目录

0 运行环境与工程结构

本实验主要在 Windows 环境下使用 CLion IDE 完成。使用的语言为 C++，标准为 C++ 11。编译器版本为 MinGW 5.0，项目构建工具为 CMake 3.10.3。

语法分析和语义分析用到了 Linux 环境中的 Flex 2.6.0 和 Bison 3.0.4。

本工程的**主要结构**如下：

- lex.l：Flex 源程序
- yacc.y：Bison 源程序
- lex.yy.cpp, yacc.cpp, yacc.hpp：Flex 和 Bison 生成的代码
- Preprocesser.h, Preprocessor.cpp：源程序预处理，去除单行注释
- ASTNode.h, ASTNode.cpp：抽象语法树定义、语法分析、IR 生成
- SymbolTable.h, SymbolTable.cpp：符号表定义
- RegManager.h, RegManager.cpp：寄存器分配
- TargetCodeGenerator.h, TargetCodeGenerator.cpp：目标代码生成
- main.cpp：主程序入口

编译运行本工程的方法如下：

- flex lex.l
- bison -d yacc.y -o yacc.cpp
- 将 lex.yy.c 改为 lex.yy.cpp
- 在 yacc.hpp 开头添加一行“#include “ASTNode.h””，否则无法通过编译
- MyCompiler.exe 需要一个命令行参数，该参数为源文件路径

1 语法定义

本实验实现的语法为 C 语法的子集。

目前支持的数据类型如下：

1. void
2. int
3. bool
4. double （仅支持到 IR）
5. 一维数组

支持的主要语法包括：

1. 函数声明、定义
2. 全局变量的定义、初始化（支持到 IR）
3. 局部变量的定义、初始化，一维数组支持使用初始化列表初始化
4. 以“//”开头的行注释
5. 语句块和局部作用域
6. 流程控制语句，包括：
    a) if
    b) if-else
    c) while
    d) do-while
    e) for：支持在初始化语句中声明本地变量，如 for (int i = 0; i < 3; i++)
7. 常用表达式，包括
    a) 算术运算符：+, -, *, /, %
    b) 关系运算符：==, !=, >, <, >=, <=
    c) 逻辑运算符：&&, ||, !
    d) 位运算符：&, |, ^, ~, <<, >>
    e) 赋值运算符：=, +=, -=, *=, /=, %=
    f) 下表访问运算发：[]
    g) 自增自减运算符：++(前缀或后缀), --(前缀或后缀)
    h) 函数调用运算符：()

2 词法分析

词法分析部分，我们整体使用 flex 来进行正则匹配与传递终结符。C 语言的词法种类可以分为五大类，C 保留关键字、标识符、运算符、字面量以及注释。

C 保留关键字包含有基本类型以及 if、else 等结构上的关键字。对于这些关键字的处理，我们直接采用字符串去完全匹配，并对每个匹配成功的符号，返回对应的 symbol 给语法分析阶段使用。

C 语言的标识符要求以下划线或字母开头，之后可接下划线、字母或者数字。对此，我们首先将下划线和所有字母归为一类 L，然后用匹配以 L 开头，后接 L 或者数字的字符串，匹配成功后，我们将匹配到的字符串作为 symbol 的语义值，与标识符对应的 symbol 一起返回给语法分析器。

C 的运算符我们实现了三目运算符以外的所有运算符，同时也包括了分号。对于运算符，由于对应的字符串也是固定的，我们也只要用确定的字符串去匹配就可以得到这些符号。匹配成功后，我们就返回对应的 symbol 给语法分析器。

C 语言的字面量包括整型字面量、浮点字面量、布尔字面量、字符字面量以及字符串字面量。对于整型字面量，我们分为十进制、二进制、八进制、十六进制三类去考虑。对于十进制，其包括 0 以及以 1-9 开头后接 0-9 的字符串，我们使用 0|[1-9]{D}*的正则表达式来匹配十进制整数，并将匹配到的字符串作为 symbol 语义值。对于二进制，我们使用 0[Bb][01]+的正则表达式去匹配，并将匹配到的字符串作为 symbol 的语义值。对于八进制，我们使用 0[0-7]+的正则表达式去匹配以 0 开头后接 0-7 数字的八进制整数，并将匹配到的字符作为 symbol 语义值。对于十六进制整数，我们使用 0[Xx]{H}+的正则表达式去匹配以 0x 或者 0X 开头，后接 0-9 或者 a-f 或者 A-F 的字符串，同时将匹配道德字符作为 symbol 语义值。对于浮点数字面量，我们考虑无小数点的科学记数法(eg. 12e8, 3e-2)，可以无整数部分的浮点数(eg. .8, .03e-3)，以及可以无小数部分的浮点数(eg. 12.E8, 2.e-5)这三种情况，采用对应的正则表达式去匹配，并将匹配到的字符串作为 symbol 的语义值。对于布尔字面量，我们可以很简单的使用“true”和“false”去直接匹配，并用对于字符串作为 symbol 语义值。对于字符字面量，我们可以使用以单引号开头后接一个任意字符，再以单引号结尾的正则表达式去匹配。对于字符串字面量，我们可以使用双引号开头，后接非换行符的任意字符，包括被反斜杠转义的双引号，最后以双引号结尾的字符串对于的正则表达式去匹配，并将该字符串作为 symbol 的语义值返回。

C 语言的注释包含行注释以及块注释。对于前者，我们直接匹配//及之后的一行，对于后者，我们先去匹配/*，之后在匹配到*/之前，我们都直接忽略字符。

剩下的还包括换行符、空格符、制表符等，我们可以直接将其读入并不返回任何 symbol。
除了上述匹配之外，我们还增加了 column 这一变量来统计当前已经读入至当前行的哪一列，用来提供给语法分析器中的错误处理提供信息。

3 语法分析

语法分析部分，我们采用的是 bison 来进行处理。出于为后续处理的方便，我们在语法分析阶段，直接为各个语法树的节点分配特定的子类节点，用于标识其类别。

语法的种类可以基本分为类型、表达式、定义、语句这四大类。

对于类型，我们统一使用非终结符 type_specifier，并根据从词法分析器返回的类型 symbol 进行标注。

对于表达式，我们用总的非终结符 expression 来表示。同时表达式下可以特化成双目表达式，单目表达式，函数调用表达式，字面量表达式，变量表达式共 5 类，分别使用不同的非终结符来表示。对于双目表达式，我们使用 expression op expression 的统一形式来表示，而 op 的优先级我们在说明阶段进行指定。对于单目表达

式，我们使用 op expression 或者 expression op 的形式来分解。上述的 op 我们根据词法分析器返回的运算符 symbol 来进行区分。对于函数调用表达式，我们使用标识符、左括号、参数列表、右括号的形式进行分解，其中参数列表我们继续分解成 expression 的合集。对于字面量表达式，我们也可以很直接地根据词法分析器返回的语义值来创建对应的节点。变量表达式，我们也直接分解成标识符作为终结符。

对于定义，可以分为简单定义，也就是标识符本身，数组定义、函数定义、带初始值的定义 4 类。对于简单定义，我们直接分解成标识符作为终结符。对于数组定义，我们分解成标识符和以中括号包围的各个维度的容量，对于容量，我们使用 expression 来表示。对于函数定义，我们分解成标识符和以括号包围的参数列表，对于参数列表，我们用类型-标识符的键值对集合来表示。对于带初始值的定义，我们又可以分成对简单变量的初始化和对数组变量的初始化。对于简单变量的初始化，我们分解成简单定义 ＝expression 的形式；对于数组变量的初始化，我们分解成数组定义和初始化列表的形式，其中初始化列表我们以大括号包围的各个 expression 表示。

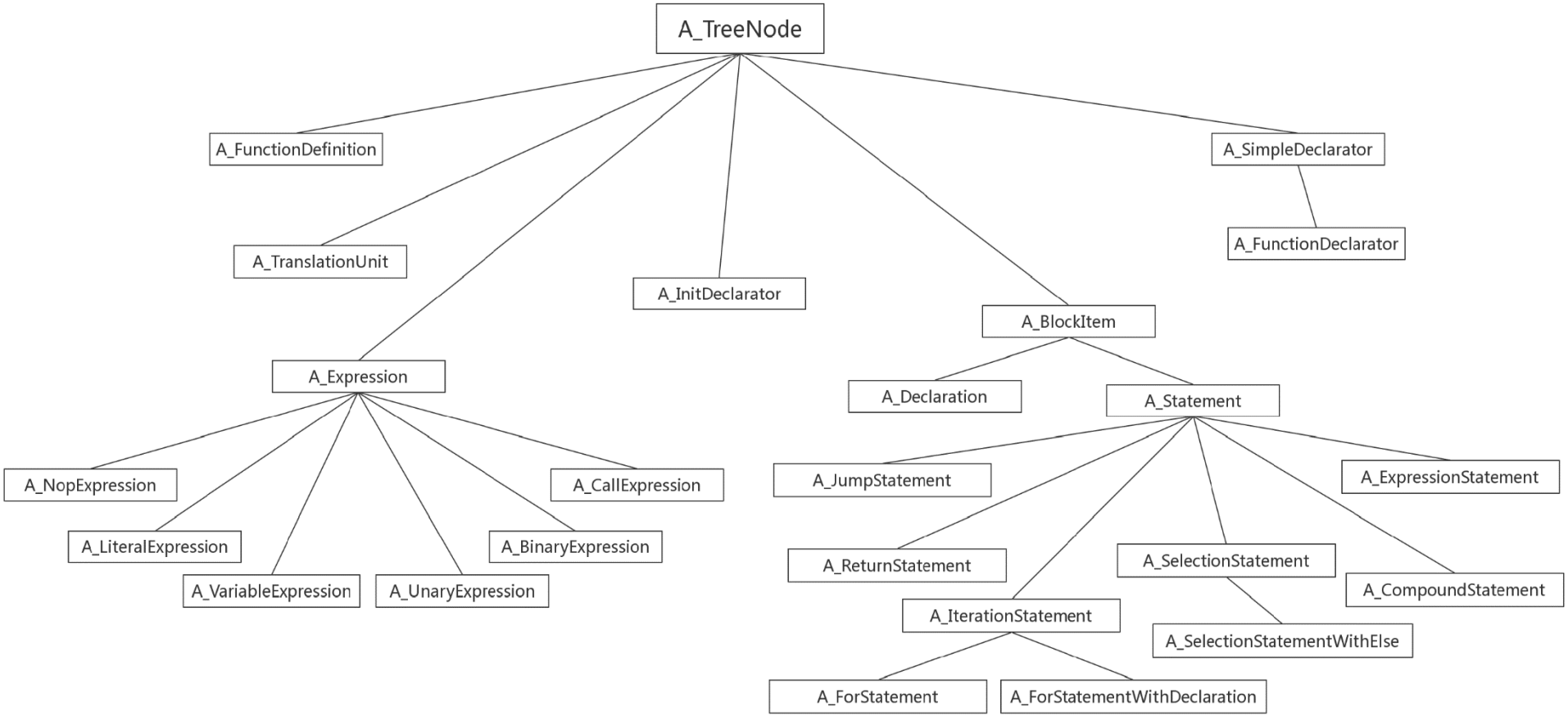
对于语句，我们可以分为声明，表达式语句，复合语句，选择语句，循环语句，跳转语句等。对于声明，我们分解成类型加上定义列表的形式，其中定义列表为多个定义的集合。对于表达式语句，我们直接分解成 expression 加分号，或者单独的分号。对于复合语句，我们分解成大括号包围的块列表，其中块列表我们分解成语句或者定义的集合。对于选择语句，我们使用 IF (expression) 语句 或者 IF (expression) 语句 ELSE 语句的形式。对于循环语句，我们分成 while 语句和 for 语句。其中 while 语句我们可以分解成 WHILE (expression) 语句 或者 DO 语句 WHILE (expression) 的形式；for 语句我们可以分解成 FOR (表达式语句 表达式语句 表达式) 语句 或者 FOR (定义 表达式语句 表达式) 语句 的形式。对于跳转语句，我们只考虑 break、continue、和 return 三种情况，其中 break 和 continue 我们直接作为一个节点，而 return 语句我们继续分解成 return 加 expression 的形式。

最后从总体角度，我们把整个程序作为一个可翻译单元，translation_unit。对于 translation_unit，我们可以分解成多个外部定义的集合，其中每个外部定义可以是函数定义或者声明。

4 抽象语法树

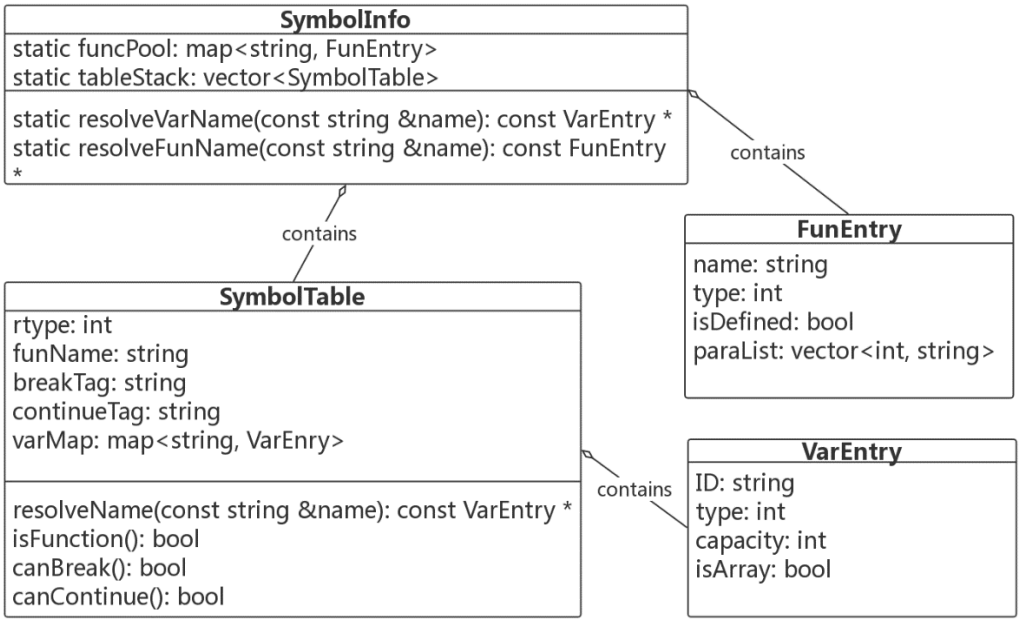
我们利用 C++的**多态**特性建立抽象语法树。下图为类的**继承关系图**。所有的节点都继承自 A_TreeNode 类。

抽象语法树主要用于语义分析和 IR 生成。A_TreeNode 类中定义了两个纯虚函数 parseRecursive 和 generateIRRecursive，分别负责语义分析和 IR 生成。其余的节点均实现了自己版本的这两个函数。这样一来，在进行语义分析和 IR 生成时，只需要不断的递归调用子节点的这两个函数，通过 C++的动态绑定，所有的 AST 节点都会被正确的处理。结构清晰、实现简洁是这种基于多态的 AST 的优势。



5 符号表

符号表定义了如下几个数据结构：



1. VarEntry

VarEntry 用来描述变量的声明信息。type 为变量的类型，capacity 为数组的大小，isArray 描述该变量是否是数组。最后，ID 为此变量对应的 IR 变量名，在语义分析时确定。

2. FunEntry

FunEntry 描述函数的声明信息。name 为函数名，type 为返回值类型，isDefined 描述此函数是否已经被定义。最后，paraList 描述函数的形参列表，包括每个参数的类型和名字。

3. SymbolTable

rtype 和 funName 仅对函数体的符号表有效，前者为函数的返回值类型，后者为函数的名字。breakTag 和 continueTag 为当前作用域 break 和 continue 的跳转标签。varMap 为当前作用域中所有变量的信息。

另外 SymbolTable 类还提供了一个函数 resolveName 来解析本作用域中的变量名。

4. SymbolInfo

SymbolInfo 类只有静态成员。

funcPool 记录了 C 程序中用户声明或定义的函数和两个预定义函数 printInt(int): void，readInt(): int。

因为 C 语言允许作用域嵌套，所以我们用了一个符号表栈 tableStack 来管理所有名字。当遇到一个新作用域时，将一个新符号表压入栈顶；当作用域结束时，弹出栈顶的符号表。

SymbolInfo 类提供了两个函数分别来解析函数名和变量名。当解析变量名时，resolveVarName 函数会自顶向下的查找整个符号表栈，返回第一个找到的结果。这符合 C 语言的名字解析规则。

6 语义分析

语义分析主要工作是类型推理和类型检查，其输入是语法树，通过构建符号表，输出带语义信息的语法树。我们通过词法分析、语法分析后可以初步得到一棵抽象语法树，这些树的节点都继承自 A_TreeNode 类。未进行语义分析时，只有某些叶节点（常量表达式节点）会标注其语义类型。语义分析的实现位于 ASTNode.cpp 中。

每一个树的节点都会有自己的 parseRecursive 方法，这些方法将会递归调用自己的子节点的 parseRecursive 函数，以后序深度优先搜索（DFS）的方式对整个树进行分析，在此期间构建符号表(Symbol Table)，并对分析到的变量、表达式或函数定义进行类型检查（向 Symbol Table 搜索），如果发现类型不匹配，或是变量及函数名未定义就使用的情况，则会进行报错并输出。由于我们的语法树节点以多态的方式构建，在调用 parseRecursive 方法时，每个节点会动态绑定自己的 parseRecursive 方法进行分析。

以下我们从三大类节点进行介绍其语义分析函数(parseRecursive)：表达式类型节点 A_Expression，声明类型节点 A_Declarator，语句类型节点 A_Statement。

6.1 表达式类型的语义分析

表达式类型节点全部继承自 A_Expression 类（它继承自 A_TreeNode），这些节点包括：双目表达式节点（A_BinaryExpression）、单目表达式节点（A_UnaryExpression）、函数调用节点（A_CallExpression）、字面常量节点（A_LiteralExpression）、变量节点（A_VariableExpression）以及空表达式节点（A_NopExpression）。

所有表达式节点的基类 A_Expression 包括一个标识自身表达式类型的成员 semanticType，通过这一成员，我们可以对操作符两边（或单边）的表达式类型进行检查。

A_BinaryExpression 的分析函数中，首先递归调用其两个子树的分析函数，获得其子树表达式的语义类型。如果是算术操作、逻辑操作、关系操作、位操作或赋值操作，检查两操作数的类型是否相同。如果是赋值操作，检查左侧操作数是不是**左值**。再根据操作符进行具体的类型检查，如逻辑运算符要求两操作数是布尔类型，这里不再赘述。最后设置本节点的语义类型。核心代码如下（switch 中的细节部分省略）：

```
void A_BinaryExpression::parseRecursive(){
    op1->parseRecursive();
    op2->parseRecursive();
    // set line
    this->setLine(op1->line);
    // type checking for arithmetic, relational, logic, bitwise and assignment operation
    if (operatorType <= 23 && op1->semanticType != op2->semanticType) {
        reportError(this->line, "Type mismatch: incompatible types for binary operation");
    }
    // check rhs of assignment operation to be rvalue: array reference or variable
    if (operatorType >= 18 && operatorType <=23 && !isLValue(op1)) {
        reportError(this->line, "assign to a rvalue");
```

```
    }
    switch (operatorType) {
        // ... omitted
    } // end of switch
}
```

A_UnaryExpression 的分析函数中，首先递归调用其子树的分析函数，获得其子树表达式的语义类型。再根据操作符进行语义检查。最后设置本节点的语义类型。核心代码如下：

```
void A_UnaryExpression::parseRecursive() {
    op->parseRecursive();
    this->setLine(op->line);
    switch (operatorType) {
        // omitted
    } // end of switch
}
```

A_CallExpression 的分析函数中，首先递归调用其参数表达式的分析函数，获得其参数表达式的语义类型。然后从SymbolTable中寻找函数定义，如果函数不存在或未定义则会进行报错。然后对各个参数类型进行检查。最终将函数调用的返回值类型设为本节点的语义类型。

```
void A_CallExpression::parseRecursive() {
    // resolve function name
    const FunEntry *entryPtr = SymbolInfo::resolveFunName(this->funcName);
    if (entryPtr == nullptr || !entryPtr->isDefinied) {
        reportError(this->line, "Name Error: function " + this->funcName + " undefined");
    }
    // check argument number
    int argNum = this->argumentList.size(), paraNum = entryPtr->paraList.size();
    if (argNum != paraNum) {
        reportError(this->line, "Call Error: expect " + to_string(paraNum) + "
                    argument(s), " + to_string(argNum) + " found");
    }
    // parse and check arguments
    for (int i = 0; i < argNum; i++) {
        auto argPtr = this->argumentList[i];
        // parse argument
        argPtr->parseRecursive();
        // type checking
        if (argPtr->semanticType != entryPtr->paraList[i].first) {
            reportError(this->line, "Type Error: at call to " + this->funcName
                + ", argument " + to_string(i + 1));
        }
    }
```

```
        this->semanticType = entryPtr->type;
    }
```

A_LiteralExpression 字面常量节点表达式在语法分析时已经赋值节点类型，因此是个空函数。

A_VariableExpression 变量表达式节点，首先从 SymbolTable 中查询变量是否事先有定义，若不存在则会进行报错。最后将查询到的变量类型赋值给当前节点。

```
    void A_VariableExpression::parseRecursive() {
        // resolve variable name
        // semanticValue stores identifier
        const VarEntry *varInfo = SymbolInfo::resolveVarName(this->semanticValue);
        // name not found, report error
        if (varInfo == nullptr) {
            reportError(this->line, "Name Error: " + this->semanticValue + " is not declared");
        }
        this->semanticType = varInfo->type;
        this->ID = varInfo->ID;
    }
```

6.2 声明类型的语义分析

声明类型节点 A_Declaration 再其分析时会获取其声明类型（函数声明、变量声明），并根据相应情况进行符号表的构建。

在函数声明中，首先递归调用其子节点，获取函数名。然后我们首先查询已有的（全局）函数符号表，若函数已经存在，则会进行报错。随后我们将在符号表中新建一个函数项，记录当前函数的参数类型及返回值。

```
    // function declaration
    if (funDeclarator != nullptr) {
        // check syntax
        if (!initializerList.empty()) {
            reportError(lineNum, "Syntax Error: non-empty initializer list for a function");
        }
        // check for duplicate declaration
        if (SymbolInfo::funcPool.find(funDeclarator->identifier) != SymbolInfo::funcPool.end()) {
            reportError(lineNum, "Redefinition of function " + funDeclarator->identifier);
        }
        // update function pool
        FunEntry newEntry(funDeclarator->identifier, this->type, false,
                          funDeclarator->parameterList);
        SymbolInfo::funcPool.insert(make_pair(funDeclarator->identifier, newEntry));
    }
```

在变量声明中，我们会取出栈顶的符号表，然后新建一项，记录当前变量的类型，将其插入。

```
// variable declaration: scalar or array
else {
    // check current symbol table for duplicate
    SymbolTable &tbl = *(SymbolInfo::tableStack.rbegin());
    if (tbl.resolveName(declarator->identifier) != nullptr) {
        reportError(lineNum, "Duplicate declaration for name " + declarator->identifier);
    }
    // parse initializer list
    if (initializerList.size() > declarator->capacity) {
        reportError(lineNum, "Syntax Error: size mismatch for initialization");
    }
    for (auto expr : initializerList) {
        expr->parseRecursive();
        if (expr->semanticType != this->type) {
            reportError(lineNum, "Type Error: type mismatch in initializer list");
        }
    }
    // assign var id
    declarator->ID = assignVarID(this->type);
    VarEntry newEntry(declarator->ID, this->type, declarator->capacity, declarator->isArray);
    tbl.varMap.insert(make_pair(declarator->identifier, newEntry));
}
```

6.3 语句类型的语义分析

语句节点包括：**表达式语句，复合语句，选择语句及循环语句**。

表达式语句只包括一个表达式，处理十分简单，故略去。

复合语句节点则会对其每个子语句进行递归调用，**新的符号表由父节点创建**。

```
void A_CompoundStatement::parseRecursive() {
    // parse children
    for (auto child: this->blockItemList) {
        child->parseRecursive();
    }
    // no type checking
}
```

选择语句分为两种，IF 和 IF-ELSE。两者都需对 TRUE 分支进行处理，所以我们定义了一个函数 parseTrueBranch 来统一处理 TRUE 分支。

```
void A_SelectionStatement::parseTrueBranch() {
    // parse & type check first child
```

```
        this->condition->parseRecursive();
        if (!A_Type::isBool(this->condition->semanticType)) {
            reportError(this->line, "Type Error: Non-boolean type as the `if` condition");
        }
        // parse the true branch
        // a statement block is found, push a new table
        if (typeid(*(this->branchTrue)) == typeid(A_CompoundStatement)) {
            if (SymbolInfo::tableStack.empty()) {
                reportError(this->line, "Internal Error: A_SelectionStatement::parseTrueBranch");
            }
            SymbolTable &currentTable = *(SymbolInfo::tableStack.rbegin());
            // inherit continueTag and breakTag from outer scope
            SymbolTable emptySymbolTable(currentTable.funName, currentTable.rtype,
                                         currentTable.breakTag, currentTable.continueTag);
            SymbolInfo::tableStack.push_back(emptySymbolTable);
            this->branchTrue->parseRecursive();
            SymbolInfo::tableStack.pop_back();
        }
        // not a block
        else {
            this->branchTrue->parseRecursive();
        }
    }
```

这个函数首先检查条件的类型是否正确。然后检查分支语句，如果是复合语句则新建一个符号表并压入栈顶。新符号表的 breakTag 和 continueTag 和当前栈顶的符号表相同，这样就允许在分支内 continue 和 break 外层循环。然后调用分支语句的 parseRecursive 函数。最后将栈顶符号表弹出。

IF 语句和 IF-ELSE 语句的处理都要用到 parseTrueBranch 函数。后者对 False 分支的处理也类似，故略去。

循环语句有三种：WHILE, DO-WHILE, FOR。循环语句均需要处理循环条件和循环体，所以我们定义了一个函数 parseSimpleLoop 来处理这部分。代码如下：

```
void A_IterationStatement::parseSimpleLoop(bool flag) {
    // parse condition
    this->condition->parseRecursive();
    // type check condition
    if (!A_Type::isBool(this->condition->semanticType) &&
        typeid(*(this->condition)) != typeid(A_NopExpression)) {
        reportError(this->line, "Type Error: Non-boolean type as loop condition");
    }
    // parse loopBody
    // set jump label for IR generation
    if (flag) {
        this->breakLabel = "break_" + to_string(labelNum++);
        this->continueLabel = "continue_" + to_string(labelNum++);
    }
```

```
    // push new table
    if (typeid(*(this->loopBody)) == typeid(A_CompoundStatement) && flag) {
        if (SymbolInfo::tableStack.empty()) {
            reportError(this->line, "Internal Error: A_IterationStatement::parseSimpleLoop");
        }
        SymbolTable &currentTable = *(SymbolInfo::tableStack.rbegin());
        SymbolTable emptySymbolTable(currentTable.funName, currentTable.rtype, this->breakLabel,
                                     this->continueLabel);
        SymbolInfo::tableStack.push_back(emptySymbolTable);
        this->loopBody->parseRecursive();
        SymbolInfo::tableStack.pop_back();
    }
    else {
        this->loopBody->parseRecursive();
    }
}
```

这个函数首先对循环条件进行类型检查。然后分配循环的 breakTag 和 continueTag。如果循环体位复合语句且 flag 位真则创建一个新符号表并压入堆栈。flag 是用来处理带声明的 FOR 语句的，稍后会有介绍。然后调用循环体的 parseRecursive 函数。最后，如果之前压入了新符号表，则将其弹出。

有了 parseSimpleLoop 函数，WHILE, DO-WHILE 和 FOR 语句的处理都较为简单，这里不详细介绍。下面介绍带声明的 FOR 语句的处理。

```
void A_ForStatementWithDeclaration::parseRecursive() {
    // expression value opt
    this->endCode->useVal = false;
    // push a new table and parse declaration
    this->breakLabel = "break_" + to_string(labelNum++);
    this->continueLabel = "continue_" + to_string(labelNum++);
    if (SymbolInfo::tableStack.empty()) {
        reportError(this->line, "Internal Error: A_ForStatementWithDeclaration::parseRecursive");
    }
    SymbolTable &currentTable = *(SymbolInfo::tableStack.rbegin());
    SymbolTable emptySymbolTable(currentTable.funName, currentTable.rtype, breakLabel,
                                 continueLabel);
    SymbolInfo::tableStack.push_back(emptySymbolTable);
    this->declaration->parseRecursive();
    // parse loopBody and conditon, DISABLE table creation
    this->parseSimpleLoop(false);
    // prase endCode
    this->endCode->parseRecursive();
}
```

因为 FOR 语句的初始化语句中带有局部变量的声明，所以在处理此类循环时一定要创建一个新的符号表（即使循环体不是复合语句）。将声明的信息写入符号表后，调用 parseSimpleLoop 函数，且 flag 参数要置为 false 来禁止 parseSimpleLoop 重复创建符号表。

7 IR 生成

7.1 IR 定义

1. IR 变量

IR 中的变量分为三类：普通变量%var、函数参数%arg 和临时变量%tmp。Var 型变量与 C 程序中的变量一一对应。其序号在整个程序中是唯一的，在语义分析阶段分配，按在源程序中声明的位置由前到后递增。Arg 型变量与函数的参数一一对应。其编号为对应参数在形参列表中的位置。Tmp 型变量用来存放运算的中间结果。每个 A_Expression 类及其子类的节点均对应一个 Tmp 型变量。其序号也是唯一的，在 IR 生成阶段分配。

2. IR 指令

我们参考了“Compilers: Principles, Techniques and Tools 2E”教材中的 IR 定义，为了生成 MIPS 代码的方便适当做了修改。最终采用的 IR 定义如下（以下描述中<>表示 IR 变量，[]表示字符串常量）：

1. 算术、逻辑、比较指令：将 src1 与 src2 进行相应的运算，将结果保存到 dst 中

```
1.  add <dst> <src1> <src2>
2.  sub <dst> <src1> <src2>
3.  mul <dst> <src1> <src2>
4.  div <dst> <src1> <src2>
5.  mod <dst> <src1> <src2>
6.  and <dst> <src1> <src2>
7.  or  <dst> <src1> <src2>
8.  xor <dst> <src1> <src2>
9.  shl <dst> <src1> <src2>
10. shr <dst> <src1> <src2>
11. equ <dst> <src1> <src2>
12. neq <dst> <src1> <src2>
13. gt  <dst> <src1> <src2>
14. lt  <dst> <src1> <src2>
15. ngt <dst> <src1> <src2>
```

16. nlt <dst> <src1> <src2>

以及一个允许常量操作数的指令：

17. addi <dst> <src1> #[constant]

2. 数据传送指令

1. li <dst> #[constant]

将常数 constant 的值存入 dst

2. assing <dst> <src>

将 src 的值存入 dst

3. mem2reg <dst> <base> <diaplacement>

将内存中基地址为 base、偏移地址为 diaplacement 处的值存入 dst

4. reg2mem <src> <base> <diaplacement>

将 src 的值存入内存中基地址为 base、偏移地址为 diaplacement 处

3. 流程控制指令

1. Label [name]

创建一个名为 name 的标签

2. goto [label]

跳转到名为 label 的标签处

3. if <condition> goto [label]

如果 condition 的值为 1，跳转到 label 处

4. ifn <condition> goto [label]

如果 condition 的值为 0，跳转到 label 处

4. 函数相关

1. Function [name]

名为 name 的函数的定义开始

2. End [num1] [num2] [num3]

名为 name 的函数的定义结束，num1 为函数参数个数，num2 为函数中用到的变量个数，num3 为函数中用到的临时变量的个数。这三个数字在生成目标代码时将被用到。

3. arg <value>

将 value 作为参数传入函数，此指令仅在 call 之前出现

4. call [name]

调用名为 name 的函数

5. return <value>

返回 value 的值

5. 辅助指令

1. $

标志函数体的开始和结束，用于输出 IR 时缩进函数体。无实际意义。

2. array [size]

表示本作用域内有一个大小为 size 的数组，分配栈帧使使用

7.2 IR 生成

在接口 A_TreeNode 中声明了一个纯虚函数 generateIRRecursive 来负责 IR 生成。每个 A_TreeNode 的子类均需实现这个函数来完成本节点的 IR 生成。

源程序中需要生成 IR 的部分包括表达式、数组生命、变量初始化、流程控制语句。下面在每类中分别举一个典型的例子来说明。

1. CALL 表达式 IR 生成

```
void A_CallExpression::generateIRRecursive(){
    // generate IR for arguments
    for (auto &arg : this->argumentList) {
        arg->generateIRRecursive();
    }
    int count = 0;
    for (auto &arg : this->argumentList) {
        IRList.push_back("arg " + to_string(count++) + " " + arg->ID);
    }
    // assign temp ID
    if (this->semanticType != A_Type::VOID) {
        this->ID = assignTmpID(this->semanticType);
    }
    else {
        this->ID = "*";
    }
    IRList.push_back(formThreeAddressIR("call", this->ID, this->funcName));
}
```

函数首先递归生成所有参数（子节点）的 IR。然后使用 arg 指令传递参数。如果函数返回类型不为空，则分配 Tmp 型变量用来存放返回值。最后使用 call 指令调用函数。

2. 初始化 IR 生成

```
void A_InitDeclarator::generateIRRecursive() {
    // generate IR for children
    this->declarator->generateIRRecursive();
    for (auto &item : this->initializerList) {
        item->generateIRRecursive();
    }
    // no initial value provided, work done
    if (this->initializerList.empty()) {
        return ;
    }
    // array initialization
    if (this->declarator->isArray) {
        for (int i = 0; i < this->initializerList.size(); i++) {
            IRList.push_back(formThreeAddressIR("reg2mem ",
                this->initializerList[i]->ID, this->declarator->ID, to_string(i)));
        }
    }
    // variable initialization
    else if (typeid(*(this->declarator)) == typeid(A_SimpleDeclarator)) {
        IRList.push_back(formThreeAddressIR("assign ", this->declarator->ID,
this->initializerList[0]->ID));
    }
    // else: function declarator, already checked in parsing phase
}
```

函数首先递归产生声明符(declarator)和初值列表的 IR。然后判断是初始化变量还是初始化数组。若是数组则遍历初值列表，使用 reg2mem 指令完成初始化；若是变量则直接使用 assign 指令进行初始化。

3. WHILE 循环 IR 生成

```
void A_IterationStatement::generateIRRecursive() {
    // write out continue label at the entrance of the loop
    IRList.push_back("Label " + this->continueLabel);
    // IR for condition
    this->condition->generateIRRecursive();
    // flow control logic at the entrance
    IRList.push_back("ifn " + this->condition->ID + " goto " + this->breakLabel);
    // IR for loop body
    this->loopBody->generateIRRecursive();
    // flow control logic at the exit
    IRList.push_back("goto " + this->continueLabel);
    // write out break label at the exit of the loop
    IRList.push_back("Label " + this->breakLabel);
}
```

函数首先写出循环的 continue 标签（注：语句块的 continue，break 标签均在语义分析阶段分配好，存储在 AST 节点中）。然后递归生成条件的 IR。写出条件判断语句，若不满足则跳转到 break 标签。产生循环体 IR。然后使用 goto 跳转到 continue 标签。最后写出 break 标签。

7.3 例子

考虑如下一个简单的例子：

```
int sum(int a, int b)
{
    return a + b;
}
int main()
{
    int a = 1, b = 2;
    int c = sum(a, b);
    return 233;
}
```

生成的 IR 如下：

```
Function sum

    add %tmp_i_0 %arg_i_0 %arg_i_1

    return %tmp_i_0

End 2 0 1

Function main

    li %tmp_i_1 #1

    assign  %var_i_0 %tmp_i_1

    li %tmp_i_2 #2

    assign  %var_i_1 %tmp_i_2

    arg 0 %var_i_0

    arg 1 %var_i_1

    call %tmp_i_3 sum

    assign  %var_i_2 %tmp_i_3

    li %tmp_i_4 #233

    return %tmp_i_4

End 0 3 4
```

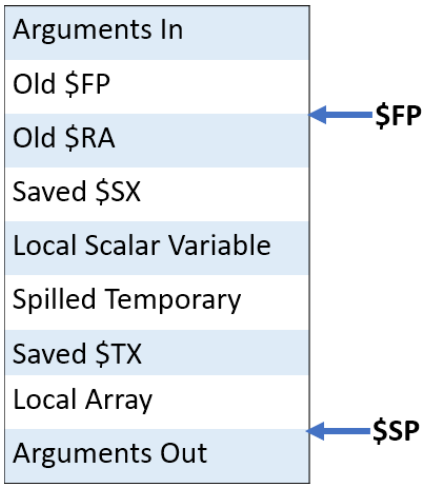
可以看出 IR 的易读性较好。在 sum 函数中，首先用前两个参数做加法，将结果存入 0 号临时变量中，然后将其返回。sum 函数末尾的 End 指令（“End 2 0 1”）后的三个数字表示本函数中有 2 个参数、0 个局部变量和 1 个临时变量。这三个数字在生成目标代码计算栈帧大小时会用到。

8 目标代码生成

这一部分负责将 IR 转换成 MIPS 目标代码，主要实现在 TargetCodeGenerator.cpp 中。生成的目标代码可以在 QtSpim 9.1.20 上运行。

8.1 栈帧结构

我们采用的栈帧结构如下图所示：



下面介绍栈帧中的各个域。

1. Arguments In

   传入参数

2. Old $FP

   保存旧的帧指针

3. Old $RA

   保存本函数的返回地址。

4. Saved $SX

   保存上层函数的$SX 寄存器

5. Local Scalar Variable

   保存本函数的局部变量

6. Spilled Temporary

   保存本函数溢出的临时变量。

7. Saved $TX

   保存本函数的$TX 寄存器，在调用其他函数时使用

8. Local Array

   保存本地数组

9. Arguments Out

   传出参数，在调用其他函数时压入

“Arguments In”域的大小（以字为单位）等于参数数目。MIPS 提供了$a0-3 四个寄存器来传递参数。但我们还是会在栈帧中为前四个参数预留空间。这样一来，当函数调用其他函数时，本函数的参数可以全部保存在 Arguments In 区域。参数的压栈顺序为右侧的参数先压栈。所以第 i 个参数的内存地址为$FP + (i + 1)×4。

“Spilled Temporary”域和“Save $TX”域的大小的和（以字为单位）等于本函数中临时变量的数目。也就是说每个 Tmp 型变量都有自己的内存空间。

“Saved $SX”域和“Local Scalar Variable”域的大小的和（以字为单位）等于函数的本地变量数目。这里要注意，“Saved $SX”用来保存上层函数的$SX 寄存器，所以并不是每个本函数的局部变量都有对应的内存空间。这会影响到稍后的寄存器分配。

综上，若不考虑传出参数，则函数的栈帧大小为：2 +本地变量数目+临时变量数目+数组大小(字)。

8.2 栈帧操作

栈帧的操作主要有四种：函数入口处构建栈帧、函数出口销毁栈帧、调用函数前压入参数、函数返回后弹出参数。

1. 函数入口（Prologue）

   函数入口处要首先保存旧的$FP 和$RA，然后让$FP = $SP – 4, $SP = $SP – Frame Size。最后将本函数中要用到的$SX 寄存器保存到“Saved $SX”区域（这些寄存器的保存由被调用者完成）。

2. 函数出口（Epilogue）

   首先写出函数的出口标签。这样在函数体中遇到 return 指令可以跳转到这里。然后令$SP = $FP + 4，并恢复保存的$SX，$FP，$RA。最后“jr $ra”返回。

3. 函数调用前

   遇到 arg 指令，如果参数序号小于 4 则用 move 指令存入$AX，否则压入栈顶（$SP 减 4）。先保存所有用到的的$TX 和$AX（这些寄存器由调用者保存），然后用 jal 指令跳转到被调函数函数。

4. 函数调用后

   函数返回后，恢复本函数用到的$TX 和$AX。从$v0 中读出返回值。

8.3 指令选择

其余的 IR 指令较好转换为 MIPS 汇编指令，总结为下表：

IR 指令	MIPS 汇编指令
mod	rem
shr	srlv
shl	sllv
lt	slt
gt	sgt
nlt	sge
ngt	sle
equ	seq
neq	sne
mem2reg	lw
reg2mem	sw
if	bnez
ifn	beqz
assign	move
li	li
goto	j

8.4 寄存器分配

寄存器分配模块的主要工作是记录每个 IR 变量在哪个寄存器中或哪个内存单元中；如果要使用内存中的 IR 变量，需要把它加载到寄存器中；如果没有多余的寄存器可以使用，则将某个寄存器的值存入内存，再将要使用的 IR 变量读入该寄存器。寄存器分配的实现在 RegManager.cpp 中。

我们采取了一种较为简单的寄存器分配方法。基本思路是$AX, $TX, $SX 三类寄存器单独**循环分配**。每一类内按序号从小到大分配，在最大序号处折回。当需要使用内存中的值且没有空闲的寄存器时（下文中称此情况为“**冲突**”），首先找到要抢占的寄存器，比如$t0。将$t0 中的值存入内存，然后将内存中的值读入$t0。下一次要抢占的寄存器变为$t1，以此类推。这种循环的分配方法可以保证最近用到的值存在寄存器中，充分利用程序的**时间局部性**。下面具体介绍三类寄存器的分配策略。

1. $AX

$a0-3 总是分配给函数的参数（Arg 型 IR 变量）。前面提到了我们在栈帧中为所有的参数都预留了内存空间，所以当冲突发生时，只需要保存要抢占的寄存器的值到该参数对应的内存空间，然后将需要的值从内从读入该寄存器即可。比如，要使用保存在内存中的第 5 个参数，要抢占的寄存器为$a0，且$a0 中目前存的是第一个参数，则分配寄存器的指令如下：

```
sw $a0, 4($fp)
lw $a0, 20($fp)
```

2. $TX

$t0-7 总是分配给中间变量（Tmp 型 IR 变量）。$t8-9 留作特殊用途。我们也为每个中间变量在内存中预留了内存空间，所以$TX 的分配策略与$AX 相同。比如，要使用内存中地址为 ADDR1 的值，要抢占寄存器$t0，且$t0 对应的中间变量的内存地址为 ADDR2，则分配寄存器的指令如下：

```
sw $t0, ADDR2($fp)
lw $t0, ADDR1($fp)
```

3. $SX

$SX 总是分配给 C 程序的局部变量和数组基地址（Var 型 IR 变量）。这里要注意，我们并没有为所有的 Var 型变量预留内存空间，因为“Saved $SX”域是用来保存上层函数的$SX 寄存器的，这部分内存不能使用。所以当冲突发生时，需要交换要读取的内存区域与要抢占的寄存器的值。MIPS 属于 SISC 架构，并不提供这样复杂的指令。所以这种内存和寄存器间的值交换必须用过第三个寄存器进行。这时候我们之前预留的$t8 就可以派上用场了。比如，要是用内存中地址为 ADDR 的值，要抢占寄存器$s0，则寄存器分配指令如下：

```
lw $t8, ADDR($fp)
sw $s0, ADDR($fp)
move $s0, $t8
```

9 测试结果

下面展示两个测试例子。

9.1 求 1-100 整数的和

1. C 语言程序

```c
int main()
{
    int sum = 0;
    for (int i = 1; i < 100; i++) {
        sum += i;
    }
    printInt(sum);
    return 233;
}
```

2. AST

```
[AST]:
translation_unit
|   function_definition -> rtype: int
|   |   function_declarator -> identifier: main, parameter_list: [ empty ]
|   |   compound_statement
|   |   |   declaration -> type: int
|   |   |   |   init_declarator
|   |   |   |   |   simple_declarator -> identifier: sum
|   |   |   |   |   literal_expression -> type: int, value: 0
|   |   |   for_statement_with_dec
|   |   |   |   declaration -> type: int
|   |   |   |   |   init_declarator
|   |   |   |   |   |   simple_declarator -> identifier: i
|   |   |   |   |   |   literal_expression -> type: int, value: 1
|   |   |   |   binary_expression -> op: lt
|   |   |   |   |   variale expression -> identifier: i
|   |   |   |   |   literal_expression -> type: int, value: 100
|   |   |   |   unary_expression -> op: postfix_inc
|   |   |   |   |   variale expression -> identifier: i
|   |   |   |   compound_statement
|   |   |   |   |   expression_statement
|   |   |   |   |   |   binary_expression -> op: assign_add
|   |   |   |   |   |   |   variale expression -> identifier: sum
|   |   |   |   |   |   |   variale expression -> identifier: i
|   |   |   expression_statement
|   |   |   |   call_expression -> name: printInt
|   |   |   |   |   variale expression -> identifier: sum
|   |   |   return_statement
|   |   |   |   literal_expression -> type: int, value: 233
```

3. IR

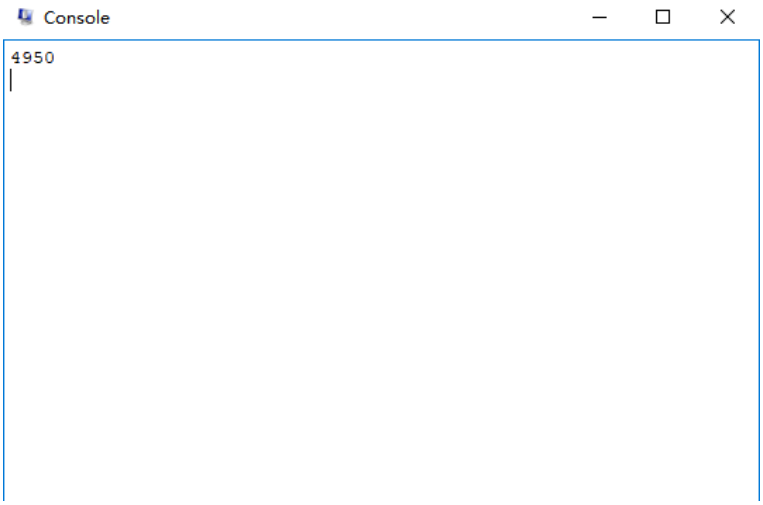
```
Function main

    li %tmp_i_0 #0

    assign  %var_i_0 %tmp_i_0

    li %tmp_i_1 #1

    assign  %var_i_1 %tmp_i_1

    Label continue_1

    li %tmp_i_2 #100

    lt  %tmp_i_3 %var_i_1 %tmp_i_2

    ifn %tmp_i_3 goto break_0

    add %var_i_0 %var_i_0 %var_i_1

    addi %var_i_1 %var_i_1 #1

    goto continue_1

    Label break_0

    arg 0 %var_i_0

    call * printInt

    li %tmp_i_4 #233

    return %tmp_i_4

End 0 2 5
```

4. MIPS 汇编

```
.globl main
.data
_newline: .asciiz "\n"

.text
readInt:
      li $v0, 5
      syscall
      jr $ra

printInt:
      li $v0, 1
      syscall
      li $v0, 4
```

```
    la $a0, _newline
    syscall
    jr $ra

main:
sw      $fp,    -4($sp)
addi    $fp,    $sp,    -4
addi    $sp,    $sp,    -36
sw      $ra,    -4($fp)
sw      $s0,    -8($fp)
sw      $s1,    -12($fp)
li $t0, 0
move $s0, $t0
li $t1, 1
move $s1, $t1
continue_1:
li $t2, 100
slt $t3, $s1, $t2
beqz $t3, break_0
add $s0, $s0, $s1
addi $s1, $s1, 1
j continue_1
break_0:
addi $sp, $sp, -4
sw $t0 -16($fp)
sw $t1 -20($fp)
sw $t2 -24($fp)
sw $t3 -28($fp)
sw $t4 -32($fp)
move $a0, $s0
jal printInt
lw $t0 -16($fp)
lw $t1 -20($fp)
lw $t2 -24($fp)
lw $t3 -28($fp)
lw $t4 -32($fp)
li $t4, 233
move $v0, $t4
j exit_main
exit_main:
lw      $s0,    -8($fp)
lw      $s1,    -12($fp)
lw      $ra,    -4($fp)
addi    $sp,    $fp,    4
lw      $fp,    ($fp)
jr      $ra
```

5. QtSpim 模拟结果



输出结果为 4950，正确。

9.2 求 6 的阶乘

1. C 语言程序

```c
int fact(int n) {
    if (n == 2) {
        return 2;
    }
    else {
        return n * fact(n - 1);
    }
}

int main()
{
    int res = fact(6);
    printInt(res);
    return 233;
}
```

2. AST

```
[AST]:
translation_unit
|   function_definition -> rtype: int
|   |   function_declarator -> identifier: fact, parameter_list: [ type: int, name: n; ]
|   |   compound_statement
|   |   |   selection_statement_with_else
|   |   |   |   binary_expression -> op: equ
|   |   |   |   |   variale expression -> identifier: n
|   |   |   |   |   literal_expression -> type: int, value: 2
|   |   |   |   compound_statement
|   |   |   |   |   return_statement
|   |   |   |   |   |   literal_expression -> type: int, value: 2
|   |   |   |   compound_statement
|   |   |   |   |   return_statement
|   |   |   |   |   |   binary_expression -> op: mul
|   |   |   |   |   |   |   variale expression -> identifier: n
|   |   |   |   |   |   |   call_expression -> name: fact
|   |   |   |   |   |   |   |   binary_expression -> op: sub
|   |   |   |   |   |   |   |   |   variale expression -> identifier: n
|   |   |   |   |   |   |   |   |   literal_expression -> type: int, value: 1
|   function_definition -> rtype: int
|   |   function_declarator -> identifier: main, parameter_list: [ empty ]
|   |   compound_statement
|   |   |   declaration -> type: int
|   |   |   |   init_declarator
|   |   |   |   |   simple_declarator -> identifier: res
|   |   |   |   |   call_expression -> name: fact
|   |   |   |   |   |   literal_expression -> type: int, value: 6
|   |   |   expression_statement
|   |   |   |   call_expression -> name: printInt
|   |   |   |   |   variale expression -> identifier: res
|   |   |   return_statement
|   |   |   |   literal_expression -> type: int, value: 233
```

3. IR

```
Function fact

    li %tmp_i_0 #2

    equ %tmp_i_1 %arg_i_0 %tmp_i_0

    ifn %tmp_i_1 goto label_0

    li %tmp_i_2 #2

    return %tmp_i_2

    Label label_0

    if %tmp_i_1 goto label_1

    li %tmp_i_3 #1

    sub %tmp_i_4 %arg_i_0 %tmp_i_3

    arg 0 %tmp_i_4

    call %tmp_i_5 fact

    mul %tmp_i_6 %arg_i_0 %tmp_i_5
```

```
    return %tmp_i_6
    Label label_1
End 1 0 7
Function main
    li %tmp_i_7 #6
    arg 0 %tmp_i_7
    call %tmp_i_8 fact
    assign  %var_i_0 %tmp_i_8
    arg 0 %var_i_0
    call * printInt
    li %tmp_i_9 #233
    return %tmp_i_9
End 0 1 3
```

---

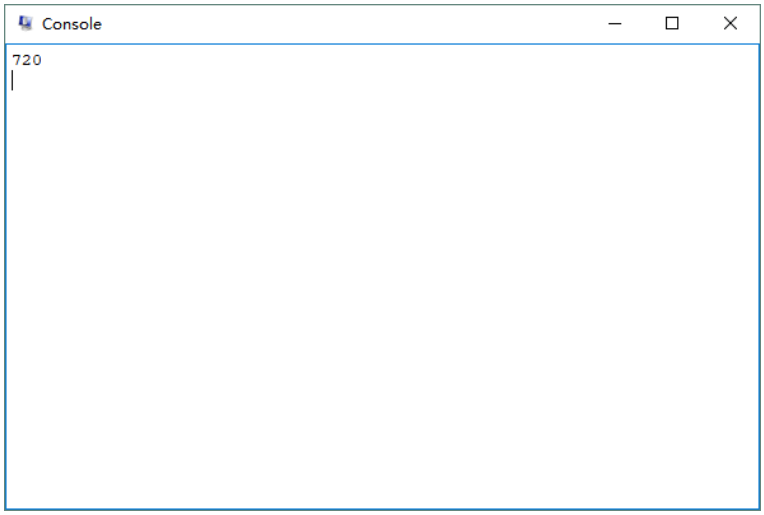
4. MIPS 汇编

```
.globl main

.data
_newline: .asciiz "\n"

.text
readInt:
      li $v0, 5
      syscall
      jr $ra

printInt:
      li $v0, 1
      syscall
      li $v0, 4
      la $a0, _newline
      syscall
      jr $ra

fact:
sw      $fp,    -4($sp)
addi    $fp,    $sp,    -4
addi    $sp,    $sp,    -36
```

```
sw      $ra,   -4($fp)
li $t0, 2
seq $t1, $a0, $t0
beqz $t1, label_0
li $t2, 2
move $v0, $t2
j exit_fact
label_0:
bnez $t1, label_1
li $t3, 1
sub $t4, $a0, $t3
addi $sp, $sp, -4
sw $t0 -8($fp)
sw $t1 -12($fp)
sw $t2 -16($fp)
sw $t3 -20($fp)
sw $t4 -24($fp)
sw $t5 -28($fp)
sw $t6 -32($fp)
sw $a0 4($fp)
move $a0, $t4
jal fact
lw $t0 -8($fp)
lw $t1 -12($fp)
lw $t2 -16($fp)
lw $t3 -20($fp)
lw $t4 -24($fp)
lw $t5 -28($fp)
lw $t6 -32($fp)
lw $a0 4($fp)
move $t5, $v0
mul $t6, $a0, $t5
move $v0, $t6
j exit_fact
label_1:
exit_fact:
lw      $ra,   -4($fp)
addi   $sp,   $fp,   4
lw      $fp,   ($fp)
jr      $ra

main:
sw      $fp,   -4($sp)
addi   $fp,   $sp,   -4
addi   $sp,   $sp,   -24
sw      $ra,   -4($fp)
```

```
sw      $s0,    -8($fp)
sw $t3, 0($fp)
lw $t3, -40($fp)
li $t3, 6
addi $sp, $sp, -4
sw $t0 -12($fp)
sw $t1 -16($fp)
sw $t2 -20($fp)
sw $t3 -40($fp)
move $a0, $t3
jal fact
lw $t0 -12($fp)
lw $t1 -16($fp)
lw $t2 -20($fp)
lw $t3 -40($fp)
sw $t4, 0($fp)
lw $t4, -44($fp)
move $t4, $v0
move $s0, $t4
addi $sp, $sp, -4
sw $t0 -12($fp)
sw $t1 -16($fp)
sw $t2 -20($fp)
sw $t3 -40($fp)
sw $t4 -44($fp)
move $a0, $s0
jal printInt
lw $t0 -12($fp)
lw $t1 -16($fp)
lw $t2 -20($fp)
lw $t3 -40($fp)
lw $t4 -44($fp)
sw $t5, 0($fp)
lw $t5, -48($fp)
li $t5, 233
move $v0, $t5
j exit_main
exit_main:
lw      $s0,    -8($fp)
lw      $ra,    -4($fp)
addi    $sp,    $fp,    4
lw      $fp,    ($fp)
jr      $ra
```

5. QtSpim 模拟结果



结果为 720，正确。