Compile Principle 编译原理期末项目报告

3210106185 金杭伟

题目要求

用lex写出一个tiger语言或者类C或者类PASCAL某个语言的词法分析器，用YACC的分析方法完成对某一个的语法分析，并生成语法树和中间代码（如果生成目标代码：可加分）。(15分)

背景知识

tiger文法

这部分的内容来自 *Tiger Language Reference Manual*

Lexical Aspects

1. 标识符由字母、数字和下划线组成，以字母开头，区分大小写。
2. 可以在标记之间使用空白字符或注释，这些将被忽略。注释以 /* 开始，以 */ 结束，并支持嵌套。
3. 整数常量由一个或多个十进制数字组成，不支持负整数常量；可以通过一元运算符 - 对整数常量进行取反得到负数。
4. 字符串常量由双引号 " 包围，可以包含可打印字符、空格或转义序列。转义序列以反斜杠 \ 开头，表示特定的字符序列，例如 \n 表示换行符。
5. 预留关键字包括 array break do else end for function if in let nil of then to type var while。
6. ☐ { } . + - * / = <> < <= > >= & | :=。

Expressions

A Tiger program is a single expr.

```
expr:
    string-constant
    integer-constant
    nil
    lvalue
    - expr
    expr binary-operator expr
    lvalue := expr
    id ( expr-listopt )
    ( expr-seqopt )
    type-id { field-listopt }
    type-id [ expr ] of expr
    if expr then expr
    if expr then expr else expr
```

```
    while expr do expr
    for id := expr to expr do expr
    break
    let declaration-list in expr-seqopt end
expr-seq:
    expr
    expr-seq ; expr
expr-list:
    expr
    expr-list , expr
field-list:
    id = expr
    field-list , id = expr
```

- Lvalues

  ```
  lvalue:
      id
      lvalue . id
      lvalue [ expr ]
  ```

  Lvalues（左值）表示可以赋值的存储位置，包括变量、参数、记录字段和数组元素。

- Return values

  返回值：过程调用、赋值、if-then、while、break以及有时的if-then-else语句不产生值，不能出现在需要值的地方。空的let表达式返回无值。用括号括起来的零个或多个表达式通过分号分隔，在括号内按顺序求值，并返回最后一个表达式的值（如果有）。

- Record and Array Literals

  记录和数组字面量：type-id { field-listopt } 创建类型为type-id的新记录实例；type-id [expr] of expr创建大小由中括号内的表达式给出的type-id新数组，初始时由of后的表达式给出的值填充。

- Function Calls

  函数调用：函数应用是一个表达式id(expr-listopt)，其中有零个或多个用逗号分隔的表达式参数。在函数调用时，实际参数的值从左到右求值，并根据常规的静态作用域规则绑定到函数的形式参数上。

- Operators

  1. 二元运算符包括 +、-、*、/、=、<>、<、>、<=、>=、& 和 |。
  2. 括号按照通常方式分组表达式。 领先的减号表示对整数表达式取反。
  3. +、-、* 和 / 需要整数操作数，并返回整数结果。
  4. 比较运算符对其操作数进行比较，如果比较成立则返回否则返回0。
  5. 逻辑运算符 & 和 | 是整数的惰性逻辑运算符。

- Assignment

  1. 赋值表达式 lvalue := expr 将表达式的值绑定到 lvalue 的内容。

2. 数组和记录的赋值是按引用而非数值进行的。
3. 记录或数组值从创建至程序结束都持续存在。

- nil

1. 空值（nil）表示可以分配给任何记录类型的值。
2. 从空值记录中访问字段会导致运行时错误。

- 流程控制

1. if-then-else：计算第一个表达式，如果结果非零，则计算第二个表达式并返回其结果；否则，计算第三个表达式并返回其结果。第二和第三个表达式必须是相同类型或都不返回值。
2. if-then：计算第一个表达式，如果结果非零，则计算第二个表达式，该表达式不得返回值。if-then表达式不返回值。
3. while-do：计算第一个表达式，如果结果非零，则计算第二个表达式，该表达式不得返回值，并重新计算while-do表达式。
4. for：计算第一和第二个表达式（循环界限），然后对于这两个表达式值之间的每个整数（包括两端），计算第三个表达式，其中由id命名的整型变量绑定到循环索引。此变量的作用域限于第三个表达式，且不可被赋值。如果循环的上界小于下界，则此表达式可能不产生结果且不被执行。
5. break：终止最内层的while或for表达式，该表达式必须与break在同一函数/过程中。在此外使用break是非法的。

- Let

let declaration-list in expr-seq-opt end：计算声明，将类型、变量和函数绑定到表达式序列的作用域内。结果是最后一个表达式的结果，如果没有表达式则为无。

声明

1. Types

declaration-list: declaration declaration-list declaration declaration: type-declaration variable-declaration function-declaration type-declaration: type type-id = type type: type-id { type-fieldsopt } array of type-id type-fields: type-field type-fields , type-field type-field: id : type-id

- Tiger语言有两种预定义类型：int和string。新类型可以通过以下方式定义或重新定义现有类型。
- 类型表达式（例如，fx:intg、array of ty）创建不同的类型，因此具有相同基础的两个数组类型或具有相同字段的两个记录是不同的。Type a=b是别名。
- 一系列类型声明（即，没有中间变量或函数声明）可以是相互递归的。这样的序列中没有两个定义的类型可以具有相同的名称。每个递归循环必须通过一个记录或数组类型。在let ... type-declaration ... in expr-seqopt end中，类型声明的作用域从它所属的类型声明序列的开始（可能是单个元素）到结束。
- 类型名称有自己的命名空间。

2. Variables

variable-declaration: var id := expr var id : type-id := expr

如果没有指定类型，则变量的类型来自表达式。在let ... variable-declaration ... in expr-seqopt end中，变量声明的作用域从声明之后开始，到结束为止。变量在其作用域内一直存在。变量和函数共享相同的命名空间。

3. Functions

function-declaration: function id (type-fieldsopt) = expr function id (type-fieldsopt) : type-id = expr

函数返回指定类型的值，而过程仅用于其副作用。两种形式都允许指定零个或多个带类型的参数，这些参数按值传递。这些参数的作用域是expr。

expr是函数或过程的主体。一系列函数声明（即，没有中间变量或类型声明）可以是相互递归的。这样的序列中没有两个函数可以具有相同的名称。在let ... function-declaration ... in expr-seqopt end中，函数声明的作用域从它所属的函数声明序列的开始（可能是单个元素）到结束。

词法实现

1. token：读入对应关键字即返回对应的token

```
42  "nil"               { adjust();return NIL; }
43  "of"                { adjust();return OF; }
44  "then"              { adjust();return THEN; }
45  "to"                { adjust();return TO; }
46  "type"              { adjust();return TYPE; }
47  "var"               { adjust();return VAR; }
48  "while"             { adjust();return WHILE; }
49  ","                 { adjust();return COMMA; }
50  ":"                 { adjust();return COLON; }
51  ";"                 { adjust();return SEMICOLON; }
52  "("                 { adjust();return LPAREN; }
53  ")"                 { adjust();return RPAREN; }
54  "["                 { adjust();return LBRACKET; }
```

2. 解析时同时返回值：使用yytext和yylval保存解析的内容

```
72  [a-zA-Z][a-zA-Z0-9_]* { adjust();yylval.sval = strdup(yytext);return ID; }
73  [0-9]+              { adjust(); yylval.ival = atoi(yytext);return INTEGER_CONSTANT; }
74  \"([^\\\"]|\\.)*\" { adjust();
75                        char* temp = strdup(yytext);
76                        temp[strlen(temp)-1] = '\0'; // 去除双引号
77                        yylval.sval = temp+1;
78                        return STRING_CONSTANT;
79                      }
```

值得注意的是，这里的sval和ival变量需要在tiger.y文件的union模块中定义：

```
%union {
    char* sval;
    int ival;

    char* id;
    tree_node node;
```

3. 嵌套注释：针对这个情况，我特意在lex中设计了COMMENT模式和对应的嵌套次数计数器，这样可以完美解决嵌套注释的问题。

```
 9  int comment_count = 0;/* 定义计数器 */
10  int charPos=1;
11  int EM_tokPos=0;
12
13  void adjust(void)
14  {
15   EM_tokPos=charPos;
16   charPos+=yyleng;
17  }
18
19  %}
20
21  %option noyywrap
22
23  /* 定义状态 */
24  %x COMMENT
25
26  %%
27
28  "/*"              { adjust();if(comment_count == 0) BEGIN(COMMENT); comment_co
29  <COMMENT>"/*"     { adjust();comment_count++; /* 增加嵌套注释的计数器 */ }
30  <COMMENT>"*/"     { adjust();comment_count--; /* 减少计数器 */ if(comment_count
31  <COMMENT>.        { adjust();/* 在注释模式下，忽略所有字符 */ }
```

文法实现

1. 文法实现是一个大工程，使用yacc完成文法分析后实际上得到的就是一个AST(抽象文法树)，那么就可以直接将这个抽象文法树转换成中间代码形式。

2. 初始的symbol是一个stm

```
prog : stm  {printf("语法分析成功! \n"); program = prog($1);
            generateDotFile(program,"tree.dot");
            }
    ;
```

3. stm可以派生出的语句如下：

```
stm : assignStm                      { $$ = stm_assignStm($1); }
    | conditionStm                   { $$ = stm_conditionStm($1); }
    | whileStm                       { $$ = stm_whileStm($1); }
    | forStm                         { $$ = stm_forStm($1); }
    | letStm                         { $$ = stm_letStm($1); }
    | LPAREN stmBlock_opt RPAREN     { $$ = stm_LPAREN_stmBlock_opt_RPAREN(
    | functionCallStm                { $$ = stm_functionCallStm($1); }
    ;
```

4. 为了体现加减乘除和取负之间的优先级关系，在这里将number表达式分成了不同的level，这样，通过不同level之间的分离，可以很好地确保算式的计算顺序和我们预先设定的相同。

```
algorithmExp: numberExp      { $$ = algorithmExp_numberExp($1); }
    | stringExp              { $$ = algorithmExp_stringExp($1); }
    ;

stringExp: STRING_CONSTANT   { $$ = stringExp_STRING_CONSTANT($1); }
    ;

numberExp: numberItem                { $$ = numberExp_numberItem($1);}
    | numberExp PLUS numberItem      { $$ = numberExp_PLUS($1, $3);}
    | numberExp MINUS numberItem     { $$ = numberExp_MINUS($1, $3);}
    ;

numberItem: numberFrator                 { $$ = numberItem_numberFrator($1);
    | numberItem MULTIPLY numberFrator   { $$ = numberItem_MULTIPLY($1, $3);
    | numberItem DIVIDE numberFrator     { $$ = numberItem_DIVIDE($1, $3);}
    ;

numberFrator: MINUS numberFrator      { $$ = numberFrator_MINUS($2);}
    | ID                              { $$ = numberFrator_ID($1);}
    | INTEGER_CONSTANT                { $$ = numberFrator_INTERGER_CONSTANT
    | LPAREN numberExp RPAREN         { $$ = numberFrator_LPAREN($2);}
    ;
```

5. 同理对relationOp也有对应的考虑，首先是OR语句，再是AND语句；同时我也考虑到了relationItem中出现NIL的情况，并特殊考虑了。

```
lvalue: ID                                          { $$ = lvalue_ID($1); }
    | lvalue DOT ID                                 { $$ = lvalue_DOT($1, $3); }
    | lvalue LBRACKET numberExp RBRACKET            { $$ = lvalue_LBRACKET($1, $3);}
    ;

relationExp: relationExpOR      { $$ = relationExp($1); }
    ;

relationExpOR: relationExpAND           { $$ = relationExpOR_relationExpAND($1); }
    | relationExpOR OR relationExpAND   { $$ = relationExpOR_OR($1, $3); }
    ;

relationExpAND: relationExpItem                 { $$ = relationExpAND_relationExpItem($1)
    | relationExpAND AND relationExpItem        { $$ = relationExpAND_AND($1, $3); }
    ;

relationItemLeft: lvalue DOT ID                     { $$ = lvalue_DOT($1, $3); }
    | lvalue LBRACKET numberExp RBRACKET            { $$ = lvalue_LBRACKET($1, $3);}
    | algorithmExp                                  { $$ = $1;}
    ;

relationItemRight: lvalue DOT ID                     { $$ = lvalue_DOT($1, $3); }
    | lvalue LBRACKET numberExp RBRACKET            { $$ = lvalue_LBRACKET($1, $3);}
    | algorithmExp                                  { $$ = $1;}
    | NIL                                           { $$ = Lable("nil");}
    ;

relationExpItem: relationItemLeft relationOp relationItemRight  { $$ = relationExpItem_al
    | algorithmExp                                      { $$ = relationExpItem_algorithmE
    ;
```

6. for语句的else需要特殊考虑，对对应的优先级进行了处理。

```
assignStm: lvalue ASSIGN algorithmExp       { $$ = assignStm($1, $3); }
    | lvalue ASSIGN NIL                     { $$ = assignStm_NIL($1); }
    ;

conditionStm: IF relationExp THEN stm %prec LOWER_THAN_ELSE { $$ = conditionStm_IF_THEN(
    | IF relationExp THEN stm ELSE stm { $$ = conditionStm_IF_THEN_ELSE($2, $4, $6); }
    ;

whileStm: WHILE relationExp DO stm                  { $$ = whileStm($2, $4); }
    ;

forStm: FOR ID ASSIGN numberExp TO numberExp DO stm { $$ = forStm($2, $4, $6, $8); }
    ;

letStm: LET declaration_list IN stm END             { $$ = letStm($2, $4); }
    ;
```

7. functionCallStm：特殊考虑了哈数调用作为参数的情况。这里的参数可选relationItemLeft主要是为了引入数组和结构体的情况，直接就用了现成的symbol.

```
call_list: call_paramater               { $$ = call_list_first($1); }
    | call_list COMMA call_paramater    { $$ = call_list($1, $3); }
    ;

call_paramater: relationItemLeft        { $$ = call_paramater_algorithmExp($1); }
    | functionCallStm                { $$ = call_paramater_functionCallStm($1); }
    ;

lvalue: ID                                          { $$ = lvalue_ID($1); }
    | lvalue DOT ID                                 { $$ = lvalue_DOT($1, $3); }
    | lvalue LBRACKET numberExp RBRACKET            { $$ = lvalue_LBRACKET($1, $3);}
    ;
```

8. 其余的函数声明的部分的文法和第一部分中给出的文法类似，此处省略。

生成中间代码

1. 参照教科书中的中间代码形式，我给出了类似的中间代码结构。可见CONST,CONSTSTRING, NAME, TEMP, BINOP, MEM, CALL, ESEQ, MOVE, EXP, JUMP, CJUMP, SEQ, LABLE语句和教科书中的大致相似。

```
enum BINOP {
    BINOP_PLUS,
    BINOP_MINUS,
    BINOP_MUL,
    BINOP_DIV,
    BINOP_AND,
    BINOP_OR,
    BINOP_LSHIFT,
    BINOP_RSHIFT,
    BINOP_ARSHIFT,
    BINOP_XOR
};
enum relop { EQ, NE, LT, GT, LE, GE, ULT, ULE, UGT, UGE};

struct tree_node_
{
    enum { CONST,CONSTSTRING, NAME, TEMP, BINOP, MEM, CALL, ESEQ, MOVE, EXP, JUMP, CJUMP, SEQ, LABLE, LET_STM} kind;
    union {
        int consti;
        string cstring;
        string name;
        tree_node temp;
        struct {enum BINOP op; tree_node left, right;} binop;
        tree_node mem;
        struct {tree_node func; tree_node_list args;} call;
        struct {tree_node stm; tree_node exp;} eseq;
        struct {tree_node dst, src;} move;
        tree_node exp;
        tree_node jump;
        struct {enum relop op; tree_node left, right; tree_node t, f;} cjump;
        struct {tree_node left, right;} seq;
        string lable;
        struct {declaration_list l; tree_node s;} let_stm;
    } u;
};
```

2. 不同之处：

   在本中间代码中的CJUMP和JUMP语句的true和false的地方不是标签值，而是tree_node的地址，亦即直接指向对应的节点。

   加入了LET_STM即Let语句对应的节点，这是为了方便统合declaration和in_block的树结构，方便输出中间代码结果。

3. 中间代码生成过程：在本程序中，中间代码生成类似模拟计算机如何执行这些指令，如whileStm先赋值，在加begin标签，再判断，在加begin_body标签，在这个结构体的末尾加上jump语句和end标签。其余的代码生成是类似的。

```
tree_node whileStm(tree_node r, tree_node s)
{
    tree_node begin_label = Lable("begin");
    tree_node begin_body = Lable("begin_body");
    tree_node end_label = Lable("end");
    tree_node temp = Seq(begin_label, r);
    temp = Seq(temp, begin_body);
    temp = Seq(temp, s);
    temp = Seq(temp, Jump(begin_label));
    temp = Seq(temp, end_label);
    while(true_call_list != NULL)
    {
        tree_node true_call_node = Tree_node_list_get(true_call_list);
        true_call_node->u.cjump.t = begin_body;
        true_call_list = Tree_node_list_pop(true_call_list);
    }
    while(false_call_list != NULL)
    {
        tree_node false_call_node = Tree_node_list_get(false_call_list);
        false_call_node->u.cjump.f = end_label;
        false_call_list = Tree_node_list_pop(false_call_list);
    }
    return temp;
}
```

4. truelist和falselist的回填

在处理CJUMP时，一开始我们不知道正确标签和错误标签的填法，需要处理到后面才能回填。

具体而言，对于AND语句，计算到下一个时，就能回填上一个的true_lable,直到遇见OR语句，即剩下一个truelable没填。

```
tree_node relationExpAND_AND(tree_node l, tree_node r)
{
    tree_node true_label = Lable("true");
    tree_node temp = Seq(l, true_label);
    temp = Seq(temp, r);

    tree_node true_call_node_save = Tree_node_list_get(true_call_list);//this is r
    true_call_list = Tree_node_list_pop(true_call_list);

    tree_node true_call_node = Tree_node_list_get(true_call_list); //this is l's right en
    true_call_node->u.cjump.t = true_label;
    true_call_list = Tree_node_list_pop(true_call_list);

    true_call_list = Tree_node_list(true_call_node_save, true_call_list);
    return temp;
}
```

对于OR语句，则可以将积累的falselable全部填完，

```
tree_node relationExpOR_OR(tree_node l, tree_node r)
{
    tree_node false_label = Lable("false");
    tree_node temp = Seq(l, false_label);
    temp = Seq(temp, r);

    tree_node false_call_node_save = Tree_node_list_get(false_call_list);//this is r
    false_call_list = Tree_node_list_pop(false_call_list);
    while(false_call_list != NULL)
    {
        tree_node false_call_node = Tree_node_list_get(false_call_list);
        false_call_node->u.cjump.f = false_label;
        false_call_list = Tree_node_list_pop(false_call_list);
    }
    false_call_list = Tree_node_list(false_call_node_save, false_call_list);
    return temp;
}
```

对于总的whileStm,由于此时都知道了truelable和falselable的去向，将所有累计的truelist和falselist全部填完即可。

```
tree_node whileStm(tree_node r, tree_node s)
{
    tree_node begin_label = Lable("begin");
    tree_node begin_body = Lable("begin_body");
    tree_node end_label = Lable("end");
    tree_node temp = Seq(begin_label, r);
    temp = Seq(temp, begin_body);
    temp = Seq(temp, s);
    temp = Seq(temp, Jump(begin_label));
    temp = Seq(temp, end_label);
    while(true_call_list != NULL)
    {
        tree_node true_call_node = Tree_node_list_get(true_call_list);
        true_call_node->u.cjump.t = begin_body;
        true_call_list = Tree_node_list_pop(true_call_list);
    }
    while(false_call_list != NULL)
    {
        tree_node false_call_node = Tree_node_list_get(false_call_list);
        false_call_node->u.cjump.f = end_label;
        false_call_list = Tree_node_list_pop(false_call_list);
    }
    return temp;
}
```

5. declaration语句：

declaration分为type,variable和function三类。这一部分的代码比较繁琐，但是基本上只是重构抽象文法树，在这个部分，我没有完成类型判断，主要是太麻烦了。

```c
typedef struct Declaration{
    enum { TYPE_DECLARATION, VARIABLE_DECLARATION, FUNCTION_DECLARATION } kind;
    union {
        TypeDeclaration *typeDeclaration;  // 对应 declaration: type_declaration
        VariableDeclaration *variableDeclaration;  // 对应 declaration: variable_declaration
        FunctionDeclaration *functionDeclaration;  // 对应 declaration: function_declaration
    };
} Declaration, *declaration;

// 定义 type_declaration_list 结构体
typedef struct DeclarationList {
    declaration head;
    struct DeclarationList *tail;
} DeclarationList , *declaration_list;
```

对应特殊的function declaration, 本程序同样将它的代码块的中间表示也生成了。

6. 最终的中间代码展示：

为了更好地可视化，我们引入了Graphviz模块生成中间代码的图示。dot文件的格式比较简单，包括节点名字内容和对应的边信息就行了，生成dot文件的generateDotFile()函数基本上是一个递归调用的过程。

tree.dot

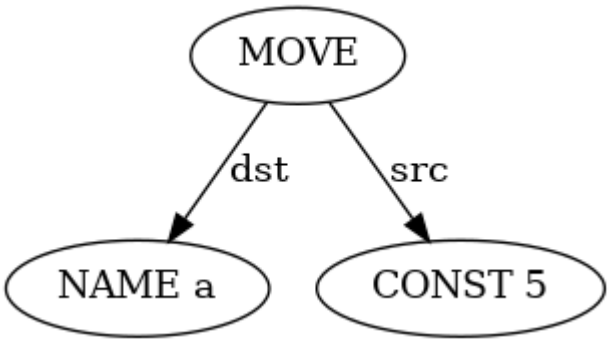
```
digraph G {
node0x5b5a85ed28d0 [label="SEQ"];
node0x5b5a85ed2890 [label="SEQ"];
node0x5b5a85ed2850 [label="SEQ"];
node0x5b5a85ed2670 [label="CJUMP GT"];
node0x5b5a85ed25f0 [label="MEM"];
node0x5b5a85ed25b0 [label="BINOP PLUS"];
node0x5b5a85ed2530 [label="NAME a"];
node0x5b5a85ed2570 [label="CONST 5"];
node0x5b5a85ed25b0 -> node0x5b5a85ed2530;
node0x5b5a85ed25b0 -> node0x5b5a85ed2570;
node0x5b5a85ed25f0 -> node0x5b5a85ed25b0;
node0x5b5a85ed2630 [label="CONST 5"];
node0x5b5a85ed27d0 [label="LABLE begin"];
node0x5b5a85ed2810 [label="LABLE end"];
node0x5b5a85ed2670 -> node0x5b5a85ed25f0 [label="left"];
node0x5b5a85ed2670 -> node0x5b5a85ed2630 [label="right"];
node0x5b5a85ed2670 -> node0x5b5a85ed27d0 [label="true "];
node0x5b5a85ed2670 -> node0x5b5a85ed2810 [label="false"];
node0x5b5a85ed27d0 [label="LABLE begin"];
node0x5b5a85ed2850 -> node0x5b5a85ed2670 [label="left"];
node0x5b5a85ed2850 -> node0x5b5a85ed27d0 [label="right"];
node0x5b5a85ed2790 [label="MOVE"];
node0x5b5a85ed2710 [label="NAME a"];
node0x5b5a85ed2750 [label="CONST 5"];
node0x5b5a85ed2790 -> node0x5b5a85ed2710 [label="dst"];
node0x5b5a85ed2790 -> node0x5b5a85ed2750 [label="src"];
node0x5b5a85ed2890 -> node0x5b5a85ed2850 [label="left"];
node0x5b5a85ed2890 -> node0x5b5a85ed2790 [label="right"];
node0x5b5a85ed2810 [label="LABLE end"];
node0x5b5a85ed28d0 -> node0x5b5a85ed2890 [label="left"];
node0x5b5a85ed28d0 -> node0x5b5a85ed2810 [label="right"];
}
```

测试结果

1. testcase1:简单赋值语句

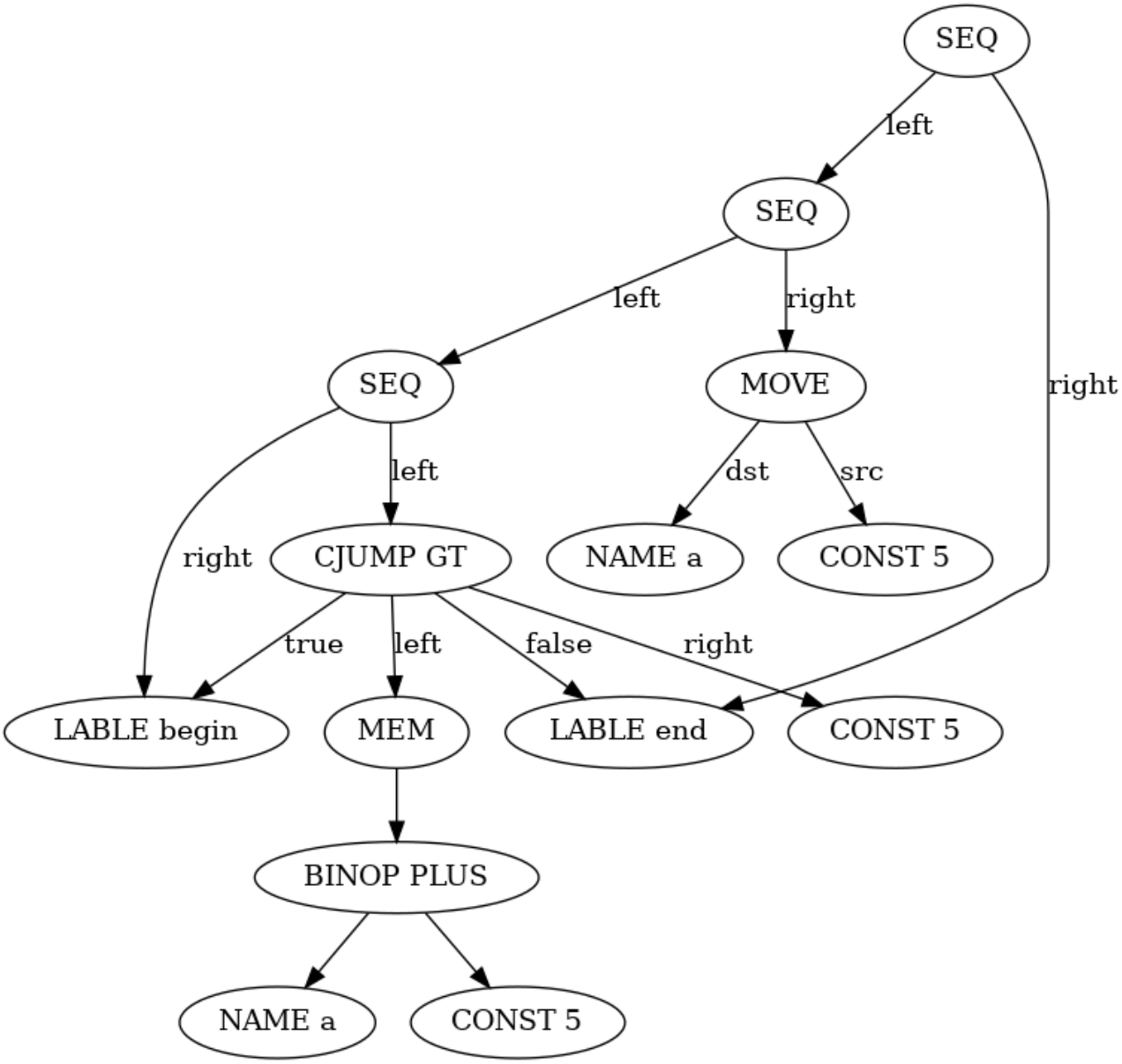
test > testcase1.txt

```
1  a:=5
```



2. testcase2:简单if语句

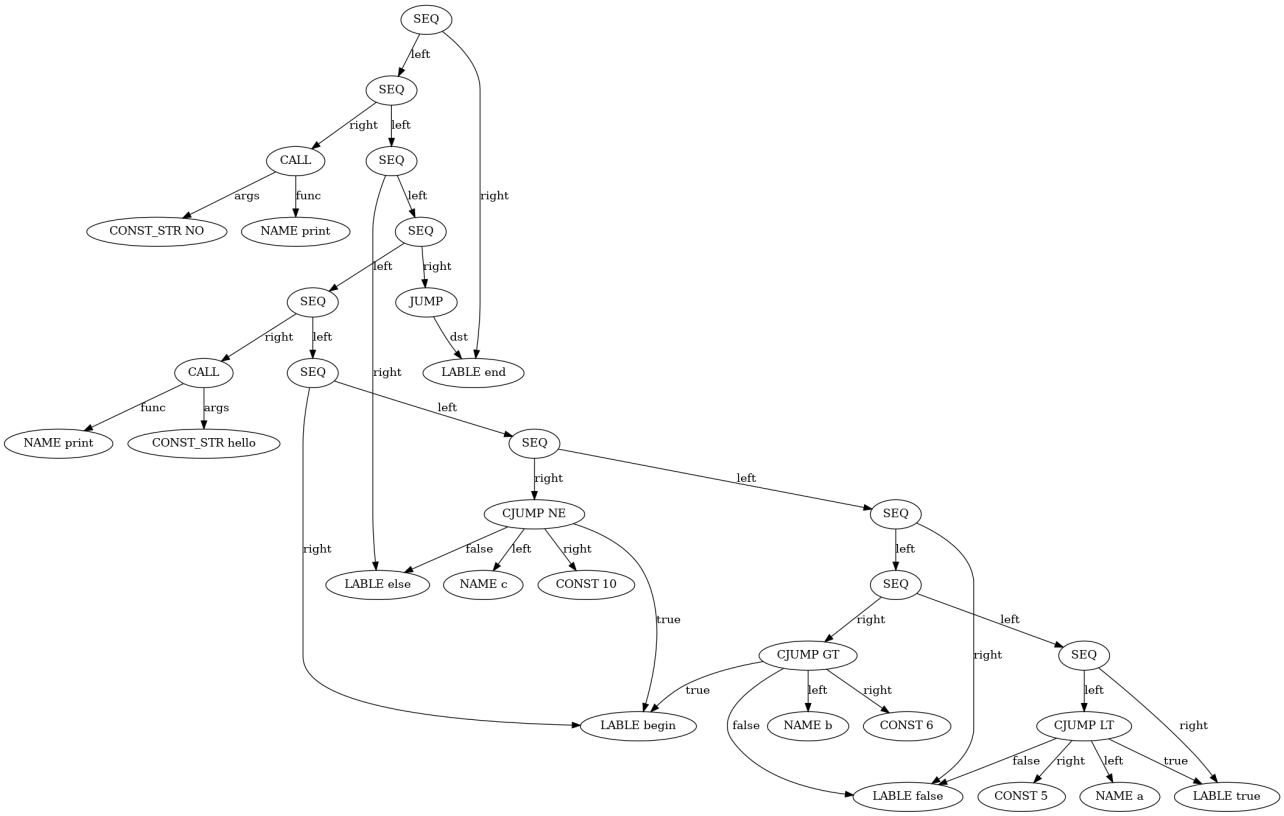
```
if a[5] > 5
then a:=5
```



3. testcase3:复杂if语句

st > testcase3.txt

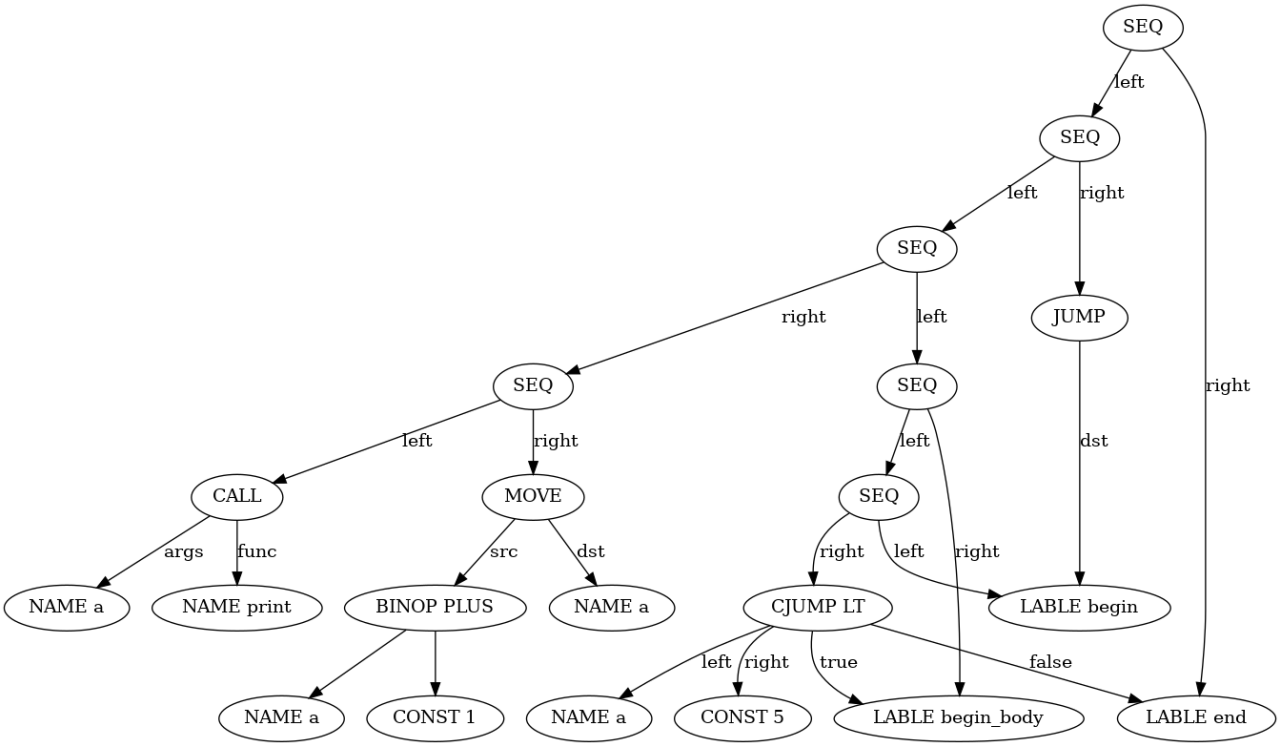
```
1  if a<5 & b>6 | c<> 10
2  then print("hello")
3  else print("NO")
```



4. testcase4:while语句

est > testcase4.txt

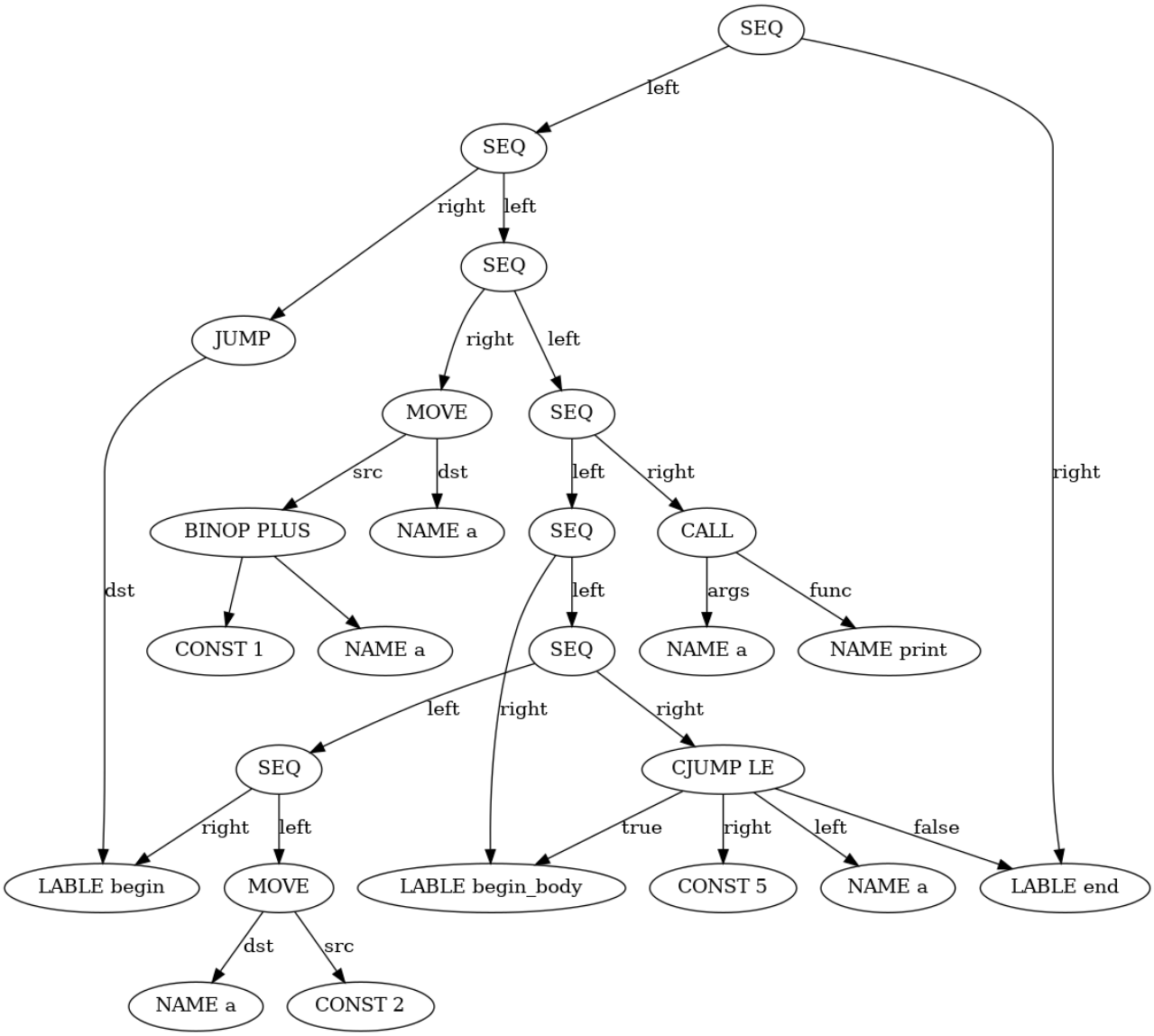
```
1  while a<5 do
2  (print(a);
3  a:=a+1
4  )
```



5. testcase5: for语句

test > testcase5.txt

```
1  for a:=2 to 5
2  do print(a)
```

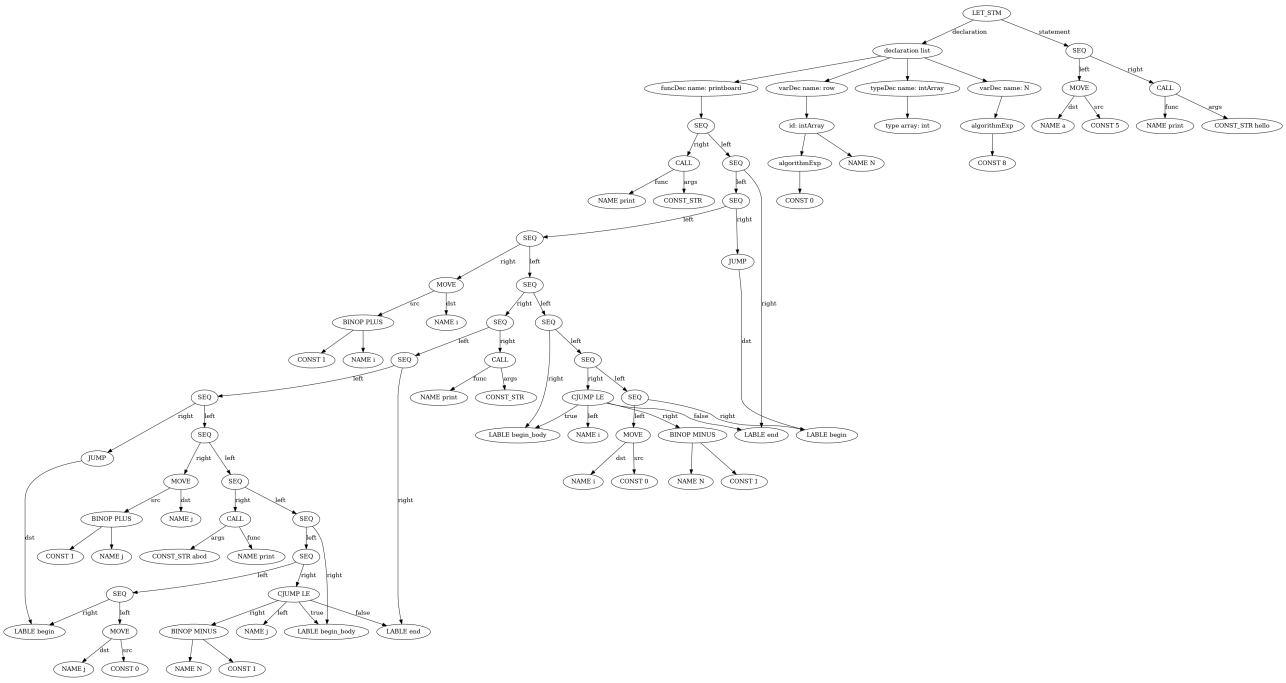


6. testcase6: 复杂let语句

testcase6.txt

```
let
    var N := 8
    type intArray = array of int
    var row := intArray [ N ] of 0
function printboard() =
    (for i := 0 to N-1
     do (for j := 0 to N-1
        do print("abcd");
        print("\n"));
    print("\n"))
in (
    a:=5;
    print("hello")
) end

```



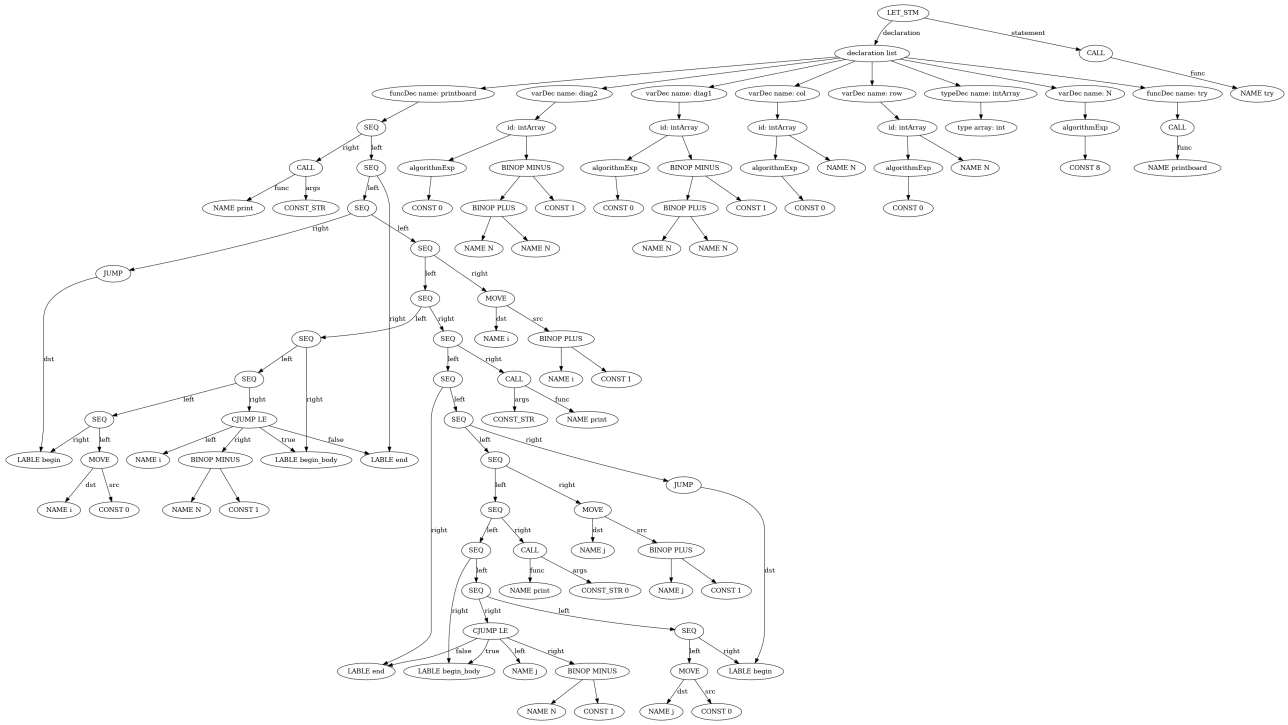
7. testcase7: 复杂let语句(2)

```
let
    var row := intArray [ N ] of 0
    var col := intArray [ N ] of 0
    var diag1 := intArray [ N+N-1 ] of 0
    var diag2 := intArray [ N+N-1 ] of 0

    function printboard() =
    (
        for i := 0 to N - 1
        do (
            for j := 0 to N - 1
            do print("0" );
            print("\n")
        );
        print("\n")
    )

    function try() =
    (
          printboard()
    )


in
    try()
end
```



8. testcase8: let语句

```
1  let
2  var N:=8
3  function fact(n:int,y:int)=
4  ( if n=0 then print("\n"))
5  in fact(n)
6  end
```