小组成员

周宇恒 55210916

孙子昱 21210320

苏浩洋 21210525

实验内容

小组经过讨论，选择参考北京大学编译实践在线文档(北京大学编译实践课程在线文档 | 北大编译实践在线文档 (pku-minic.github.io))，它指导我们由浅入深地进行编译器的设计，逐步对功能进行完善、丰富。该文档旨在指导学生实现一个可将 SysY 语言编译到 RISC-V汇编的编译器，其中SysY 语言是一种精简版的 C 语言, RISC-V 则是一种新兴且热门的指令系统 (ISA)。我们完成了文档Lv1-Lv8的功能，包含：一元表达式、算术表达式、比较和逻辑表达式、常量、变量和赋值、语句块和作用域、if语句、while语句以及函数和全局变量等，可以成功地由包含上述语句的源程序生成中间代码(Koopa IR形式)。

实现细节

词法分析

sysy.l文件负责词法分析，将源代码中的文本（例如，关键字、标识符、注释等）转换成词法单元（tokens），下面将介绍一下实现的部分细节：

空白符和注释

```
WhiteSpace    [ \t\n\r]*
LineComment   "//".*
BlockComment  "/*"([^*]*|(\*+[^/]))*"*/"
```

标识符

```
Identifier    [a-zA-Z_][a-zA-Z0-9_]*
```

整数字面量

```
Decimal       [1-9][0-9]*
Octal         0[0-7]*
Hexadecimal   0[xX][0-9a-fA-F]+
```

部分转token操作

```
{WhiteSpace}    { /* 忽略, 不做任何操作 */ }
{LineComment}   { /* 忽略, 不做任何操作 */ }
{BlockComment}   { /* 忽略, 不做任何操作 */ }
"int"           { return INT; }
"void"          { return VOID; }
"return"        { return RETURN; }
"const"         { return CONST; }
"if"            { return IF; }
"else"          { return ELSE; }
"while"         { return WHILE; }
"break"         { return BREAK; }
```

```
"continue"      { return CONTINUE; }
"<="            { return LE; }
">="            { return GE; }
"=="            { return EQ; }
"!="            { return NE; }
"&&"            { return AND; }
"||"            { return OR; }
.               { return yytext[0]; }
...
```

其中`.  { return yytext[0]; }`表示对未识别字符的处理，对于任何没有被之前所有特定规则匹配到的单个字符，直接将该字符的ASCII值作为词法单元返回。

语法分析

sysy.y文件负责语法分析的内容，语法规范如下：

```
CompUnit      ::= [CompUnit] (Decl | FuncDef);

Decl          ::= ConstDecl | VarDecl;
ConstDecl     ::= "const" BType ConstDef {"," ConstDef} ";";
BType         ::= "int";
ConstDef      ::= IDENT "=" ConstInitVal;
ConstInitVal  ::= ConstExp;
VarDecl       ::= BType VarDef {"," VarDef} ";";
VarDef        ::= IDENT | IDENT "=" InitVal;
InitVal       ::= Exp;

FuncDef       ::= FuncType IDENT "(" [FuncFParams] ")" Block;
FuncType      ::= "void" | "int";
FuncFParams   ::= FuncFParam {"," FuncFParam};
FuncFParam    ::= BType IDENT;

Block         ::= "{" {BlockItem} "}";
BlockItem     ::= Decl | Stmt;
Stmt          ::= LVal "=" Exp ";"
                | [Exp] ";"
                | Block
                | "if" "(" Exp ")" Stmt ["else" Stmt]
                | "while" "(" Exp ")" Stmt
                | "break" ";"
                | "continue" ";"
                | "return" [Exp] ";";

Exp           ::= LOrExp;
LVal          ::= IDENT;
PrimaryExp    ::= "(" Exp ")" | LVal | Number;
Number        ::= INT_CONST;
UnaryExp      ::= PrimaryExp | IDENT "(" [FuncRParams] ")" | UnaryOp UnaryExp;
UnaryOp       ::= "+" | "-" | "!";
FuncRParams   ::= Exp {"," Exp};
MulExp        ::= UnaryExp | MulExp ("*" | "/" | "%") UnaryExp;
AddExp        ::= MulExp | AddExp ("+" | "-") MulExp;
```

```
RelExp        ::= AddExp | RelExp ("<" | ">" | "<=" | ">=") AddExp;
EqExp         ::= RelExp | EqExp ("==" | "!=") RelExp;
LAndExp       ::= EqExp | LAndExp "&&" EqExp;
LOrExp        ::= LAndExp | LOrExp "||" LAndExp;
ConstExp      ::= Exp;
```

重难点

在实现的过程中，我们发现北大的语法树设计中也有一些不清楚不完善的地方，于是我们在实现上述规则的基础上也对语法树的结构进行设计。

1.Btype和FuncType

北大的文档中使用了`Btype`和`FuncType`两种类型，具体规则如下：

```
BType ::= "int";
FuncType ::= "void" | "int";
```

在实际运行时会出现归约归约冲突，因为它们都含有"int"，所以我们通过如下规则消除冲突：

```
BType:
    INT
    {
        string *type = new string("int");
        $$ = type;
    }
    ;
VoidType:
    VOID
    {
        string *type = new string("void");
        $$ = type;
    }
```

2.Stmt

文档中的Stmt规则如下所示：

```
Stmt ::= LVal "=" Exp ";"
    | [Exp] ";"
    | Block
    | "if" "(" Exp ")" Stmt ["else" Stmt]
    | "while" "(" Exp ")" Stmt
    | "break" ";"
    | "continue" ";"
    | "return" [Exp] ";";
```

然而，如果程序中包含嵌套的if/else语句，该规则将会出现二义性，我们通过实现`Stmt`,`ClosedStmt`,`OpenStmt`来消除文法的二义性：

```
Stmt:
```

```
    OpenStmt
    {
        auto stmt = ($1);
        $$ = stmt;
    }|

    ClosedStmt
    {
        auto stmt = ($1);
        $$ = stmt;
    }
    ;

ClosedStmt:

    ////////////////////////////////////////////////////////////////////////
    RETURN Exp ';'
    {
        auto stmt = manager.create_StmtAST();
        stmt->type = "ret";
        stmt->block_exp = (BaseAST*)($2);
        $$ = stmt;
    }|

    RETURN ';'
    {
        auto stmt = manager.create_StmtAST();
        stmt->type = "ret";
        stmt->block_exp = nullptr;
        $$ = stmt;
    }|

    LVal '=' Exp ';'
    {
        auto stmt = manager.create_StmtAST();
        stmt->type = "lval";
        stmt->lval = *($1);
        delete $1;
        stmt->block_exp = (BaseAST*)($3);
        $$ = stmt;
    }|

    Block
    {
        auto stmt = manager.create_StmtAST();
        stmt->type = "block";
        stmt->block_exp = (BaseAST*)($1);
        $$ = stmt;
    }|

    Exp ';'
    {
        auto stmt = manager.create_StmtAST();
        stmt->type = "exp";
        stmt->block_exp = (BaseAST*)($1);
        $$ = stmt;
```

```
    }|

    ';'
    {
        auto stmt = manager.create_StmtAST();
        stmt->type = "exp";
        stmt->block_exp = nullptr;
        $$ = stmt;
    }|

    BREAK ';'
    {
        auto stmt = manager.create_StmtAST();
        stmt->type = "break_";
        $$ = stmt;
    }|

    CONTINUE ';'
    {
        auto stmt = manager.create_StmtAST();
        stmt->type = "continue_";
        $$ = stmt;
    }|

    ////////////////////////////////////////////////////////////////////////

    IF '(' Exp ')' ClosedStmt ELSE ClosedStmt
    {
        auto stmt = manager.create_StmtAST();
        stmt->type = "ifelse";
        stmt->exp_simple = (BaseAST*)($3);
        stmt->if_stmt = (BaseAST*)($5);
        stmt->else_stmt = (BaseAST*)($7);
        $$ = stmt;
    }|

    WHILE '(' Exp ')' ClosedStmt
    {
        auto stmt = manager.create_StmtAST();
        stmt->type = "while_";
        stmt->exp_simple = (BaseAST*)($3);
        stmt->while_stmt = (BaseAST*)($5);
        $$ = stmt;
    }
    ;

OpenStmt:
    IF '(' Exp ')' Stmt
    {
        auto stmt = manager.create_StmtAST();
        stmt->type = "if_";
        stmt->exp_simple = (BaseAST*)($3);
        stmt->if_stmt = (BaseAST*)($5);
        $$ = stmt;
    }|
```

```
    IF '(' Exp ')' ClosedStmt ELSE OpenStmt
    {
        auto stmt = manager.create_StmtAST();
        stmt->type = "ifelse";
        stmt->exp_simple = (BaseAST*)($3);
        stmt->if_stmt = (BaseAST*)($5);
        stmt->else_stmt = (BaseAST*)($7);
        $$ = stmt;
    }|

    WHILE '(' Exp ')' OpenStmt
    {
        auto stmt = manager.create_StmtAST();
        stmt->type = "while_";
        stmt->exp_simple = (BaseAST*)($3);
        stmt->while_stmt = (BaseAST*)($5);
        $$ = stmt;
    }
    ;
```

3.[]和{}

对于文档中的如下规则，`[]`表示0次或多次，`{}`表示1次或多次

```
ConstDecl     ::= "const" BType ConstDef {"," ConstDef} ";";
FuncDef       ::= FuncType IDENT "(" [FuncFParams] ")" Block;
```

我们通过如下规则，通过左递归的方法进行实现：

```
//ConstDecl     ::= "const" BType ConstDef {"," ConstDef} ";";

ConstDecl:
    CONST BType ConstDefList ';'
    {
        auto const_decl = manager.create_ConstDeclAST();
        const_decl->b_type = *($2);
        delete $2;
        vector<BaseAST*> *v_ptr = ($3);
        for (auto it = v_ptr->begin(); it != v_ptr->end(); it++)
            const_decl->const_def_list.push_back(*it);
        $$ = const_decl;
    }
    ;
ConstDefList:
    ConstDef
    {
        vector<BaseAST*> *v = manager.create_vector();
        v->push_back((BaseAST*)($1));
        $$ = v;
    }|

    ConstDefList ',' ConstDef
```

```
    {
        vector<BaseAST*> *v = ($1);
        v->push_back((BaseAST*)($3));
        $$ = v;
    }
    ;

//FuncDef       ::= FuncType IDENT "(" [FuncFParams] ")" Block;

FuncDef:
    VoidType IDENT '(' ')' Block   //空参数
    {
        auto func_def = manager.create_FuncDefAST();
        func_def->func_type = *($1);
        delete $1;
        func_def->ident = *($2);
        delete $2;
        func_def->block = (BaseAST*)($5);
        $$ = func_def;
    }|

    VoidType IDENT '(' FuncFParams ')' Block
    {
        auto func_def = manager.create_FuncDefAST();
        func_def->func_type = *($1);
        delete $1;
        func_def->ident = *($2);
        delete $2;
        vector<BaseAST*> *v_ptr = ($4);
        for (auto it = v_ptr->begin(); it != v_ptr->end(); it++)
            func_def->params.push_back(*it);
        func_def->block = (BaseAST*)($6);
        ((BlockAST*)(func_def->block))->func = func_def->ident;
        $$ = func_def;
    }|

    BType IDENT '(' ')' Block
    {
        auto func_def = manager.create_FuncDefAST();
        func_def->func_type = *($1);
        delete $1;
        func_def->ident = *($2);
        delete $2;
        func_def->block = (BaseAST*)($5);
        $$ = func_def;
    }|

    BType IDENT '(' FuncFParams ')' Block
    {
        auto func_def = manager.create_FuncDefAST();
        func_def->func_type = *($1);
        delete $1;
        func_def->ident = *($2);
        delete $2;
        vector<BaseAST*> *v_ptr = ($4);
        for (auto it = v_ptr->begin(); it != v_ptr->end(); it++)
```

```
            func_def->params.push_back(*it);
        func_def->block = (BaseAST*)($6);
        ((BlockAST*)(func_def->block))->func = func_def->ident;
        $$ = func_def;
    }
    ;
    ;

FuncFParams:
    FuncFParam
    {
        vector<BaseAST*> *v = manager.create_vector();
        v->push_back((BaseAST*)($1));
        $$ = v;
    }|

    FuncFParams ',' FuncFParam
    {
        vector<BaseAST*> *v = ($1);
        v->push_back((BaseAST*)($3));
        $$ = v;
    }
    ;

FuncFParam:
    BType IDENT
    {
        auto param = manager.create_FuncFParamAST();
        param->b_type = *($1);
        delete $1;
        param->ident = *($2);
        delete $2;
        $$ = param;
    }
    ;
```

除此之外，我们以下面的规则为例，详细解释我们如何实现语法分析。

```
Start:
    CompUnit
    {
        manager.root = (BaseAST*)($1);
    }
    ;

CompUnit:
    FuncDef
    {
        auto comp_unit = manager.create_CompUnitAST();
        auto func_def = $1;
        comp_unit->func_def_list.push_back(func_def);
        $$ = comp_unit;
```

```
    }|

    Decl
    {
        auto comp_unit = manager.create_CompUnitAST();
        auto decl = (BaseAST*)($1);
        comp_unit->decl_list.push_back(decl);
        $$ = comp_unit;
    }|

    CompUnit FuncDef
    {
        auto comp_unit = (CompUnitAST*)($1);
        auto func_def = (BaseAST*)($2);
        comp_unit->func_def_list.push_back(func_def);
        $$ = comp_unit;
    }|

    CompUnit Decl
    {
        auto comp_unit = (CompUnitAST*)($1);
        auto decl = (BaseAST*)($2);
        comp_unit->decl_list.push_back(decl);
        $$ = comp_unit;
    }
    ;
```

- `start`规则是解析器的起始规则。它定义了解析的入口点。当 Bison 解析到 `CompUnit` 时，将解析得到的 `CompUnit` AST 节点赋值给 `manager.root`，这样，整个程序的根节点就被设置为这个编译单元。`manager` 是用于管理和维护AST节点的类实例
- `CompUnit` 规则
  - 一个编译单元可以直接从一个函数定义开始。这里，当Bison匹配到`FuncDef`语法模式时，规则表示一个编译单元可以是一个函数定义（`FuncDef`）。当解析器匹配到一个函数定义时，它会创建一个新的编译单元AST节点（`CompUnitAST`），然后将函数定义添加到该节点的函数定义列表中（`func_def_list`）。最后，将这个新的编译单元节点返回（`$$`）
  - 一个编译单元可以是一个声明（`Decl`），如变量声明或常量声明。类似地，当解析器匹配到一个声明时，它会创建一个新的编译单元AST节点，将声明添加到该节点的声明列表中（`decl_list`），并返回这个新的编译单元节点
  - 一个编译单元可以由另一个编译单元（`CompUnit`）和一个函数定义（`FuncDef`）组成。当解析器匹配到这两个部分时，它会获取已经存在的编译单元AST节点（`$1`），将新的函数定义（`$2`）添加到该节点的函数定义列表中，并返回更新后的编译单元节点
  - 一个编译单元可以由另一个编译单元（`CompUnit`）和一个声明（`Decl`）组成。类似地，解析器会获取已经存在的编译单元AST节点，将新的声明添加到该节点的声明列表中，并返回更新后的编译单元节点

总而言之，我们定义了如下规则，其余不再详述。

规则名称
Start

规则名称
CompUnit
FuncDef
FuncFParams
FuncFParam
FuncRParams
Block
Stmt
OpenStmt
ClosedStmt
Exp
LOrExp
LAndExp
EqExp
RelExp
AddExp
MulExp
UnaryExp
PrimaryExp
Decl
ConstDecl
ConstDef
ConstInitVal
BlockItem
ConstExp
VarDecl
VarDef
InitVal
BlockItemList
ConstDefList
VarDefList

规则名称
Number
LVal
BType
VoidType
UNARYOP

语义分析和中间代码生成

Nodes.h和NodesManager.h是我们实现的编译器的核心部分，用于构建抽象语法树进行语义分析和中间代码生成。

Nodes.h

Nodes.h定义了一些通用函数和变量：

- `StrToInt(const string& str)`：将字符串转换为整数，支持正负号
- `static` 变量（`count_temp`，`count_ifelse`，`count_while` 等）用于生成唯一的标签和临时变量名，帮助生成中间代码

Nodes.h也定义了一些类，其中

- `MyVar` 类管理不同类型的变量，如整数和字符串，提供方法来获取和设置变量的值
- `BaseAST` 类为所有AST节点的基类，定义了输出中间表示（IR）和计算表达式值的虚拟方法
- `CompUnitAST` 类代表整个程序的顶层结构，包含函数和全局声明的列表，`outputIR()` 方法生成顶层声明和函数定义的IR代码
- `DeclAST` 和相关类（`ConstDeclAST`，`VarDeclAST`，`ConstDefAST`，等)管理声明和定义，每个类都有方法来生成相应的IR代码和计算常量表达式的值
- 表达式类（`ExpAST`，`LOrExpAST`，`LAndExpAST`，等)管理各种表达式，包括逻辑、关系和算术表达式，每个类处理特定的操作符和表达式类型，并能生成相应的IR代码
- `BlockAST` 和 `BlockItemAST` 管理代码块和代码块中的条目（声明或语句），负责局部作用域的符号表管理
- `FuncDefAST` 类管理函数定义，包括参数、返回类型和函数体，生成函数的IR代码，包括处理参数和管理局部作用域
- `StmtAST` 和相关子类处理各种语句类型，如条件语句、循环语句和返回语句，生成控制流语句的IR代码

除此之外，我们使用 `vector` 和 `map` 来管理变量和函数的作用域和定义。

NodesManager.h

NodesManager.h提供了`NodesManager` 类，用于管理和创建各种类型的 AST（抽象语法树）节点。成员变量中提供了指向整个AST的根节点的指针(代表整个程序的顶层结构)，存储所有创建的AST节点的向量以及存储AST节点向量的集合。类方法中则提供了各种节点创建方法以及向量创建方法，具体见代码，不再详述。

实现功能

在该部分，我们将详细介绍我们支持的各类语句，并提供实例。

表达式

- 一元表达式：`+`, `-` 和 `!` 运算, 同时支持括号表达式(注意这里的`+`和`-`值的是取正和取负而非加减)
- 算术表达式：加减乘除取模，`+`，`-`，`*`，`/`，`%`
- 比较和逻辑表达式：`<`,`>`,`<=`,`>=`,`==`,`!=`,`&&`,`||`

```
+(- -!6)          //一元表达式
1 + 2 * -3        //算术表达式
1 <= 2            //比较和逻辑表达式
```

常量和变量

- 在表达式的基础上，额外处理常量/变量定义和赋值语句

```
const int x = 233 * 4;    //常量
int y = 10;               //变量和赋值
y = y + x / 2;
```

语句块和作用域

- 在表达式、常量和变量的基础上，额外处理语句块和作用域

```
int a = 1, b = 2;
{
    int a = 2;
    b = b + a;
}
```

if语句

- 在表达式、常量和变量、语句块和作用域的基础上，额外处理if/else语句

```
int a = 1;
int y;
if (a == 2 || a == 3) {
    y = 1;
} else {
    y = 2;
}
```

while语句

- 在表达式、常量和变量、语句块和作用域、if/else语句的基础上，额外处理while语句

```
int i = 0, pow = 1;
while (i < 7) {
    pow = pow * 2;
    i = i + 1;
}
```

函数和全局变量

- 在表达式、常量和变量、语句块和作用域、if/else语句、while语句的基础上，额外处理函数和全局变量

```
int var;

int func(int x) {
  var = var + x;
  return var;
}

int main() {
  var = var * 10;
  return func(var);
}
```

运行结果展示

我们将如下源代码作为输入进行测试：

```
int var;

int func(int x) {
    var = var + x;
    return var;
}

int main() {
    // 注释
    /*我是注释块*/
    const int x = 233 * 4;
    int a = 1;
    int b = 1 + 2 * -3, c = 3;
    {
        int a = 0;
        a = a + b / 2;
    }
    if (a == 2 || a >= 4) {
        return 0;
    } else {
        b = a + 1;
    }
    int i = 0, pow = 1;
    while (i < 7) {
        pow = pow * 2;
```

```
        i = i + 1;
        if (i > 3){
            break;
        }
    }
    a + 3;
    putint(func(1));
    var = var * 10;
    putint(func(var));
    putch(10);
    return +(- -!6); // 注释
}
```

运行结果：

中间代码：

```
decl @getint(): i32
decl @getch(): i32
decl @getarray(*i32): i32
decl @putint(i32)
decl @putch(i32)
decl @putarray(i32, *i32)
decl @starttime()
decl @stoptime()

global @var_0 = alloc i32, zeroinit

fun @func(@x_0: i32): i32 {
%entry_func:
    %x_0 = alloc i32
    store @x_0, %x_0
    %0 = load @var_0
    %1 = load %x_0
    %2 = add %0, %1
    store %2, @var_0
    %3 = load @var_0
    ret %3
}

fun @main(): i32 {
%entry_main:
    @a_0 = alloc i32
    store 1, @a_0
    @b_0 = alloc i32
    %4 = sub 0, 3
    %5 = mul 2, %4
    %6 = add 1, %5
    store %6, @b_0
    @c_0 = alloc i32
    store 3, @c_0
    @a_1 = alloc i32
    store 0, @a_1
    %7 = load @a_1
    %8 = load @b_0
```

```
    %9 = div %8, 2
    %10 = add %7, %9
    store %10, @a_1
    %11 = load @a_0
    %12 = eq %11, 2
    %13 = alloc i32
    br %12, %then_0, %else_0
%then_0:
    store 1, %13
    jump %end_0
%else_0:
    %15 = load @a_0
    %16 = ge %15, 4
    %14 = ne %16, 0
    store %14, %13
    jump %end_0
%end_0:
    %17 = load %13
    br %17, %then_1, %else_1
%then_1:
    ret 0
%else_1:
    %18 = load @a_0
    %19 = add %18, 1
    store %19, @b_0
    jump %end_1
%end_1:
    @i_0 = alloc i32
    store 0, @i_0
    @pow_0 = alloc i32
    store 1, @pow_0
    jump %while_0
%while_0:
    %20 = load @i_0
    %21 = lt %20, 7
    br %21, %do_0, %while_end_0
%do_0:
    %22 = load @pow_0
    %23 = mul %22, 2
    store %23, @pow_0
    %24 = load @i_0
    %25 = add %24, 1
    store %25, @i_0
    %26 = load @i_0
    %27 = gt %26, 3
    br %27, %then_2, %end_2
%then_2:
    jump %while_end_0
%end_2:
    jump %while_0
%while_end_0:
    %28 = load @a_0
    %29 = add %28, 3
    %30 = call @func(1)
    call @putint(%30)
    %31 = load @var_0
```

```
    %32 = mul %31, 10
    store %32, @var_0
    %33 = load @var_0
    %34 = call @func(%33)
    call @putint(%34)
    call @putch(10)
    %35 = eq 6, 0
    %36 = sub 0, %35
    %37 = sub 0, %36
    ret %37
}
```

目标代码(通过调库生成)

```
    .data
    .globl var_0
var_0:
    .zero 4

    .text
    .globl func
func:
    addi  sp, sp, -32
entry_func:
    sw    a0, 0(sp)
    la    t0, var_0
    lw    t0, 0(t0)
    lw    t1, 0(sp)
    add   t2, t0, t1
    la    s11, var_0
    sw    t2, 0(s11)
    la    t3, var_0
    lw    t3, 0(t3)
    mv    a0, t3
    addi  sp, sp, 32
    ret

    .text
    .globl main
main:
    addi  sp, sp, -176
    sw    ra, 172(sp)
entry_main:
    li    t0, 1
    sw    t0, 0(sp)
    li    t0, 3
    sub   t1, x0, t0
    li    t0, 2
    mul   t0, t0, t1
    li    t2, 1
    add   t2, t2, t0
    sw    t2, 4(sp)
    li    t3, 3
    sw    t3, 8(sp)
    sw    x0, 12(sp)
```

```
    lw    t3, 12(sp)
    lw    t4, 4(sp)
    li    t5, 2
    div   t6, t4, t5
    add   t5, t3, t6
    sw    t5, 12(sp)
    lw    t3, 0(sp)
    li    a0, 2
    xor   a1, t3, a0
    seqz  a1, a1
    bnez  a1, then__0
    j     else__0
then__0:
    li    t0, 1
    sw    t0, 16(sp)
    j     end__0
else__0:
    lw    t0, 0(sp)
    li    t1, 4
    slt   t2, t0, t1
    xori  t2, t2, 1
    snez  t1, t2
    sw    t1, 16(sp)
    j     end__0
end__0:
    lw    t0, 16(sp)
    bnez  t0, then__1
    j     else__1
then__1:
    mv    a0, x0
    lw    ra, 172(sp)
    addi  sp, sp, 176
    ret
else__1:
    lw    t0, 0(sp)
    li    t1, 1
    add   t2, t0, t1
    sw    t2, 4(sp)
    j     end__1
end__1:
    sw    x0, 56(sp)
    li    t0, 1
    sw    t0, 60(sp)
    j     while__0
while__0:
    lw    t0, 56(sp)
    li    t1, 7
    slt   t2, t0, t1
    bnez  t2, do__0
    j     while_end__0
do__0:
    lw    t0, 60(sp)
    li    t1, 2
    mul   t2, t0, t1
    sw    t2, 60(sp)
    lw    t0, 56(sp)
```

```
    li    t1, 1
    add   t3, t0, t1
    sw    t3, 56(sp)
    lw    t0, 56(sp)
    li    t1, 3
    sgt   t4, t0, t1
    bnez  t4, then__2
    j     end__2
while_end__0:
    lw    t0, 0(sp)
    li    t1, 3
    add   t2, t0, t1
    sw    t2, 80(sp)
    li    t0, 1
    mv    a0, t0
    call  func
    sw    a0, 84(sp)
    lw    t0, 84(sp)
    mv    a0, t0
    call  putint
    la    t0, var_0
    lw    t0, 0(t0)
    li    t1, 10
    mul   t2, t0, t1
    la    s11, var_0
    sw    t2, 0(s11)
    la    t1, var_0
    lw    t1, 0(t1)
    sw    t0, 88(sp)
    sw    t1, 92(sp)
    sw    t2, 96(sp)
    lw    t0, 92(sp)
    mv    a0, t0
    call  func
    sw    a0, 100(sp)
    lw    t0, 100(sp)
    mv    a0, t0
    call  putint
    li    t0, 10
    mv    a0, t0
    call  putch
    li    t0, 6
    seqz  t0, t0
    sub   t1, x0, t0
    sub   t2, x0, t1
    mv    a0, t2
    lw    ra, 172(sp)
    addi  sp, sp, 176
    ret
then__2:
    j     while_end__0
end__2:
    j     while__0
```

后续改进

由于时间的限制，我们虽然对Lv9中的数组功能以及从Koopa IR生成目标代码进行了尝试，然而最后并没有成功。故在最终版本中，我们的测试程序中并不包含数组，同时Koopa IR到目标代码的部分通过调用现有的库来实现。