**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 查找和建立标识符

3.1.4 内建符号的处理

3.2 词法分析

3.2.1词法分析算法的设计

3.2.2 词法单元的识别

3.3 语法分析

3.3.1 语言文法的定义

3.3.2 语法分析算法的设计

3.4 语义分析与代码生成

3.4.1 语法制导的翻译技术

3.4.2 代码生成

**第4章 编译器实现与测试**

4.1 词法分析器的实现

4.1.1 标识符的识别

4.1.2 数字的识别

4.1.3 注释的识别

4.1.4 字符与字符串的识别

4.1.5 其他符号的识别

4.2 语法制导的解析程序实现

4.2.1 变量的解析

4.2.2 enum的解析

4.2.3 函数的解析

4.2.4 语句的解析

4.2.5 编译错误

4.2.6 编译警告

4.3编译器测试

4.3.1 变量测试

4.3.2 enum测试

4.3.3 函数测试

4.3.4 语句的测试

4.3.5 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，或硬编码方式。基于有限自动机的方式能够实现词法分析程序的自动构造，可以根据正则表达式动态地更改词法单元描述。硬编码的方式则根据给定词法单元描述直接在程序中硬编码识别，较基于有限自动机的方式更加简单，但是不可动态配置词法单元。

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。

自顶向下分析包括确定分析和不确定分析，不确定分析方法是带有回溯的分析方法，实际上是一种穷举的试探方法，因此效率低而代价高。确定分析方法中递归下降方法是常用的语法分析方法，其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，但是实现较为简单。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。

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>

<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

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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 查找和建立标识符

3.1.4 内建符号的处理

3.2 词法分析

3.2.1词法分析算法的设计

3.2.2 词法单元的识别

3.3 语法分析

3.3.1 语言文法的定义

3.3.2 语法分析算法的设计

3.4 语义分析与代码生成

3.4.1 语法制导的翻译技术

3.4.2 代码生成

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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》]。

3.1.1 符号的表示

在词法分析阶段，从源代码中截取出单词时，就需要将部分信息写入符号中。词法单元有标识符、数字、关键字等类型，对于标识符，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。符号类的各个属性定义见表3-1：

表3-1

|  |  |  |
| --- | --- | --- |
| 属性名 | 描述 | 数据类型 |
| type | 类型，如ID | int |
| name | 名字 | string |
| hash | hash值 | int |
| klass | 种类，如FUNC | int |
| dataType | 数据类型 | int |
| value | 值 | value |
| argsDataType | 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