



南开大学
Nankai University

南 开 大 学

计算机与网络空间安全学院

编译原理实验报告

定义你的编译器 & ARM 汇编编程

2014074 费泽锟

2011763 黄天昊

年级：2020 级

指导教师：王刚

2022 年 10 月 13 日

摘要

本次实验确定了实现的编译器支持的 SysY 语言特性，运用上下文无关文法（CFG）描述 SysY 语言子集。本次实验还要对 ARM 汇编程序进行学习，了解 ARM 汇编语言的逻辑，编写几个简单的 SysY 程序，使用 ARM 汇编语言将其复写。

在 CFG 设计和 ARM 汇编编程的基础上，本小组还探究了如何将表达式翻译成汇编指令，成功实现了简单表达式到 x86 汇编语句的 Bison 翻译。

关键字：CFG，ARM 汇编，符号化，Bison

目录

一、 小组分工	1
二、 实验内容	1
(一) CFG 设计	1
1. 声明	2
2. 函数	2
3. 语句	3
4. 表达式	3
5. 注释	3
6. 常量	4
(二) ARM 汇编	4
(三) Bison 实现简单表达式翻译	10
三、 总结	16

一、 小组分工

1. CFG 设计

黄天昊：负责设计数据类型的实现，浮点数，变量、常量声明与初始化，语句（即后文中实现的 SysY 语言特性 1-3 与 7-9）

费泽锟：负责设计注释、输入输出、表达式（即后文中实现的 SysY 语言特性 4-6）

2. ARM 汇编

黄天昊：负责完成求阶乘的简单 SysY 程序到 ARM 汇编程序的翻译与复写，并通过生成可执行文件验证了对应 ARM 汇编程序的正确性。

费泽锟：负责完成数组循环或的简单 SysY 程序到 ARM 汇编程序的翻译与复写，验证了对应 ARM 汇编程序的逻辑正确性。（SysY 程序同预备工作 1）

3. Bison 简单表达式翻译

黄天昊：负责实现包含常量和变量的简单表达式翻译。

费泽锟：负责实现包含变量，常量和变量符号表的简单表达式翻译。

小组 GitLab 代码链接如下：

<https://gitlab.eduxiji.net/nku-ikun/principles-of-compiler-design/-/tree/master/>

二、 实验内容

（一） CFG 设计

在编译器的设计过程中，我们希望编译器能够将 SysY 语言编写的程序翻译成对应平台的汇编指令，首先就需要确定想要实现的 SysY 语言特性都有哪些 [3,4]。

本组在设计过程中，确定 SysY 语言特性如下：

1. 数据类型的实现：int, float
2. 变量声明、常量声明，常量、变量的初始化
3. 语句：赋值（=）、表达式语句、语句块、if、while、for、return
4. 表达式：算术运算（+、-、*、/、%，其中 +、- 都可以是单目运算符）、关系运算（==, >, <, >=, <=, !=）和逻辑运算（&&（与）、||（或）、！（非））
5. 注释（包含单行注释和多行注释，与 C 语言相同）
6. 输入输出（实现连接 SysY 运行时库）
7. 函数、语句块：函数声明、函数调用、变量、常量作用域
8. 数组：数组（一维、二维、…）的声明和数组元素访问
9. 浮点数：浮点数常量识别、变量声明、存储、运算

在确定了要实现的 SysY 语言特性之后，就可以将其抽象化、符号化，参考巴克斯瑙尔范式定义，根据上下文无关文法设计出相应的产生式，再通过 Bison 工具来实现简单的源程序的翻译了。

具体的 CFG 设计是由小组分工设计，再通过讨论综合得到的，具体设计将在以下展开。

1. 声明

这一部分主要包括函数、常量、变量的声明

抽象语义	表达式
编译单元	$\text{CompUnit} \rightarrow \text{CompUnit Decl} \mid \text{CompUnit FuncDef} \mid \text{Decl} \mid \text{FuncDef}$
声明	$\text{Decl} \rightarrow \text{ConstDecl} \mid \text{VarDecl}$
基本类型	$\text{BType} \rightarrow \text{'int'} \mid \text{'float'}$
常量声明	$\text{ConstDecl} \rightarrow \text{'const'} \text{ BType ConstDef ConstDefList ';'}$
常数定义中间符号	$\text{ConstDefList} \rightarrow \text{ConstDefList ',' ConstDef} \mid \epsilon$
常数定义	$\text{ConstDef} \rightarrow \text{Ident ConstExpList '=' ConstInitVal}$
常量表达式中间符号	$\text{ConstExpList} \rightarrow \text{ConstExpList '[' ConstExp ']' } \mid \epsilon$
常量初值	$\text{ConstInitVal} \rightarrow \text{ConstExp} \mid \text{'{' ConstInitVal ConstInitValList '}' } \mid \text{'{' '}'}$
常量初值中间符号	$\text{ConstInitValList} \rightarrow \text{ConstInitValList ',' ConstInitVal} \mid \epsilon$
变量声明	$\text{VarDecl} \rightarrow \text{BType VarDef VarDefList ';'}$
变量声明中间符号	$\text{VarDefList} \rightarrow \text{VarDefList ',' VarDef} \mid \epsilon$
变量定义	$\text{VarDef} \rightarrow \text{Ident ConstExpList} \mid \text{Ident ConstExpList '=' InitVal}$
变量初值	$\text{InitVal} \rightarrow \text{Exp} \mid \text{'{' InitVal InitValList '}' } \mid \text{'{' '}'}$
变量初值中间符号	$\text{InitValList} \rightarrow \text{InitValList ',' InitVal} \mid \epsilon$

表 1: 声明（函数、变量、常量）部分的 CFG 设计

2. 函数

抽象语义	表达式
函数定义	$\text{FuncDef} \rightarrow \text{FuncType Ident '(' FuncFParams ')'} \text{ Block}$ $\mid \text{FuncType Ident '(' ')} \text{ Block}$
函数类型	$\text{FuncType} \rightarrow \text{'void'} \mid \text{'int'} \mid \text{'float'}$
函数形参表	$\text{FuncFParams} \rightarrow \text{FuncFParam FuncFParamList}$
函数形参中间符号	$\text{FuncFParamList} \rightarrow \text{FuncFParamList ',' FuncFParam} \mid \epsilon$
函数形参	$\text{FuncFParam} \rightarrow \text{BType Ident '[' ']' ExpList} \mid \text{BType Ident}$
函数中间符号	$\text{ExpList} \rightarrow \text{ExpList '[' Exp ']' } \mid \epsilon$

表 2: 函数部分的 CFG 设计

3. 语句

抽象语义	表达式
语句块	Block \rightarrow '{' BlockItemList '}'
语句块项中间符号	BlockItemList \rightarrow BlockItemList BlockItem $\mid \epsilon$
语句块项	BlockItem \rightarrow Decl \mid Stmt
语句	Stmt \rightarrow LVal '=' Exp ';' \mid Exp ';' \mid ';' \mid Block \mid 'if' '(' Cond ')' Stmt 'else' Stmt \mid 'if' '(' Cond ')' Stmt \mid 'while' '(' Cond ')' Stmt \mid 'break' ';' \mid 'continue' ';' \mid 'return' Exp ';' \mid 'return' ';' ;

表 3: 语句部分的 CFG 设计

4. 表达式

部分表达式采用了扩展的 Backus 范式表达, 以简化产生式表达。其中, 符号 (...) 表示可以匹配圆括号内的任一字符。

抽象语义	表达式
表达式	Exp \rightarrow AddExp
条件表达式	Cond \rightarrow LOExp
左值表达式	LVal \rightarrow identifier Explist
基本表达式	PrimaryExp \rightarrow '(' Exp ')' \mid LVal \mid Number
数值	Number \rightarrow integer-const \mid floating-const
一元表达式	UnaryExp \rightarrow PrimaryExp \mid identifier '(' ')' \mid identifier '(' FuncRParams ')'
单目运算符	UnaryOp \rightarrow '+' \mid '-' \mid '!'
函数实参表	FuncRParams \rightarrow Func-Explist
函数实参列表	Func-Explist \rightarrow Func-Explist ',' Exp \mid Exp $\mid \epsilon$
乘除模表达式	UnaryExp \mid MulExp '*' UnaryExp
乘除模表达式	\mid MulExp '/' UnaryExp
乘除模表达式	\mid MulExp '%' UnaryExp
加减表达式	MulExp \mid AddExp '+' MulExp
加减表达式	MulExp \mid AddExp '-' MulExp
关系表达式	RelExp \rightarrow AddExp \mid RelExp ('<' \mid '>' \mid '<=' \mid '>=') AddExp
相等性表达式	EqExp \rightarrow RelExp \mid EqExp ('==' \mid '!=') RelExp
逻辑与表达式	LAndExp \rightarrow EqExp \mid LAndExp '&&' EqExp
逻辑或表达式	LOrExp \rightarrow LAndExp \mid LOrExp ' ' LAndExp
常量表达式	ConstExp \rightarrow AddExp

表 4: 表达式部分的 CFG 设计

5. 注释

注释部分的 CFG 设计, 包含了以 '//' 符号开始的单行注释 (不包含换行符) 和以 '/*' 符号开始、'*/' 符号结束的多行注释 (包含换行符)。

抽象语义	表达式
注释	Comments \rightarrow <code>'/'</code> strings-nonLBreak <code>'/*'</code> strings <code>'*/'</code> ϵ
无换行符字符串	strings-nonLBreak \rightarrow char-nonLBreak-list
无换行符字符串列表	char-nonLBreak-list \rightarrow char-nonLBreak-list char-nonLBreak char-nonLBreak ϵ
含换行符字符串	strings \rightarrow char-list
含换行符字符串列表	char-list \rightarrow char-list char ϵ

表 5: 注释部分的 CFG 设计

6. 常量

常量的表达包含了 int 型常量的表达和 float 型常量的表达, 虽然这一部分看上去很像词法分析的流程, 这里仍旧给出了常量表达的 CFG 设计 (参考英文文档: ISO/IEC 9899)

抽象语义	表达式
整型常量	integer-const \rightarrow decimal-const
十进制整型常量	decimal-const \rightarrow nonzero-digit decimal-const digit
浮点型常量	floating-const \rightarrow decimal-floating-const
十进制浮点型常量	decimal-floating-const \rightarrow fractional-const exponent-part digit-sequence exponent-part
浮点部分	fractional-const \rightarrow digit-sequence <code>'.'</code> digit-sequence <code>'.'</code> digit-sequence digit-sequence <code>'.'</code>
指数部分	exponent-part \rightarrow <code>'e'</code> Sign digit-sequence <code>'E'</code> Sign digit-sequence <code>'e'</code> digit-sequence <code>'E'</code> digit-sequence ϵ
符号	Sign \rightarrow <code>'+'</code> <code>'-'</code>
数字序列	digit-sequence \rightarrow digit-sequence digit

表 6: 常量部分的 CFG 设计

(二) ARM 汇编

对于 ARM 汇编部分, 通过对实验文档的学习 [1,2], 我们可以了解到, 如果要使用 ARM 汇编语言编写函数的话, 需要先使用 `.global functionname` 和 `.type functionname, %function` 指令对函数进行声明, 我们也可以使用 `__bridge` 标签桥接在源代码中隐性的全局变量的地址, 但是这些都是基于 arch armv5t 版本之下的特性, 接下来将介绍简单的 ARM 汇编指令以及 armv7-a 版本之下的 ARM 汇编程序。

ARM 汇编指令	含义
LDR	字数据加载指令
LDRB	字节数据加载指令
LDRH	半字数据加载指令
CMP	比较指令，根据运算结果设置了各个标志位
TST	逻辑处理指令，用于把两个操作数进行按位与运算
BNE	数据跳转指令，标志寄存器中 Z 标志位等于零时跳转
BEQ	数据跳转指令，标志寄存器中 Z 标志位不等于零时跳转
STR	字数据存储器指令
STRB	字节数据存储器指令
STRH	半字数据存储器指令
B	无条件跳转指令
BL	跳转到标号 Label 处执行，同时将当前的 PC 值保存到 R14 中
BX	目标地址处的指令既可以是 ARM 指令，也可以是 Thumb 指令
LSL	算术左移指令
LSR	算术右移指令
AND	逻辑与操作指令
ORR	逻辑或操作指令
EOR	逻辑异或操作指令

表 7: 常用的 ARM 汇编指令

接下来我们来观察一下 armv7-a 版本下的编译器的具体翻译的操作特性。我们用以下的一段简单的 SysY 代码为例：

简单的变量赋值

```

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

接下来我们来看看，这段简单的 C 代码使用 arm-linux-gnueabi-gcc test.c -S -o test.S 命令后得到的 ARM 汇编结果。

简单的变量赋值 (ARM 汇编)

```

1 .arch armv7-a
2 .fpu vfpv3-d16
3 .eabi_attribute 28, 1
4 .eabi_attribute 20, 1
5 .eabi_attribute 21, 1
6 .eabi_attribute 23, 3
7 .eabi_attribute 24, 1
8 .eabi_attribute 25, 1
9 .eabi_attribute 26, 2
```

```

10      .eabi_attribute 30, 6
11      .eabi_attribute 34, 1
12      .eabi_attribute 18, 4
13      .file      "test.c"
14      .text
15      .align    1
16      .global   main
17      .syntax   unified
18      .thumb
19      .thumb_func
20      .type     main, %function
21 main:
22      @ args = 0, pretend = 0, frame = 8
23      @ frame_needed = 1, uses_anonymous_args = 0
24      @ link register save eliminated.
25      push     {r7}
26      sub      sp, sp, #12
27      add      r7, sp, #0
28      movs     r3, #1
29      str      r3, [r7, #4]
30      movs     r3, #0
31      mov      r0, r3
32      adds     r7, r7, #12
33      mov      sp, r7
34      @ sp needed
35      ldr      r7, [sp], #4
36      bx       lr
37      .size    main, .-main
38      .ident   "GCC: (Ubuntu 11.2.0-17ubuntu1) 11.2.0"
39      .section .note.GNU-stack,"",%progbits

```

我们可以发现在 ARM 汇编中，定义了字符集类型以及 thumb 模式，以及初始化了接口属性，这些显然在我们自己编程的过程之中是不需要的。

armv7-a 的一个很显著的不同的特征就是在函数实现的过程中（该段代码中为 main 函数）并没有像我们学习的那样，使用 fp 作为栈底指针，使用 sp 作为栈顶指针。而是只使用 sp 作为栈顶指针，并且将 r7 寄存器的值赋值为 sp 的地址，再通过 r7 寄存器进行函数内的寻址，这一点与我们应当实现的截然不同，本实验中，小组仍然保持的是栈底和栈顶指针的结构，手写简单的 ARM 汇编程序。

armv7-a 的另一个很显著的不同的特征就是在于栈中的变量地址的分配问题，用以下两段 SysY 代码为例：

为变量 a 赋值

```

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

```


为变量 b 赋值

```

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

```

这两段代码很简单，就是为不同的变量赋值，根据我们学习到的关于栈的知识，这两段汇编代码应当是不一样的，因为 a 和 b 变量的地址是不同的，局部变量中 a 变量应当为高地址，但是这两段代码在 armv7-a 的版本编译下，ARM 汇编代码是完全一样的，也就是说 a 和 b 变量均为 int 类型变量，编译器就没有在意地址问题，而是哪一个变量被先赋值了，那么就从高地址顺序给变量分配空间。ARM 汇编结果如下：

赋值部分的汇编代码

```

1 movs    r3, #1
2 str     r3, [r7, #4]

```

所以说编译器直接翻译的结果与手动编写 ARM 汇编程序的结果完全不同，我们小组编写了两个简单的 SysY 程序，手动复写了其 ARM 汇编程序。

首先是数组循环执行逻辑或操作的简单 SysY 程序，具体代码如下（同预备工作 1）

数组循环或操作

```

1 int main() {
2     int a[5];
3     int ans;
4     int i;
5     i = 0;
6     ans = getint();
7     //cin >> ans;
8     while (i < 5) {
9         a[i] = i + 1;
10        ans = ans | a[i]; //todo 'or' operation
11        i++;
12    }
13    //cout << ans;
14    putint(ans);
15    return 0;
16 }

```

该段代码体现了 SysY 语言中的一维数组的实现、输入输出、注释、while 循环语句、return 语句等特性，手动复写的 ARM 汇编程序如下：

ARM 汇编程序

```

1 .arch armv7-a
2 .text

```

```

3      .align 2
4      .global main
5      .type   main, %function
6  main:
7      push    {fp, lr}
8      mov     fp, sp
9      sub     sp, sp, #32
10     mov     r4, #0
11     str     r4, [fp, #-32]           @i=0
12     bl      getint(PLT)
13     str     r0, [fp, #-28]          @ans=getint()
14  .L2:
15     ldr     r4, [fp, #-32]
16     cmp     r4, #5                  @i<5    blt
17     bge     .L3
18     ldr     r4, [fp, #-32]
19     add     r2, r4, #1              @r2=i+1
20     ldr     r4, [fp, #-32]
21     lsl     r4, r4, #2              @4i
22     add     r4, r4, fp
23     str     r2, [r4, #-24]          @a addr
24     ldr     r4, [fp, #-32]
25     lsl     r4, r4, #2
26     add     r4, r4, fp
27     ldr     r4, [r4, #-24]
28     ldr     r2, [fp, #-28]
29     orr     r4, r4, r2              @| instruction
30     str     r4, [fp, #-28]
31     ldr     r4, [fp, #-32]
32     add     r4, r4, #1
33     str     r4, [fp, #-32]          @i++
34     b       .L2
35  .L3:
36     ldr     r0, [fp, #-28]          @ans=putint()
37     bl      putint(PLT)
38  .L4:
39     mov     sp, fp
40     pop     {fp, pc}
41
42     .section      .note.GNU-stack,"",%progbits

```

在该 ARM 汇编程序中实现了具体的 while 跳转逻辑，并且在函数初始化的过程中，初始化了 fp 和 sp 寄存器，来体现具体的栈结构，并且维持了变量间高地址和低地址之间的关系，需要注意的是需要使用 PLT 来调用外部函数（即 sylib.c 中的函数）。

使用 arm-linux-gnueabi-gcc test.S -o test 指令后，可见运行可执行文件的结果正确，验证图：



```

zzekun@zzekun-virtual-machine: ~/Compilers-Work/ARM-As...
zzekun@zzekun-virtual-machine:~/Compilers-Work/ARM-Assembly$ qemu-arm -L /usr/ar
m-linux-gnueabihf/ ./test
100
TOTAL: 0H-0M-0S-0us
103zzekun@zzekun-virtual-machine:~/Compilers-Work/ARM-Assembly$

```

图 1: 循环或操作结果验证

第二段代码是在之前的手册中出现的基础样例程序，体现了 SysY 语言中输入输出、while 循环语句、return 语句等特性，代码如下：

求阶乘程序

```

1  int main()
2  {
3      int i, n, f;
4      n = getint();
5      i = 2;
6      f = 1;
7      while (i <= n)
8      {
9          f = f * i;
10         i = i + 1;
11     }
12     putint(f);
13     return 0;
14 }

```

手动翻译对应的 ARM 汇编程序如下：

ARM 汇编程序

```

1      .arch armv7-a
2      .text
3      .global main
4      .type    main, %function
5  main:
6      push    {fp, lr}
7      push    {r7}
8      mov fp, sp
9      sub     sp, sp, #16
10     add     r7, sp, #0
11     bl      getint(PLT)
12     str     r0, [r7, #12]    @ n=getint(PLT)
13     movs    r3, #2
14     str     r3, [r7, #4]     @ i=2
15     movs    r3, #1
16     str     r3, [r7, #8]     @ f=1
17     b       .L2

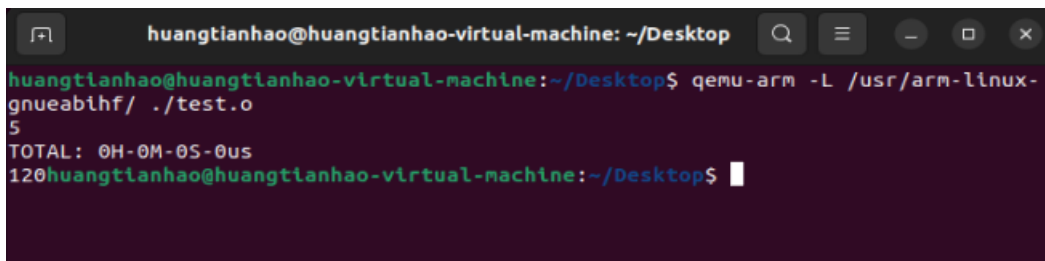
```

```

18 .L3:
19     ldr    r3, [r7, #8]
20     ldr    r2, [r7, #4]
21     mul    r3, r2, r3
22     str    r3, [r7, #8]
23     ldr    r3, [r7, #4]
24     adds   r3, r3, #1
25     str    r3, [r7, #4]
26 .L2:
27     ldr    r2, [r7, #4]
28     ldr    r3, [r7, #12]
29     cmp    r2, r3
30     ble    .L3
31     ldr    r0, [r7, #8]
32     bl     putint(PLT)
33     movs   r3, #0
34     mov    r0, r3
35     adds   r7, r7, #16
36     mov    sp, r7
37     pop    {r7}
38     pop    {fp, pc}
39     .size   main, .-main
40     .section        .note.GNU-stack,"",%progbits

```

使用 `arm-linux-gnueabi-gcc t.S sylib.c -o test.o` 指令联合编译 `sylib.c` 文件生成 `test` 可执行文件，运行结果正确，如下图



```

huangtianhao@huangtianhao-virtual-machine: ~/Desktop
huangtianhao@huangtianhao-virtual-machine:~/Desktop$ qemu-arm -L /usr/arm-linux-gnueabi/ ./test.o
5
TOTAL: 0H-0M-0S-0us
120huangtianhao@huangtianhao-virtual-machine:~/Desktop$

```

图 2: 求阶乘结果验证

(三) Bison 实现简单表达式翻译

对于思考问题，思考如何设计语法制导定义和翻译模式，实现简单的 SysY 程序到汇编程序的翻译，并通过 Bison 进行实验。

对于该问题的思考，我们需要根据设计的上下文无关文法结合相应的语法制导定义来实现将合法的 SysY 程序都能正确地翻译成汇编程序。因为真正设计的产生式集合以及非终结符集合中，集合的元素都不少，需要在学习了词法分析工具及 `lex` 编程后才能更加完善、更加简便地完成。所以本次实验，小组主要结合了 Bison 编程练习中的思考问题，实现了包含符号表的简单算术表达式的汇编语言翻译工作。

在实现包含符号表的简单表达式的翻译过程之中，使用的是能够调用 C++ 中的 `string` 库、`map` 库的 `.ypp` 格式文件，因为如果使用 `.y` 格式的文件，会找不到相应的库函数调用。对于符号

表的实现则是使用 map 类型的数据结构，保存字符串与数值的对应关系。

在对 expr 等非终结符的 yylval 返回值的定义与处理上，采用了自定义的 struct 结构来定义 yylval 的类型，在结构体变量中包含了对应的汇编代码以及自身的地址。

而在变量，常量以及临时变量的地址分配和表达之中：

1. 对于变量的地址表达，这里采用了类似于 80386 指令集中的伪指令形式，使用变量的字符串值来指示变量的地址。
2. 对于常量数值的表达形式，这里采用了形如 MOV EAX, 1D 的指令来表达对常量数值的运用，以简化汇编代码的复杂度（这里的'D' 是指十进制格式）。
3. 对于临时变量的存储和地址分配，本次实验中统一使用了从 0x1 地址进行编址，通过地址递增的方式为临时变量分配地址。

最后在翻译模式部分，也就是 Bison 中的产生式设计那里，设计对应语句的翻译动作，包含汇编代码的连接，部分汇编代码的补充以及地址的分配等内容。

具体的 Bison 代码如下：

Bison 实现代码

```

1  %{
2  /*****
3  test.ypp
4  YACC file
5  Date: 2022/10/08
6  zzekun <2014074@mail.nankai.edu.cn>
7  *****/
8  #include <iostream>
9  #include <map>
10 #include <string>
11 using namespace std;
12 #ifndef YYSTYPE
13 #define YYSTYPE Assemble
14 #endif
15
16 int yylex();
17 extern int yyparse();
18 FILE* yyin;
19 void yyerror(const char* s);
20 char idStr[50];
21 map<string, string>character_table;
22 int basic_addr = 0;
23
24 struct Assemble{
25     string addr = "";
26     string code = "";
27     int dval = 0;
28     string strval = "";
29 };
30 %}

```

```

31
32 /*
33 %union {
34     double dval;
35     string strval;
36     Assemble Assemval;
37 }
38 */
39
40 %token ID
41 %token NUMBER
42 %type expr
43 %type statement
44 %type statement_list
45 %token ASSIGN
46 %token ADD
47 %token SUB
48 %token MUL
49 %token DIV
50 %token LEFT_PRA
51 %token RIGHT_PRA
52
53
54 %right ASSIGN
55 %left ADD SUB
56 %left MUL DIV
57 %right UMINUS
58
59 %%
60
61 statement_list: statement ';'      { $$code = $1.code; } //cout<<$$code;
62               | statement_list statement ';'  { $$code = $1.code + $2.code;
63               } //cout<<$$code;
64
65 ;
66 statement:      ID ASSIGN expr { character_table[$1.addr] = "okk"; //若消除
        空行, 这里也应进行判断
67
68                                     $$code = $3.code + "\nMOV EAX, " + $3.addr
69                                     + "\nMOV " + $1.addr + ", EAX";
70                                     cout<<$$code; }
71
72 |      expr      { $$code = $1.code;
73                                     cout<<$$code; }
74
75 ;
76
77 expr : expr ADD expr
78 { basic_addr = basic_addr + 1; //这里也可以不用tempaddr
79   $$addr = "0x" + to_string(basic_addr); //可以在这里设置条件判断消除空行的
80   输出
81   if($1.code!=" " && $3.code!=" ")

```

```

75     $$code = $1code + '\n' + $3code + "\nMOV EAX, " + $1addr + "\nMOV EBX
    , " + $3addr + "\nADD EAX, EBX\n" + "MOV " + $$addr + ", EAX";
76 else if($1code==" && $3code=="")
77     $$code = "\nMOV EAX, " + $1addr + "\nMOV EBX, " + $3addr + "\nADD EAX,
    EBX\n" + "MOV " + $$addr + ", EAX";
78 else
79     $$code = $1code + $3code + "\nMOV EAX, " + $1addr + "\nMOV EBX, " +
    $3addr + "\nADD EAX, EBX\n" + "MOV " + $$addr + ", EAX"; }
80
81
82 | expr SUB expr
83 { basic_addr = basic_addr + 1;
84  $$addr = "0x" + to_string(basic_addr);
85  $$code = $1code + '\n' + $3code + "\nMOV EAX, " + $1addr + "\nMOV
    EBX, " + $3addr + "\nSUB EAX, EBX\n" + "MOV " + $$addr + ", EAX";
    }
86
87
88 | expr MUL expr
89 { basic_addr = basic_addr + 1;
90  $$addr = "0x" + to_string(basic_addr);
91  $$code = $1code + '\n' + $3code + "\nMOV EAX, " + $1addr + "\nMOV
    EBX, " + $3addr + "\nMUL EAX, EBX\n" + "MOV " + $$addr + ", EAX";
    }
92
93
94 | expr DIV expr
95 { basic_addr = basic_addr + 1;
96  $$addr = "0x" + to_string(basic_addr);
97  $$code = $1code + '\n' + $3code + "\nMOV EAX, " + $1addr + "\nMOV
    EBX, " + $3addr + "\nDIV EAX, EBX\n" + "MOV " + $$addr + ", EAX";
    }
98
99
100 | LEFT_PRA expr RIGHT_PRA { $$addr = $2addr;
101                               $$code = $2code; }
102
103
104 | SUB expr %prec UMINUS
105 { basic_addr = basic_addr + 1;
106  $$addr = "0x" + to_string(basic_addr);
107  $$code = $2code + "\nMOV EAX, " + $2addr + "NEG EAX\n" + "MOV " + $$
    .addr + ", EAX"; }
108
109
110 | NUMBER { $$addr = to_string($1.dval) + "D"; }
111
112

```

```

113 | ID
114 { $$ .addr = $1 .addr;
115   if (character_table[$1 .addr] == "")
116     $$ .code = "MOV EAX, " + to_string(0) + '\n' + "MOV " + $1 .addr + ",
        EAX";
117   else
118     ; }
119   ;
120
121 %%
122
123 // programs section
124
125
126 int yylex()
127 {
128   // place your token retrieving code here
129   int t;
130   while(1)
131   {
132     t = getchar();
133     if (t == ' ' || t == '\t' || t == '\n')
134       ;
135     else if (isdigit(t)){
136       yylval.dval = 0;
137       while (isdigit(t)){
138         yylval.dval = yylval.dval * 10 + t - '0';
139         t = getchar();
140       }
141       ungetc(t, stdin);
142       return NUMBER;
143     }
144     else if ((t >= 'a' && t <= 'z' ) || ( t >= 'A' && t <= 'Z' ) || ( t ==
        '_' )){
145       int i = 0;
146       while (( t >= 'a' && t <= 'z' ) || ( t >= 'A' && t <= 'Z' ) || ( t
        == '_' ) || ( t >= '0' && t <= '9' )){
147         idStr[i] = t;
148         t = getchar();
149         i++;
150       }
151       idStr[i] = '\0';
152       yylval.addr = idStr;
153       //cout<<yylval.addr<<endl;
154       ungetc(t, stdin);
155       //printf("%f\n", character_table[yylval.strval]);
156       return ID;
157     }

```



```
158     else{
159         switch (t)
160         {
161             case '+':
162                 return ADD;
163                 break;
164             case '-':
165                 return SUB;
166                 break;
167             case '*':
168                 return MUL;
169                 break;
170             case '/':
171                 return DIV;
172                 break;
173             case '(':
174                 return LEFT_PRA;
175                 break;
176             case ')':
177                 return RIGHT_PRA;
178                 break;
179             case '=':
180                 return ASSIGN;
181                 break;
182             default:
183                 return t;
184         }
185     }
186 }
187 }
188
189 int main(void)
190 {
191     yyin = stdin ;
192     do {
193         yyparse();
194     }
195     while (! feof (yyin));
196     return 0;
197 }
198 void yyerror(const char* s) {
199     fprintf (stderr , "Parse error : %s\n", s );
200     exit (1);
201 }
```

通过上述的 Bison 程序，我们就能够将包含变量的表达式成功地翻译成类似 x86 格式的汇编代码了，但是翻译得到的汇编代码还有不小的问题。虽然在寄存器的分配，以及运算的逻辑上，测试样例已经测试正确，但是该段代码却不能够直接使用汇编器生成可执行文件来进行验证，因

为我们知道，汇编代码中的伪指令需要提前定义，而且一个完整的汇编程序，需要设置运行环境和确定指令集，且需要定义各个段（如.text 段，.data 段等）。

接下来展示一个该 Bison 程序的运行实例：

简单的 stmt 翻译

```
1      a=1;
2      b=2;
3      c=(a+b)*2;
```

使用 Bison 翻译成汇编代码后：

stmt 翻译后的汇编代码

```
1      MOV EAX, 1D
2      MOV a, EAX
3      MOV EAX, 2D
4      MOV b, EAX
5      MOV EAX, a
6      MOV EBX, b
7      ADD EAX, EBX
8      MOV 0x1, EAX
9
10     MOV EAX, 0x1
11     MOV EBX, 2D
12     MUL EAX, EBX
13     MOV 0x2, EAX
14     MOV EAX, 0x2
15     MOV c, EAX
```

可证明汇编代码的逻辑是正确的。

三、 总结

本次实验确定了最终要实现的编译器应该支持的 SysY 语言特性，运用上下文无关文法（CFG）描述 SysY 语言子集。本次实验还对 ARM 汇编程序进行了学习，了解了 ARM 汇编语言的逻辑，编写了两个简单的 SysY 程序并使用 ARM 汇编语言将其复写。在 CFG 设计和 ARM 汇编编程的基础上，本小组还探究了如何将表达式翻译成汇编指令，成功实现了简单表达式到 x86 汇编语句的 Bison 翻译，为未来实现完整的编译器打下了基础。

参考文献

- [1] 预备工作 2——定义你的编译器 & 汇编编程, 杨保哲, 李煦阳, 2020 年 10 月, 费迪, 2021 年 9 月
- [2] 预备工作 3-熟悉语法分析器辅助构造工具, 潘宇, 2021 年 10 月
- [3] SysY2022 语言定义-V1
- [4] SysY2022 运行时库-V1