编译器设计实验报告

2019202463李厚润

1. 实验目标

- 完成一个简单的基于PL0语言的编译器
- 拓展功能：**支持for语句**

2. PL0文法

```
<程序> ::= <分程序>.
<分程序> ::= [<常量说明部分>][<变量说明部分>][<过程说明部分>]<语句>
<常量说明部分> ::= CONST<常量定义>{,<常量定义>};
<常量定义> ::= <标识符>=<无符号整数>
<无符号整数> ::= <数字>{<数字>}
<变量说明部分> ::= VAR<标识符>{,<标识符>};
<标识符> ::= <字母>{<字母>|<数字>}
<过程说明部分> ::= <过程首部><分程序>{;<过程说明部分>};
<过程首部> ::= PROCEDURE<标识符>;
<语句> ::= <赋值语句>|<复合语句>|<条件语句>|<当型循环语句>|<过程调用语句>|<读语句>|<写语句>|<空>
<赋值语句> ::= <标识符>:=<表达式>
<复合语句> ::= BEGIN<语句>{;<语句>}END
<条件> ::= <表达式><关系运算符><表达式>|ODD<表达式>
<条件语句> ::= IF<条件>THEN<语句>
<表达式> ::= [+|-]<项>{<加法运算符><项>}
<项> ::= <因子>{<乘法运算符><因子>}
<因子> ::= <标识符>|<无符号整数>|'('<表达式>')'
<加法运算符> ::= +|-
<乘法运算符> ::= *|/
<关系运算符> ::= =|#|<|<=|>|>=
<当型循环语句> ::= WHILE<条件>DO<语句>
<过程调用语句> ::= CALL<标识符>
<读语句> ::= READ'('<标识符>{,<标识符>}')'
<写语句> ::= WRITE'('<表达式>{,<表达式>}')'
<字母> ::= a|b|...|X|Y|Z
<数字> ::= 0|1|...|8|9
```

3. 文件说明

- lexical_analysis .l 是词法分析部分，和语法分析类似
- syntax_analysis.y是语法/语义分析部分

4. 词法分析器设计

- 第一次词法分析实验代码在后面几乎没有任何作用，删除重写，所以就不按照之前的讲了。
- 利用正则表达式来匹配标识符、关键词等等字符：
    - 单字符类型

```
digit_ch   [0-9]
alpha_ch   [A-Za-z]
delimiter_ch    (\.\.\.)|[\,\.\:\;\(\)\[\]\{\}]
blank_ch    (" ")|(\\t)|(\\r)|(\\n)|(\\a)|(\\b)|(\\f)|(\\v)|(\\0)|(\\\\)
special_ch  [\!\@\#\$\%\^\&\"\'\~\|\_]
operator_ch    [\+\-\*\/\=\<\>]|(\>\=)|(\<=)|(\:\=)|(\!\=)
```

- 组合类型

```
keyword
if|then|while|do|read|write|call|begin|end|const|var|procedure|odd|integer|float|char|integerarray|for|to|downto|repeat|until
identifier  ({alpha_ch}|\_)({alpha_ch}|{digit_ch}|\_)*
integer {digit_ch}+
float   {integer}(\.{integer})?([E|e][+|-]?{integer})?
character   (\'({digit_ch}|{alpha_ch}|[\,\.\:\;\(\)\[\]\{\}]|{blank_ch}|{special_ch}|[\+\-\*\/\=\<\>])\')|(#{integer})
constant {integer}
```

- **匹配串并处理**：如果匹配到了其中的某些特定类别字符串，比如关键词、常数、标识符等等，我们需要进一步细化匹配究竟是匹配了这一大类里的具体哪一个，然后return返回；或者我们需要进一步对yyvalue进行赋值。yyvalue的类型是yystype，默认是int类型，其中对类型的修改见.y语法/语义分析文件。其中每次调用一次yylex函数就会return匹配到的一个特定字符串。其中yylex函数具体应用在后续的词法分析和语义分析都会不断调用yylex。
- return的大写关键词本质是宏，定义在syntax_analysis.y中。

```
{keyword} {
    if(!strcmp(yytext, "if")) return IF;
    if(!strcmp(yytext, "then")) return THEN;
    if(!strcmp(yytext, "while")) return WHILE;
    if(!strcmp(yytext, "do")) return DO;
    if(!strcmp(yytext, "read")) return READ;
    if(!strcmp(yytext, "write")) return WRITE;
    if(!strcmp(yytext, "call")) return CALL;
    if(!strcmp(yytext, "begin")) return BEGIN_;
    if(!strcmp(yytext, "end")) return END;
    if(!strcmp(yytext, "const")) return CONST;
    if(!strcmp(yytext, "var")) return VAR;
    if(!strcmp(yytext, "procedure")) return PROCEDURE;
    if(!strcmp(yytext, "odd")) return ODD;
    if(!strcmp(yytext, "integer")) return INTEGER;
    if(!strcmp(yytext, "integerarray")) return INTEGERARRAY;
    if(!strcmp(yytext, "for")) return FOR;
    if(!strcmp(yytext, "to")) return TO;
    if(!strcmp(yytext, "downto")) return DOWNTO;
    if(!strcmp(yytext, "repeat")) return REPEAT;
    if(!strcmp(yytext, "until")) return UNTIL;
}

{identifier} {
    yylval.name = strdup(yytext);
    return ID;
}

```

```
{integer} {
    int t = atoi(yytext);
    yylval.valInteger = t;
    return CONSTANT;
}
```

- 对于空白符，移动指针足矣

```
[" "\t\r] ;
[\n] {yrow++;}
. ;
```

- 其中定义了`%option noyywrap`表示优化选项，不需要wrap函数。

5. 语法分析器的设计

- **由于语法分析器不支持[] 表示可选项， {} 表示重复项等符号，以及存在一些冲突，所以需要我们改写。**
- 只需要根据老师词法分析的课件上所给的PL0文法改写即可。
- 比如`<表达式>{,<表达式>}`我们可以用`<表达式列表>`来代替，然后`<表达式列表> ::= <表达式>`
  `<表达式列表> ::= <表达式>, <表达式列表>`即可表示原来的含义。

1. **基础支持**

```
<表达式> ::= <项列表>
<表达式> ::= <加法运算符> <项列表>

<项列表> ::= <项>
<项列表> ::= <项> <加法运算符> <项>

<无符号整数> ::= CONSTANT

<项> ::= <因子>
<项> ::= <因子> <乘法运算符> <因子>

<因子> ::= <标识符>
<因子> ::= <无符号整数>
<因子> ::= '('<表达式>')'

<标识符列表> ::= <标识符>
<标识符列表> ::= <标识符>, <标识符列表>

<表达式列表> ::= <表达式>
<表达式列表> ::= <表达式>, <表达式列表>
```

2. **程序架构**

```
<程序> ::= <分程序>.

<分程序> ::= <常量说明部分> <变量说明部分> <过程说明部分> <语句>

<常量说明部分> ::= <空>
<常量说明部分> ::= CONST <常量定义>;

<变量说明部分> ::= <空>
<变量说明部分> ::= VAR <变量定义>;

```

```
<过程说明部分> ::= <空>
<过程说明部分> ::= <过程首部> <分程序>; <过程说明部分>

<过程首部> ::= PROCEDURE <标识符>;

<语句列表> ::= <语句>
<语句列表> ::= <语句> ; <语句列表>

<语句> ::= <赋值语句>
<语句> ::= <复合语句>
<语句> ::= <条件语句>
<语句> ::= <while语句>
<语句> ::= <过程调用语句>
<语句> ::= <for语句>
<语句> ::= <读语句>
<语句> ::= <写语句>
<语句> ::= <空>

<条件> ::= <表达式> <关系运算符> <表达式>
<条件> ::= ODD<表达式>

<加法运算符> ::= +
<加法运算符> ::= -
<乘法运算符> ::= *
<乘法运算符> ::= /

<关系运算符> ::= =
<关系运算符> ::= #
<关系运算符> ::= <
<关系运算符> ::= <=
<关系运算符> ::= >
<关系运算符> ::= >=
```

3. **常量变量定义**

```
<常量定义> ::= <普通常量定义>
<常量定义> ::= <普通常量定义>, <常量定义>

<普通常量定义> ::= <标识符> = <无符号整数>

<变量定义> ::= <标识符>
<变量定义> ::= <标识符>, <变量定义>
```

4. **语句**

```
<while语句> ::= WHILE <条件> DO <语句>

<过程调用语句> ::= CALL <标识符>

<复合语句> ::= BEGIN <语句列表> END

<读语句> ::= READ'('<标识符列表>')'

<写语句> ::= WRITE'('‘<表达式列表>’)’

<赋值语句> ::= <标识符>:=<表达式>

```

```
<条件语句> ::= IF <条件> THEN <语句>

<for语句> ::= FOR <标识符> := <表达式> TO <表达式> DO <语句>
<for语句> ::= FOR <标识符> := <表达式> DOWNTO <表达式> DO <语句>
```

5. 终结符：
    - 用**%token标注的就是终结符**，我们需要到词法分析器中加以修改，然后**每一次调用yylex函数返回一个这样的token**，这就是在词法分析器中提到的宏。
    - ```
      %token IF THEN WHILE DO READ WRITE CALL BEGIN_ END CONST VAR
      PROCEDURE ODD INTEGER FLOAT CHAR INTEGERARRAY FOR TO DOWNTO REPEAT
      UNTIL
      %token ID CONSTANT PLUS MINUS MULTIPLY DIVIDE EQ GT LT GE LE ASSIGN
      NEQ
      %token DOTS COMMA DOT SEMICOLON COLON L0 R0 L1 R1 L2 R2
      ```
    - 对于其他的非终结符，我们只需要一个符号来表示，不需要定义%type（这是后续语义分析需要做的事）。然后定义相关的规则动作就可以了。其中这个工作难度低，重复度高，就是**按照给的EBNF范式机械翻译**，没啥意思。
    - 举个栗子：
        - 对于文法`<程序> ::= <分程序>.`，我们只需要如下翻译即可。其中有两个儿子节点，分别是`<分程序>`和`.`。那么我们通过sechild加入即可。
        - 由于这个文法是最终的规约，规约开始符号program，所以我们还需要额外的动作：打印树。其他的规约只需要输出利用到的规约文法即可。
        - ```
          program : subprogram DOT
                    {
                        $$ = newnode("<程序>");
                        setchild($$, $1);
                        setchild($$, $2);
                        fprintf(opraw, "<程序> -> <分程序>.\n");
                        fprintf(optree, "digraph pic{\n");
                        outtree($$, NULL, 0);
                        fprintf(optree, "}\n");
                    }
                  ;
          ```
        - 具体详情操作见syntax_analysis.y文件

5.2 主函数main说明

- ```
  int main(int argc, char **argv){
      yyin = fopen(argv[1], "r");
      opraw = fopen(argv[2], "w");
      optree = fopen(argv[3], "w");
      yyparse();
      return 0;
  }
  ```
- 接受main的参数，对输入输出进行重定向
- 其中yyparse会默认调用yylex函数。
  ```

5.3语法树说明

- 语法树的每一个节点是node类型
- 其中基本操作是：

```
#define YYSTYPE node*
node* newnode(char *str){
    node* tmp = new node;
    tmp->text = strdup(str);
    return tmp;
}

void setchild(node *fa, node *ch){
    (fa->ch).push_back(ch);
}

void outtree(node *cur, node *fa, int faid){
    int id = ++num;
    fprintf(optree, "Node%d[label=\"%s\"]\n", id, cur->text);
    if(fa != NULL)
        fprintf(optree, "Node%d -> Node%d\n", faid, id);
    fprintf(optree, "\n");
    for(auto i: cur -> ch){
        outtree(i, cur, id);
    }
}
```

- 我们需要重新定义**YYSTYPE为node*类型**，方便我们在词法和语法分析中使用。
- **每一个node节点，包含一个字符串信息和所有孩子指针。**孩子指针是通过vector存储，方便记录不同数量的孩子。

5.4语法树的绘制

- 我们利用outtree来输出语法树的文字形式
- 利用graphviz 的 dot 模块绘制树形图 ，其中按照先后顺序遍历编号，用->控制连边关系。
- label是标签，用来展示在图形中
- 用`dot optree.out -T png -Gdpi=100 -o optree.png`一条命令即可

5.5编译方式指令

```
bison -vd syntax_analysis.y
flex lexical_analysis.l
g++ -std=c++11 -o parse lex.yy.c syntax_analysis.tab.c
./parse test.pl0 opraw.out optree.out
dot optree.out -T png -Gdpi=100 -o optree.png
```

- 由于我们使用了vector等c++独有结构以及new函数，所以我们需要g++编译。
- 但是由于其实之前的flex和bison只是对c进行支持，**在bison的.y文件中对yylex等外部函数或者变量进行调用的时候，需要添加extern “C”字样**，例如：

```
extern FILE * yyin;extern int yylineno;extern "C" int yylex();
```

6. 语义分析

1. **栈值类型更改**：类似于语法分析，我们同样需要更改语法分析栈值的类型，不同于语法分析（语法分析只需要按部就班分析语法和输出就可以），语义分析需要相关的综合属性，所以我们需要更改这个类型：`#define YYSTYPE SuperNode`。Supernode就是我们更改后的栈值类型，定义如下：

```
struct SuperNode{
    char* name;
    int typeH; /*the first tyoe*/
    int typeS; /*the second type*/
    int codeAddr;/*the location of the first code of a procedure*/
    int valInteger; /*the value of variable of constant*/
    Node *list; /*the nextlist*/
};
```

2. **非终结符类型说明**：类似于语义分析对终结符进行一系列声明，我们还需要对部分非终结符进行类型说明：

```
/*the name of identifier*/
%type<name> ID
/*the type of expression*/
%type<typeS> plusop multop expression termlist term factor relop
/*the value of a caontant variable*/
%type<valInteger> CONSTANT
/*the nextlist, used for backpatch*/
%type<list> statement assignsta compsta statementlist ifsta condition whilesta forsta progcallsta readsta writesta
/*the anchor, used for backpatch*/
%type<codeAddr> checkpoint1
```

3. **代码表示**：我们对于每一条指令代码，f表示指令，l表示层差，a表示偏移，通过如下形式表达：

```
struct Instructions{
    char *f;
    int l;
    int a;
}code[MAX_INSTRUCTIONS];
```

4. **代码生成**：我们通过如下函数生成代码：

```
/*generate the code*/
void gen(char *x, int l, int a){
    code[curIns++] = (Instructions){strdup(x),l,a};
    /*make the f in lower case*/
    for(int i = 0; code[curIns - 1].f[i] != 0; i++){
        if(code[curIns - 1].f[i] <= 'Z')
            code[curIns - 1].f[i] += 'a' - 'A';
    }
}
```

5. **标识符表示**：我们定义了结构体Symbol来存放标识符，其中在symbolTable中存放大量标识符信息。

```
struct Symbol{
    char* name;
    int typeH;/*the first type*/
    int typeS;/*the second type*/
    int level;/*the level it lays in*/
    int offset;
    int size;
    int codeAddr;
    int valInteger;/*value*/
}symbolTable[MAX_SYMBOL_TABLE_ENTRIES];
```

6. **标识符注册**：我们有了如上的表示记录，我们就需要专门的注册函数对新出现的标识符进行注册。

```
Symbol *registSymbol(char *name, int typeH, int typeS, int val){
    /*only the variable will occupy the stack*/
    if(typeH == VARIABLE_T)
        curDataStackSize++;
    Symbol tmp;
    tmp.name = strdup(name);
    tmp.typeH = typeH;
    tmp.typeS = typeS;
    tmp.level = curLevel;
    tmp.size = 1;
    if(typeH == VARIABLE_T){
        /*if it is the first variable or the local first variable in a procedure*/
        if(curSymbolTable == 0 || symbolTable[curSymbolTable - 1].level != curLevel || symbolTable[curSymbolTable - 1].typeH != VARIABLE_T)
            tmp.offset = PROG_OCCUPIED;
        else
            tmp.offset = symbolTable[curSymbolTable - 1].offset + symbolTable[curSymbolTable - 1].size;
    }else if(typeH == CONSTANT_T){
        tmp.offset = 0;
        tmp.valInteger = val;
    }
    symbolTable[curSymbolTable++] = tmp;
    return &symbolTable[curSymbolTable - 1];
}
```

7. **标识符存取、查找**：通过lookupSymbol来查找标识符，stoid来存储标识符，lodid来取出标识符

```
Symbol *lookupSymbol(char *name){
    for(int i = curSymbolTable - 1; i >= 0; i--)
        if(strcmp(name, symbolTable[i].name) == 0)
        return &symbolTable[i];
    return NULL;
}
Symbol* lodid(char *name){
    //fprintf(stderr, "%s\n", name);
    Symbol *tmp = lookupSymbol(name);
    if(tmp == NULL){
        fprintf(stderr, "404 %s\n", name);
        exit(0);
    }
    if(tmp -> typeH == VARIABLE_T)
```

```
        gen("LOD", curLevel - tmp->level, tmp->offset);
    else{
        /*If the var is a constant var*/
        gen("LIT", 0, tmp->valInteger);
    }
    return tmp;
}
void stoid(char *name){
    Symbol *tmp = lookupSymbol(name);
    if(tmp == NULL){
        fprintf(stderr, "404 %s\n", name);
        exit(0);
    }gen("STO", curLevel - tmp->level, tmp->offset);
}
```

8. backpatch代码回填管理：由于条件表达式和控制流跳转都需要回填技术，所以我们需要维护一些链表，在某一时刻填入跳转目标地址。

```

/*There will be a cycle*/
struct Node* linkList(Node *ptr1, Node *ptr2){
    if(ptr1 == NULL) return ptr2;
    if(ptr2 == NULL) return ptr1;
    Node *ptr1n = ptr1 -> next;
    Node *ptr2n = ptr2 -> next;
    ptr1 -> next = ptr2n;
    ptr2n -> pre = ptr1;
    ptr2 -> next = ptr1n;
    ptr1n -> pre = ptr2;
    return ptr1;
}

struct Node* makeList(int *ptr){
    Node *res = (Node *)malloc(sizeof(Node));
    res -> ptr = ptr;
    res -> pre = res -> next = res;
    return res;
}

struct Node* insList(Node *list, int *ptr){
    Node *list2 = makeList(ptr);
    return linkList(list, list2);
}

void backPatch(Node *list, int x){
    if(list == NULL) return;
    Node *tmp = list;
    do{
        *(tmp->ptr) = x;
        tmp = tmp->next;
    }while(tmp != list);
}
```

9. **基于SDT进行语法动作翻译**：
    - 整个语法规约方法和上次几乎一模一样，稍有一丁点改变，这里就不详细列举了。

- 但是其中会加入许多语义动作，而且不同于语法分析的固定翻译，这次会更加具体且会更难，稍不注意就会有bug。这里的内容很多，我们只需要挑一些具有代表性的介绍一下：

```
subprogram : const_cv var_cv {
            /*Figure out the DataStackSize when compiling*/
                dataStackSize[curLevel] = curDataStackSize;
             } proc_cv {
                if(curLevel > 0)
                /*the first ins of the current procedure*/
                    funcStack[curLevel - 1] -> codeAddr = curIns;
                else
                /*directly enter the master procedure*/
                    code[0].a = curIns;
                /*Initialize the stack*/
                gen("INT", 0, dataStackSize[curLevel]);
             } statement {
                /*backpatch the list of statement*/
                backPatch($6, curIns);
                gen("OPR", 0, 0);
                curDataStackSize = PROG_OCCUPIED;
                /*delete the symbols occured in the current but ending procedure*/
                deleteSymbol();
             }
           ;
```

```
forsta : /*for to formula*/
          FOR ID ASSIGN expression{
                stoid($2);
          } TO checkpoint1{
                lodid($2);
          } expression checkpoint1{
              /*check boundary*/
                gen("OPR", 0, 10);
                gen("JPC", 0, 0);
          } DO statement {
              /*the next code of statement is modifying the value of ID*/
                backPatch($13, curIns);
                lodid($2);
                gen("LIT", 0, 1);
                gen("OPR", 0, 2);
                stoid($2);
                /*goto the first chckpoint1 and check the boundary*/
                gen("JMP", 0, $7);
                /*the nextlist of forsta equals to the nextlist of falselist of expression*/
                $$ = makeList(&code[($10)+1].a);
          }
        /*for downto formula*/
       | FOR ID ASSIGN expression{
                stoid($2);
          } DOWNTO checkpoint1{
                lodid($2);
          } expression checkpoint1{
                gen("OPR", 0, 12);
```

```
                gen("JPC", 0, 0);
          } DO statement {
                backPatch($13, curIns);
                lodid($2);
                gen("LIT", 0, 1);
                gen("OPR", 0, 3);
                stoid($2);
                gen("JMP", 0, $7);
                $$ = makeList(&code[($10)+1].a);
          }
        ;
```

10. **pcode代码与代码解释器**：
    - 我们需要一个代码解释器生成c语言，然后用c语言进行编译。interpret只需要不足一些结构和主函数即可。

7. 编译运行方法：

- 编译命令

```
flex -o LA.cpp lexical_analysis.l
yacc -d syntax_analysis.y
g++ y.tab.c -o y
g++ interpret.cpp -o I
```

- 运行方法：

```
./y test.pl0 test.pcode
./I test.pcode
```

8.测试文件说明

- test.pl0是一个有三层嵌套的复杂pl0测试代码
- for.pl0是一个测试for语句的代码

完结!

END!