编译原理实验—— C--语言编译器

邓开圣 161220031

lab1: 词法分析、语法分析

实验内容

使用C语言编写程序，对使用C--语言书写的源代码进行词法分析和语法分析，并输出语法树。

实现思路

- 借助词法分析工具Flex和语法分析工具GNU Bison
- Flex使用用户定义的正则表达式对词素进行匹配，Bison根据用户定义的语法规则对识别的词法单元进行归约
- 每进行一次归约，定义构造语法树的动作，直到语法分析过程结束，得到一棵完整的语法树
- 关键数据结构与算法：

```c
// Tree node type
typedef struct TreeNode{
    char* name;             // name of node, e.g. ID, LP, Stmt
    char* propertyValue;    // property value of node, can be empty, e.g. myVar, int, float
    int n_children;         // number of children node
    int first_line;         // first line number of current node
    struct TreeNode* parent;           // parent of current node
    struct TreeNode** children;        // children of current node
    bool if_leaf;           // note if current node is leaf node
} Node;

```

语法树节点结构体

```c
// Create a node
// Set first_line to -1 if not need to print out
Node* createNode(char* name, char* propertyValue, int first_line){

    Node* n = (Node*)malloc(sizeof(Node));    // allocate space in memory
    n->name = (char*)malloc(strlen(name) + 1);
    strcpy(n->name, name);
    n->propertyValue = (char*)malloc(strlen(propertyValue) + 1);
    strcpy(n->propertyValue, propertyValue);
    n->first_line = first_line;
    n->n_children = 0;
    n->parent = NULL;
    n->children = NULL;
    n->if_leaf = false;     // not leaf node by default
    return n;
}

```

创建语法树节点

```c
// Construct tree according to parent-children relationship
void constructTree(Node* parent, int n_children, ...){
    va_list p;
    va_start(p, n_children);
    parent->children = (Node**)malloc(sizeof(Node*) * n_children);
    parent->n_children = n_children;
    for(int i = 0; i < n_children; ++i){
        parent->children[i] = (Node*)va_arg(p, Node*);
        if(parent->children[i]) parent->children[i]->parent = parent;
    }
    va_end(p);
}
```

根据父子关系构建语法树

输出示例

```
// a3.cmm
int main(){
    int a, b;
    int result;
    int start;
    int found = 0;
    a = read();
    b = read();
    if(a > b){
        start = a;
    }else {
        start = b;
    }
    while(found == 0){
        if(start == (start / a) * a){
            if(start == (start / b) * b){
                result = start;
                found = 1;
            }else {
                start = start + 1;
            }
        }else {
            start = start + 1;
        }
    }
    write(result);
    return 0;
}
```

```
// a3.out
Program (1)
  ExtDefList (1)
    ExtDef (1)
      Specifier (1)
        TYPE: int
      FunDec (1)
        ID: main
        LP
        RP
      CompSt (1)
        LC
        DefList (2)
          Def (2)
            Specifier (2)
              TYPE: int
            DecList (2)
              Dec (2)
                VarDec (2)
                  ID: a
              COMMA
              DecList (2)
                Dec (2)
                  VarDec (2)
                    ID: b
            SEMI
          DefList (3)
```

```
          Def (3)
            Specifier (3)
              TYPE: int
            DecList (3)
              Dec (3)
                VarDec (3)
                  ID: result
            SEMI
          DefList (4)
            Def (4)
              Specifier (4)
                TYPE: int
              DecList (4)
                Dec (4)
                  VarDec (4)
                    ID: start
              SEMI
            DefList (5)
              Def (5)
                Specifier (5)
                  TYPE: int
                DecList (5)
                  Dec (5)
                    VarDec (5)
                      ID: found
                    ASSIGNOP
                    Exp (5)
                      INT: 0
                SEMI
        StmtList (6)
          Stmt (6)
            Exp (6)
              Exp (6)
                ID: a
              ASSIGNOP
              Exp (6)
                ID: read
                LP
                RP
            SEMI
          StmtList (7)
            Stmt (7)
              Exp (7)
                Exp (7)
                  ID: b
                ASSIGNOP
                Exp (7)
                  ID: read
                  LP
                  RP
              SEMI
            StmtList (8)
              Stmt (8)
                IF
                LP
                Exp (8)
                  Exp (8)
                    ID: a
```

```
      RELOP
      Exp (8)
        ID: b
    RP
    Stmt (8)
      CompSt (8)
        LC
        StmtList (9)
          Stmt (9)
            Exp (9)
              Exp (9)
                ID: start
              ASSIGNOP
              Exp (9)
                ID: a
            SEMI
        RC
    ELSE
    Stmt (10)
      CompSt (10)
        LC
        StmtList (11)
          Stmt (11)
            Exp (11)
              Exp (11)
                ID: start
              ASSIGNOP
              Exp (11)
                ID: b
            SEMI
        RC
  StmtList (13)
    Stmt (13)
      WHILE
      LP
      Exp (13)
        Exp (13)
          ID: found
        RELOP
        Exp (13)
          INT: 0
      RP
      Stmt (13)
        CompSt (13)
          LC
          StmtList (14)
            Stmt (14)
              IF
              LP
              Exp (14)
                Exp (14)
                  ID: start
                RELOP
                Exp (14)
                  Exp (14)
                    LP
                    Exp (14)
                      Exp (14)
```

```
                ID: start
              DIV
              Exp (14)
                ID: a
            RP
          STAR
          Exp (14)
            ID: a
      RP
      Stmt (14)
        CompSt (14)
          LC
          StmtList (15)
            Stmt (15)
              IF
              LP
              Exp (15)
                Exp (15)
                  ID: start
                RELOP
                Exp (15)
                  Exp (15)
                    LP
                    Exp (15)
                      Exp (15)
                        ID: start
                      DIV
                      Exp (15)
                        ID: b
                    RP
                  STAR
                  Exp (15)
                    ID: b
              RP
              Stmt (15)
                CompSt (15)
                  LC
                  StmtList (16)
                    Stmt (16)
                      Exp (16)
                        Exp (16)
                          ID: result
                        ASSIGNOP
                        Exp (16)
                          ID: start
                      SEMI
                    StmtList (17)
                      Stmt (17)
                        Exp (17)
                          Exp (17)
                            ID: found
                          ASSIGNOP
                          Exp (17)
                            INT: 1
                        SEMI
                  RC
              ELSE
              Stmt (18)
```

```
                                CompSt (18)
                                  LC
                                  StmtList (19)
                                    Stmt (19)
                                      Exp (19)
                                        Exp (19)
                                          ID: start
                                        ASSIGNOP
                                        Exp (19)
                                          Exp (19)
                                            ID: start
                                          PLUS
                                          Exp (19)
                                            INT: 1
                                      SEMI
                                  RC
                            RC
                          ELSE
                          Stmt (21)
                            CompSt (21)
                              LC
                              StmtList (22)
                                Stmt (22)
                                  Exp (22)
                                    Exp (22)
                                      ID: start
                                    ASSIGNOP
                                    Exp (22)
                                      Exp (22)
                                        ID: start
                                      PLUS
                                      Exp (22)
                                        INT: 1
                                  SEMI
                              RC
                        RC
                    StmtList (25)
                      Stmt (25)
                        Exp (25)
                          ID: write
                          LP
                          Args (25)
                            Exp (25)
                              ID: result
                          RP
                        SEMI
                      StmtList (26)
                        Stmt (26)
                          RETURN
                          Exp (26)
                            INT: 0
                          SEMI
RC
```

lab2: 语义分析

实验内容

- 在词法分析和语法分析程序的基础上编写一个程序，对C--代码进行语义分析和类型检查，并打印分析结果。
- 需要识别的错误类型：
  - 错误类型1： 变量在使用时未经定义。
  - 错误类型2： 函数在调用时未经定义。
  - 错误类型3： 变量出现重复定义，或变量与前面定义过的结构体名字重复。
  - 错误类型4： 函数出现重复定义（即同样的函数名出现了不止一次定义）。
  - 错误类型5： 赋值号两边的表达式类型不匹配。
  - 错误类型6： 赋值号左边出现一个只有右值的表达式。
  - 错误类型7： 操作数类型不匹配或操作数类型与操作符不匹配（例如整型变量与数组变量相加减，或数组（或结构体）变量与数组（或结构体）变量相加减）。
  - 错误类型8： return语句的返回类型与函数定义的返回类型不匹配。
  - 错误类型9： 函数调用时实参与形参的数目或类型不匹配。
  - 错误类型10： 对非数组型变量使用“[…]”（数组访问）操作符。
  - 错误类型11： 对普通变量使用“(…)”或“()”（函数调用）操作符。
  - 错误类型12： 数组访问操作符“[…]”中出现非整数（例如a[1.5]）。
  - 错误类型13： 对非结构体型变量使用“.”操作符。
  - 错误类型14： 访问结构体中未定义过的域。
  - 错误类型15： 结构体中域名重复定义（指同一结构体中），或在定义时对域进行初始化（例如struct A { int a = 0; }）。
  - 错误类型16： 结构体的名字与前面定义过的结构体或变量的名字重复。
  - 错误类型17： 直接使用未定义过的结构体来定义变量。

实现思路

- 前序遍历上一步生成的语法树，在遍历过程中生成语义信息，同时检查语义是否合法
- 主要数据结构有3个：哈希表、符号表、类型表。其中哈希表用于检查某个符号当前是否已经定义；符号表为链表结构，用于获取当前已经定义的变量或函数的信息；类型表也为链表结构，用于获取当前已经定义的类型信息。
- 关键数据结构与算法：

```
4 // hash table
5 // both variable name and type name will be hashed into this table, since each two of them can not be identical
6 bool hash_table[HASH_TABLE_SIZE];
```

哈希表

```
51 // A variable should be added to SYMBOL_LIST after it has been defined
52 // A type should be added to TYPE_LIST after it has been defined
53
54 struct VAR_INFO {
55     char* varType;  // int->"int", float->"float", array->element type name, function->"function", stru->"struct"
56     char* varName;
57     bool ifArray;   // array->true, non-array->false
58 };
```

变量信息描述符

```
59 // information of certain type
60 // for basic type, the typeDetail pointer is NULL
61 // allow 2-D array (enumeration in code when defining)
62 // don'e allow nested structure/function
63 struct TYPE_INFO {
64     char* typeName;
65     char* typeCategory;     // "array", "function" or "struct"
66     union TYPE_DETAIL* typeDetail;
67 };
68 // detail of complex type
69 union TYPE_DETAIL {
70     struct ARRAY_INFO* array_info;
71     struct STRUCT_INFO* struct_info;
72     struct FUNC_INFO* func_info;
73 };
```

类型信息描述符

只有复杂类型（数组、函数和结构体）会用到`typeDetail`字段

```
// array info
struct ARRAY_INFO {
    int size;               // size of array
    // type info of array element
    // to get detailed type info, use getTypeInfo(eleTypeName)
    char* eleTypeName;      // type name of element
};

// struct info
struct STRUCT_INFO {
    int n_fields;   // num of fields
    struct VAR_INFO** fields;         // fields of struct (var type and var name)
};

// func info
struct FUNC_INFO {
    int n_params;       // num of parameters
    struct VAR_INFO** params;     // parameters of function (var type and var name)
    char* returnTypeName;   // type name of return value
};
```

数组、结构体和函数的描述符

输出示例

```
// a1.cmm
int main(int x1, int x2){
    int p = x1;
    int i = 3;
    int k = x2 + p;
    i = x1 + x3;
    return k;
}
```

```
dks@ubuntu:~/Desktop/compiler_lab/lab2/Tests$ ls
a1.cmm  parser
dks@ubuntu:~/Desktop/compiler_lab/lab2/Tests$ ./parser a1.cmm
Error type 1 at Line 5: Undefined variable "x3".
```

```
// a2.cmm
struct Student{
    int id;
    int weight;
    int grades;
};

int newStudent(int a, int b){
    struct Student st;
    st.id = a;
    st.weight = b;
    st.grades = 0;
    return 0;
}

int main(){
    int k = 3;
    int p = 14;
    newStdent(k, p);
    return 0;
}
```

```
dks@ubuntu:~/Desktop/compiler_lab/lab2/Tests$ ls
a1.cmm  a2.cmm  parser
dks@ubuntu:~/Desktop/compiler_lab/lab2/Tests$ ./parser a2.cmm
Error type 2 at Line 18: Undefined function "newStdent".
dks@ubuntu:~/Desktop/compiler_lab/lab2/Tests$
```

```
// b1.cmm
struct Leaf {
    int number;
    int isGreen;
};

struct Tree {
    struct Leaf leaves[100];
    int height;
    float weight;
    int hasFruit;
};

struct AppleTree {
    struct Tree t;
    int numberApple;
    float priceApple;
    int highQuality;
};

struct AppleTree newAppleTree() {
    int i = 0;
    int x = 1;
    struct AppleTree at = x();
    struct Tree tt;
    struct Leaf sto[100];
    while(i < 100){
        sto[i].number = x;
        tt.leaves[i].number = sto[i].number;
        sto[i].isGreen = 1;
        tt.leaves[i].isGreen = sto[i].isGreen;
        i = i + 1;
    }
    tt.height = 100;
    tt.weight = 2.5;
    tt.hasFruit = 0;
    at.numberApple = 1;
    at.priceApple = 1.2;
    i = 0;
    while(i < 100){
        at.t[i].leaves[i].number = tt.leaves[i].number;
        at.t.leaves[i].isGreen = tt.leaves[i].isGreen;
        i = i + 1;
    }
    at.t.height = tt.height;
    at.t.weight = tt.weight;
    at.hasFruit = 1;
    at.highQuality = at.t.hasFruit * at.t.weight;
    return at;
```

```
}

int main(){
    struct AppleTree aat = newAppleTree();
    return 0;
}
```

```
dks@ubuntu:~/Desktop/compiler_lab/lab2/Tests$ ls
a1.cmm  a2.cmm  b1.cmm  parser
dks@ubuntu:~/Desktop/compiler_lab/lab2/Tests$ ./parser b1.cmm
Error type 11 at Line 23: "x" is not a function.
Error type 10 at Line 40: illegal subscript with a non-array variable.
Error type 14 at Line 46: Non-existent field "hasFruit".
Error type 7 at Line 47: Type mismatched for operands.
dks@ubuntu:~/Desktop/compiler_lab/lab2/Tests$
```

lab3: 中间代码生成（与优化）

实验内容

在词法分析、语法分析和语义分析的基础上，将C--源代码翻译为三地址代码形式的中间代码。

语法	描述
LABEL x :	定义标号x。
FUNCTION f :	定义函数f。
x := y	赋值操作。
x := y + z	加法操作。
x := y - z	减法操作。
x := y * z	乘法操作。
x := y / z	除法操作。
x := &y	取y的地址赋给x。
x := *y	取以y值为地址的内存单元的内容赋给x。
*x := y	取y值赋给以x值为地址的内存单元。
GOTO x	无条件跳转至标号x。
IF x [relop] y GOTO z	如果x与y满足[relop]关系则跳转至标号z。
RETURN x	退出当前函数并返回x值。
DEC x [size]	内存空间申请，大小为4的倍数。
ARG x	传实参x。
x := CALL f	调用函数，并将其返回值赋给x。
PARAM x	函数参数声明。
READ x	从控制台读取x的值。
WRITE x	向控制台打印x的值。

中间代码的形式及操作规范

实现思路

- 前序遍历语法树，过程类似语义分析，遍历到特定位置时根据当前使用的产生式创建中间代码

```
142 // translate functions
143 void translate_Program(Node* program);
144 void translate_ExtDefList(Node* extdeflist);
145 void translate_ExtDef(Node* extdef);
146 void translate_ExtDecList(Node* extdeclist);
147 void translate_Specifier(Node* specifier);
148 void translate_OptTag(Node* opttag);
149 void translate_VarDec(Node* vardec);
150 void translate_FunDec(Node* fundec);
151 void translate_VarList(Node* varlist);
152 void translate_ParamDec(Node* paramdec);
153 void translate_CompSt(Node* compst);
154 void translate_StmtList(Node* stmtlist);
155 void translate_Stmt(Node* stmt);
156 void translate_DefList(Node* deflist);
157 void translate_Def(Node* def);
158 void translate_DecList(Node* declist);
159 void translate_Dec(Node* dec, char* varName);
160
161 void translate_Exp(Node* exp, Operand* place);
162 void translate_Cond(Node* exp, Operand* label_true, Operand* label_false);
163 void translate_Args(Node* args);
164
165 void translate_ArrayArg(char* arrayName);
166
167 void translate_ArrayAddr(Node* exp, Operand* place);
168
```

执行中间代码生成的一系列translate函数

- 关键数据结构与算法：

```
19 typedef enum IRType {
20     Label_IR,               // label
21     Function_IR,            // function
22     Assign_IR,              // =
23     Plus_IR,                // +
24     Minus_IR,               // -
25     Multiply_IR,            // *
26     Divide_IR,              // /
27     Address_IR,             // &
28     ReadMemory_IR,          // read memory (x := *y)
29     WriteMemory_IR,         // write memory (*x := y)
30     Goto_IR,                // goto
31     ConJump_IR,             // conditional jump
32     Return_IR,              // return statement
33     DecMemory_IR,           // ask for memory
34     Arg_IR,                 // parameter pass
35     Call_IR,                // call function
36     Param_IR,               // parameter declaration
37     Read_IR,                // read from console
38     Write_IR,               // write to console
39 } IRType;
40
```

中间代码的种类，用于在输出时设置不同的语句形式

```
41 // Operand
42 typedef struct Operand {
43     // OperandType type;
44     char* value;
45 }Operand;
46
```

操作数，一开始为了实现比较好的可拓展性用了一个结构体，后来发现只需要一个char*就行了，但保留了这个结构体

```
48 // IR
49 typedef struct InterCode {
50     int n_operand;          // number of operands
51     IRType type;            // type of IR
52     union{
53         // 1 operand
54         // label, function, goto, return, arg, param, read, write
55         struct {
56             Operand* op1;
57         }o1;
58         // 2 operands
59         // assign, getAddr, readMem, writeMem, functionCall
60         struct {
61             Operand* op1;
62             Operand* op2;
63         }o2;
64         // 3 operands
65         // add, minus, multiply, divide
66         struct {
67             Operand* op1;
68             Operand* op2;
69             Operand* op3;
70         }o3;
71         // conditional jump
72         struct {
73             Operand* op1;
74             Operand* op2;
75             Operand* op3;
76             char* relop;
77         }conJump;
78         // memory dec
79         struct {
80             Operand* op1;    // owner of this declared space
81             int size;
82         }memDec;
83     }ops;
84 }InterCode;
```

中间代码描述符，包含了操作数、操作符、操作符数量、中间代码类型等成员

```
86 typedef struct InterCodeNode{
87     InterCode* interCode;
88     struct InterCodeNode* next;
89     // struct InterCodeNode* prev;
90 }InterCodeNode;
```

生成的中间代码用单向链表组织

```
92 // variable name map of source code and intercode
93 typedef struct NameMap{
94     char* varName;
95     char* interCodeVarName;
96 }NameMap;
97
98 typedef struct NameMapNode{
99     NameMap nameMap;
100    struct NameMapNode* next;
101 }NameMapNode;
102
103 typedef struct ArrayDesc{
104    char* arrayName;
105    int n_dim;
106    int dim1;
107    int dim2;
108 }ArrayDesc;
109
110 typedef struct ArrayDescNode{
111    ArrayDesc* arrayDesc;
112    struct ArrayDescNode* next;
113 }ArrayDescNode;
114
115 typedef struct SpecialParam{
116    char* name;
117    bool ifAddr;
118 }SpecialParam;
119
120 typedef struct SpecialParamNode{
121    SpecialParam* param;
122    struct SpecialParamNode* next;
123 }SpecialParamNode;
124
```

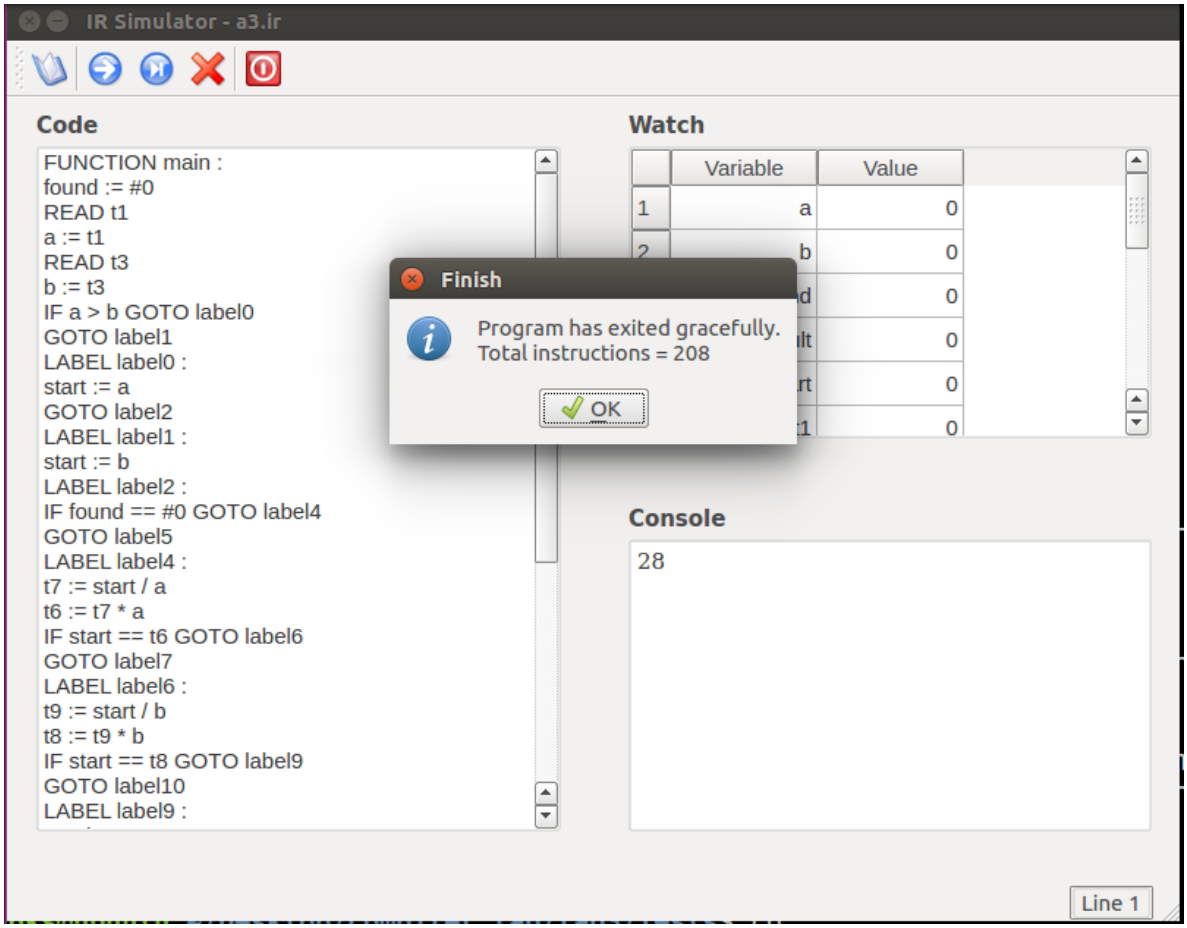
以上几个数据结构分别表示源代码与中间代码的变量名映射表节点、数组描述符节点和特殊参数节点，均用单向链表组织

输出示例

```
// a3.cmm
// 求a, b的最小公倍数
int main(){
    int a, b;
    int result;
    int start;
    int found = 0;
    a = read();
    b = read();
    if(a > b){
        start = a;
    }else {
        start = b;
    }
    while(found == 0){
        if(start == (start / a) * a){
            if(start == (start / b) * b){
                result = start;
                found = 1;
            }else {
                start = start + 1;
            }
        }else {
            start = start + 1;
        }
    }
    write(result);
    return 0;
}
```

```
FUNCTION main :
found := #0
READ t1
a := t1
READ t3
b := t3
IF a > b GOTO label0
GOTO label1
LABEL label0 :
start := a
GOTO label2
LABEL label1 :
start := b
LABEL label2 :
IF found == #0 GOTO label4
GOTO label5
LABEL label4 :
t7 := start / a
t6 := t7 * a
IF start == t6 GOTO label6
GOTO label7
LABEL label6 :
t9 := start / b
t8 := t9 * b
```

```
IF start == t8 GOTO label9
GOTO label10
LABEL label9 :
result := start
found := #1
GOTO label2
LABEL label10 :
t13 := start + #1
start := t13
GOTO label2
LABEL label7 :
t15 := start + #1
start := t15
GOTO label2
LABEL label5 :
t17 := result
WRITE t17
RETURN #0
```



在基于PyQt4的模拟器上运行中间代码

lab4: 目标代码生成

实验内容

在词法分析、语法分析、语义分析和中间代码生成的基础上，将C--源代码翻译为MIPS32指令序列（可以包含伪指令），并在SPIM Simulator上运行。

实现思路

- 关键数据结构和算法

本实验使用的主要数据结构有如下3个： VarDesc 表示变量描述符， op 表示其内容（操作数），reg_no 表示其存储的寄存器的下标； Register 是寄存器描述符，包含寄存器内容、寄存器名字、寄 存器索引以及寄存器当前年龄（用于寄存器分配），后续介绍寄存器分配使用的算法。 StackDesc 是 栈描述符，用数组模拟一个在内存中的栈结构，用于函数调用的目标代码生成。

```
#define STK_SIZE 100

typedef struct VarDesc{
    Operand* op;
    int reg_no;
}VarDesc;

typedef struct Register{
    VarDesc* content;
    char* name;
    int index;
    int age;
}Register;

typedef struct StackDesc{
    VarDesc* stack[STK_SIZE];
    int top;
}StackDesc;

```

关键的函数有两个： `writeAssemblyCode` 和 `getReg` `writeAssemblyCode` 用于将传入的中间代码翻译成目标代码并输出到目标文件中。主要思想是用 `switch ... case...` 语句根据中间代码的类型分别处理。 `getReg` 函数用于分配寄存器，返回分配的寄存器的索引。主要思想是：首先遍历所有可用寄存器看当 前操作数是否已经分配到了某个寄存器中，若是，则直接返回该寄存器的索引；若否，先给每个当前已 经已有内容的寄存器老化（即 `age` +1），然后寻找是否有空寄存器，若有，则直接使用找到的第一个空 寄存器；若无，则找当前 `age` 值最大的一个寄存器，将其内容换出，使用这个寄存器存储参数中的操作 数。并且每次在访问某个寄存器中都要将该寄存器的 age 置为0，这样每次换出的都是最近最不常使用 的操作数

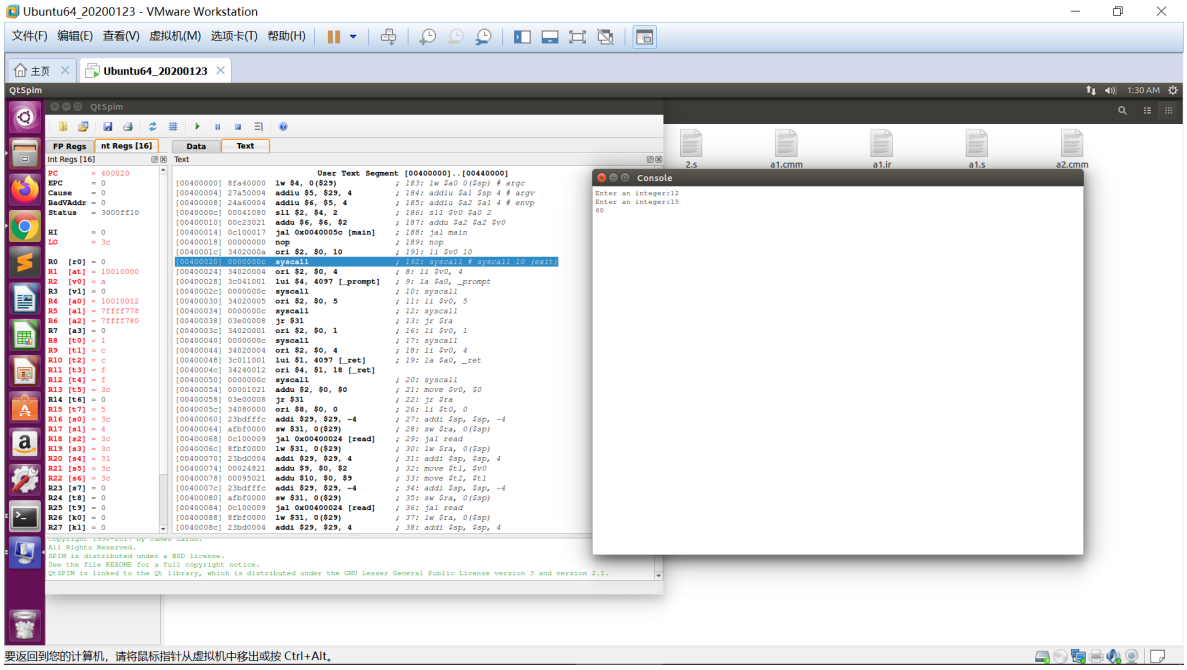
```
void initRegs();
void initStk();
void pushStack(VarDesc* vardesc);
VarDesc* popStack();
int getReg(Operand* operand, FILE* fp);
void writeAssemblyCode(char* fileName);


VarDesc* copyVarDesc(VarDesc* src);
void saveVarIntoStack(VarDesc* var, FILE* fp);
VarDesc* loadVarFromStackTop(FILE* fp);

void saveAllVarIntoStack(FILE* fp);
void loadAllVarFromStack(FILE* fp);
```

输出示例

```
@ a3.cmm 生成的目标代码
.data
_prompt: .asciiz "Enter an integer:"
_ret: .asciiz "\n"
.globl main
.text

read:
  li $v0, 4
  la $a0, _prompt
  syscall
  li $v0, 5
```

```
  syscall
  jr $ra

write:
  li $v0, 1
  syscall
  li $v0, 4
  la $a0, _ret
  syscall
  move $v0, $0
  jr $ra


main:
  li $t0, 0
  addi $sp, $sp, -4
  sw $ra, 0($sp)
  jal read
  lw $ra, 0($sp)
  addi $sp, $sp, 4
  move $t1, $v0
  move $t2, $t1
  addi $sp, $sp, -4
  sw $ra, 0($sp)
  jal read
  lw $ra, 0($sp)
  addi $sp, $sp, 4
  move $t3, $v0
  move $t4, $t3
  bgt $t2, $t4, label0
  j label1
label0:
  move $t5, $t2
  j label2
label1:
  move $t5, $t4
label2:
  li $t6, 0
  beq $t0, $t6, label4
  j label5
label4:
  div $t5, $t2
  mflo $t7
  mul $s0, $t7, $t2
  beq $t5, $s0, label6
  j label7
label6:
  div $t5, $t4
  mflo $s1
  mul $s2, $s1, $t4
  beq $t5, $s2, label9
  j label10
label9:
  move $s3, $t5
  li $t0, 1
  j label2
label10:
  addi $s4, $t5, 1
```

```
  move $t5, $s4
  j label2
label7:
  addi $s5, $t5, 1
  move $t5, $s5
  j label2
label5:
  move $s6, $s3
  move $a0, $s6
  addi $sp, $sp, -4
  sw $ra, 0($sp)
  jal write
  lw $ra, 0($sp)
  addi $sp, $sp, 4
  move $v0, $0
  jr $ra
```



在SPIM Simulator上的运行结果