**目 录**

**第1章 绪论**

1.1 研究背景和意义

1.2 编译器技术的研究现状

1.3 主要研究内容

1.3.1 虚拟机

1.3.2 编译系统

**第2章 高级语言与目标机**

2.1 高级语言文法定义

2.2 虚拟机的设计

2.2.1 指令集设计

2.2.2 数据与指令的存储

**第3章 编译器设计**

3.1 符号表设计

3.1.1 符号的表示

3.1.2 作用域的管理

3.1.3 符号的查找与建立

3.2 词法分析算法设计

3.3 语法分析算法设计

3.4 语义分析与代码生成算法设计

3.4.1 语义分析方法

3.4.2 代码生成算法设计

**第4章 编译器实现与测试**

4.1 词法分析器的实现

4.1.2 处理内建符号

4.1.3 符号的识别

4.2解析程序的实现

4.2.1 全局变量的解析

4.2.2 全局数组的解析

4.2.3 全局enum的解析

4.2.4 函数的解析

4.2.5 语句的解析

4.2.6 编译错误和编译警告

4.3 编译选项与目标代码分析

4.3.1 编译选项

4.3.2目标代码分析

4.4 编译器测试

4.4.1 变量测试

4.4.2 enum测试

4.4.3 函数测试

4.4.4 语句的测试

4.4.5 整体测试

**第5章 总结与展望**

5.1 本文总结

5.2 未来展望

**致谢**

**参考文献**

摘要

编译器能将以某种语言写的程序转换为等价的目标语言写成的程序。编译器是计算机科学中理论与实践相结合的最好典范，通过编写编译器，不仅能更好地掌握编程语言本身，还能锻炼组织大量代码的能力、以及系统设计的能力。

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。通过对编译器进行阶段划分，可以让整体结构更加清晰，易于编写和维护。编译器有多趟式编译器和单趟式编译器之分，多趟式编译器通常实现为多个相对独立的阶段，每个阶段都将对源代码进行扫描，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，只对源代码扫描一次，就完成了编译过程，实现过程更加简单，但是生成的代码没有多趟式编译器生成的代码高效。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，因此主要内容集中在编译器前端部分，不生成中间代码，也不进行代码优化，直接生成目标虚拟机指令，采用单趟式实现。

主要的内容总结如下：

（1）定义C语言子集的文法，定义一个实现变量定义、函数定义、各种语句支持的C语言子集。并设计一个虚拟机，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，根据每个单词可能的符号组成，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融合在一起进行的程序，语法分析采用自顶向下的LL(1)方法进行语句的解析，语义分析则是简单的检查类型匹配，代码生成部分，则参考《编译原理》中各种语句的代码生成方法直接生成虚拟机指令。

关键字：编译器，虚拟机，单趟式

第一章 绪论

1.1 研究背景和意义

编译器是程序员使用的关键工具，必须生成高效的目标代码，但更重要的是，编译器必须生成正确的目标代码，只有可靠的编译器才能生成可靠的应用程序。编译器本身是一个大而复杂的应用程序。编译器构造原理和技术可以说是计算机科学理论与实践相结合的最好典范。

实现一个编译器，涉及各种编译知识的学习，如词法分析、语法分析、代码生成等，还能更加清晰地了解程序的运行逻辑，例如变量定义时如何存储变量、函数调用时参数传递规则、各种语句的运行方式等。因为编译器是一个比较复杂的程序，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的，也能有效地提升编程能力。

1.2 编译器技术的研究现状

上世纪50年代，IBM的John Backus带领了一个研究小组对FORTRAN语言及其编译器进行开发。由于当时人们对编译理论的了解不多，开发编译器既复杂又艰苦。与此同时Noam Chomsky也开始他对自然语言结构的研究。Chomsky研究出根据语言文法的难易程度以及识别它们所需的算法来对语言分类，即Chomsky架构。它包括了文法的四个层次：0型文法、1型文法、2型文法和3型文法。其中2型文法（上下文无关文法）被证明是程序设计语言中最有用的。解析问题（用于上下文无关文法识别的有效算法）的研究是在60年代和70年代，它相当完善地解决了这个问题，现在它已是编译原理中的一个标准部分。人们接着又深入研究有效生成目标代码的方法，人们通常称为优化技术，并被一直使用至今。

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（GUN Compiler Collection）是一套由GNU开发的编程语言编译器，是以GPL及LGPL许可证发布的自由软件，现已被大多数操作系统采纳为标准的编译器。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。

LLVM这个名字源于Lower Level Virtual Machine，但这个项目并不局限于创建一个虚拟机，它已经发展成为当今炙手可热的编译器基础框架。LLVM最初以C/C++为编译目标，近年来经过众多机构和开源社区的努力，LLVM已经能够为ActionScript、D、Fortran、Haskell、Java、Objective-C、Swift、Python、Ruby、Rust、Scala等众多语言提供编译支持，而一些新兴语言则直接采用了LLVM作为后端。可以说，LLVM对编译器领域的发展起到了举足轻重的作用。[《LLVM CookBook---需要英文版引用，否则不是进5年英文文献》]

Lex是一个词法分析器生成器，yacc是一个语法分析器生成器，它们结合使用可以完成编译器前端的大部分工作，生成的LLVM IR由LLVM进行代码优化、机器码生成等，为实现编程语言提供了方便，避免了很多重复的工作。

1.3 主要研究内容

编译器是一个语言翻译程序，能够把源语言书写的程序翻译成等价的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，每个阶段都将源程序的一种表示形式转换成另一种表示形式，并提供给下一个阶段使用。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。本编译器的目的是将源代码正确地翻译成目标代码，不考虑代码优化，因此不涉及中间代码生成及代码优化，主要实现了词法分析、语法分析、语义分析及目标代码生成四个阶段。

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就是进行类型检查。

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，并根据目标代码的格式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并解决错误。

第二章 高级语言与目标机

2.1 高级语言文法定义

一个程序设计语言是一个记号系统，如同自然语言一样，它的完整定义应包括语法和语义方面。一个语言的语法是指一组规则，用它可以形成和产生一个合适的程序。自然语言存在不同程度的二义性，这种模糊、不确定的方式无法精确定义一门程序设计语言。必须设计一种准确无误地描述程序设计语言的语法结构，这种严谨、简洁、易读的形式规则描述的语言结构模型称为文法。

目前广泛使用的手段是上下文无关文法，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。本高级语言是C语言的子集，因此参考C语言的语法结构。

程序可由任意多个全局定义组成。其文法定义为：

<program> = { <global\_decl> }\*

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，因此将它们合并为其他定义，再在其他定义中，将它们进行区分。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>

本高级语言将enum的功能设定为方便地定义多个值可自增长的全局变量，因此其变量都处于全局作用域中。其文法定义为：

<enum\_decl> = 'enum' '{' <id> [ '=' <number> ] { ',' <id> [ '=' <number> ] }\* '}' ';'

变量定义中，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> ] }\* ';'

数组定义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> }\* ] '}' ] ';'

函数定义包括函数参数定义和函数返回值定义。其文法定义为：

<func\_decl> = '(' <func\_param> ')' '{' <func\_body> '}'

<func\_param> = <null> | <type> { '\*' }\* <id> { ',' <type> { '\*' }\* <id> }\*

<func\_body> = { <local\_var> | <statement> }\*

局部变量定义类似于全局变量定义，但是允许使用表达式进行初始化，而不仅仅是常量。其文法定义为：

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

语句包含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> = <if\_stat> | <while\_stat> | 'return' <expr> ';' | '{' <statement> '}' | <expr> ';'

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

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

在进行表达式文法定义之前，需要明确表达式中各种运算符的优先级。下面对高级语言中优先级进行了定义：（//TODO:三线表）

表2-1 运算符优先级

|  |  |  |
| --- | --- | --- |
| 运算符 | 含义 | 优先级 |
| = | 赋值 | 10 |
| || | 逻辑或 | 9 |
| && | 逻辑与 | 8 |
| > < >= <= == != | 比较 | 7 |
| + - | 加法、减法 | 6 |
| \* / % | 乘法、除法、取模 | 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 |

定义好优先级后，就可以对表达式进行文法定义。其文法定义为：

<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>

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。[x]

2.2 虚拟机的设计

本文的虚拟机指的是高级语言虚拟机，其有自己的指令集架构，意义是实现高级语言的语义。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，其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。基于栈的虚拟机更加适合于内存和CPU处理速度等方面有限的系统，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，而基于栈的虚拟机则省去了这些不必要的麻烦，因此虚拟机设计部分采用基于栈的虚拟机。图2-1展示了栈式虚拟机的结构：

2.2.1 指令集的设计

指令可分为存取数指令、跳转指令、函数调用指令、算术运算指令和内建函数指令。存取数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。算术运算指令用以执行各种运算，如四则运算、取反、位移等。内建函数指令则是为了提供C语言部分库函数的功能而提供的指令。

存取数指令包括I\_IMM、I\_LEA、I\_LI、I\_LC、I\_SI、I\_SC、I\_PUSH，I\_IMM用于取立即数，I\_LEA用于取局部变量，I\_LI和I\_LC分别用于取int型数据的char型数据，I\_SI和I\_SC分别用于存储int型数据和char型数据，I\_PUSH则将ax寄存器中的内容入栈。

跳转指令包括I\_JMP、I\_JZ、I\_JNZ，分别完成无条件跳转、零跳转和非零跳转功能。

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.[《LLVM CookBook》]，即设计指令时需要设计好调用约定。在函数调用前，需要将参数入栈，即在编译器识别到函数调用时，需要先将函数参数入栈，然后使用I\_CALL指令用于保存现场并调用函数。函数调用结束后，由I\_ADJ将参数出栈，完成函数调用。在解析函数定义时，需要为函数局部变量预留栈空间，由I\_ENT来实现。在函数定义结束时生成I\_LEV指令将参数出栈，并恢复调用之前的现场。因此I\_CALL和I\_ADJ指令在函数调用及函数结束时生成，不属于函数本身的代码，而I\_ENT指令和I\_LEV指令在函数定义的解析时生成，作为函数本身的代码。

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。这些指令都不带参数，其第一个操作数在栈顶，第二个操作数在ax寄存器中，计算时栈顶元素出栈，与ax寄存器中的值计算，计算结果存放于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。因为编译器不支持多文件支持，因此无法直接使用C语言的各种库函数，因此特地设计几个内建函数指令，用于提供对常用C语言库函数的支持。其不带参数，而调用的库函数的参数存放在栈上，调用结果存放于ax寄存器中。

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。[《汇编语言》—王爽]。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形数据。

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，但是字符串以字符’\0’结尾，因此需要在末尾添加’\0’。

在存储整形数据时，则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。[《深入理解计算机系统》]。因为整形数据大小在32位机器上是4字节，因此在存储整形数据时需要进行数据对齐，让整形数据的起始地址为4的倍数，否则将无法正确取出整型数据。

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形的方式存储在代码段中。

第三章 编译器设计

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。在实现之前，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，以及需要保存的信息。

3.1 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。[《可变目标C编译器—设计与实现》]。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。

3.1.1 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，其有标识符、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 | ( ) |
| RBRACK / 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 / ADD / SUB / MUL / DIV / MOD | << >> + - \* / % |
| INC / DEC | ++ -- |
| STRING | 字符串 |

仅仅词法单元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：

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 |

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。

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

|  |  |  |
| --- | --- | --- |
| 属性名 | 描述 | 数据类型 |
| type | 类型 | int |
| name | 名字 | string |
| hash | hash值，用于符号名字的快速比较 | int |
| klass | 种类 | int |
| dataType | 数据类型 | int |
| value | 值 | value |
| argsDataType | 函数参数类型列表 | vector<int> |
| scope | 所在作用域 | vector<int> |

3.1.2 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。示例如图3-1：



图3-1 作用域的标识

3.1.3 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。先再当前作用域内查找，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了用户定义的标识符。

3.2 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，以及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。

3.3 语法分析器算法设计

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。[《可变目标C编译器—设计与实现》]。

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，其递归调用多且运行速度慢，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。

3.4 语义分析及代码生成算法设计

3.4.1 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。

3.4.2 代码生成算法设计

编译过程中涉及代码生成的只有函数。函数调用时虚拟机内部需要创建函数调用栈帧，对于函数定义：

void func(int arg1, int arg2, int arg3) {

int i = 0;

int j = 1;

//…

}

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



图3-2 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。函数生成的目标代码结构如图3-2所示：

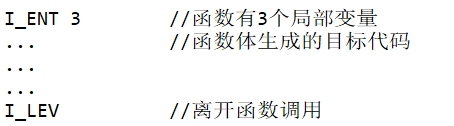


图3-2 函数生成的目标代码结构

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3-3所示：

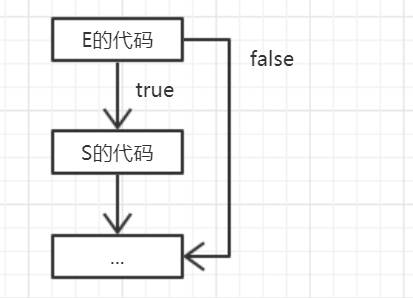


图3-3 if语句的代码结构

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3-4所示：

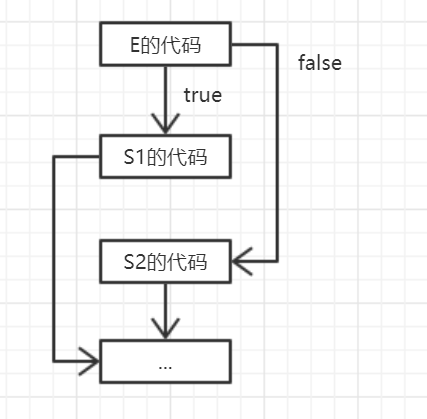


图3-4 if-else语句的代码结构

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3-5所示：

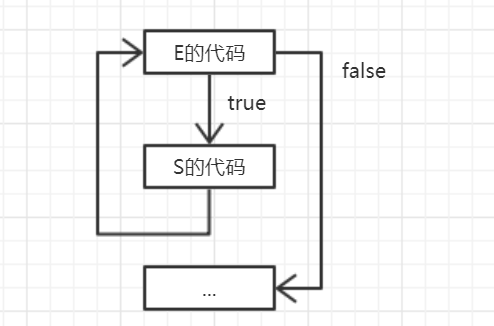


图3-5 while语句的代码结构

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。一元表达式中先求出子表达式的值，再生成其代码。如函数调用语句中，先生成各个参数表达式的代码，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。

第四章 编译器实现与测试

4.1 词法分析器的实现

词法分析器采用硬编码方式实现，根据各个词法单元的符号组成，直接进行匹配截取出词法单元，并完善符号的信息。

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解析，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。

4.1.3 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：

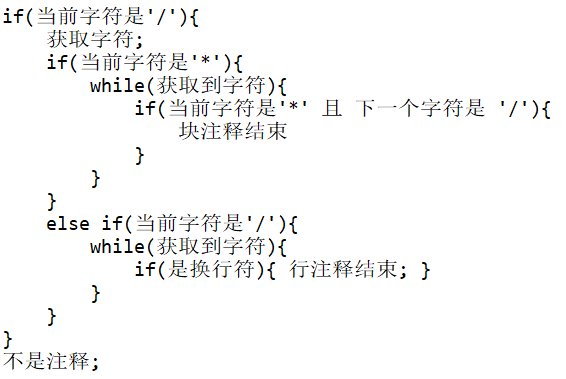


图4-3 注释的识别

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。

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，其余空白符则直接跳过，不视作任何词法单元。

4.2 解析程序的实现

因为本编译器采用单趟式实现，因此语法分析、语义分析及代码生成同时进行，在进行语法分析的同时，通过查看符号表中的信息进行语义分析，并在合适的位置生成目标代码。

全局定义中，变量定义、数组定义及enum定义不会生成任何代码，它们的初始化在编译时完成。函数定义会生成函数自身的代码。

4.2.1 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。

4.2.2 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：

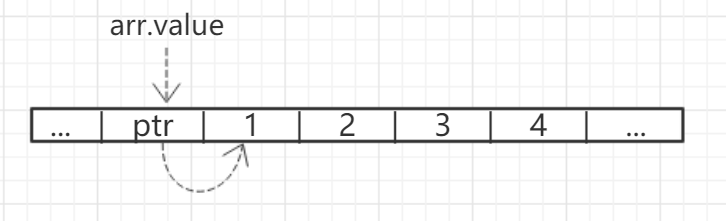


图4-4 数组的存储

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。

4.2.3 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，则值为初始值，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化值，则使用上一个变量的值加1，否则，使用初始化值。例如enum定义：

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

a默认值为0。b因为提供初始值，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。

4.2.4 函数的解析

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。

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

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：

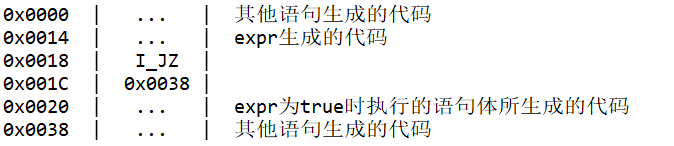


图4-5 if语句生成的代码

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：

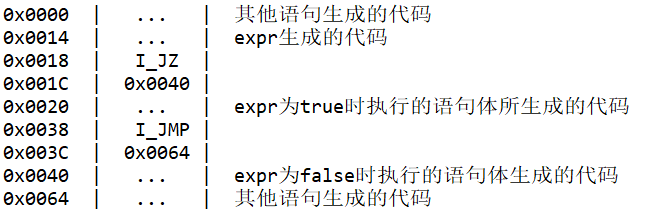


图4-6 if-else语句生成的代码

上述两种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进行回填。

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深度，并在离开语句体时恢复作用域深度。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：

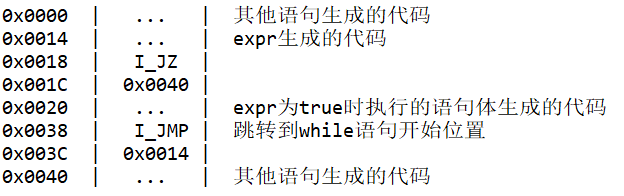


图4-7 while语句生成的代码

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：

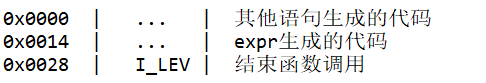


图4-8 return语句生成的代码

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：

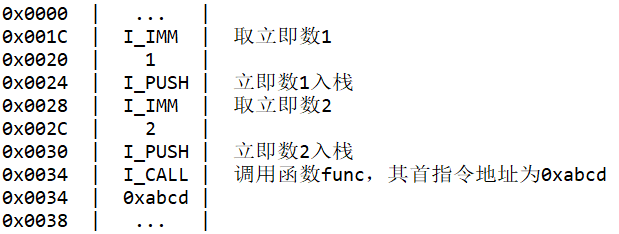


图4-9 函数调用func(1,2)生成的代码

取整型局部变量的值时，生成的代码结构如图3-12所示：

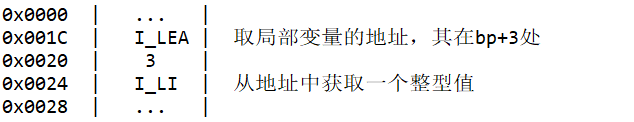


图4-10 取整型局部变量值所生成的代码

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：

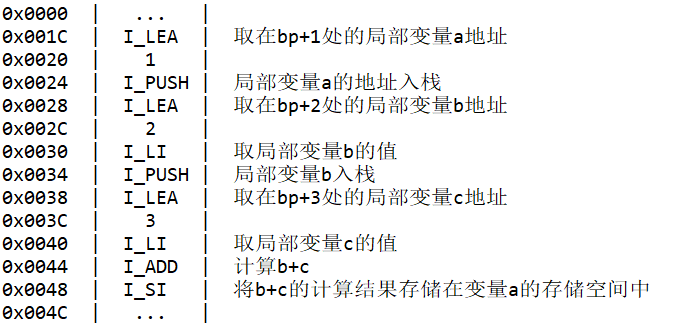


图4-11 表达式a = b + c生成的代码

4.2.6 编译错误与警告

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，因此不符合文法规范，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，并将其作为异常抛出。

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该错误，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可以检测到。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型值有数据溢出的风险，从而提示警告。

4.3 编译选项与目标代码分析

4.3.1 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 | 查看生成的所有目标代码 |
| -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 | 显示少量编译细节 |
| -dt2 | 显示适量编译细节 |
| -dt3 | 显示所有编译细节 |
| -df | 将输出内容定向到debug.txt |

4.3.2 目标代码分析

编译选项-gg可以查看每个全局定义生成的目标代码。下面通过这个选项来对编译器生成的目标代码进行分析。源程序的定义如图4-12所示：

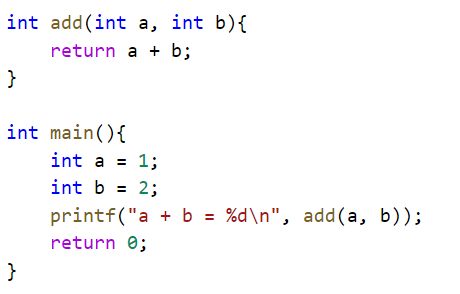


图4-12 源程序

通过-gg选项编译，可以看到每个全局定义生成的目标代码，如图4-13所示：

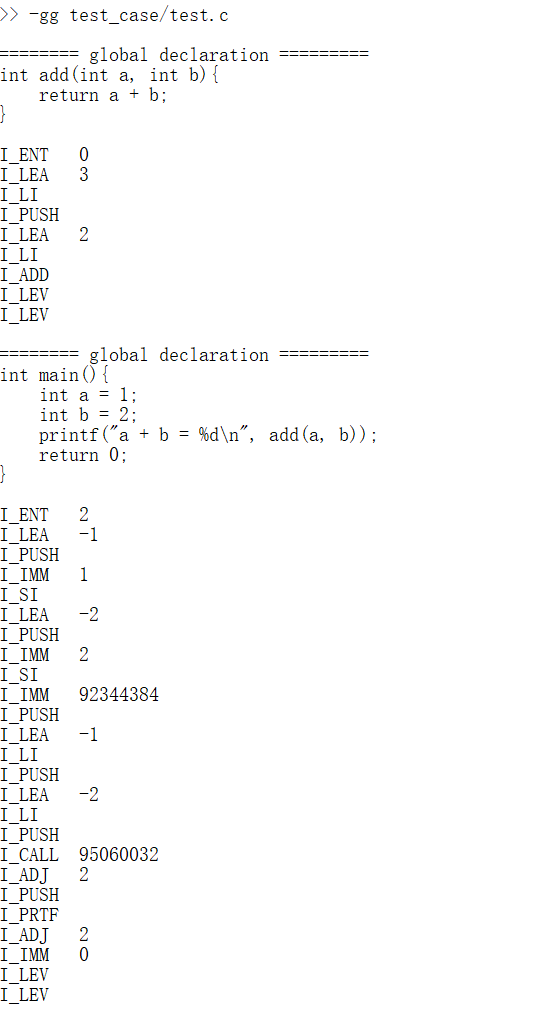


图4-13 每个全局定义生成的目标代码

全局函数add生成的目标代码中，首部的I\_ENT 2表示其有两个局部变量，即函数参数a和b。末尾的第一个I\_LEV指令由return语句生成，第二个I\_LEV指令在离开函数体时生成。I\_LEA 3指令获取变量a的地址，然后I\_LI指令将该地址视为一个int型变量地址并取值，再由I\_PUSH指令将变量a的值入栈。之后的I\_LEA 2和I\_LI指令获取变量b的值并放入ax寄存器中。I\_ADD则将栈顶元素出栈并与ax寄存器中的值相加，结果放回ax寄存器中，完成a+b的操作。函数返回时，返回值存放在ax寄存器中。

Main函数生成的目标代码中，首部的I\_ENT 2表示其有两个局部变量，即函数体首部定义的变量a和b。指令I\_LEA -1获取局部变量a的地址，I\_PUSH指令将a的地址入栈。I\_IMM 1指令获取立即数1，I\_SI指令则将该立即数1存储在栈顶元素所指的整型存储空间中，即将立即数1赋予变量a。后续的指令I\_LEA -2获取局部变量b的地址，I\_PUSH指令将b的地址入栈，I\_IMM 2指令获取立即数2，I\_SI指令将立即数2存储在变量b的存储空间中。指令I\_IMM 92344384中92344384是字符串“a + b = %d\n”的首地址，I\_PUSH指令将该地址入栈，完成printf函数的第一个参数传递。之后到I\_CALL 95060032之前是将add函数的两个参数入栈，然后I\_CALL 95060032指令调用add函数，实现add(a, b)的函数调用，其中95060032为add函数的首地址。因为add函数有两个参数，因此在退出add函数调用时，需要使用I\_ADJ 2指令将add函数的两个参数从栈上清除。add函数成功调用后，执行结果存放在ax寄存器中，再使用I\_PUSH指令将结果入栈。此时栈上有两个参数，第一个是字符串“a + b = %d\n”的首地址，第二个add(a, b)的执行结果。printf的两个参数都已入栈，再使用I\_PRTF指令调用printf函数。因为printf函数是内置函数，所以有其专有指令I\_PRTF。结束printf函数调用时也需要将使用指令I\_ADJ 2将两个参数出栈。最后的I\_IMM 0和第一个I\_LEV指令是由return 0语句生成，最后一条I\_LEV语句是在离开main函数时生成。

通过对源程序生成的目标代码进行分析，能够更加清晰地了解变量定义、变量使用、参数传递、函数调用等的实际执行过程。

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。先对各个功能单独进行测试，然后再进行整体测试。

测试的方式为：若可以获取测试对象的值，则将其值与预期的值进行比较，若相同则视为成功，否则视为失败；若不可以获取测试对象的值，则根据输出各种辅助信息来进行判断。如变量定义语句int a=1，则将a与值1进行比较看是否相等，若测试printf函数，则调用printf函数看其输出的结果是否与预计相同。

每个通过检测值是否相等的来判断测试是否成功的文件，如变量测试文件、函数参数测试文件等，中都有都有一部分测试框架代码，其结构如图4-14所示：

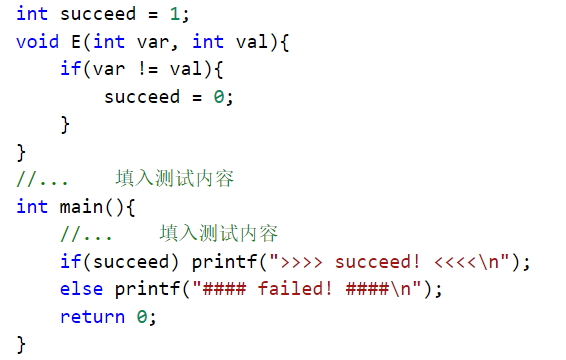


图4-14 测试框架代码

定义succeed为1，表示测试成功。函数E检测两个变量是否相等，不相等则将succeed置0。最后在main函数末尾检测succeed的值，来判断对该内容的测试是否成功。

4.4.1 变量测试

全局变量只允许使用数字、字符或字符串进行初始化，局部变量允许使用表达式进行初始化。

全局变量的定义如图4-15所示。然后再使用测试函数E对其进行测试，如图4-16所示。不启用Warning的测试结果如图4-17所示。因为全局变量定义中有各种类型，如char型、int\*型等，类型检测会产生大量Warnning，因此使用-nw编译选项警用Warning，只看最终测试结果。作为对比，不使用-nw编译选项的输出如图4-18所示。后续除非是测试类型检测机制，否则都将使用-nw选项避免输出结果混乱。

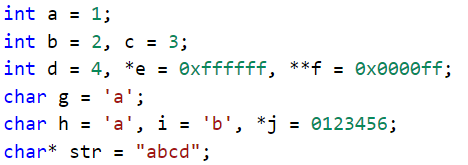


图4-15 全局变量定义

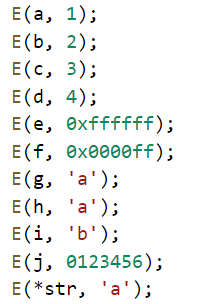


图4-16 全局变量测试

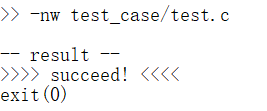


图4-17 禁用Warning的全局变量测试结果

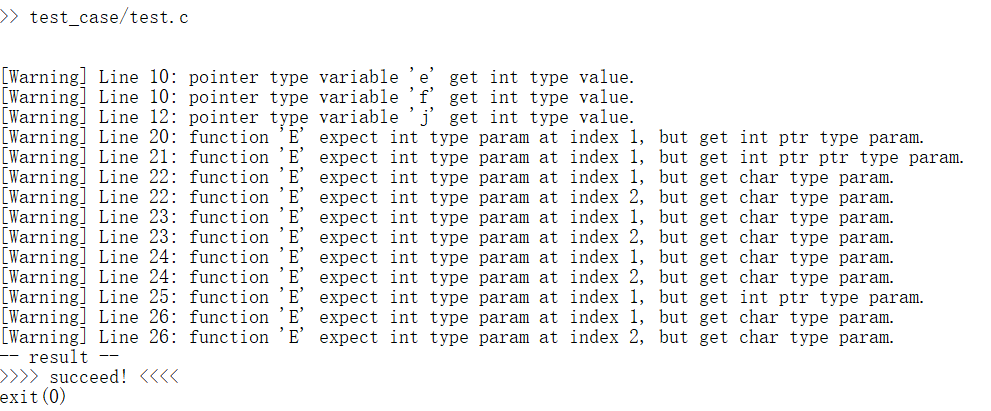


图4-18 启用Warning的全局变量测试结果

后续各种语句的测试结果都成功了，测试结果都同图4-17类似，因此后续测试就不再展示测试结果图，只展示测试方式。

4.4.2 enum测试

enum变量只允许使用数字进行初始化，并且默认为int型。对enum变量的测试主要是测试默认情况下后一个变量的值是否为前一个变量值加1。若变量有初始值，则变量的值为该初始值。

enum变量定义如图4-19所示。然后再使用测试函数E对各个enum变量进行检测，如图4-20所示。

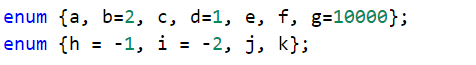


图4-19 enum变量定义

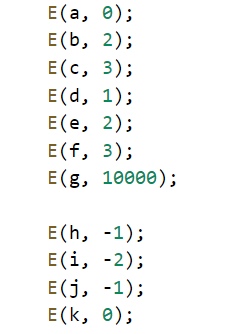


图4-20 enum变量测试

4.2.3 函数测试

函数测试则是测试函数在调用过程中，参数是否正确传递，函数体是否完成指定的功能。测试函数参数时的函数定义如图4-21所示，测试代码如图4-22所示，测试结果如图4-22所示。

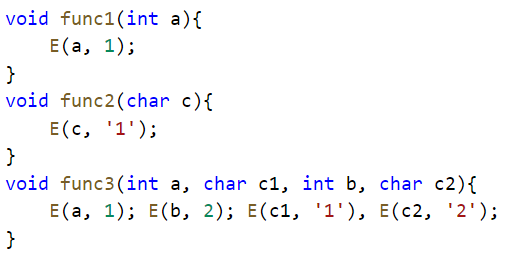


图4-21 函数定义

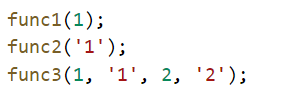


图4-22 函数参数测试

测试函数体时，需测试其是否完成了指定的功能，可以通过检测返回值，或者检测其修改过的变量等内容来进行。函数定义如图4-23所示，测试代码如图4-24所示。

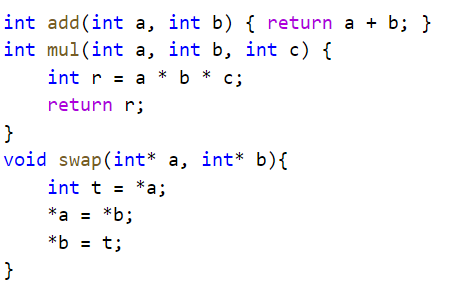


图4-23 函数定义

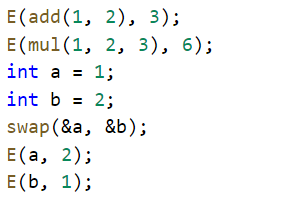


图4-24 函数功能测试

4.4.4 语句的测试

各种语句的测试则语句包括if语句、while语句、return语句、块语句、表达式语句。

if语句可通过在不同分支中修改某个变量为不同的值，来确定其执行了哪个分支。其测试代码如图4-25所示。通过设置不同的expr值，来让if执行不同的分支，从而将a修改为不同的值，最后通过检测a的值是否与期望的相同来检验if语句是否正确完成其功能。

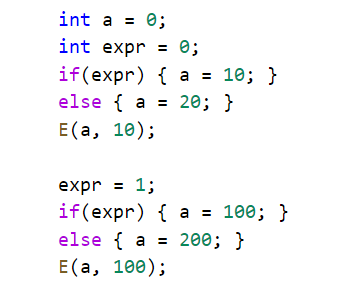


图4-25 if语句测试

while语句是循环语句，可以对其语句块循环执行多次，根据这个特点，使用在语句块中累加变量来测试其是否正确运行。测试代码如图4-26 所示。

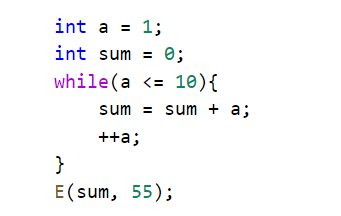


图4-26 while语句测试

return语句已在函数测试中得到检验。因此不再单独进行测试。

块语句将创建一个更深层的作用域，可以通过在内部定义与外部同名的变量，来检测作用域规则是否正确。块语句的测试代码如图4-27 所示。

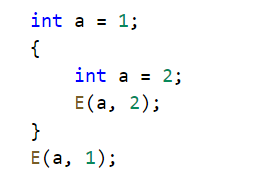


图4-27 块语句测试

表达式语句包括各种表达式，如四则运算、sizeof、类型转换、函数调用等，下面只展示四则运算以及sizeof的测试。四则运算表达式的测试如图4-28所示，sizeof表达式的测试如图4-29所示。

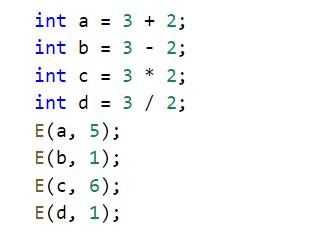


图4-28 四则运算表达式测试

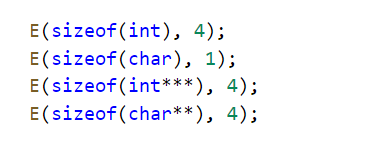


图4-29 sizeof表达式测试

4.4.5 整体测试

尽管各个部分都能通过测试，但是整体测试往往能检测出一些隐藏的问题。整体测试是通过写一段有实际用途的代码来进行测试，与实际情况更加符号。排序程序和斐波那契数计算程序可以用来检验程序的健壮性。排序程序涉及了循环语句，以及大量数据移动交换等情况，斐波那契数计算程序使用递归实现，涉及大量函数调用，这两种情况可以模拟大部分程序的运行情况。

排序程序采用插入排序算法，算法主体代码如图4-30所示，测试代码如图4-31所示。

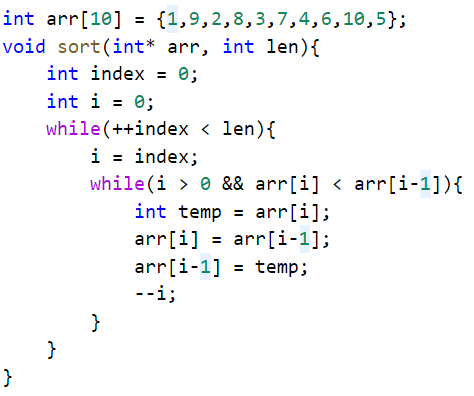


图4-30 插入排序算法

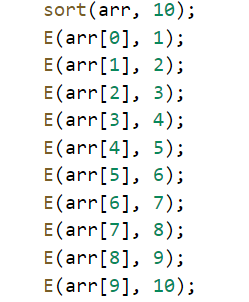


图4-31 插入排序算法测试

程序正确运行，说明编译器在各个方面都比较健壮。

斐波那契数计算程序将使用递归的方式计算第10个斐波那契数，其定义如图4-32所示，最终的测试结果如图4-33所示。

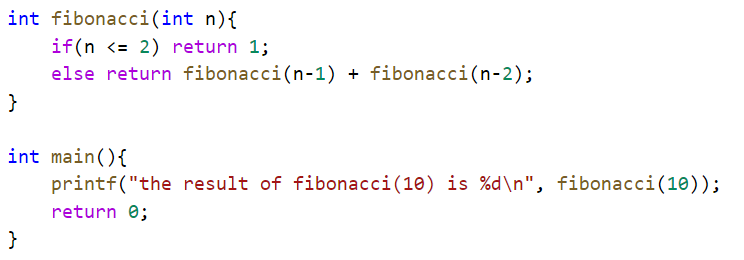


图4-32 fibonacci数计算程序

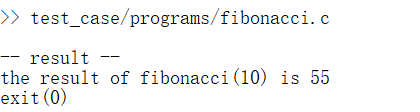


图4-33 fibonacci程序测试结果

fibonacci函数计算出了正确的结果55，说明本编译器对大量递归的函数调用也能正确处理。

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且更为简单的实现方式。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，避免了编译到汇编语言可能涉及的各种复杂而无关的细节。

编译器后端优化部分比较复杂且与我的学习目标不重叠，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，整体测试中的比较复杂的程序也能正确运行。

//TODO: 子集简单、五脏俱全

5.2 未来展望

本编译器采用的是单趟式编译方式，虽然实现了定义的高级语言的功能，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：

《C程序设计语言》: 学习C语言基本语法

《C++ Primer》: C++语言的使用

《The C++ Standard Library: A Tutorial and Reference》: C++标准库的使用

《Effective C++》: C++编码规范

《Effective Modern C++》: C++编码规范

《数据结构与算法--C语言描述》: 基本算法

《算法导论》: 基本算法

《Computer Systems: A Programmer's Perspective》: 指令集的设计参考

《汇编语言》：指令集的设计参考

《LLVM Cookbook》：指令集设计参考

《深度探索C++对象模型》: 编程语言中功能的实现方式

《Compiler: Principles, Techniques, and Tools》: 编译器的介绍与实现

《可变目标C编译器：设计与实现》: 编译器的实现

《Parsing Techniques》: 编译器前端

《自己动手构造编译系统：编译、汇编与链接》：编译器实现

《自己动手写编译器、链接器》：编译器实现