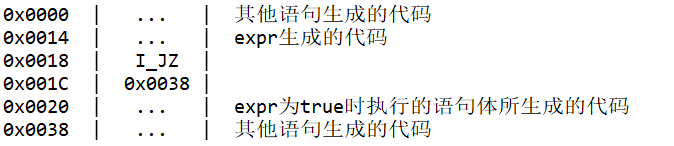


图4-5 if语句生成的代码

图中expr为if语句的条件控制表达式。对于有else分支的if语句，其代码生成结构如图4-6所示，最终生成的代码结构如图3-7所示：

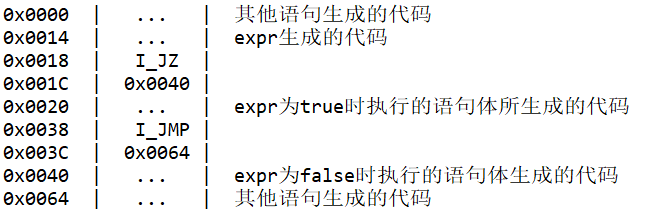


图4-6 if-else语句生成的代码

两种if语句中，I\_JZ和I\_JMP指令的操作数都需要在解析完后续语句后再回填。

while语句会在控制条件为true时循环执行其语句体，并且在进入语句体内部前需要更新作用域，并在离开语句体时恢复作用域。若控制条件为false，则需要跳转到其语句体生成代码的后方，也需要在解析完语句体后才能知道跳转位置从而进行回填。while语句最终生成的代码结构如图4-7所示：

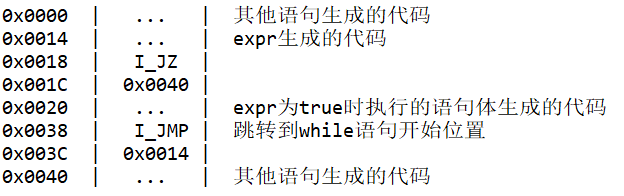


图4-7 while语句生成的代码

return语句会生成I\_LEV指令，结束函数调用，若return语句带有表达式，则先生成表达式的代码，再生成I\_LEV指令。Return语句最终生成的代码结构如图4-8所示：

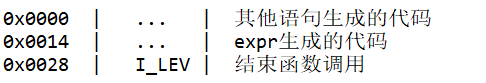


图4-8 return语句生成的代码

表达式语句分为一元表达式和二元表达式。一元表达式通常形如expr或op expr，如+a、sizeof(a)、!a、3、”abcd”等，会中先求出子表达式a的值，再生成相应代码。对于立即数如100，会生成指令I\_IMM 100。对于字符串，会生成I\_IMM addr形式，addr为该字符串首个字符的存储位置。

一元表达式中需要特别注意的是标识符的解析，标识符可能是函数名、局部变量、全局变量、enum变量。若标识符是函数名，则需要先解析每个实际参数，并用I\_PUSH指令将参数入栈，然后生成I\_CALL指令来调用函数。若标识符是enum变量，则使用I\_IMM指令直接取其值。否则使用I\_LEA指令获取局部变量在栈上的地址，或用I\_IMM指令获取全局变量的存储地址，然后再使用I\_LI或I\_LC取其值。

对于函数调用func(1, 2)，其生成的代码结构如图4-9所示：

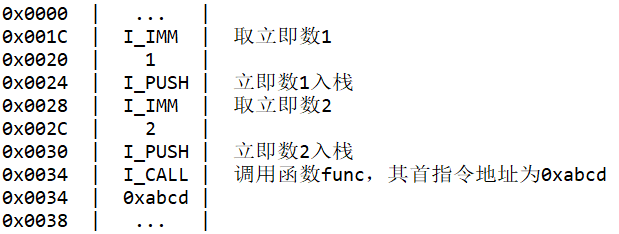


图4-9 函数调用func(1,2)生成的代码

取整型局部变量的值时，生成的代码结构如图3-12所示：

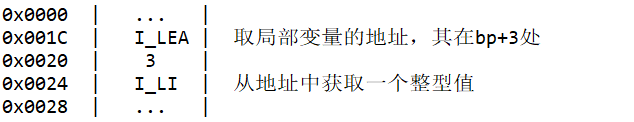


图4-10 取整型局部变量值所生成的代码

二元表达式由多个一元表达式或二元表达式组成，形如expr1 op expr2，如 a||b、a=b+c等，会先生成expr1的代码，并将计算结果入栈，然后生成expr2的代码，再生成op对应的代码来进行计算。如表达式a=b+c，a、b、c都是整型局部变量，则生成的代码结构如图4-11所示：

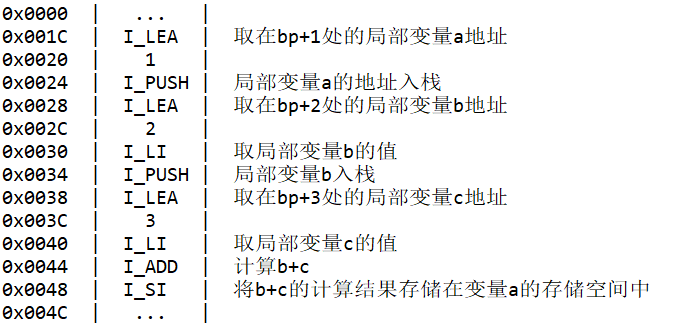


图4-11 表达式a = b + c生成的代码

4.2 错误处理

在语法分析过程中，会根据语言文法对源程序进行解析，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匹配等。若参数类型不同会提示错误，而表达式类型不匹配则进行警告。

程序中定义了Error类和Warning类来分别记录错误和警告。它们会记录错误或警告的位置及描述，在解析过程中检测到源程序语法错误或类型不匹配时，提供行号和错误描述即可。语法错误即源程序的编写方式不符合文法定义，在遇到语法错误时，可根据词法分析器提供的行号，以及错误位置，来构建Error。例如有函数调用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:

<args> = <expr> { ',' <expr> ] }\* | <null>

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。

对于函数调用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，然后将实际参数列表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型值赋予char型值有溢出的风险，从而提示警告。

4.3 编译选项

为了方便调试及测试，为编译器增加编译选项。共支持10个编译选项，且可以同时使用多个选项。详细信息如表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选项可以直接使用，会编译并运行项目中test\_case目录下所有的测试文件，方便对编译器进行测试。

-gen可以查看源文件编译生成的目标代码。

-gg选项可以将源文件生成的目标代码，并以全局定义进行分割，可以对照源代码看生成的目标代码。

-e选项可以查看当前运行的程序执行的指令。

-ev选项不仅可以查看当前运行程序执行的指令，还可以查看每条指令执行过程中，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。

4.4 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。测试方式为编写一个包含大部分本语言支持功能的源文件，查看其运行结果是否正确，然后再使用-gg编译选项对每个全局定义生成的目标代码进行分析。

测试代码定义如图4-12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。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、判断等内容的测试。函数judgeEnum用于判断enum变量是否有正确的值，实现对enum、逻辑或语句、多语句嵌套、内建函数printf等内容的测试。judgeArray函数则判断对数组的求和操作是否得到了正确的结果，测试了带有else分支的if语句。main函数则是整个程序的入口，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4-13所示。因为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。

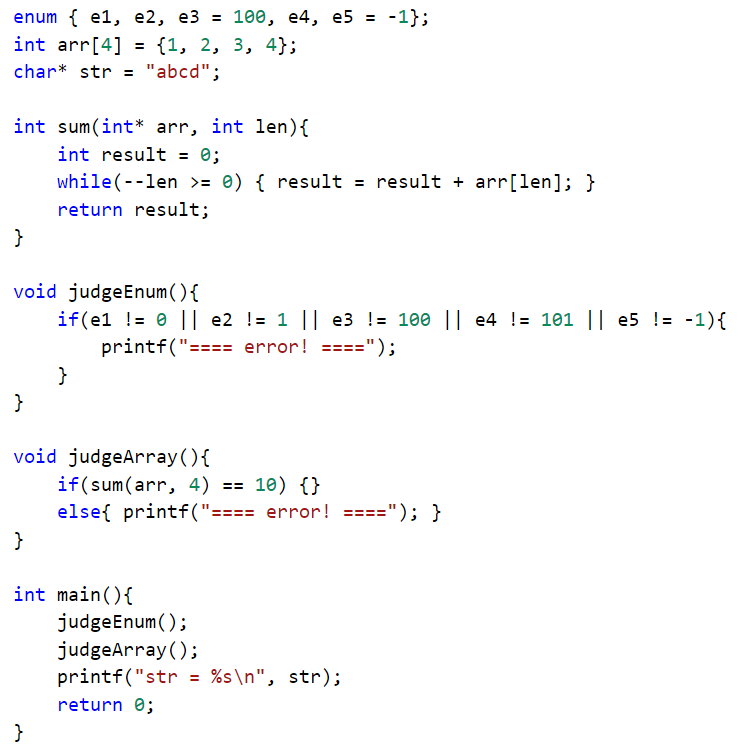


图4-12 测试代码

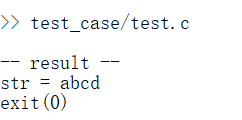


图4-13 运行结果

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编译选项来分析测试代码中每个全局定义生成的目标代码。 图4-14展示了sum函数定义生成的目标代码：

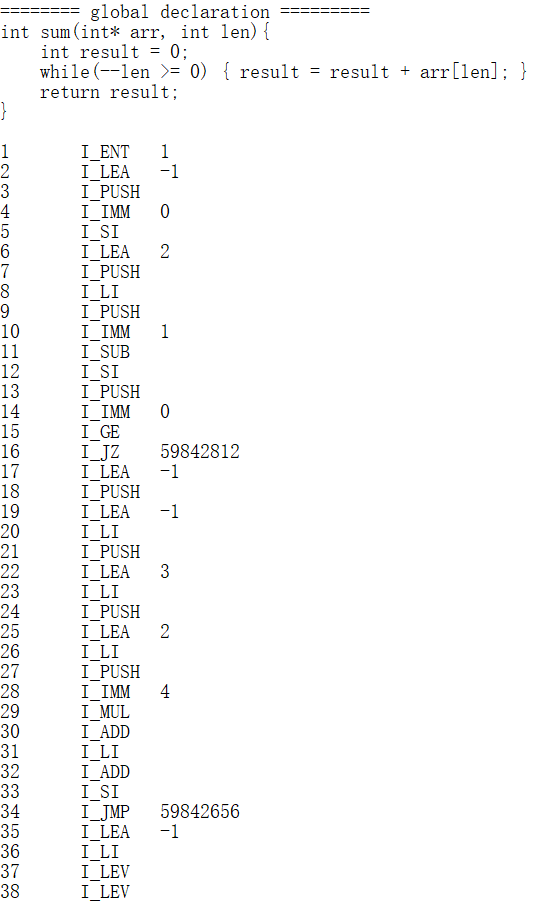


图4-14 sum函数定义生成的目标代码

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。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。第14条指令I\_IMM 0则将参与>=比较的0放入ax寄存器中。I\_GE指令实现>=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>=操作为false时才会执行该条语句。编号为17到33的指令为while语句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4-15展示了judgeEnum函数定义生成的目标代码：

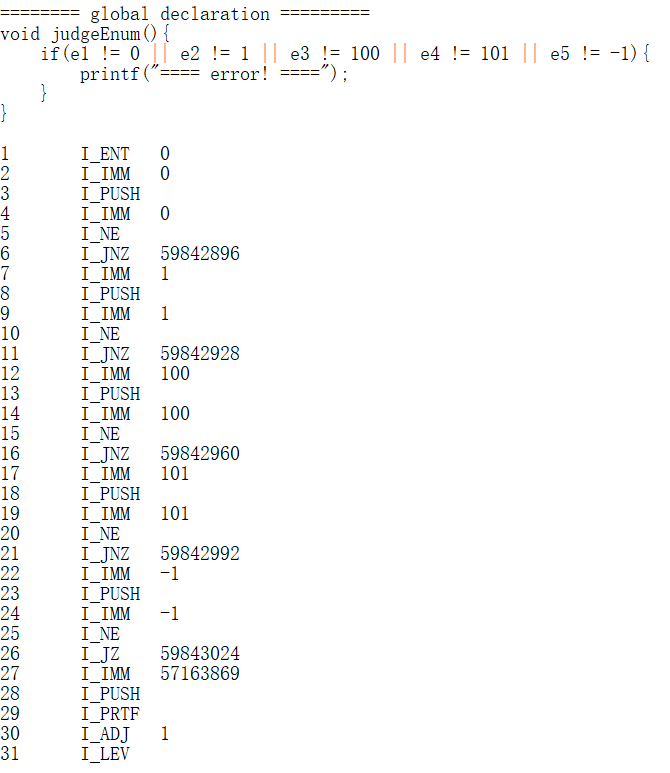


图4-15 judgeEnum函数定义生成的目标代码

第一条指令I\_ENT 0表示judgeEnum函数内没有局部变量。后续编号从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寄存器中的值便是整个if语句中条件判断表达式的结果。I\_JZ 59843024的功能是若ax寄存器中值为0，则跳转到地址59843024，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，即对printf函数进行调用。

图4-16展示了judgeArray函数定义生成的目标代码：

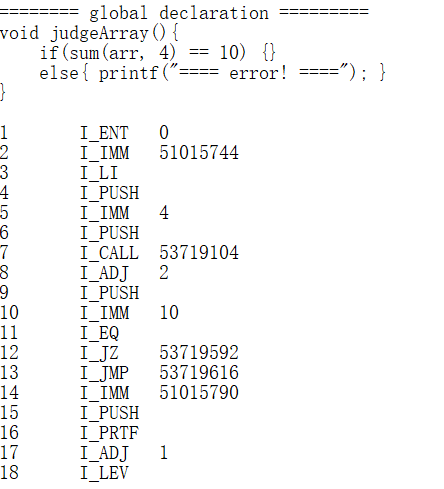


图4-16 judgeArray函数定义生成的目标代码

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 53719592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4-17展示了main函数定义生成的目标代码：

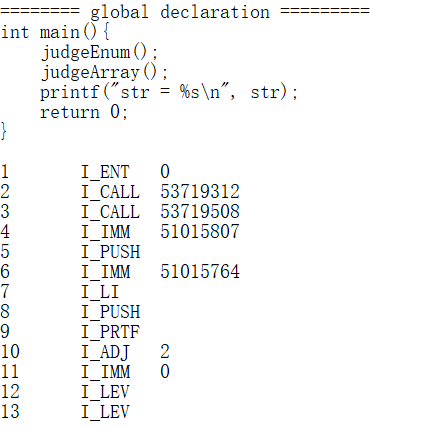


图4-17 main函数定义生成的目标代码

I\_ENT 0指令表示main函数没有局部变量，I\_CALL 53719312和I\_CALL 53719508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指令则完成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。

通过对测试代码生成的目标代码进行分析，可以看出函数调用、变量使用，甚至是十分复杂的表达式等都能够生成正确的目标代码，说明编译器的正确性和健壮性都十分良好。

第5章 总结与展望

5.1 本文总结

编译器是一个比较复杂的程序，涉及多个阶段如词法分析、语法分析、语义分析、代码生成等，每个阶段也有各自的多种算法。经过多方面的比较，最终选择了单趟式编译方式，使用硬编码方式的词法分析、递归下降的语法分析以及语法制导的翻译方式进行实现。

在设计编译器的过程中，发现将自定义虚拟机作为目标机是一个更加有意义的实现方式。实现一个虚拟机，能够学习到计算机内部的各种知识，如内存分配、数据存储、指令执行等。

编译器后端优化部分比较复杂且与学习目标不重叠，因此编译器的实现中没有代码优化阶段，中间代码生成阶段也直接替换成目标代码生成，因此编译器只有四个阶段。这四个阶段也让我对整个编译器的实现流程，以及对源程序到目标程序的转换有了更加深入的了解。

通过比较完善的测试，能够确定本编译器还算比较健壮，语法错误能够准确地进行定位并给出错误描述，源代码中可能出现的问题也能够以Warning的方式给出提示，测试中的比较复杂的程序也能正确运行。说明编译器整体功能比较完善。

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，则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，从而提高性能与可读性。

(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，因此目标程序可能存在部分多余指令。可以增加中间代码阶段和代码优化阶段，先生成中间代码，然后进行优化，将优化后的中间代码生成目标代码，可以提高程序的运行性能。

致 谢

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，Ceriel 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．