

# Project 2 语法分析实验报告

小组：罗旭川、万伟霖

日期：2020.12.7

## 一、文件组织结构

- `\input` 存放了14个测试样例的文件夹
- `\output` 存放每个样例的输出结果
- `yacc.y` 语法分析代码
- `lexer.l` 词法分析代码
- `main.cpp` 主程序
- `Makefile`

## 二、代码执行方法

```
make
./main
```

执行完后 `case_1` 到 `case_14` 的结果就会生成在 `/output` 文件夹中。

其中 `case_10` 中的输入格式不符合语法要求（全部为词法错误的测例），但为了显示出其中的词法错误确实能被找到，我们在项目中特别地为这个用例生成了特殊的语法树。

## 三、bison的用法

- `bison`命令可以根据编写好的 `.y` 文法说明文件生成语法分析器C代码。主程序通过调用由该代码提供的 `yyparse()` 函数即可进行语法分析。
- 另外，语法分析器需要调用 `yylex()` 来获取每一个输入记号，`yylex()` 可以由 `flex` 工具生成，具体见project 1。
- 文法说明文件 `.y` 文件的写法

```
[定义段]
%%
[规则段]
%%
[用户代码段]
```

### ◦ 定义段

进行终结符和非终结符的声明，以及它们属性值类型的声明等。

可用的声明类型有：`%token`，`%left`，`%right`，`%nonassoc`，`%union`，`%type`，`%start`。在本次实验中只用到了`%token`、`%left`、`%union`、`%type`。

- **%token**：声明为一般的终结符
- **%left**：声明具有左关联合性的终结符
- **%union**：声明可用的属性值类型，在本次实验中，我们构造了一个Node类型的结构体表示语法树的结点类型，并将它作为可用的一个属性值类型

- **%type**: 声明非终结符
- **规则段**

在这里进行文法规则的书写:

```
[产生式左部]: [产生式右部1] {语义动作1}
              | [产生式右部2] {语义动作2}
              ....
              ;
```

语义动作即为相应的产生式进行规约时进行的C代码。在这里可以用 **\$\$** 表示左部符号的属性值, 分别用 **\$1**、**\$2**、**\$3** ....表示产生式右部各符号的属性值。在本实验中, 我们在这里进行语法树的构造。

- **用户代码段**

一些额外的C代码可以写在这里, 也可以写在用 **{%、%}** 包围起来的区域中。

在本实验中我们大部分额外C代码书写在主程序 **main.cpp** 中。

## 四、语法规则实现

```
program      -> PROGRAM IS body ';'
body         -> {declaration} BEGIN {statement} END
declaration  -> VAR {var-decl}
              -> TYPE {type-decl}
              -> PROCEDURE {procedure-decl}
var-decl     -> ID {',' ID} [ ':' type ] ':'=' expression ';'
type-decl    -> ID IS type ';'
procedure-decl -> ID formal-params [ ':' type ] IS body ';'
type        -> ID
              -> ARRAY OF type
              -> RECORD component {component} END
component    -> ID ':' type ';'
formal-params -> '(' fp-section {',' fp-section } ')'
              -> '(' ')'
fp-section   -> ID {',' ID} ':' type
statement    -> lvalue ':'=' expression ';'
              -> ID actual-params ';'
              -> READ '(' lvalue {',' lvalue} ')' ';'
              -> WRITE write-params ';'
              -> IF expression THEN {statement}
                  {ELSIF expression THEN {statement}}
                  [ELSE {statement}] END ';'
              -> WHILE expression DO {statement} END ';'
              -> LOOP {statement} END ';'
              -> FOR ID ':'=' expression TO expression [ BY expression ] DO {statement} END ';'
              -> EXIT ';'
              -> RETURN [expression] ';'

```

```

write-params    -> '(' write-expr ',' write-expr ')'
                -> '(' ')'
write-expr       -> STRING
                -> expression
expression       -> number
                -> l-value
                -> '(' expression ')'
                -> unary-op expression
                -> expression binary-op expression
                -> ID actual-params
                -> ID comp-values
                -> ID array-values
l-value          -> ID
                -> l-value '[' expression ']'
                -> l-value '.' ID
actual-params    -> '(' expression ',' expression ')'
                -> '(' ')'
comp-values      -> '{ ID ':' expression { ';' ID ':' expression } '}'
array-values     -> '[< array-value { ',' array-value } >]'
array-value      -> [ expression 'OF' ] expression
number           -> INTEGER | REAL
unary-op         -> '+' | '-' | NOT
binary-op        -> '+' | '-' | '*' | '/' | DIV | MOD | OR | AND
                -> '>' | '<' | '=' | '>=' | '<=' | '<>'

```

语法规则的书写全部参照 [PCAT语言参考指南.pdf](#) 进行书写。

下面分普通的产生式、带重复项的产生式、带可选项的产生式、复杂的产生式几类来说明书写方法（为了看着清晰，下面所有的语义动作会省略，语义动作将在后面讲语法树如何构建时再细说）：

## 1. 普通的产生式

在定义段中已经声明好所有的保留字和非终结符后，基本照抄即可，如：

```

type -> ID
      -> ARRAY OF type
      ...

```

转换成Bison代码即为：

```

type: ID { ... /* 语义动作 */ }
      | ARRAY OF type { ... /* 语义动作 */ }
      | ... // 其它产生式子
      ;

```

## 2. 带重复项的产生式

### • 可以为空的重复项

在参考指南中，用 {、} 包围起来的表示重复项，可以出现0次或多次。对于这样的产生式，需要用递归的方式来书写，如：

```
body -> {declaration} BEGIN {statement} END
```

这条产生式的非终结符 `declaration` 和 `statement` 可以重复0次或多次，这时我们通过构造对应的额外非终结符 `some_declaration` 和 `some_statement` 来解决，并且书写这两个新非终结符的产生式（我们在本实验中统一采用右递归的形式书写它们）。

即上述产生式转换成Bison代码为：

```

body: some_declaration _BEGIN some_statement END { ... /* 语义动作 */ }
;

some_declaration: /* 空 */ { ... /* 语义动作 */ }
| some_declaration declaration { ... /* 语义动作 */ }
;

some_statement: /* 空 */ { ... /* 语义动作 */ }
| some_statement statement { ... /* 语义动作 */ }
;

```

这里由于 `some_declaration` 和 `some_statement` 可以为空，所以它们的第一条产生式为空。

- 不能为空的重复项

另一种情况是，某个非终结符可以重复1次或多次，即不能为空，如：

```
fp-section -> ID {' , ' ID} ':' type
```

这条产生式右部的非终结符 `ID` 可以产生一次或多次，并用 `,` 隔开，对于这种产生式子，也使用类似的方法，区别仅在于额外产生式 `some_ID` 的书写不一样。

这种产生式转化为Bison代码为：

```

fp-section: some_ID ':' type { ... /* 语义动作 */ }
;
some_ID: ID { ... /* 语义动作 */ }
| some_ID ',' ID { ... /* 语义动作 */ }
;

```

### 3. 带可选的产生式

在指南中，可选项用 `[` 和 `]` 包围起来，表示可有可无。在这里，我们的处理方法就是简单地将原本的产生式拆分成两个产生式，分别代表“有”和“无”的两种情况，如：

```
array-value -> [expression 'OF'] expression
```

上述产生式转换成Bison代码为：

```

array-value: expression OF expression { ... /* 语义动作 */ }
| expression { ... /* 语义动作 */ }
;

```

### 4. 复杂的产生式

即既带有重复项，又带有可选的产生式，这里只要将上述的两种方法混合同时使用即可，如：

```
var-decl -> ID {' , ' ID} [ ':' type] ' := ' expression ';' 
```

上述产生式转换成Bison代码为：

```

var-decl: some_ID ':' type ASSIGN expression ';' { ... /* 语义动作 */ }
        | some_ID ASSIGN expression ';' { ... /* 语义动作 */ }
        ;

some_ID: ID { ... /* 语义动作 */ }
        | some_ID ',' ID { ... /* 语义动作 */ }
        ;

```

由于 `PCAT` 语言比较简单，因此只要运用上述的方法即可按照指南写出所有的语法规则了。其它未列举规则的写法参见代码 `yacc.y`。

## 五、词法及语法错误检测

### 1. 词法错误检测

简单地在词法分析器中通过调用 `yyerror` 来打印错误信息即可。如下面这个整数超出范围的错误：

```

{DIGIT}+ { // 整数
    yylval.node = new Node(yytext);
    if(yyleng > 10 || atoll(yytext) >= (1LL << 31)) {
        yyerror("integer out of range.");
        return INT_OUT_OF_RANGE;
    }
    return INTEGER;
}

```

其它词法错误方法类似，不再赘述。

至于 `yyerror` 函数，这里通过在 `.l` 文件中引入 `%option yylineno` 的选项来开启行数统计，这样一来就可以实现错误位置的显示：

```

void yyerror(const char* msg)
{
    cout << "line " << yylineno << ": " << msg << endl;
}

```

### 2. 语法错误检测

产生语法错误时，语法分析器会自动调用 `yyerror` 函数，并输出 `syntax error` 的信息。但这个信息并没有说明出错误的类型。

通过查阅文档，我们发现只需要在 `.y` 文件中引入 `%error-verbose` 的选项，即可实现一个简易的错误类型说明，这已经足够使用。

错误信息的打印效果如下：

```

line 7: syntax error, unexpected WRITE, expecting ';'

```

## 六、语法树实现

### 1. 方法概览

- 主要通过将每个非终结符的属性值定义为一个 `Node` 类型来实现，表示语法树的结点。
- 然后在每条产生式的语义动作中实现结点的创建和连接，最终产生树状结构。为了书写方便，我们还编写了一个 `insert` 函数来方便为父结点添加子结点。
- 产生的语法树数据结构将用一个全局指针 `root` 来指引。然后由主程序来进行语法树的打印。

### 2. Node结构

```
struct Node { // 语法树节点
    string name;
    vector<Node*> children;
    Node() {}
    Node(const char* name) : name(name) {}
    Node(const string& name): name(name) {}
};
```

每个结点包含 `name`、`children` 两个字段。`name` 表示该结点最终打印出来时要书写的字符串；`children` 表示该结点的所有子结点的指针数组。

### 3. insert函数

```
void insert(Node* fa, Node* child) // 添加一个孩子节点
{
    fa->children.push_back(child);
}

void insert(Node* fa, const vector<Node*>& children) // 添加若干孩子节点
{
    for(Node* child : children) fa->children.push_back(child);
}

void insert(Node* fa, const string& str) // 添加一个叶子节点
{
    fa->children.push_back(new Node(str));
}
```

这里的 `insert` 重载了3个方法，分别用于给 `fa` 结点（即语义动作中的 `$$` 结点）添加一个子结点、多个子结点或一个特定的叶子结点。

### 4. 语义动作写法

语义动作我们分了3类，分别为普通产生式的语义动作、起始产生式的语义动作、递归产生式的语义动作、“管道”产生式的语义动作。

#### (1). 普通产生式的语义动作

```

var-decl: some_ID ':' type ASSIGN expression ';' {
    $$ = new Node("var_decl");
    insert($$, vector<Node*>{$1, $3, $5});
}
| some_ID ASSIGN expression ';' {
    $$ = new Node("var_decl");
    insert($$, vector<Node*>{$1, $3});
}
;

```

对于一般的产生式，我们首先通过 `new Node(...)` 语句来初始化该产生式左部对应的节点。然后通过前面定义的 `insert()` 函数来将产生式右部的非终结符和部分终结符（需要在语法树中表示出来的那些）的结点添加作为该父节点的子结点。

这样，当语法分析器进行规约时，就会至下而上的构造结点，直至生成根节点。并且错误的规约尝试并不会影响最终的语法树的正确性，因为生成的错误的那些子树最终不会连接到根节点下。

## (2). 起始产生式的语义动作

```

program: PROGRAM IS body ';' {
    $$ = new Node("prograrm");
    insert($$, $3);
    root = $$; // 储存语法树指针
}
;

```

大部分和普通产生式的语义动作一样，只是起始产生式还需要将最终的父结点指针存放到全局变量 `root` 中，以便后面的程序来继续进行语法树的打印。

## (3). 递归产生式的语义动作

前面提到，为了能书写一些表示重复项的产生式，我们使用了递归的写法。对于这种产生式的语法子树，我们不希望它也是递归的（这样语法树会很长），我们希望产生的语法子树是简单的（即只有两层，即父节点直接连接到所有的子结点）。对于这样的需求，以 `some_declaration` 为例：

```

some_statement: { // 叶子
    $$ = new Node("statement_list");
}
| some_statement statement {
    $$ = new Node("statement_list");
    insert($$, $1->children);
    insert($$, $2);
}
;

```

即在第二条产生式中，我们会把 `$1` 结点的所有子结点加到 `$$` 结点中，而不是它本身。这样就能让 `$$` 结点直接连接到所有的 `statement` 子结点。

## (4). “管道”产生式的语义动作

对于一些只是起到“管道”作用的产生式，为了能简化语法树（即不要产生太多只有一个孩子的结点），我们这样处理它们，以 `declaration` 结点为例：

```

declaration: VAR some_var-decl {
    $$ = $2;
}
| TYPE some_type-decl {
    $$ = $2;
}
| PROCEDURE some_procedure-decl {
    $$ = $2;
}
;

```

即直接将那个唯一的子结点 `$2` 作为当前结点 `$$`。

## 5. 语法树打印方法

- 递归打印 `root` 所指向的树状结构即可。需要维护一个偏移量 `offset`，每进入下一层，偏移量就增加一定量。
- 为了方便图形的美化（补充竖线），我们先用一个 `vector<string>` 来存图，美化完毕后才进行打印。
- 代码如下，调用 `print_tree(root)` 即可：

```

struct Node { // 语法树节点
    string name;
    vector<Node*> children;
};
extern "C" Node *root;

void print_subtree(Node* root, int offset, vector<string>& graph)
{
    string row = string(offset - 5, ' ') + "+--- " + root->name;
    graph.push_back(row);
    for(Node* child : root->children)
    {
        print_subtree(child, offset + 5, graph);
    }
}

void print_tree(Node* root)
{
    vector<string> graph;
    graph.push_back(root->name);
    for(Node* child : root->children)
    {
        print_subtree(child, 5, graph);
    }
    for(int i = 0; i < graph.size(); i++) // 美化（补充完整树形图上的竖线）
    {
        for(int j = 0; j < graph[i].size(); j++) if(graph[i][j] == '+')
        {
            for(int k = i - 1; k >= 0 && j < graph[k].size() && graph[k][j]
            == ' '; k--) graph[k][j] = '|';
        }
    }
    for(const string& row : graph) // 打印语法树
    {
        cout << row << endl;
    }
}

```



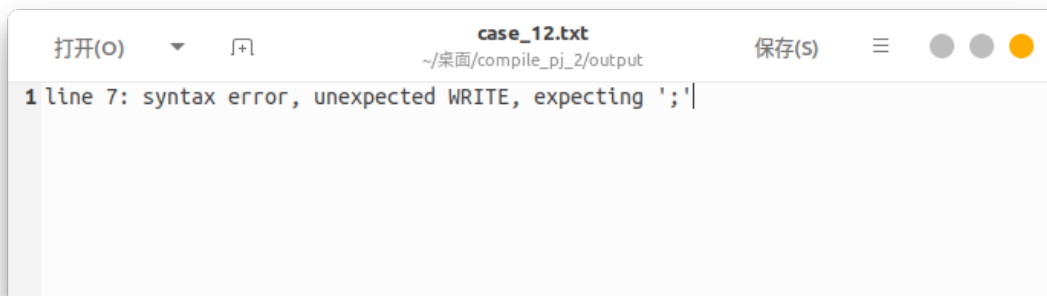
```
}  
}
```

- 打印效果:



The screenshot shows a text editor window titled 'case\_1.txt' with the path '~/桌面/compile\_pj\_2/output'. The editor displays a parse tree for a program. The root node is 'progarm' (line 1), which has a single child 'body' (line 3). 'body' has a child 'decl\_list' (line 5). 'decl\_list' has two children: 'var\_decl\_list' (line 7) and 'var\_decl' (line 9). The first 'var\_decl\_list' (line 7) has a child 'var\_decl' (line 9), which in turn has a child 'ID\_list' (line 11). This 'ID\_list' (line 11) has two children: 'i' (line 13) and 'j' (line 15). The second 'var\_decl' (line 9) has a child 'type' (line 17), which has a child 'INTEGER' (line 19). The second 'var\_decl\_list' (line 27) has a child 'var\_decl' (line 29), which has a child 'ID\_list' (line 31). The 'ID\_list' (line 31) has a single child 'number' (line 23), which has a child '1' (line 25). The tree is visualized with vertical dashed lines and horizontal lines connecting the nodes.

```
1 progarm  
2 |  
3 +--- body  
4 |  
5 +--- decl_list  
6 |  
7 +--- var_decl_list  
8 |  
9 +--- var_decl  
10 |  
11 +--- ID_list  
12 |  
13 +--- i  
14 |  
15 +--- j  
16 |  
17 +--- type  
18 |  
19 +--- INTEGER  
20 |  
21 +--- expression  
22 |  
23 +--- number  
24 |  
25 +--- 1  
26 |  
27 +--- var_decl_list  
28 |  
29 +--- var_decl  
30 |  
31 +--- ID_list  
32 |
```



The screenshot shows a text editor window titled 'case\_12.txt' with the path '~/桌面/compile\_pj\_2/output'. The editor displays a syntax error message: '1 line 7: syntax error, unexpected WRITE, expecting ';' |'. The message is on the first line of the editor.

```
1 line 7: syntax error, unexpected WRITE, expecting ';' |
```

## 七、makefile

```

main: $(main_cpp) yacc.o
    $(CC) $(main_cpp) yacc.o -o $(target)
    rm yacc.c yacc.h lexer.c *.o

yacc.o: yacc.c
    $(CC) -c yacc.c -o yacc.o

yacc.c: $(yacc_y) lexer.c
    $(BISON) -o yacc.c -d $(yacc_y)

lexer.c: $(lexer_l)
    $(FLEX) -o lexer.c $(lexer_l)

```

上面时 `Makefile` 文件中最主要的部分。从下到上来解读，就是：

- 用 `Flex` 工具来将 `lexer.l` 词法分析文件转换为 `lexer.c` 文件（里面提供了 `yylex()`）
- 然后用 `Bison` 工具来将 `yacc.y` 语法分析文件转换为 `yacc.c` 文件（里面提供了 `yyparse()`），并且通过 `-d` 参数来将定义的类型符号传递给 `lexer.c`
- 将 `yacc.c` 编译为 `yacc.o`
- 编译主程序 `main.cpp` 并链接 `yacc.o`，这样以来主程序中就可以调用 `yyparse()` 来开始进行语法分析了。

## 八、思考题回答

- 回忆曾经学过的“中缀表达式”、“后缀表达式”，联系算法优先分析法，思考程序语言使用LR分析法的好处。

答：

- 首先要对程序语言构造算符文法（产生式右部没有两个相连的非终结符）就很困难，因为程序语言比算式要复杂得多。因此不使用算法优先分析法。
- 程序语言并不同于算式，程序语言中的“优先级”的区分其实比较少。一个程序的大部分其实是保留字或自定义的ID等，这些终结符都是没有优先级的区分的。因此无法使用算法优先分析法。
- LR分析法能够在分析的同时及时发现错误。它的适用范围广、分析速度快、报错准确，因此程序语言用这种分析法非常适合。
- 已经有专用的工具 `yacc` 来自动产生分析程序（就像本实验），因此这种方法人工成本也很低。

## 九、组员与分工贡献百分比

姓名	学号	贡献百分比
罗旭川	17307130162	50%
万伟霖	17307130106	50%