# 实验四 目标代码生成 实验报告

匡亚明学院 陈劭源 161240004

## 文件夹结构

注意：**Makefile**在根目录下（而不是在**Code**文件夹内）

```
.
├── Code                    // 源代码文件
│   ├── ast                 // 抽象语法树相关代码
│   │   └── ...
│   ├── ir                  // 中间代码相关
│   │   ├── optimize.c      // 实现了简单的窥孔优化
│   │   └── ...
│   ├── mips                // 代码生成相关
│   │   └── mips.c
│   ├── error.c             // 错误处理代码
│   ├── lexical.l           // flex词法文件
│   ├── memory.c            // 内存管理代码
│   ├── symtbl.c            // 符号表代码
│   ├── main.c              // 主程序
│   └── syntax.y            // bison语法文件
├── Include                 // 头文件
│   └── ...
├── Makefile                // Makefile文件
├── parser                  // 语法分析器可执行文件
├── README.txt              // README文件
├── report.md               // 本实验报告的源代码
├── report.pdf              // 本实验报告
├── Test
│   ├── sample              // 提供的测试用例
│   │   └── ...
│   └── secret              // 自行构造的测试用例
│       └── ...
└── testrun.sh              // 测试用例运行脚本
```

## 编译和运行方法

### 编译环境

- OS: Ubuntu 18.04.2 LTS
- gcc: gcc 7.3.0
- flex: flex 2.6.4
- bison: GNU Bison 3.0.4
- make: GNU Make 4.1
- shell: GNU bash 4.4.19

## 编译方法

切换到根目录，输入

```
make
```

即可从源代码生成可执行文件parser（位于根目录）。

## 运行方法

输入

```
./parser
```

或

```
make run
```

即可运行语法分析器。语法分析器默认从标准输入读入c--源代码，可以通过参数指定从文件读入：

```
./parser source_file dest_file
```

运行

```
make clean
```

可以清除所有中间文件和目标文件。

# 完成的功能点

将没有语义错误的c--源代码翻译成MIPS32指令序列，并能够在SPIM Simulator上运行。

# 实现方法

本次实验是基于中间代码生成目标代码的。本实验中，采用如下调用约定

1. $fp中存放栈基地址指针，0($fp)存放返回地址， -4($fp)存放上层函数的栈基地址指针；
2. 函数参数按照从右向左的顺序依次压入栈中；
3. 将函数参数弹出栈是被调用者的责任。

本次实验只需要把中间代码逐条翻译成MIPS代码即可。具体的翻译规则如下：

| IR Code | MIPS Code |
| --- | --- |
| LABEL x : | x : |
| FUNCTION f : | f :<br>push($ra)<br>push($fp)<br>addiu $fp, $sp, 8 |
| (end function) | f_ret :<br>addiu $sp, $fp, -8<br>pop($fp)<br>pop($ra)<br>addiu $sp, $sp, #argsize<br>jr $ra<br>nop |
| x := y | getvalue($t0, y)<br>assignto($t0, x) |
| x := y [op] z | getvalue($t0, y)<br>getvalue($t1, z)<br>[op] $t0, $t0, $t1<br>assignto($t0, x) |
| GOTO x | j x<br>nop |
| IF x [relop] y GOTO z | getvalue($t0, x)<br>getvalue($t1, y)<br>b[relop] $t0, $t1, z<br>nop |
| RETURN x | getvalue($v0, x)<br>j f_ret<br>nop |
| DEC x [size] | addiu $sp, $sp, -size |
| ARG x | getvalue($t0, x)<br>push($t0) |
| x := CALL f | jal f<br>nop<br>assignto($v0, x) |
| PARAM x | lw $t0, offset($fp)<br>assignto($t0, x) |

其中，push(reg)表示把reg压入栈中，pop(reg)表示把栈顶元素弹至reg中，getvalue(reg, x)表示把x的值加载到reg中，assignto(reg, x)表示把reg中的值保存到x中。

本次实验采用了朴素寄存器分配算法，即仅当变量的值使用时才加载入内存中，改变内存的值时立即将新值写入内存。

## 实验总结

本次实验在上一次实验的基础上进行，只需要将IR语句逐条翻译成MIPS指令即可，较为简单。