C-语言项目总结

一．实验目的

本项目使用 C-语言文法作为实验材料，用计算机模拟出 C-语言的实现。本次实验为构造 C-语言的代码生成模块，起承之前的语义语法分析模块 ,把完善信息后的 AST 配合符号表以每类节点为单位生成目标语言，以配合虚拟机进行实际的执行。

二．实验环境

语言：ANSI C

系统：Linux 3.16.0-36-generic #48-Ubuntu SMP x86_64

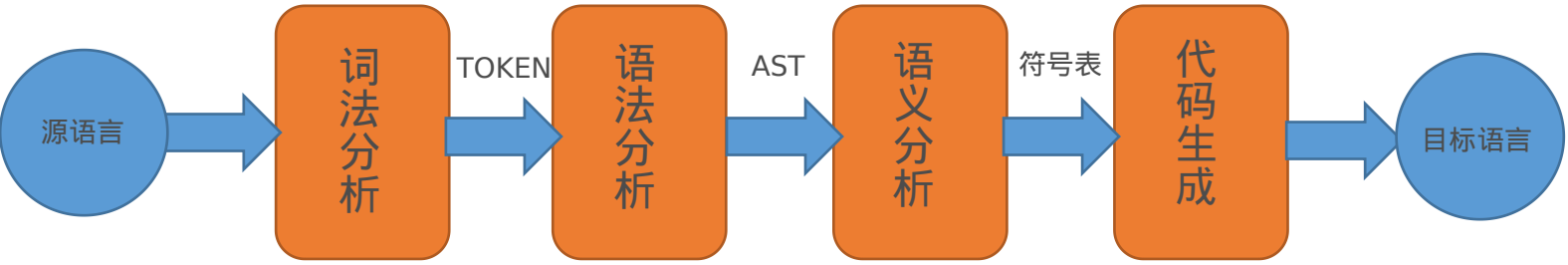
编译工具：gcc version 4.9.2 (Ubuntu 4.9.2-10ubuntu13)

词法分析工具：flex 2.5.39

语法分析工具：bison (GNU Bison) 3.0.2

三．技术框架

系统执行策略如下：



工程文件结构如下：

```
.
├── flex
│   └── cminus.l
├── yacc
│   └── cminus.y
├── main.c
├── code.c
├── code.h
├── cgen.c
├── cgen.h
├── globals.h
├── analyze.c
├── analyze.h
├── symtab.c
├── symtab.h
├── parse.c
├── parse.h
├── scan.c
├── scan.h
├── util.c
├── util.h
├── Makefile
├── TestFlie.tny
└── TestErrorFile.tny

2 directories, 21 files.
```

四．技术线路

现模拟程序执行：

main.c 调用 scan.c 把输入字符流转换成 Token 流

main.c 调用 parse.c 把 Token 流转换成 AST 树

main.c 调用 analyze.c 完善 AST 树，构造符号表

main.c 调用 cgen.c 把 AST 树打印成目标代码

词法分析模块：

采用了 lex 工具，以正则表达式的形式完成程序书写

语法分析模块：

采用了 yacc 工具，以 BNF 范式的形式完成程序书写

语义分析模块：

采用广度优先搜索的方式；符号表架构使用了栈结构

代码生成模块：

借鉴 TINY 虚拟机的指令集

五．测试结果

正确性测试：

以下为输入文档

```
1 int gcd (int u, int v)
2 {
3         if (v == 0) return u ;
4         else return gcd(v,u-u/v*v);
5         /* u-u/v*v == u mod v */
6 }
7 void main(void)
8 {
9         int x; int y;
10        x = input(); y = input();
11        output (gcd(x,y)) ;
12 }
```

以下为输出结果

```
* TINY Compilation to TM Code
* File: test.tm
* Standard prelude:
  0:     LD  5,0(0)     load gp with maxaddress
  1:    LDA  6,0(5)     copy gp to mp
  2:     ST  0,0(0)     clear location 0
* End of standard prelude.
* -> Function (gcd)
  4:     ST  1,-2(5)    func: store the location of func. entry
* func: unconditional jump to next declaration belongs here
* func: function body starts here
  3:    LDC  1,6(0)     func: load function location
  6:     ST  0,-1(6)    func: store return address
* -> param
* u
* <- param
* -> param
* v
* <- param
* -> compound
* -> if
* -> Op
* -> Id (v)
  7:    LDC  0,-3(0)    id: load varOffset
  8:    ADD  0,6,0      id: calculate the address
  9:     LD  0,0(0)     load id value
* <- Id
 10:     ST  0,-4(6)    op: push left
* -> Const
 11:    LDC  0,0(0)     load const
* <- Const
 12:     LD  1,-4(6)    op: load left
```

```
 13:    SUB  0,1,0      op ==
 14:    JEQ  0,2(7)     br if true
 15:    LDC  0,0(0)     false case
 16:    LDA  7,1(7)     unconditional jmp
 17:    LDC  0,1(0)     true case
* <- Op
* if: jump to else belongs here
* -> return
* -> Id (u)
 19:    LDC  0,-2(0)    id: load varOffset
 20:    ADD  0,6,0      id: calculate the address
 21:     LD  0,0(0)     load id value
* <- Id
 22:     LD  7,-1(6)    return: to caller
* <- return
* if: jump to end belongs here
 18:    JEQ  0,5(7)     if: jmp to else
* -> return
* -> Call
* -> Id (v)
 24:    LDC  0,-3(0)    id: load varOffset
 25:    ADD  0,6,0      id: calculate the address
 26:     LD  0,0(0)     load id value
* <- Id
 27:     ST  0,-6(6)    call: push argument
* -> Op
* -> Id (u)
 28:    LDC  0,-2(0)    id: load varOffset
```

```
 29:    ADD  0,6,0      id: calculate the address
 30:     LD  0,0(0)     load id value
* <- Id
 31:     ST  0,-4(6)    op: push left
* -> Op
* -> Op
* -> Id (u)
 32:    LDC  0,-2(0)    id: load varOffset
 33:    ADD  0,6,0      id: calculate the address
 34:     LD  0,0(0)     load id value
* <- Id
 35:     ST  0,-5(6)    op: push left
* -> Id (v)
 36:    LDC  0,-3(0)    id: load varOffset
 37:    ADD  0,6,0      id: calculate the address
 38:     LD  0,0(0)     load id value
* <- Id
 39:     LD  1,-5(6)    op: load left
 40:    DIV  0,1,0      op /
* <- Op
 41:     ST  0,-5(6)    op: push left
* -> Id (v)
 42:    LDC  0,-3(0)    id: load varOffset
 43:    ADD  0,6,0      id: calculate the address
 44:     LD  0,0(0)     load id value
* <- Id
 45:     LD  1,-5(6)    op: load left
 46:    MUL  0,1,0      op *
* <- Op
```

```
 47:     LD  1,-4(6)    op: load left
 48:    SUB  0,1,0      op -
* <- Op
 49:     ST  0,-7(6)    call: push argument
 50:     ST  6,-4(6)    call: store current mp
 51:    LDA  6,-4(6)    call: push new frame
 52:    LDA  0,1(7)     call: save return in ac
 53:     LD  7,-2(5)    call: relative jump to function entry
 54:     LD  6,0(6)     call: pop current frame
* <- Call
 55:     LD  7,-1(6)    return: to caller
* <- return
 23:    LDA  7,32(7)    jmp to end
* <- if
* <- compound
 56:     LD  7,-1(6)    func: load pc with return address
  5:    LDA  7,51(7)    func: unconditional jump to next declaration
* -> Function (gcd)
* -> Function (main)
 58:     ST  1,-3(5)    func: store the location of func. entry
* func: unconditional jump to next declaration belongs here
* func: function body starts here
 57:    LDC  1,60(0)    func: load function location
 60:     ST  0,-1(6)    func: store return address
* -> compound
* -> assign
* -> Id (x)
 61:    LDC  0,-2(0)    id: load varOffset
 62:    ADD  0,6,0      id: calculate the address
 63:    LDA  0,0(0)     load id address
* <- Id
 64:     ST  0,-4(6)    assign: push left (address)
* -> Call
 65:     IN  0,0,0      read integer value
* <- Call
 66:     LD  1,-4(6)    assign: load left (address)
 67:     ST  0,0(1)     assign: store value
* <- assign
* -> assign
* -> Id (y)
 68:    LDC  0,-3(0)    id: load varOffset
 69:    ADD  0,6,0      id: calculate the address
 70:    LDA  0,0(0)     load id address
* <- Id
 71:     ST  0,-4(6)    assign: push left (address)
* -> Call
 72:     IN  0,0,0      read integer value
* <- Call
 73:     LD  1,-4(6)    assign: load left (address)
 74:     ST  0,0(1)     assign: store value
* <- assign
* -> Call
* -> Call
* -> Id (x)
 75:    LDC  0,-2(0)    id: load varOffset
```

```
 76:    ADD  0,6,0      id: calculate the address
 77:     LD  0,0(0)     load id value
* <- Id
 78:     ST  0,-6(6)    call: push argument
* -> Id (y)
 79:    LDC  0,-3(0)    id: load varOffset
 80:    ADD  0,6,0      id: calculate the address
 81:     LD  0,0(0)     load id value
* <- Id
 82:     ST  0,-7(6)    call: push argument
 83:     ST  6,-4(6)    call: store current mp
 84:    LDA  6,-4(6)    call: push new frame
 85:    LDA  0,1(7)     call: save return in ac
 86:     LD  7,-2(5)    call: relative jump to function entry
 87:     LD  6,0(6)     call: pop current frame
* <- Call
 88:     ST  0,-6(6)    call: push argument
 89:     LD  0,-6(6)    load arg to ac
 90:    OUT  0,0,0      write ac
* <- Call
* <- compound
 91:     LD  7,-1(6)    func: load pc with return address
 59:    LDA  7,32(7)    func: unconditional jump to next declaration
* -> Function (main)
 92:    LDC  0,-2(0)    init: load globalOffset
 93:    ADD  6,6,0      init: initialize mp with globalOffset
* -> Call
 94:     ST  6,0(6)     call: store current mp
 95:    LDA  6,0(6)     call: push new frame
 96:    LDA  0,1(7)     call: save return in ac
 97:    LDC  7,60(0)    call: unconditional jump to main() entry
 98:     LD  6,0(6)     call: pop current frame
* <- Call
* End of execution.
 99:   HALT  0,0,0
```

六．项目总结

这次实验让我熟悉了一个编译器前端的大部分行为，收获颇多。

在词法方面，熟悉了 lex 的语法及功能。

在语法方面，处理了分解 BNF 范式，悬挂 else 等语言设计问题；同时还熟悉了 yacc 的语法及功能。初步开始设计 AST 结构。

在语义方面，对数据结构的使用进行了权衡，对一些新增功能，如

作用域，声明等制定了实现策略，正式把源语言转换成了内部形式。

在代码生成方面，体会到需要合理的规划我们的资源，如几个寄存器 / 采用的数据抽象方式 / 如何选取指令集，不仅要考虑我们实现语言，更要考虑目标机的状态。

在本次实验中，实现了编译器前端的绝大部分内容，接下来可能还会考虑继续深入后端，或者做代码优化。