

天津大学



任务三 GO 语言编译器设计文档

学院 智能与计算学部

专业 计算机科学与技术

组长 胡轶然

成员 许振东 沈昊 李研 胡书豪

学号 3019244355

3019244356

3019234165

3019205150

3019205412

目录

1 任务概述	3
2 生成语法分析树	3
2.1 GO 语言的语法描述	3
2.2 语法分析树生成原理	4
2.3 构建生成语法分析树	6
3 中间代码表示形式的设计	6
3.1 语义分析与中间代码生成的基础知识	6
3.2 中间代码的实现——四元式	7
3.3 中间代码的实现内容	8
3.4 将语法分析树转化为中间代码的过程实例	11
4 将中间代码转化为汇编指令	13
4.1 基本原理	13
4.2 实现代码结构	14
4.3 实例分析	15
5 错误分析与处理的实现	16
5.1 Yacc 语法说明文件的错误处理规则	16
5.2 三地址码生成中的错误处理	17
6 生成可执行文件	19
6.1 机器语言	19
6.1.1 机器语言的格式	19
6.1.2 汇编语言转化为机器语言	20
6.2 链接	20
7 可视化编译 IDE 的实现	21
7.1 实现方法	21
7.2 效果展示	23

1 任务概述

本任务为设计 GO 语言的编译器，具体内容如下所示：

- 1、查找资料，给出 GO 语言的语法描述
- 2、设计中间代码表示形式。
- 3、编译器的基本功能包括：
 - 根据输入程序得到对应语法分析树；
 - 设计中间代码表示形式，并根据输入程序生成三地址码；
 - 按照 MIPS 规范，根据输入程序生成汇编指令；
- 4、编译器的附加功能
 - 对输入程序错误的部分实现分析和汇报；
 - 将汇编指令生成二进制可执行文件；
 - 部分的代码优化功能；

任务分工：

胡轶然：整理会议纪要，协助完成三地址码的生成和过程中的错误分析处理；协助完成汇编指令的生成与代码优化；整理制作 PPT 展示以及讲稿文档；审核设计文档；安装运行 IDE 演示界面；24%

许振东：制作 IDE 界面进行展示，实现语法分析树的生成，协助实现代码优化；提供工作总结和协助 PPT 的制作；19%

沈昊：合作实现汇编指令的生成与代码优化，帮助实现程序二进制文件的生成，制作项目整体的脚本文件，提供工作总结和协助 PPT 的制作；19%

李研：协助完成三地址码的生成过程中的错误处理，提供工作总结和代码的解释文档；帮助制作 IDE 界面进行展示，审核会议纪要；19%

胡书豪：帮助设计三地址码的表达形式和三地址码的生成，提供工作总结和协助 PPT 的制作；撰写设计文档；19%

2 生成语法分析树

2.1 GO 语言的语法描述

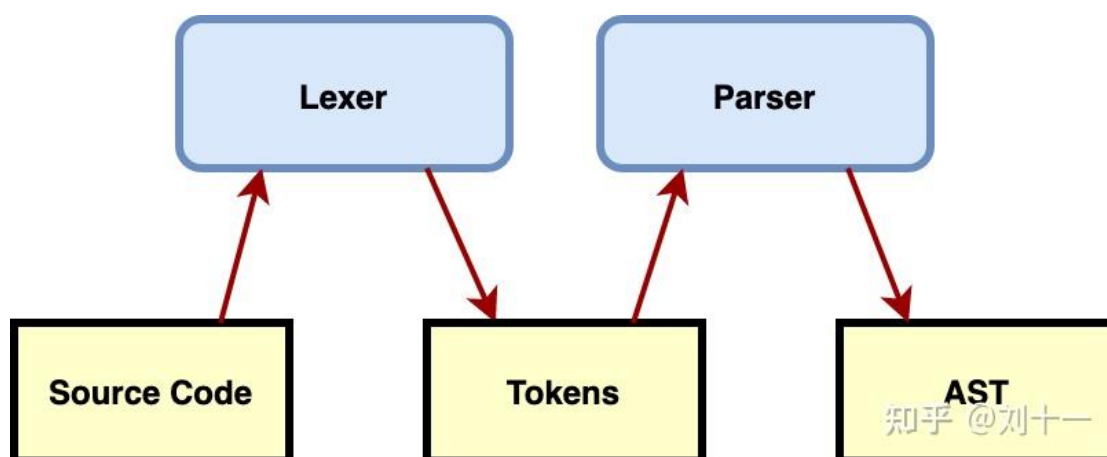
通过查找 GO 语言官网上的相关资料（<https://golang.org/ref/spec>），得到 Golang 语言的语法描述的 g4 文件如下所示：

```
活动 文本编辑器 6月30日 03:24
Golang.g4
~/golang

1 /*
2  [The "BSD licence"]
3  Copyright (c) 2016 Sasa Coh
4  All rights reserved.
5  Redistribution and use in source and binary forms, with or without
6  modification, are permitted provided that the following conditions
7  are met:
8  1. Redistributions of source code must retain the above copyright
9  notice, this list of conditions and the following disclaimer.
10 2. Redistributions in binary form must reproduce the above copyright
11 notice, this list of conditions and the following disclaimer in the
12 documentation and/or other materials provided with the distribution.
13 3. The name of the author may not be used to endorse or promote products
14 derived from this software without specific prior written permission.
15 THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR
16 IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
17 OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
18 IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
19 INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
20 NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
21 DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
22 THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
23 (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
24 THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
25 */
26 /**
27  * A Go grammar for ANTLR 4 derived from the Go Language Specification
28  * https://golang.org/ref/spec
29  *
30  */
31 grammar Golang;
32
33 //SourceFile = PackageClause ";" { ImportDecl ";" } { TopLevelDecl ";" } .
34 sourceFile
35 : packageClause ';' ? ( importDecl ';' ? ) * ( topLevelDecl ';' ? ) *
36 ;
37
38 //PackageClause = "package" PackageName .
39 //PackageName = identifier .
40 packageClause
41 : 'package' IDENTIFIER
42 ;
43
44 importDecl
45 : 'import' ( importSpec | '(' ( importSpec ';' ? ) * ')' )
46 ;
47
48 importSpec
49 : ( '.' | IDENTIFIER ) ? importPath
50 ;
51
52 importPath
53 : STRING_LITERAL
54 ;
55
```

(因 g4文件较大, 约1200行, 这里只展示了部分)

2.2 语法分析树生成原理



首先, 计算机先用一个词法分析器(Lexer)对文本(Source Code)进行词法分析, 生成 Token。一般接下来是将它传给一个解析器, 然后检索生成语法分析树。上图中有两个关键的工具: lexer 和 parser 以及中间结果 Token。

Lexer-又名词法分析器: 词法分析器用来将字符序列转换为单词(Token)。词法分析主要是完成:

1. 对源程序的代码进行从左到右的逐行扫描, 识别出各个单词, 从而确定单词的类型;
2. 将识别出的单词转换为统一的机内表示——词法单元 (Token) 形式。

token 是一种类型 Map 的 key/value 形式, 它由<种别码, 属性值>组成, 种别码就是类型、属性值就是值。例如下述代码中:

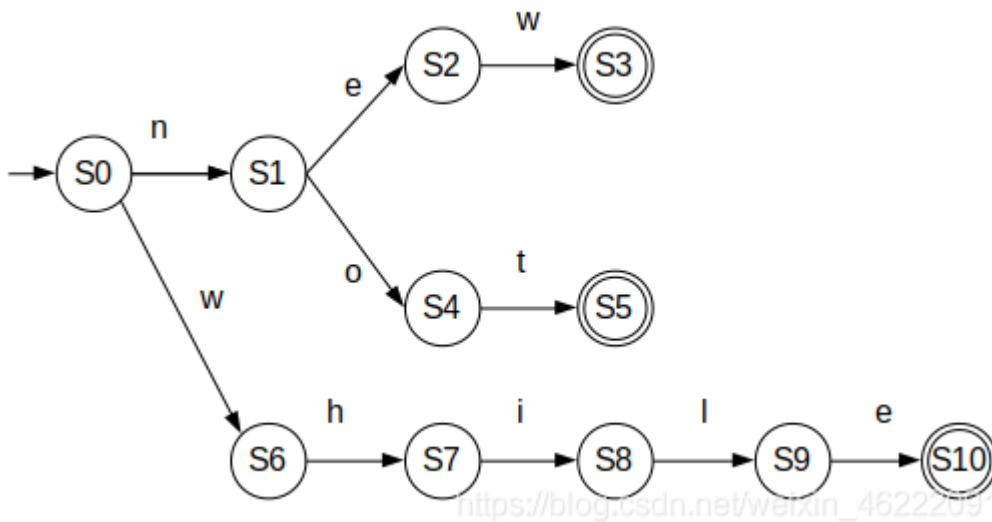
1. go package main
2. **const** s = "foo"

转换为 token 之后:

1. PACKAGE(package)
2. IDENT(main)
3. CONST(**const**)
4. IDENT(s)
5. ASSIGN(=)
6. STRING("foo")
7. EOF()

根据文法生成 文法分析表，再由文法分析表和输入串（token）在语法分析器中生成语法分析树。

1、词法分析器：形式语言自动机



2、语法分析器：解释器使用 LALR 来解析 go 语言文法文件生成 ACTION-GOTO 表 然后使用上述生成的 tokens 流移进规约 生成 AST（穿线表）

我们设计了 Go 语言的解析器使用了 LALR(1) 的文法来解析词法分析过程中输出的 Token 序列

语法分析器最终会使用不同的结构体来构建抽象语法树中的节点，根节点和其它节点如下：

```
1. type File struct {
2.     PkgName *Name
3.     DeclList []Decl
4.     Lines   uint
5.     node
6. }
7.
8.
9. type (
10.     Decl interface {
11.         Node
12.         aDecl()
13.     }
14.
15.     FuncDecl struct {
16.         Attr map[string]bool
17.         Recv  *Field
18.         Name *Name
```

```

19.     Type *FuncType
20.     Body *BlockStmt
21.     Pragma Pragma
22.     decl
23.   }
24. }
25.

```

2.3 构建生成语法分析树

终端执行：

```

1.  antlr4 Golang.g4
2.  javac Golang*.java
3.  grun Golang sourceFile -gui example.go

```

3 中间代码表示形式的设计

在完成编译器生成其对应的语法分析树功能后，小组进行进一步的学习，了解三地址指令代码生成的相关知识，设计中间代码表示形式。

3.1 语义分析与中间代码生成的基础知识

项目计划使用三地址代码的中间表示形式。

- 三地址代码的形式

三地址码的得名原因是每条语句通常包含三个地址，两个是操作数地址，一个是结果地址。三地址代码是下列一般形式的语句序列：

$$x := y \text{ op } z$$

其中， x 、 y 和 z 是名字，常量或编译器生成的临时变量， op 代表任何操作符（定点运算符、浮点运算符、逻辑运算符等）。这里不允许组合的算术表达式，因为语句右边只有一个操作符。也就是说， $x+y*z$ 这样的表达式是不被允许的，在三地址代码中要拆分成这样的形式

$$\begin{aligned} T_1 &:= y * z \\ T_2 &:= x + T_1 \end{aligned}$$

其中 T_1 ， T_2 为编译时产生的临时变量。

这种复杂算术表达式和嵌套控制流语句的拆解使得三地址码适用于目标代码生成及优化。三地址码的得名原因是每条语句通常包含三个地址，两个是操作数地址，一个是结果地址。

- 三地址语句的类型

形如 $x := y \text{ op } z$ 的赋值语句，其中 op 为二元算术算符或逻辑算符

形如 $x := op \ y$ 的赋值语句，其中 op 为一元算符。

形如 $x := y$ 的复制语句，将 y 的值赋给 x

形如 `goto L` 的无条件跳转语句，即下一条将被执行的语句是带有标号 L 的三地址语句

- 三地址语句的实现

三地址语句是中间代码的一种抽象形式，这些语句可以以带有操作符和操作数域的记录来实现。四元式、三元式及简介三元式是三种这样的表示。

- 四元式

一个四元式是带有四个域的记录结构，这四个域分别称为 op , $arg1$, $arg2$ 及 $result$ 。域 op 包含

一个代表运算符的内部码，三地址语句 $x:=y \text{ op } z$ 通过将 y 放入 arg1 ， z 放入 arg2 ，并且将 x 放入 result ， $:=$ 为算符。

像 $x:=y$ 或 $x:=-y$ 这样的一元操作符语句不使用 arg2 ，像 param 这样的运算符仅使用 arg1 域。条件和无条件语句将目标标号存入 result 域，临时变量也要填入符号表中。

产生式	语义规则
$S \rightarrow \text{id} := E$	$S.\text{code} := E.\text{code} \parallel \text{gen}(\text{id.place} \text{ ':=' } E.\text{place})$
$E \rightarrow E_1 + E_2$	$E.\text{place} := \text{newtemp};$
	$E.\text{code} := E_1.\text{code} \parallel E_2.\text{code} \parallel \text{gen}(E.\text{place} \text{ ':=' } E_1.\text{place} \text{ '+' } E_2.\text{place})$
$E \rightarrow E_1 * E_2$	$E.\text{place} := \text{newtemp};$
	$E.\text{code} := E_1.\text{code} \parallel E_2.\text{code} \parallel \text{gen}(E.\text{place} \text{ ':=' } E_1.\text{place} \text{ '*' } E_2.\text{place})$
$E \rightarrow - E_1$	$E.\text{place} := \text{newtemp};$
	$E.\text{code} := E_1.\text{code} \parallel \text{gen}(E.\text{place} \text{ ':=' 'uminus' } E_1.\text{place})$
$E \rightarrow (E_1)$	$E.\text{place} := E_1.\text{place}$ $E.\text{code} := E_1.\text{code}$
$E \rightarrow \text{id}$	$E.\text{place} := \text{id.place}$ $E.\text{code} := \text{''}$

$a := b * -c + b * -c$

	op	arg1	arg2	result
(0)	uminus	c		T_1
(1)	*	b	T_1	T_2
(2)	uminus	c		T_3
(3)	*	b	T_3	T_4
(4)	+	T_2	T_4	T_5
(5)	assign	T_5		a

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	0
(2)	uminus	c	
(3)	*	b	2
(4)	+	1	3
(5)	assign	a	4

3.2 中间代码的实现——四元式

中间代码表示形式的设计：仿照四元式进行

四元式是一种“三地址语句”的等价表示。这里我们的基本形式为：

(op , result , arg1 , arg2)

即<操作符>，<结果>，<操作数 1>，<操作数 2>

以上四元式运算规则即，将操作数 1 与操作数 2 进行操作符代表的运算，将运算结果存放在 result 位置。如对于 四元式 $(*, T_1, B, C)$ ，表示的运算即： $T_1:=B * C$

特别的，对于单目运算符，常常将运算对象定义为 arg1 ， arg2 位置空出。除此之外，对于有些运算符不生成结果的（如 print ，将操作数打印输出），则将 result 位置空出。像 $x:=y$ 或 $x:=-y$ 这样的一元操作符语句不使用 arg2 ，像 param 这样的运算符仅使用 arg1 域。条件和无条件语句将目标标号存入 result 域，临时变量也要填入符号表中。

中间代码表示形式的具体设计部分列出如下：

- (1) (op, c, a, b) op 为运算符如“+、-、*、/、%、^、==、<、>”等，单目运算 arg2 位置空出。例如：
(+,t1,a,b) 即 t1 = a + b
(-,t2,c) 即 t2 = -c
- (2) (call, function_name, return_value) 调用函数 function，如果有返回值，会储存在 return_value 中。例如：
(call,main,0) 即调用 main 函数，返回值为 0
- (3) (exit) 退出函数体或调用函数等（注意没有操作数和结果，位置空出）
- (4) (return, value) 函数返回值 value。例如：
(return,t1) 即 return t1;
- (5) (print, value) 将操作数 value 打印输出。例如：
(print,t1) 即将操作数 t1 打印输出
- (6) (label, label_name) 将当前位置命名为 label_name，配合 goto 和 ifgoto 使用。
- (7) (goto, label_name) 无条件跳转至 label_name 所在的位置。例如：
17 行(goto label_1)
.....
20 行(label, label_1) 则从 17 行直接跳转到 20 行执行
- (8) (ifgoto, op, args1, args2, label_name) 有条件地跳转至 label_name 所在的位置，args1 op args2 为真才进行跳转，其中 op 为“=、>、<、<=、>=等”。
例如： 17 行(ifgoto, =, t3, 1, label_2)
.....
20 行(label, label_2) 则如果 t3 = 1 则从 17 行直接跳转到 20 行执行

3.3 中间代码的实现内容

在 parserIR.y 中，定义终结符：

在其中，定义终结符：

```
1. %token <sval> PACKAGE IMPORT FUNC BREAK CASE CONST CONTINUE D
   EFAULT
2. %token <sval> ELSE FOR GO IF RANGE RETURN STRUCT SWITCH TYPE VA
   R VAR_TYPE
3. %token <sval> BOOL_CONST NIL_VAL IDENTIFIER BYTE STRING ELLIPSIS
4. %token <sval> SHL SHR INC DEC
5. %token <sval> INTEGER
6. %token <sval> FLOAT
7. %left <sval> ADD SUB MUL QUO REM
8. %token <sval> ASSIGN AND NOT DEFINE AND_NOT
9. %left <sval> OR XOR ARROW //Identifier
10. %right <sval> ADD_ASSIGN SUB_ASSIGN MUL_ASSIGN QUO_ASSIGN REM
    _ASSIGN
11. %right <sval> AND_ASSIGN OR_ASSIGN XOR_ASSIGN
12. %right <sval> SHL_ASSIGN SHR_ASSIGN AND_NOT_ASSIGN COLON
```


13. %left <sval> LAND LOR EQL NEQ LEQ GEQ SEMICOLON
14. %left <sval> GTR LSS LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK COMMA PERIOD

非终结符:

1. %type <nt> StartFile Expression Expression1 Expression2 Expression3 Expression4 Expression5
2. %type <nt> Block StatementList Statement SimpleStmt int_lit
3. %type <nt> EmptyStmt /*ExpressionStmt*/ IncDecStmt OPENB CLOSEB
4. %type <nt> Assignment Declaration ConstDecl VarSpec string_lit
5. %type <nt> Signature Result Parameters ParameterList ParameterDecl
6. %type <nt> ConstSpec MethodDecl Receiver TopLevelDecl TopLevelDeclList LabeledStmt
7. %type <nt> ReturnStmt BreakStmt ContinueStmt StructType
8. %type <nt> FunctionDecl FunctionName TypeList
9. %type <nt> Function FunctionBody FunctionCall ForStmt ForClause /*RangeClause*/
10. InitStmt ArgumentList Arguments
11. %type <nt> PostStmt Condition UnaryExpr PrimaryExpr
12. %type <nt> Selector Index /*Slice*/TypeDecl TypeSpecList TypeSpec VarDecl
13. %type <nt> TypeAssertion ExpressionList ArrayType assign_op rel_op add_op mul_op
14. unary_op Tag byte_lit float_lit Label
15. /*%type <nt> ExprCaseClauseList ExprCaseClause
16. %type <nt> Operand Literal BasicLit OperandName ImportSpec IfStmt
17. %type <nt> ImportPath
18. %type <nt> PackageClause PackageName PackageName2 ImportDecl ImportDeclList
19. ImportSpecList
20. %type <nt> FieldDeclList FieldDecl Type TypeLit ArrayLength
21. %type <nt> TypeName
22. %type <nt> /*QualifiedIdent*/ PointerType IdentifierList IdentifierLIST BaseType

剩下大多数代码的核心思想是，在遇到传过来的 character 是非终结符时，编写代码将其进行拆解为终结符。在 lexer.l 中，三地址码的生成一共实现以下字符：

1. "=" {yylval.sval = strdup(yytext); return ASSIGN;}
2. "+" {yylval.sval = strdup(yytext); return ADD;}
3. "-" {yylval.sval = strdup(yytext); return SUB;}
4. "*" {yylval.sval = strdup(yytext); return MUL;}
5. "/" {yylval.sval = strdup(yytext); return QUO;}
6. "%" {yylval.sval = strdup(yytext); return REM;}
7. "&" {yylval.sval = strdup(yytext); return AND;}
8. "|" {yylval.sval = strdup(yytext); return OR;}
9. "^" {yylval.sval = strdup(yytext); return XOR;}
10. "<<" {yylval.sval = strdup(yytext); return SHL;}
11. ">>" {yylval.sval = strdup(yytext); return SHR;}
12. "&^" {yylval.sval = strdup(yytext); return AND_NOT;}
13. "+=" {yylval.sval = strdup(yytext); return ADD_ASSIGN;}
14. "-=" {yylval.sval = strdup(yytext); return SUB_ASSIGN;}
15. "*=" {yylval.sval = strdup(yytext); return MUL_ASSIGN;}
16. "/=" {yylval.sval = strdup(yytext); return QUO_ASSIGN;}
17. "%=" {yylval.sval = strdup(yytext); return REM_ASSIGN;}

```
18. "&=" {yylval.sval = strdup(yytext); return AND_ASSIGN;}
19. "|=" {yylval.sval = strdup(yytext); return OR_ASSIGN;}
20. "^=" {yylval.sval = strdup(yytext); return XOR_ASSIGN;}
21. "<<=" {yylval.sval = strdup(yytext); return SHL_ASSIGN;}
22. ">>=" {yylval.sval = strdup(yytext); return SHR_ASSIGN;}
23. "&^=" {yylval.sval = strdup(yytext); return AND_NOT_ASSIGN;}
24. "&&" {yylval.sval = strdup(yytext); return LAND;}
25. "||" {yylval.sval = strdup(yytext); return LOR;}
26. "->" {yylval.sval = strdup(yytext); return ARROW;}
27. "++" {yylval.sval = strdup(yytext); return INC;}
28. "--" {yylval.sval = strdup(yytext); return DEC;}
29. "==" {yylval.sval = strdup(yytext); return EQL;}
30. ">" {yylval.sval = strdup(yytext); return GTR;}
31. "<" {yylval.sval = strdup(yytext); return LSS;}
32. "!" {yylval.sval = strdup(yytext); return NOT;}
33. "!=" {yylval.sval = strdup(yytext); return NEQ;}
34. "<=" {yylval.sval = strdup(yytext); return LEQ;}
35. ">=" {yylval.sval = strdup(yytext); return GEQ;}
36. "[:=" {yylval.sval = strdup(yytext); return DEFINE;}
37. "..." {yylval.sval = strdup(yytext); return ELLIPSIS;}
38. "(" {yylval.sval = strdup(yytext); return LPAREN;}
39. ")" {yylval.sval = strdup(yytext); return RPAREN;}
40. "{" {yylval.sval = strdup(yytext); return LBRACE;}
41. "}" {yylval.sval = strdup(yytext); return RBRACE;}
42. "[" {yylval.sval = strdup(yytext); return LBRACK;}
43. "]" {yylval.sval = strdup(yytext); return RBRACK;}
44. "," {yylval.sval = strdup(yytext); return COMMA;}
45. ";" {yylval.sval = strdup(yytext); return SEMICOLON;}
46. ":" {yylval.sval = strdup(yytext); return COLON;}
47. "." {yylval.sval = strdup(yytext); return PERIOD;}
48. "package" {yylval.sval = strdup(yytext); return PACKAGE;}
49. "import" {yylval.sval = strdup(yytext); return IMPORT;}
50. "func" {yylval.sval = strdup(yytext); return FUNC;}
51. "break" {yylval.sval = strdup(yytext); return BREAK;}
52. "case" {yylval.sval = strdup(yytext); return CASE;}
53. "const" {yylval.sval = strdup(yytext); return CONST;}
54. "continue" {yylval.sval = strdup(yytext); return CONTINUE;}
55. "default" {yylval.sval = strdup(yytext); return DEFAULT;}
56. "else" {yylval.sval = strdup(yytext); return ELSE;}
57. "for" {yylval.sval = strdup(yytext); return FOR;}
58. "go" {yylval.sval = strdup(yytext); return GO;}
59. "if" {yylval.sval = strdup(yytext); return IF;}
60. "range" {yylval.sval = strdup(yytext); return RANGE;}
61. "return" {yylval.sval = strdup(yytext); return RETURN;}
62. "struct" {yylval.sval = strdup(yytext); return STRUCT;}
63. "switch" {yylval.sval = strdup(yytext); return SWITCH;}
64. "type" {yylval.sval = strdup(yytext); return TYPE;}
65. "var" {yylval.sval = strdup(yytext); return VAR;}
66. {VAR_TYPE} {yylval.sval = strdup(yytext); return VAR_TYPE;}
67. {BOOL_CONST} {yylval.sval = strdup(yytext); return BOOL_CONST;}
68. {NIL_VAL} {yylval.sval = strdup(yytext); return NIL_VAL;}
```

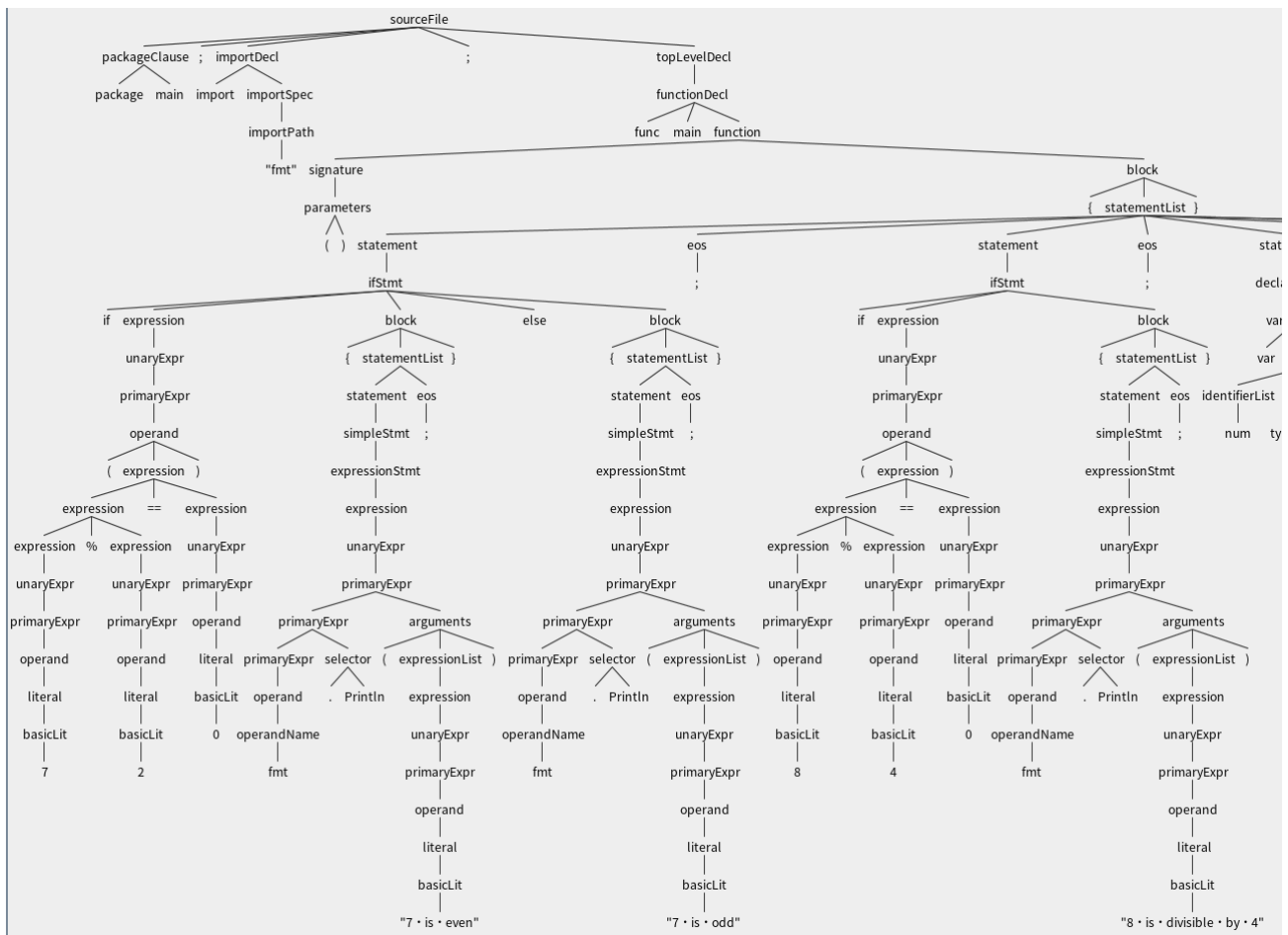
```
69. {LETTER}({LETTER}|{DIGIT})* {yyval.sval = strdup(yytext); return IDENTIFI  
    ER;}  
70. {DIGIT}+ {yyval.sval = strdup(yytext); return INTEGER;}  
71. {DIGIT}+ "."{DIGIT}+ {yyval.sval = strdup(yytext); return FLOAT;}  
72. \"{UNICODE_CHAR}\" {yyval.sval = strdup(yytext); return BYTE;}  
73. \"([^\n\"]|(\\\.))*\" {yyval.sval = strdup(yytext); return STRING;}
```

3.4 将语法分析树转化为中间代码的过程实例

下面将介绍如何将前面生成的语法分析树转化为上面设计的中间代码四元式
对于样例 Go 语言代码 test1.go，如下所示：

```
1. package main;  
2. import "fmt";  
3. func main() {  
4.  
5.     if (7%2 == 0){  
6.         fmt.Println("7 is even");  
7.     }else {  
8.         fmt.Println("7 is odd");  
9.     };  
10.  
11.    if (8%4 == 0) {  
12.        fmt.Println("8 is divisible by 4");  
13.    } ;  
14.  
15.    var num int = 45;  
16.    if (num < 0) {  
17.        fmt.Println(num, "is negative");  
18.    } else if (num < 10) {  
19.        fmt.Println(num, "has 1 digit");  
20.    } else {  
21.        fmt.Println(num, "has multiple digits");  
22.    };  
23.  
24. }
```

通过 2 中介绍的方法，生成的语法分析树（部分）如下所示：



将语法分析树转化为四元式的方法如下：

从根节点出发深度遍历整棵语法树，维护一个栈，每当遍历到一个叶子结点时，将结点入栈，然后对比此时栈中代表的语句是否与前面所述的某条中间代码形式规则相匹配，若匹配将栈中这些对应的结点弹出，并输出其对应的中间代码。

具体按照 test1.go 的例子来看，深度遍历语法分析树到“function”“main” 此时栈中语句匹配前面 call funtion 规则和 exit 规则，两个结点退栈，输出中间代码1 2 3行，继续深度遍历至 $7 \% 2 == 0$ 几个结点时，对应中间代码的 (op,c,a,b)规则，输出中间代码第4行，这些元素退栈，……重复这样类似的过程，直到遍历完整棵语法分析树。最后生成的四元式如下所示：

```

1,call,main,0
2,exit
3,function,main
4,%,t0,7,2
5,==,t1,t0,0
6,ifgoto,=,t1,1,label0
7,goto,label1
8,label,label0
9,string,t_str0,"7 is even"
10,print,t_str0
11,goto,label2
12,label,label1
13,string,t_str1,"7 is odd"
14,print,t_str1
15,goto,label2
16,label,label2
17,%,t2,8,4
18,==,t3,t2,0
19,ifgoto,=,t3,1,label3
20,goto,label4
21,label,label3
22,string,t_str2,"8 is divisible by 4"
23,print,t_str2
24,label,label4
25,=,num,45
26,<,t4,num,0
27,ifgoto,=,t4,1,label5

```

4 将中间代码转化为汇编指令

4.1 基本原理

三地址码转汇编码关键步骤概括：

- （1）指令结果生成
 - 规定用于不同类型指令和数据段的汇编码生成方法
 - 检查三地址码类型并进行相应转换
- （2）符号表生成
 - 划分基本块，记录基本块中每条语句的活动及下一步使用信息后，逐块生成汇编代码
 - 为每个基本块中的变量及对应值建立符号表
- （3）寄存器描述符设置
 - 启发式地为每条指令查找合适寄存器，实时更新寄存器描述符

包含从 `ir` 代码生成指令结果、生成符号表、寄存器描述符和处理数组表示法的代码。

我们的主汇编码生成程序，包含：

- 一个包含用于不同类型指令和数据段的 `x8` 代码生成方法的 `codegenerator` 类，它能够检查指令类型并进行相应的转换。
- `Next_use` 函数，它通过向后传递来确定基本块中每条语句的活动和下一步使用信息。然后，它通过向前传递为每个基本块生成汇编代码。最后，它将所有寄存器内容保存到内存中，并重置符号表的活动和下一次使用信息。

主要功能是读取输入的 `ir` 文件并调用代码生成器的方法。`Get_Reg` 包含为使用 `next_use` 启发式生成的下一个 `x86` 指令查找合适寄存器的代码。除此之外，它还包括用于寄存器描述符更新的代码，用于跳转调用的上下文保存等。

4.2 实现代码结构

包含四个文件：

syntab.h：定义空寄存器，符号表，寄存器、地址以及代码段的 `map`。

tac.h：`main` 函数的初始化，另含判断操作数是变量还是整数的 `isInteger` 函数。

tac.cpp：符号表生成以及寄存器描述符的设置。

根据三地址码运算符 (`op`) 确定操作类型，判断操作数个数并赋值（记为 `in1`、`in2` 和 `out`）。寄存器生成部分，包括寄存器的初始化、分配以及清空（`$a1-$a3`，`$t1-$t9`，`$s1-$s7`）。符号表生成部分，包括基本块划分（每块起始与终止位置的确定）、三地址与汇编码逐行 `map`、变量赋值与 `map`、内存地址计算与 `map`、语句下一步使用情况的记录。

translate.cpp：指令结果生成。

初始化 `blockcode`（基本块），调用 `tac.cpp` 的方法向 `blockcode` 里添加汇编语句。分为赋值、运算以及跳转三个部分（均由大量 `if` 实现）。

赋值：支持寄存器、存储器赋值以及立即数赋值，使用 `isInteger` 函数判断赋值方式。对于寄存器（`reg` 类型）、存储器赋值（`mem` 类型），调用符号表 `syntab` 得到所需值。

运算：支持 `+-*/%&|^`（加减乘除整除与或非）。汇编码直接翻译即可，例：三地址码 `op1 out in1 in2` 直接转为汇编码 `op2 reg1 reg2 reg3`。

跳转：支持无条件跳转 `goto`：直接翻译为 `j target`；

有条件跳转 `ifgoto`：根据语句上下文得到跳转条件，并翻译为助记符，包括 `ble`、`bge`、`blt`、`bgt`、`beq`、`bne`；

标签设置 `label`：初始设置 `label0`；

函数调用 `call` 以及返回 `return`（立即数、`reg`、`mem`）：保存现场（记录当前地址）并跳转至符号表对应的地址处。

此外，包含 `getreturn` 以及 `print`。

目前可实现的部分三地址转换为汇编指令如下：

1. `decl_array, arr[N]`

定义数组

`arr` -> 数组名

`N` -> 数组大小

2. `=, x, <symbol>`

赋值运算

`x = <symbol>` `symbol` 可以是数字、变量或数组元素

注：只允许以下两种操作：

`a[i] = t`

`t = a[i]`

3. `call, func_name, optional_var`

调用函数 `func_name`，如果有返回值，会储存在 `optional_var` 中

4. `ret, optional_return_var`

函数返回, optional_return_var 为返回的可选变量

5. op,a,b,c op in [&|^+,-,%,/,*,<<,>>]

算数及按位逻辑操作, 允许的操作符如上

6. ifgoto, leq,a,b,<line no. or label>

跳转至某一行或某个标签

支持无条件跳转和有条件跳转

7. print,n

打印整数 n

8. input,n

从 stdin 中扫描整数 n

9. op,a,b op in [~,!,++,--]

对 b 进行一元操作并把结果储存在 a 中

10. label, label_name

设置标签, 功能不变

4.3 实例分析

3AC

call, main, 0
exit
function main

%, t0, 7, 2
==, t1, t0, 0
ifgoto, =, t0, 0

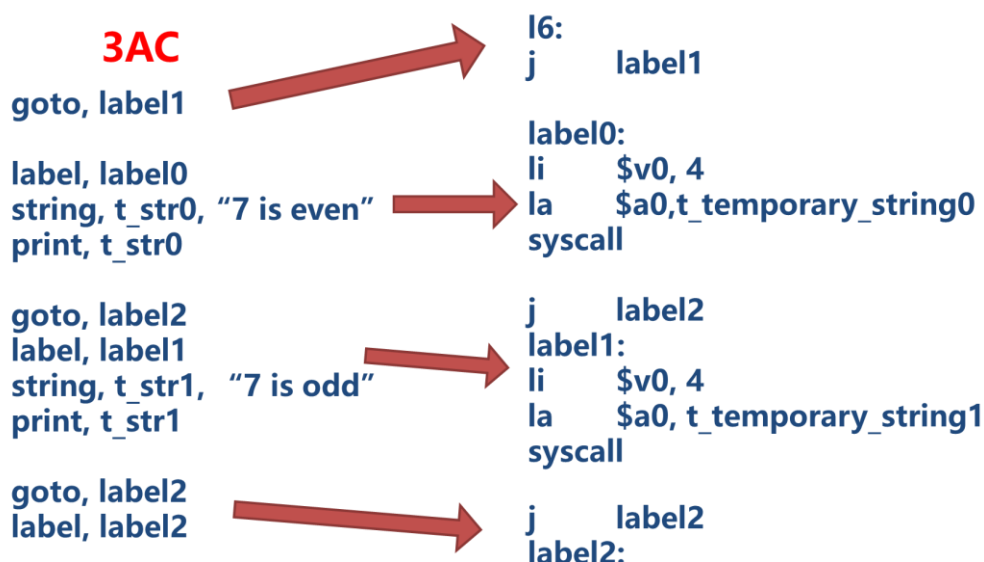
Assembly

l1:
j main_prog
main_prog:

li \$t8, 7
rem \$a1, \$t8, 2
seq \$a2, \$a1, 0
sw \$a2, t1
sw \$a1, t0
lw \$a3, t1
beq \$a3, 1, label0

Key:
变量的保存
与寄存器的分配

接续前一部分的 go 语言转三地址码部分。按基本块划分汇编码，此实例中为 11 和 16，之后逐段检查三地址码类的类型，并将功能相同的 MIPS 汇编码输出，图中箭头表示转化前与转化后的情况。另需处理的是变量值的保存与寄存器的合理分配。接下来解释汇编码的含义，首先进入基本块 11，跳转至 main 部分，进入第一个条件判断。将立即数 7 保存至启发式生成的寄存器 \$t8 中，之后 rem 指令计算其除以 2 的余数并保存在寄存器 \$a1 中。seq 指令用于判断(\$a1)是否为 0，若为 0 则(\$a2)为 1。sw 指令将寄存器中保存的值转储到内存地址 t1 和 t0 处。之后从内存将原(\$a2)加载入 \$a3 中，beq 指令判断(\$a3)是否为 1，是则跳转至 label0。由于计算至此(\$a3)为 0，所以不跳转，而是继续顺序执行。



之后进入基本块 16，根据上一个块中最终结果执行条件跳转。如进入 label0 后，首先将执行的调用名称 4（即 print）加载入 \$v0，□之后将标记为 0 号的字符串调入 \$a0，最后执行系统调用。label2 开始第二组条件判断。其余转化过程类似。

5 错误分析与处理的实现

5.1 Yacc 语法说明文件的错误处理规则

Yacc 中，语法说明文件（即 Yacc 源文件，一般表示为 xxx.y 的格式）分为三个部分：

第一部分：

定义段，可分为两部分，第一部分用 '%{' 和 '%}' 包围起来，其内部是 C 语法写的一些定义和声明。

第二部分是对文法的终结符和非终结符做一些相关声明，声明中包含其部分特性，包括是否为终结符，有无结合性，若有，是左结合还是右结合，等等。

第二部分：

规则段，其定义了文法的非终结符以及产生式集合，以及当归约整个产生式时应该执行的操作。例如：

```

1 | expr: expr PLUS term      {语义动作}
2 |   | term                  {语义动作}
3 |   ;
```


产生式后大括号括起的语义动作表示该条产生式归约时应当执行的操作。（按照这条规则归约时，程序实际要做什么）

第三部分：

辅助函数部分，使用 C 语言语法来写，一般是规则段中或者语法分析器的其他部分用到的函数。

“一般来说，除规则段用到的函数外，辅助函数段一般包括如下一些例程：yylex(), yyerror(), main() 。”（摘自 Yacc 手册，也是使用代码中不规范的一点：yylex()和 yyerror()似乎被写在了定义段）

（关于 yylex()的功能在此暂且不表）

yyerror()是 Yacc 提供的错误信息报告程序，其库中源程序如下：

```
#include <stdio. h>
yyerror (s) char * s{
fprintf (stderr, "%s n",s);
}
```

如果用户需要更复杂的自定义错误信息报告程序，可以自己覆写这一函数。

错误处理：

Yacc 处理错误的方法是：在其发现语法错误时，抛弃那些导致错误的符号来调整状态栈，然后从出错处的后一个符号处或是跳过若干符号直到遇到指定的某符号时继续分析。（即在遇到错误时直接开始抛弃符号，直到出错处全部清除或者遇到用户指定的某个符号才继续）Yacc 拥有一个保留终结符 **error**，不需声明也可使用。将其写在某个产生式（或者说某条规则）右部，Yacc 就认为这个地方是可能出错的，如果确实出错，就进行上述错误处理。

例如：

stat: error;

这使得 Yacc 在分析 stat 推导出的句型时可以在遇到语法错误时直接跳过出错部分继续分析（但仍然会打印语法错误信息）

又例如

stat: error';';

这使得 Yacc 在分析 stat 推导出的句型时，在遇到语法错误时开始不断抛弃符号流中的符号，直到找到下一个分号为止。

有时我们会需要用到'yerrorok'这一语句，用于在一个操作中将解析器重置为常规模式，从而避免编译器在重新同步的过程中错误地在删除不合法的 token 时不报错，这使得一些错误无法在一次编译尝试中被看到。

5.2 三地址码生成中的错误处理

本次项目制实践代码的做法：

```
if(vec_identifier_list.size()!= vec_expr_list.size()){
cout<<"ERROR : Unequal # of expressions on LHS "<<vec_identifier_list.size()<<" & RHS "<<vec_expr_list.size()<<endl;
}
```

使用了不规范的做法：并没有使用 yyerror()函数来进行错误信息的报告，对发生的如上错误也并没有使用 yerrorok 或 yerrclearin 等操作，也没有在语义动作里进行额外的错误处理，仅仅是在控制台上打印一条错误信息，同时因为没有 error 规则，Yacc 在错误恢复过程中选择直接抛弃出错部分。

在symtab.h 中，主要解决变量未声明的问题。

定义是否处于错误状态的变量：

```
1.  bool error_status = 0;
```

变量未定义的错误：

```
1.  void check_entry(string ident,table *t){
2.      if(!lookup(ident,t)){
3.          cout << "ERROR: variable " << ident << " NOT declared.\n";
4.          error_status = 1;
5.      }
6.  }
```

其中的lookup 函数如下：

```
1.  bool lookup(string ident,table * t1){
2.      int check = 0;
3.      //while(t1!=NULL){
4.      table * temp1 = t1;
5.      //cout << "LOOKING FOR " << ident << endl;
6.      while(temp1->table_type!=1){
7.          if(look_in_map(ident,temp1->symtab)==1){
8.              check=1;
9.              break;
10.         }
11.
12.         temp1=temp1->parent;
13.     }
14.
15.     return check;
16. }
```

look_in_map 是我们定义的一个布尔函数：

```
1.  bool look_in_map(string ident,symboltable *symtab1){
2.      int check = 0;
3.      symboltable &symtab = *symtab1;
4.      unordered_map<string, s_entry>:: iterator itr;
5.      if(symtab.size()==0){
6.          check=0;
7.      }
8.      else{
9.          for(itr=symtab.begin();itr!=symtab.end();itr++){
10.             if(itr->first==ident){
11.                 check = 1;
12.                 break;
13.             }
14.         }
15.     }
16.
17.     return check;
18. }
```

table 的结构：

```
1.  typedef struct table{
2.      symboltable* symtab;
3.      table *parent;//=NULL;
4.      int table_type=0;
5.      //table *child=NULL;
6.  }table;
```

而在parserIR.y 文件中，关于错误处理的部分，Yacc自己提供的错误信息报告函数为yyerror()，源程序如下：

```
1.  void yyerror (const char *s){
2.      fprintf (stderr, "%s\n", s);
```

6 生成可执行文件

因为软件和硬件之间隔着一个操作系统，我们将我们写好的源代码编译生成了汇编代码，但是不可执行，因为不同的操作系统操作要求不同，这时我们就需要通过汇编得到二进制文件，进而链接生成可执行文件来实现不同操作系统需要的要求。

6.1 机器语言

6.1.1 机器语言的格式

1. 机器语言指令有操作码(OP)和地址码两部分组成

| _____ OP _____ | | d | | w |

| _____ OP _____ | | s | | w | <--此格式用于立即寻址方式

在多数操作码中，常使用某些位来指示某些信息：

如图上结构里的： w=1 时 对字来操作

w=0 时 对字节来操作

d值在双操作数指令中才有效

当 d=1 时 有且只有一个寄存器用于目的操作数

d=0 时 有且只有一个寄存器用于源操作数

s=1 时 立即数为8位，但要求扩展成16位数

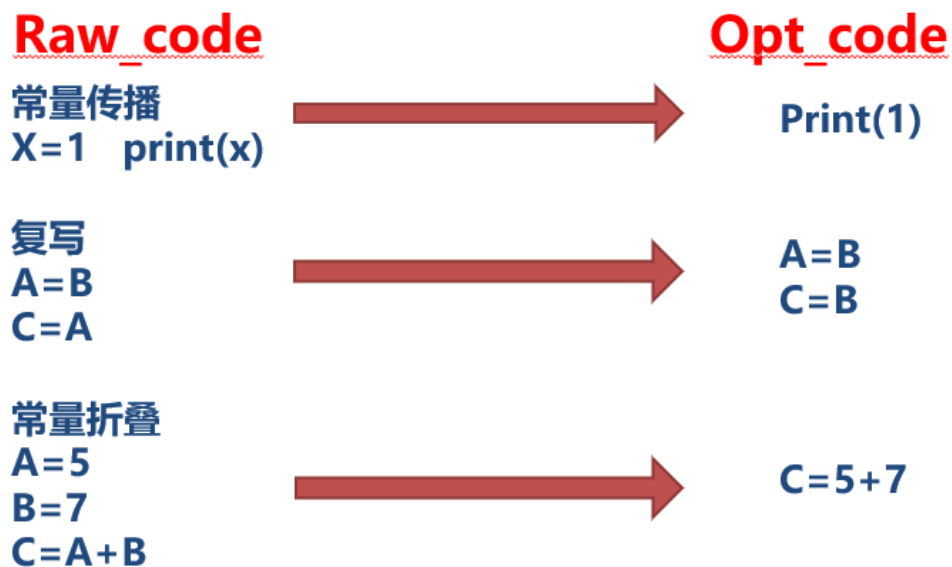
s=0 时 当指令作字节操作/有16位立即数

2. 寻址方式的机器语言表示：

| mod | reg | r/m |
| _____ |

reg 表示寄存器方式，在不包括立即数的双操作数指令的情况下，规定必须有一个操作数在寄存器中，该寄存器由reg字段指定，并与操作码字节中的w位相组合确定的寄存器

代码优化部分主要有下列三个板块：



可是存在的困惑是，代码优化的最终目标之一，是加快程序的执行速度，缩短程序的执行时间。但是因为小组实现的编译器还不够完善与成熟，很难确保在源程序的基础上进行代码优化的总执行时间会快于原始的程序执行时间。因此这里是需要进一步提升的地方。

6.1.2 汇编语言转化为机器语言

事实上，因为汇编代码和机器代码基本一一对应，这个过程并不困难。通过搜索linux的调试工具,radare2有附加的工具来转化。执行代码如下：

1. sudo apt-get install radare2 #安装 radare2 工具
2. rasm2 -L #查看 architectures
3. rasm2 -a mips -b 32 test1.s -o test1.o #转机器码
4. rasm2 -a mips -b 32 test2.s -o test2.o
5. rasm2 -a mips -b 32 test3.s -o test3.o

6.2 链接

链接过程就是要把编译器生成的一个个目标文件链接成可执行文件。最终得到的文件是分成各种段的，比如数据段、代码段、BSS段等等，运行时会被装载到内存中。各个段具有不同的读写、执行属性，保护了程序的安全运行。具体步骤如下：

1. 链接器收集main包引用的所有其它包中的符号信息，并将它们装载到一个大的字节数组中
2. 对于每个符号，链接器计算它在（数组）镜像中的地址。
3. 然后他为每个符号应用重定位，
4. 链接器准备所有ELF格式（linux系统中）文件所需的文件头。然后它再生成一个可执行的文件。

7 可视化编译 IDE 的实现

7.1 实现方法

实现的代码如下所示，利用文件对话框 `QFileDialog`，打开文本文件，将文本文件的内容显示到窗口上。除此之外，还实现了在IDE窗口中直接编辑。最后点击run按钮，即可弹出窗口生成抽象语法树、三地址代码和汇编程序。

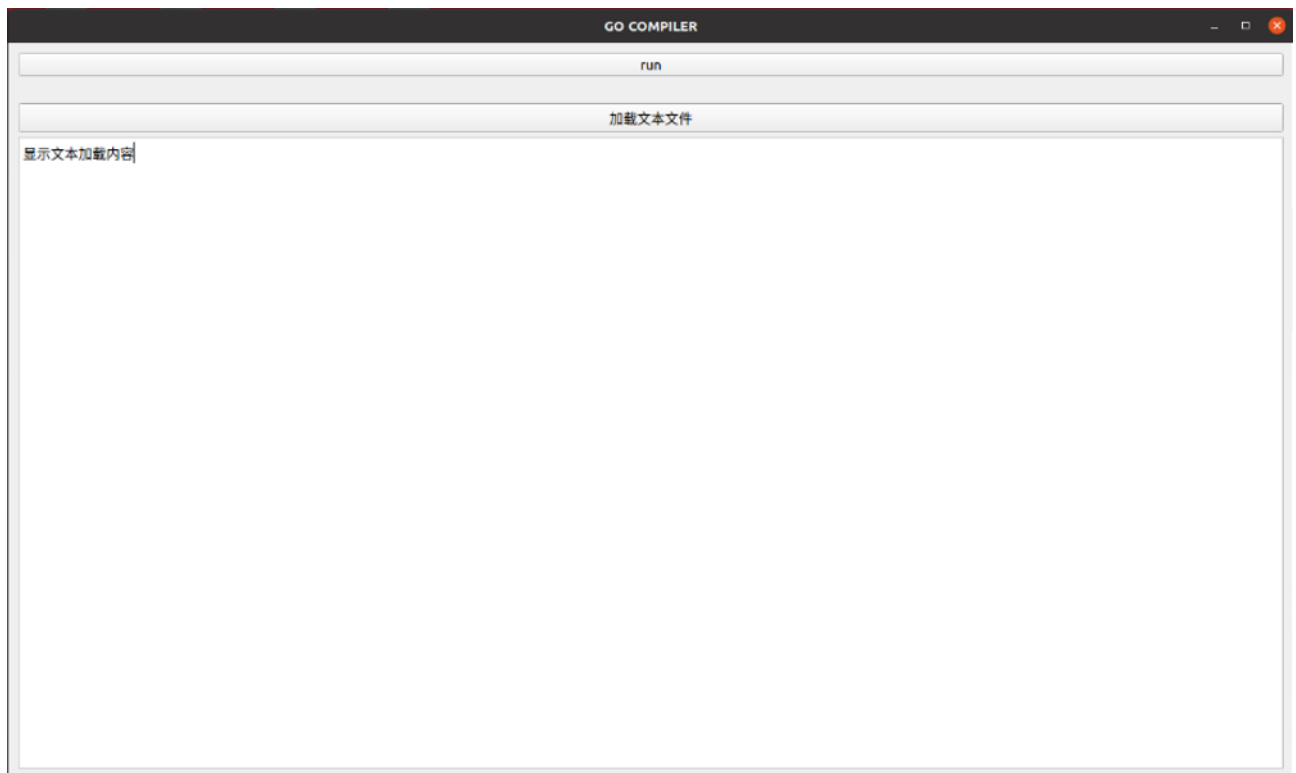
```
1. """
2. 文件对话框 QFileDialog
3. 最常用的是打开文件和保存文件对话框
4. """
5. # 需求：
6. # 1.打开文件，显示到窗口上
7. # 2.打开文本文件，将文本文件的内容显示到窗口上
8.
9. import sys
10. from PyQt5.QtCore import *
11. from PyQt5.QtGui import *
12. from PyQt5.QtWidgets import *
13. import subprocess
14.
15.
16. class QFileDialogDemo(QWidget):
17.     def __init__(self):
18.         super(QFileDialogDemo, self).__init__()
19.         self.initUI()
20.
21.     # 编写初始化方法
22.     def initUI(self):
23.         # 设置窗口标题
24.         self.setWindowTitle('GO COMPILER')
25.
26.         # 创建垂直布局
27.         layout = QVBoxLayout()
28.
29.         # 创建 button1 控件，用于加载图片
30.         self.button1 = QPushButton('run')
31.
32.         # 创建 label 控件，把图像显示到 label 控件上
33.         self.imageLabel = QLabel()
34.
35.         # 创建 button2 控件，用于加载文件
36.         self.button2 = QPushButton('加载文本文件')
37.
38.         # 创建 QTextEdit 控件，来显示文本加载的内容
39.         self.contents = QTextEdit('显示文本加载内容')
40.         # 连接信号槽
```

```
41.     self.button1.clicked.connect(self.runTest)
42.     self.button2.clicked.connect(self.loadText)
43.
44.     # 把控件添加到垂直布局里
45.     layout.addWidget(self.button1)
46.     layout.addWidget(self.imageLabel)
47.     layout.addWidget(self.button2)
48.     layout.addWidget(self.contents)
49.
50.     # 应用于垂直布局
51.     self.setLayout(layout)
52.
53.     # 槽方法
54.     def runTest(self):
55.         #print("wuhuqifei")
56.
57.
58.         #generate final code
59.         ss = self.contents.toPlainText()
60.
61.         gogo = open('./go/gogo.go', encoding='utf-8', mode='w+')
62.         gogo.write(ss)
63.         gogo.close()
64.         print(ss)
65.         #execute test.sh
66.         subprocess.call("./test.sh", shell=True)
67.
68.     def loadText(self):
69.         # 直接创建 QFileDialog, 第二种方法
70.         # 创建对象
71.         dialog = QFileDialog()
72.         # 设置文件创建模式
73.         dialog.setFileMode(QFileDialog.AnyFile)
74.         # 选择文件
75.         dialog.setFilter(QDir.Files)
76.
77.         #打开文件
78.         if dialog.exec():
79.             # 如果打开成功
80.             filename = dialog.selectedFiles()
81.             print(filename)
82.             # 打开文件, 可以打开多个, 取第一个
83.             f = open(filename[0], encoding='utf-8', mode='r')
84.
85.             # cmd = open('./test.sh', encoding='utf-8', mode='w+')
86.             # cmd.write('#! /bin/bash\n')
87.             # cmd.write('cd ./go\n')
88.             # cmd.write('sh compile.sh ')
89.             # cmd.write(filename[0])
90.             # cmd.close()
91.
```

```
92.
93.
94.
95.     # 读取
96.     # 使用 with 的原因，自动关闭，当 with 读取结束后，会自动调用 f 里面
    的 close 方法关闭文档
97.     with f:
98.         data = f.read()
99.         self.contents.setText(data)
100.
101.
102.
103. # 防止别的脚本调用，只有自己单独运行时，才会调用下面代码
104. if __name__ == '__main__':
105.     # app 实例化，并传递参数
106.     app = QApplication(sys.argv)
107.     # 创建对象
108.     main = QFileDialogDemo()
109.     # 创建窗口
110.     main.show()
111.     # 进入程序的主循环，通过 exit 函数
112.     sys.exit(app.exec_())
```

7.2 效果展示

1. 可视化IDE窗口效果图，包括两个功能按钮和一个文本编辑区。加载文本文件按钮可以选择系统中的某项文件，将文本文件的内容加载到窗口中；Run按钮将加载文本编辑区中的内容生成抽象语法树、三地址代码和汇编程序。



2. 点击加载文本文件按钮，选择系统中的go语言文件test1.go，将文本文件的内容加载到窗口中。也可以在文本编辑区中直接编辑，写代码。

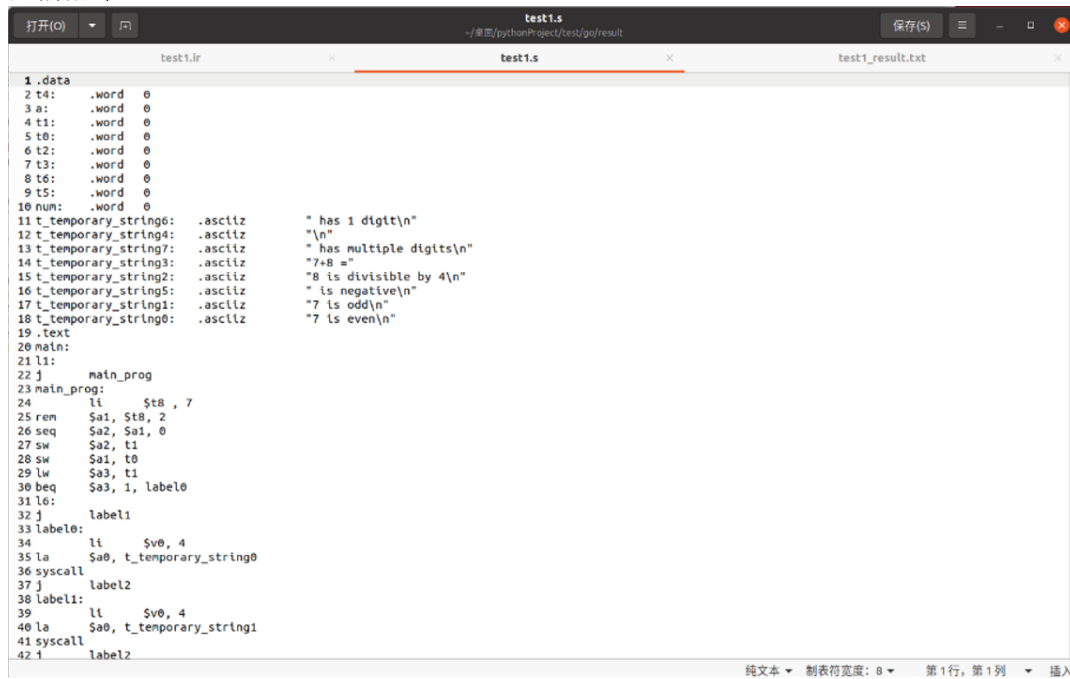
```
package main;
import "fmt";
func main() {

    if (7%2 == 0){
        fmt.Println("7 is even\n");
    } else {
        fmt.Println("7 is odd\n");
    };

    if (8%4 == 0) {
        fmt.Println("8 is divisible by 4\n");
    };
    var a int = 7+8;
    fmt.Println("7+8 =",a,"\n");
    var num int = 45;
    if (num < 0) {
        fmt.Println(num, " is negative\n");
    } else if (num < 10) {
        fmt.Println(num, " has 1 digit\n");
    } else {
        fmt.Println(num, " has multiple digits\n");
    };
}
```

3. 点击Run按钮将加载文本编辑区中的内容生成抽象语法树、三地址代码和汇编程序。

Test1.s: 汇编指令



```
1 .data
2 t4: .word 0
3 a: .word 0
4 t1: .word 0
5 t0: .word 0
6 t2: .word 0
7 t3: .word 0
8 t6: .word 0
9 t5: .word 0
10 num: .word 0
11 t_temporary_string6: .asciiz " has 1 digit\n"
12 t_temporary_string4: .asciiz "\n"
13 t_temporary_string7: .asciiz " has multiple digits\n"
14 t_temporary_string3: .asciiz "7+8 ="
15 t_temporary_string2: .asciiz "8 is divisible by 4\n"
16 t_temporary_string5: .asciiz " is negative\n"
17 t_temporary_string1: .asciiz "7 is odd\n"
18 t_temporary_string0: .asciiz "7 is even\n"
19 .text
20 main:
21 li:
22 j main_prog
23 main_prog:
24 li $t8, 7
25 rem $a1, $t8, 2
26 seq $a2, $a1, 0
27 sw $a2, t1
28 sw $a1, t0
29 lw $a3, t1
30 beq $a3, 1, label0
31 li:
32 j label1
33 label0:
34 li $v0, 4
35 la $a0, t_temporary_string0
36 syscall
37 j label2
38 label1:
39 li $v0, 4
40 la $a0, t_temporary_string1
41 syscall
42 i label2
```

Test1_result.txt: 程序执行结果



```
1 7 is odd
2
3 8 is divisible by 4
4
5 7+8 = 15
6
7 45 has multiple digits
8
```