School of Data and Computer Science, Sun Yat-sen University

编译原理实验三：词法分析和语法分析

18340180 谢健聪

2021 年 4 月 12 日

目录

1 任务要求

词法分析、语法分析程序实验:

1. 实验目的:扩充已有的样例语言 TINY,为扩展 TINY 语言 TINY +构造词法分析和语法分析程序,从而掌握词法分析和语法分析程序的构造方法

2. 实验内容:了解样例语言 TINY 及 TINY 编译器的实现,了解扩展 TINY 语言 TINY +,用 EBNF 描述 TINY +的语法,用 C 语言扩展 TINY 的词法分析和语法分析程序,构造 TINY +的语法分析器。

3. 实验要求:将 TINY +源程序翻译成对应的 TOKEN 序列,并能检查一定的词法错误。将 TOKEN 序列转换成语法分析树,并能检查一定的语法错误

2 编译器构造过程

此次任务采用两个工具实现,分别是 Lex 和 YACC,前者完成词法分析工作,后者则完成语法分析。然后结合 C 语言读取文本进行分析并且输出 TOKEN 序列和打印语法分析树。在借助 Lex 和 YACC 工具的前提下,整个过程分为两个步骤,词法定义和语法定义。

2.1 词法定义

在 Lex 中,代码分为 3 个模块,第一个模块为声明段,其中声明头文件和定义符号。对于符号的定义,采用正则表达式的方式进行定义。为了满足 TINY 基本的词法需求,我定义了如下符号:

```
1         /*非数字由大小写字母、下划线组成*/
2         nondigit    ([_A-Za-z])
3
4         /*一位数字,可以是0到9*/
5         digit       ([0-9])
6
7         /*整数由1至多位数字组成*/
8         integer     ({digit}+)
9
10        /*标识符,以非数字开头,后跟0至多个数字或非数字*/
11        identifier  ({nondigit}({nondigit}|{digit})*)
12
13        /*不带双引号的任意字符串*/
14        qstring (\"[^"\n]*["\n])
15
16        /*一个或一段连续的空白符*/
17        blank_chars ([ \f\r\t\v]+)
18
19        /*关键字*/
20        key_words (READ|WRITE|IF|ELSE|RETURN|BEGIN|END|STRING|REAL|INT|TO|FOR|WHILE|ENDIF|ENDFOR|ENDWHILE)
```

```
21        mul_char (:=|==|!=)
```

接下来的模块为规则段，即 Lex 在处理真实文本，遇到上述定义的符号时采取的动作。此处我采取两个动作，分别是输出遇到的 TOKEN，并将该 TOKEN 传递给语法分析器。对于后者，承载的变量为 **yylval**，Lex 与 YACC 默认定义为一个整型，对于 char 类型数据，**yylval** 为其实际 ASCII 值，对于其他数据，**yylval** 从 256 开始进行计数。

为了方便构造语法树，我重定义该变量为结构体，其中包含三种数据类型，分别为 int,char,string 和 Node 指针（实际上 char 类型可以去掉）。

```
1         struct Type
2         {
3                 string m_Str;
4                 char m_cOp;
5                 int m_nInt;
6                 Node* nptr;
7         };
8         #define YYSTYPE Type
```

其中 int,char,string 表示终结符，而 Node 指针表示非终结符。Node 结构体定义如下，Node 实则语法树中的节点，所以包含不同类型（int,string,char）的节点，并且用 vector 存储分支节点。

```
1         typedef enum { typeCon, typeOpr, typeKey} nodeEnum;
2         struct Node{
3                 nodeEnum type;
4                 int interger;
5                 string key_word;
6                 char opr;
7                 vector<Node*> child;
8         };
```

此时可以定义每个 TOKEN 所采取的动作。遇到符合规则的 TOKEN 时，将其中的值放入 yylval 结构体中，并定义一个 TOKEN 符号返回给 Yacc，并且输出该 TOKEN。以关键字和标识符为例。

```
1         {key_words} {
2                 yylval.m_Str = yytext;
3                 cout<<"<key word,"<<yytext<<">\n";
4                 if(!strcmp(yytext,"READ"))
5                         return READ;
6                 else if(!strcmp(yytext,"WRITE"))
7                         return WRITE;
8                 else if(!strcmp(yytext,"IF"))
9                         return IF;
10                else if(!strcmp(yytext,"ELSE"))
11                        return ELSE;
12                else if(!strcmp(yytext,"RETURN"))
13                        return RETURN;
```

```
14                 else if(!strcmp(yytext,"BEGIN"))
15                         return BEGIN_;
16                 else if(!strcmp(yytext,"END"))
17                         return END;
18                 else if(!strcmp(yytext,"STRING"))
19                         return STRING;
20                 else if(!strcmp(yytext,"REAL"))
21                         return REAL;
22                 else if(!strcmp(yytext,"INT"))
23                         return INT;
24         }
25
26         {identifier}    {
27                 yylval.m_Str=yytext;
28                 cout<<"<id,"<<yytext<<">\n";
29                 return IDENTIFIER;
30         }
```

对于注释段，也在词法分析模块里处理。

```
1          "/*" {
2                  BEGIN COMMENT;
3          }
4          <COMMENT>"*/" {
5                  BEGIN INITIAL;
6          }
```

2.2 语法定义

在 Yacc 对语法进行预处理之前，首先要“接受”或者说“定义”Lex 返回过来的变量。对于非终结符，也要定义其类型为 Node 指针，如下：

```
1          %token<m_nInt>INTEGER
2          %token<m_Str>QSTRING
3          %token<m_Str>IDENTIFIER
4          ...
5
6          %type<nptr>MethodDecl
7          %type<nptr>FormalParams
8          ...
9          %type<nptr>Type
```

接下来，就可以定义文法的产生式，Yacc 会采用自底向上的移进-规约算法，根据产生式进行匹配。首先用 EBNF（巴科斯范式）定义文法。

首先明确一些终端符号：

- 常量：实数 number，字符串 qstring
- 单字符号：( ) , ; + - * /
- 多字符号：:= !=
- 关键字：READ,WRITE,IF,ELSE,RETURN,BEGIN,END,STRING,REAL,INT

3 TINY+ 语法（自定义）

在原有 TINY 语法的基础上，我做出了如下扩展：

- IF 条件语句更改对于原有的 IF 条件语句：

$$IfStmt \rightarrow IF\ '('\ BoolExpression\ ')'\ Statement\ |$$
$$\rightarrow IF\ '('\ BoolExpression\ ')'\ Statement\ ELSE\ Statement$$

问题在于出现移进-规约冲突，若出现 IF...IF...ELSE 的语法，则无法判断 ELSE 属于哪个 IF 语句。为了解决冲突，我引入条件语句结束符 **ENDIF** 来划分 IF 的作用范围，则可以解决上述冲突问题。若是 IF...IF...ELSE...ENDIF...ENDIF，则 ELSE 匹配第二个 IF，若 IF...IF...ENDIF..ELSE...ENDIF，则 ELSE 匹配第一个 IF。IF 条件语句更改为：

$$IfStmt \rightarrow IF\ '('\ BoolExpression\ ')'\ Statement\ ENDIF|$$
$$\rightarrow IF\ '('\ BoolExpression\ ')'\ Statement\ ELSE\ Statement\ ENDIF$$

- For 循环语句

$$ForStmt \rightarrow$$
$$For\ '('\ identifier\ ":="\ integer\ to\ integer\ ')'\ Statement\ EndFor$$

- While 循环语句

$$WhileStmt \rightarrow$$
$$while\ '('\ BoolExpression\ ')'\ Statement\ EndWhile$$

4 语法树构建

在 Yacc 中，每个产生式后可以执行相应的动作，便可以在匹配一个产生式，或者说归结该产生式时，构建一棵语法子树。由于 Yacc 采用的是自底向上的移进规约算法，则涉及符号栈，其中 Yacc 定义：**$$** 表示归结后的栈顶，**$n** 表示归结前从栈顶数往下第 n 个值。由于我们先前定义每个符号的类型（非终结符为 Node 指针，终结符为 int 或者 string 或者 char），则我们可以从符号得知栈里相应的值的类型。

对于非终结符，我们要构造其 Node 节点：

```
        //integer终结符生成Node节点
        Node* int_node(int value){
                Node* p = new Node();
                p->interger = value;
                p->type = typeCon;
                return p;
        }
        //关键字终结符
        Node* key_node(string keyword){
                Node* p = new Node();
                p->key_word = keyword;
                p->type = typeKey;
                return p;
        }
        //单个符号终结符
        Node* op_node(char c){
                Node* p = new Node();
                p->opr = c;
                p->type = typeOpr;
                return p;
        }
```

语法子树的生成，利用 merge 函数，第一个参数为该节点的名字，第二个参数表示有 n 个分支，接下来 n 个参数表示 n 个分支节点。

```
        Node* merge(string s, int n,...){
                Node* p = new Node();
                p->type = typeKey;
                p->key_word = s;
                va_list ap;
                va_start(ap, n);
                for(int i=0;i<n;i++)
                        p->child.push_back(va_arg(ap, Node*));
                va_end(ap);
                return p;
        }
```

则在产生式进行归结时，则构建语法子树，以

$$Block \rightarrow BEGIN\ Statement\ END$$

为例，在归结前，栈中从顶往下分别为（关键字 BEGIN，string 类型），（非终结符 Statement，Node 指针类型）,(关键字 END，string 类型)，则对于终结符，需要构造出 Node 节点，对于非终结符，直接使用其 Node 节点，然后构建父节点 Block，构造出子树。

```
Block:
```

```
2         BEGIN_ Statement END {$$=merge("Block",3,key_node("BEGIN"),$2,key_node("END"));}
```

对于关键字终结符，我们知道该关键字是什么，则构造 Node 节点时可以直接赋值关键字的值，对于标识符我们不知道其值为多少，如对于产生式

$$FormalParam \rightarrow Type\ IDENTIFIER$$

此时标识符我们可以从栈中取出，为 string 类型，则同样构建 Node 节点和语法子树

```
1         FormalParam:
2     Type IDENTIFIER {$$=merge("FormalParam",2,$1,key_node($2));}
```

对于整个语法的起始状态，进行归结时说明语法正确，则此时生成整个语法树的根节点，并输出语法树。

```
1         MethodDecl:
2     Type IDENTIFIER '(' FormalParams ')' Block MethodDecl
3     {
4         Node* root = merge("MethodDecl",7,$1,key_node($2),op_node('('),$4,op_node(')'),$6,$7);
5         print(root,(height(root)+1)*15);
6     }
7         | {...}
```

5 结果展示

由于语法树过于庞大，我们先测试简单的一些语句，观测 token 序列和语法树的结构。

- TEST1

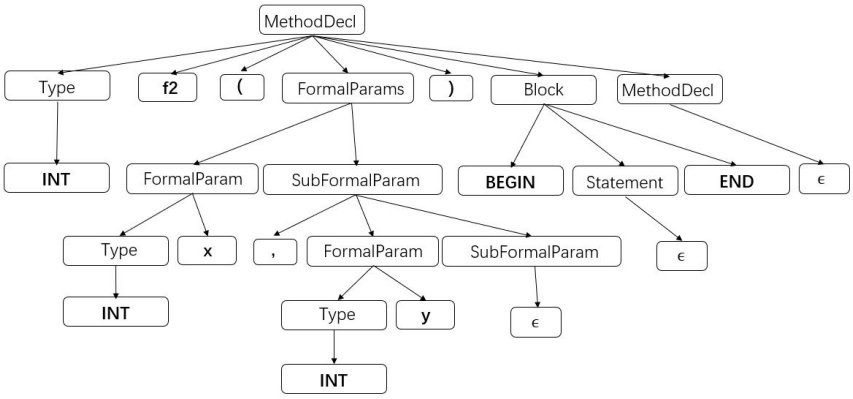
```
1  INT f2(INT x, INT y )
2  /*this is comment*/
3  BEGIN
4  END
```

结果：其中第一部分为 token 序列，第二部分为语法树，最右边的为根节点，整棵树的结果实则往顺时针旋转了 90 度。可视化后语法树如下：

- TEST2

```
1  INT f1(INT x)
2  BEGIN
3          INT z := x*x ;
4          RETURN z;
5  END
```

```
$ ./main file.txt
-----begin parsing file.txt
<key word,INT>
<id,f2>
<Single-character operators,(>
<key word,INT>
<id,x>
<Single-character operators,,>
<key word,INT>
<id,y>
<Single-character operators,)>
this is comment<key word,BEGIN>
<key word,END>
-----the file is end
-------------------------------------------------------------------------------Type
-----------------------------------------------------------------------INT
-------------------------------------------------------------------------------f2
-------------------------------------------------------------------------------(
-------------------------------------------------------------------------------------------------------------MethodDecl
-----------------------------------------------------------Type
-------------------------------------------INT
-----------------------------------------------------------------------FormalParam
-------------------------------------------------------x
-------------------------------------------------------------------------------FormalParams
-------------------------------------------------------,
-----------------------------------------------------------------------SubFormalParam
-------------------------------------------Type
----------------------------INT
-----------------------------------------------------------FormalParam
-------------------------------------------y
-----------------------------------------------------------SubFormalParam
-------------------------------------------NULL
-------------------------------------------------------------------------------)
-----------------------------------------------------------------------BEGIN_
-------------------------------------------------------------------------------Block
-----------------------------------------------------------------Statement
-----------------------------------------------------------NULL
-----------------------------------------------------------------------END
-------------------------------------------------------------------------------MethodDecl
-----------------------------------------------------------------------NULL
-----end parsing
```

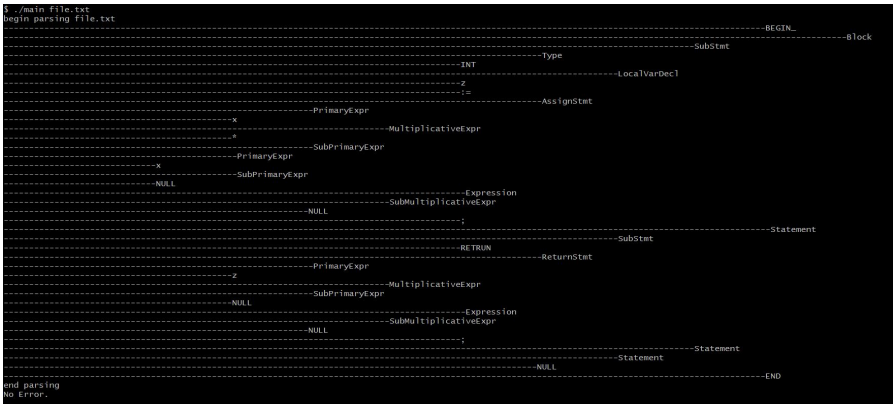


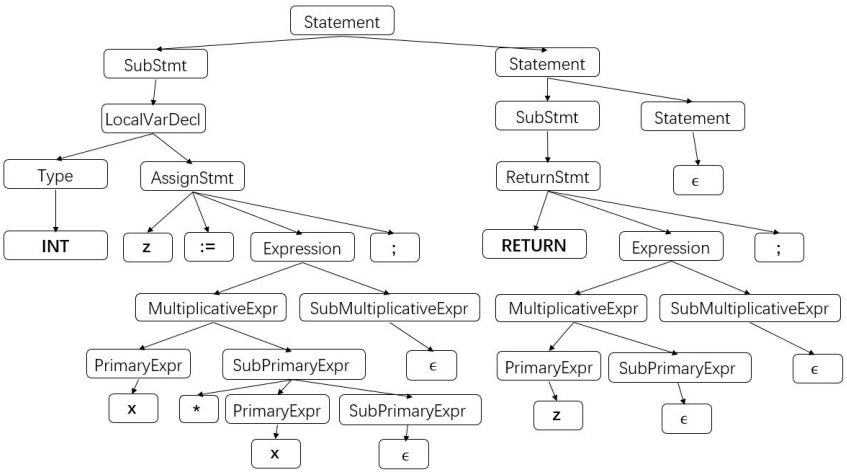
结果：只展示 Statement 部分的语法树

可视化语法树如下：

- TEST3 下述 TINY+ 语法基本展示了所有语法，由于语法树过于庞大，不再展示

```
1  INT f1(INT x, INT y )
2  /*this is comment*/
3  BEGIN
4          INT z;
5          z := x*x - y*y;
6          RETURN z;
7  END
8
9  INT f2()
10 BEGIN
11         STRING x;
```






```
12          READ(x, "A41.input");
13          INT y:=9;
14          READ(y, "A42.input");
15          INT z;
16          z := f1(x,y) + f1(y,x);
17          WRITE (z, "A4.output");
18  END
19
20  INT f3()
21  BEGIN
22          FOR(i := 2 TO 10)
23                  INT x := 3;
24                  INT y := i;
25                  IF(f2(x,y)!=9)
26                          STRING z := "error!";
27                          WRITE(z, "err.txt");
28                  ENDIF
29          ENDFOR
30          WHILE(2+3!=5)
31                  INT t := 2 + 3;
32          ENDWHILE
```

```
33 END
```

结果: