Extended PL/0 Compiler

胡京徽(11061191)

November 12, 2013

1 拓展的PL/0文法

鉴于中文的文法在编程时的对应关系容易混乱，我就先将所给的文法翻译成了英文。具体文法如下：

program ::= *block* .

block ::= [*constdec*][*vardec*]{[*procdec*]|[*fundec*]} *compstmt*

constdec ::= **const** *constdef* {, *constdef* };

constdef ::= *ident* = *const*

const ::= [+|−] *unsign* | *character*

character ::= ′ *letter* ′|′ *digit* ′

string ::= "{*ASCII characters with decimal code number varys from* 32 *to* 126 *exclude* 34}"

unsign ::= *digit* { *digit* }

ident ::= *letter* { *letter* | *digit* }

vardec ::= **var** *vardef* ;{ *vardef* ;}

vardef ::= *ident* {, *ident* } : *type*

type ::= *basictype* |**array**′[′ *unsign* ′]′**of** *basictype*

basictype ::= **integer**|**char**

procdec ::= *prochead* *block* {; *prochead* *block* };

fundec ::= *funhead* *block* {; *funhead* *block* };

prochead ::= **procedure** *ident* ′(′[*paralist*]′)′;

funhead ::= **function** *ident* ′(′[*paralist*]′)′ : *basictype* ;

paralist ::= [**var**] *ident* {, *ident* } : *basictype* {; *paralist* }

statement ::= *assignstmt* | *ifstmt* | *repeatstmt* | *pcallstmt*
| *compstmt* | *readstmt* | *writestmt* | *forstmt* | *nullstmt*

assignstmt ::= *ident* := *expression* | *funident* := *expression*
| *ident* ′[′ *expression* ′]′ := *expression*

funident ::= *ident*

expression ::= [+|−] *term* { *addop* *term* }

term ::= *factor* { *multop* *factor* }

factor ::= *ident* | *ident* ′[′ *expression* ′]′| *unsign* |′(′ *expression* ′)′| *fcallstmt*

fcallstmt ::= *ident* ′(′[*arglist*]′)′

arglist ::= *argument* {, *argument* }

argument ::= *expression*

addop ::= +|−

multop ::= ∗|/

condition ::= *expression relop expression*

relop ::= < | <= | > | >= | = | <>

ifstmt ::= **if** *condition* **then** *statement*
|**if** *condition* **then** *statement* **else** *statement*

repeatstmt ::= **repeat** *statement* **until** *condition*

forstmt ::= **for** *ident* := *expression* (**to**|**downto**) *expression* **do** *statement*

pcallstmt ::= *ident* '(' [*arglist*]')'

compstmt ::= **begin** *statement* {; *statement* }**end**

readstmt ::= **read**'(' *ident* {, *ident* }')'

writestmt ::= **write**'(' *string* , *expression* ')'|**write**'(' *string* ')'|**write**'(' *expression* ')'

letter ::= $a|b|c|...|z|A|B|C|...|Z$

digit ::= $0|1|2|3|...|9$

2 文法解读

接下来对文法进行具体解释:

2.1 程序的解读

program ::= *block* .

这句定义了程序是由分程序加“.”组成，其中“.”可以判定程序的结束。这句的语法图见Figure 1。 具体实例:

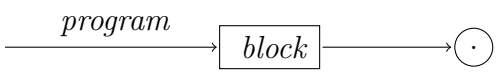

Figure 1: program 语法图

```
count a = 2; .
```

这就是一个完整的程序，“.”表示程序的结束。

2.2 分程序的解读

block ::= [*constdec*][*vardec*]{[*procdec*]|[*fundec*]} *compstmt*

constdec ::= **const** *constdef* {, *constdef* };

constdef ::= *ident* = *const*

vardec ::= **var** *vardef* ;{ *vardef* ;}

vardef ::= *ident* {, *ident* } : *type*

procdec ::= *prochead block* {; *prochead block* };

fundec ::= *funhead block* {; *funhead block* };

prochead ::= **procedure** *ident* '('[*paralist*]')';

funhead ::= **function** *ident* '('[*paralist*]')' : *basictype* ;

compstmt ::= **begin** *statement* {; *statement* }**end**

这几句定义了 *block* 文法的完整视图，语法图见Figure 2。 在分程序 *block* 中先进行常量定义 *constdec* ，

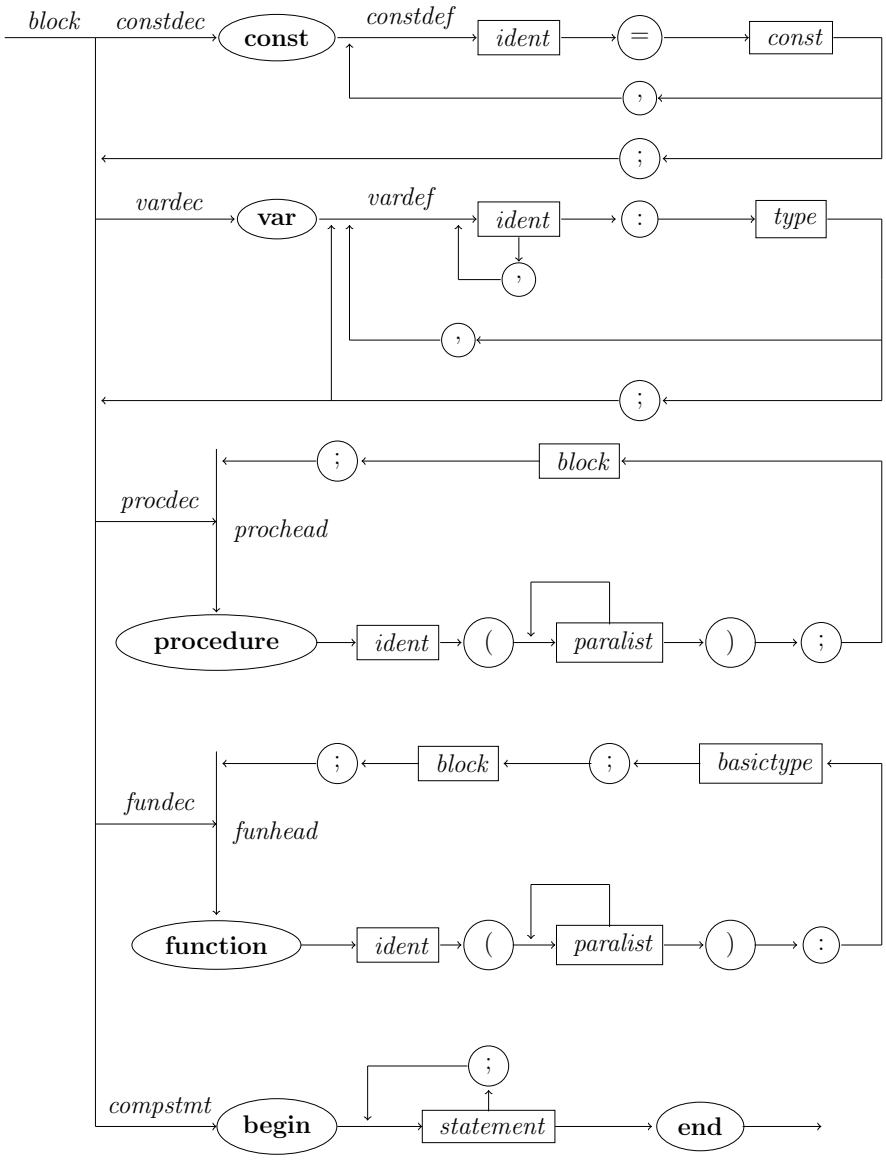

Figure 2: block 语法图

然后是变量定义 *vardef* ，接着是过程的声明 *procdec* 和函数声明 *fundec* 最终的 *block* 通过一个复合语句 *compstmt* 退出。这样的顺序是不能改变的。具体实例:

```
const numbera=0, numberb=1; // 常量定义
var i, j, sum: integer;     // 变量定义
procedure  test();          // 过程声明部分
  test 的程序（略）
function add():integer;     // 函数声明部分
begin
  sum := numbera + numberb; // add 的分程序部分
  add := sum
end;
                            // 复合语句部分
begin
  i := sum
end
```

说明:

1. 常量定义必须在变量前面，这种顺序不能改变。如：`var i:integer;const a=1;`就是错误的。
2. 相连的过程和函数的声明是可以打乱顺序的，过程和函数都可以右参数列表，用于传入参数。
3. 常量是可以连续定义的，之间使用逗号隔开，最后以分号结束常量的定义.
4. 变量的定义也可以连续定义，之间使用逗号隔开，另外使用冒号后跟变量类型来说明定义的变量的类型。变量的定义的结束是使用分号。
5. 常量定义，变量定义，过程声明，函数声明对一个分程序来说是可有可无的，只有复合语句是必须部分。

3 语句的解读

statement ::= *assignstmt* | *ifstmt* | *repeatstmt* | *pcallstmt*
| *compstmt* | *readstmt* | *writestmt* | *forstmt* | *nullstmt*

assignstmt ::= *ident* := *expression* | *funident* := *expression*
| *ident* '[' *expression* ']' := *expression*

ifstmt ::= **if** *condition* **then** *statement*
|**if** *condition* **then** *statement* **else** *statement*

repeatstmt ::= **repeat** *statement* **until** *condition*

forstmt ::= **for** *ident* := *expression* (**to**|**downto**) *expression* **do** *statement*

pcallstmt ::= *ident* '(' [*arglist*] ')'

compstmt ::= **begin** *statement* {; *statement* }**end**

readstmt ::= **read**'(' *ident* {, *ident* }')'

writestmt ::= **write**'(' *string* , *expression* ')'|**write**'(' *string* ')'|**write**'(' *expression* ')'

这几句定义了语句的文法的完整视图,语句的文法图见Figure 3。 具体实例:

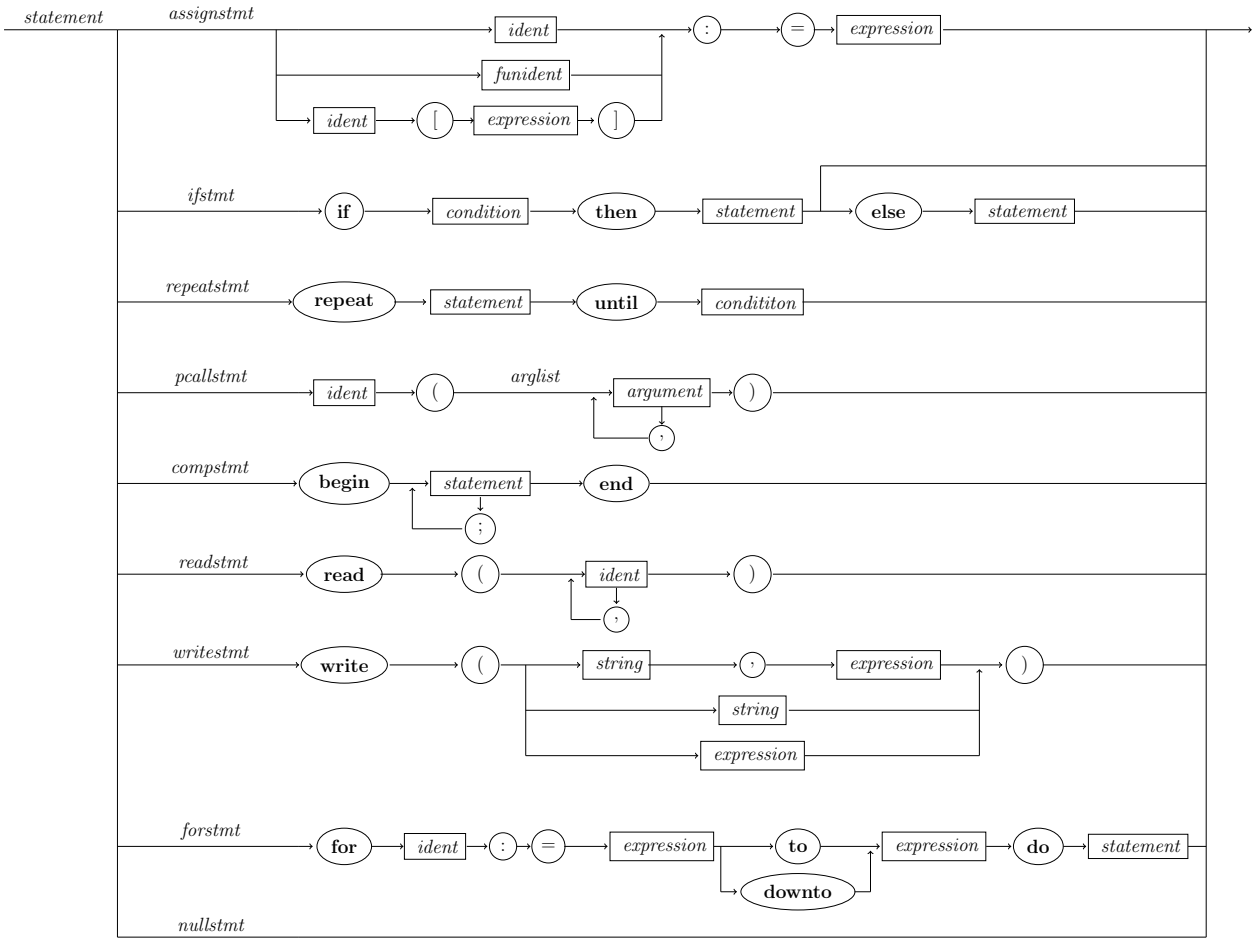

Figure 3: statement 语法图

```
const a=1, b=2, test=11, i=1;
var sum, k: integer; A:array[3] of of integer;
procedure testproc();
  testproc具体实现略;
begin
  // 赋值语句
  sum := a + b;
  A[i] := b;
  // 条件语句
  if a > b then
    sum := a;
  eles
    sum := b;
  // 重复语句
  repeat
```

```
    sum := sum + 1;
  until sum > 4
  // for语句 (步长为一)
  for i := 1 to 10 do
    sum := sum + i
  //过程调用语句
  testproc();
  //读语句
  read (k, sum);
  //写语句
  write("hello world!");
  write(sum)
end
```

说明:

1. 赋值语句可以是表达式给变量的赋值，可以是表达式给数组赋值，还可以给表达式给函数给返回值。
2. 条件语句的**else**悬挂的解决方法是总将**else** 和最近的**if**进行匹配。
3. for语句的步长设为一，**to**表示变量值加一，**downto** 表示变量值减一。
4. 读语句，写语句和过程调用语句比较简单。
5. 复合语句被**begin**和**end**围起来，之间是以分号隔开。
6. 语句可以为空，即什么都没有。

4 类型的解读

$type$ ::= $basictype$ |**array**$'['$ $unsign$ $']'$**of** $basictype$

$basictype$ ::= **integer**|**char**

$const$ ::= $[+|-]$ $unsign$ | $character$

$character$ ::= $'$ $letter$ $'|'$ $digit$ $'$

$string$ ::= “$\{ASCII\ characters\ with\ decimal\ code\ number\ varys\ from\ 32\ to\ 126\ exclude\ 34\}$”

$unsign$ ::= $digit$ $\{$ $digit$ $\}$

$letter$ ::= $a|b|c|...|z|A|B|C|...|Z$

$digit$ ::= $0|1|2|3|...|9$

类型的定义比较简单，就不画语法图了。直接说明。

1. 具体的类型分为两类：基本类型和数组。
2. 基本类型包括**integer**和**char**。整型包含正负的整数；char型包含数字位和大小写字母，定义时以单引号隔开。
3. 字符串包含十进制ASCII码值从32到126的所有的值但是得剔除值为34的双引号。这样有利于词法分析的状态机的设计。另外字符串不是一种类型，只用于写语句的打印。
4. 常量的定义可以是正负整数，也可以是字符。若是字符，则会使用ASCII 码值进行运算。

5 表达式，项，因子的解读

5.1 表达式

expression ::= [+|−] *term* { *addop* *term* }

表达式的语法图见Figure 4。下面是一些表达式的样例:

```
- 3 + a
a + b
1 - 10
a + 1 - 5
```

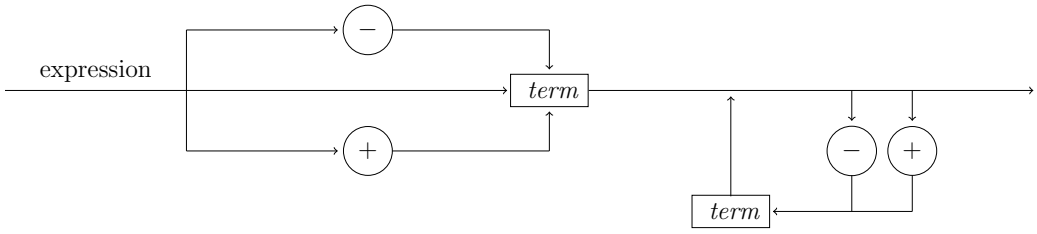

Figure 4: expression 语法图

5.2 项

term ::= *factor* { *multop* *factor* }

项的语法图见Figure 5。下面是一些项的样例:

```
a * b
1 / 10
a / 1 * 5
```

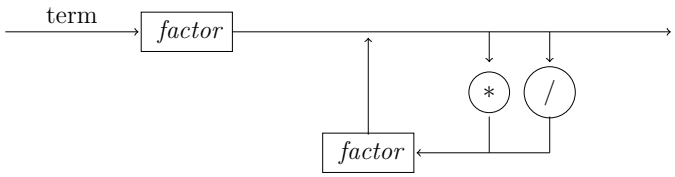

Figure 5: term 语法图

5.3 因子

factor ::= *ident* | *ident* '[' *expression* ']' | *unsign* | '(' *expression* ')' | *fcallstmt*

fcallstmt ::= *ident* '(' [*arglist*] ')'

因子的语法图见Figure 6。下面是一些因子的样例:

```
const a = 1;
var A:array[3] of integer;
function somefun(): integer;
  somefun程序略;
// 下面的是因子
a
A[2]
(a + b)
somefun()
```

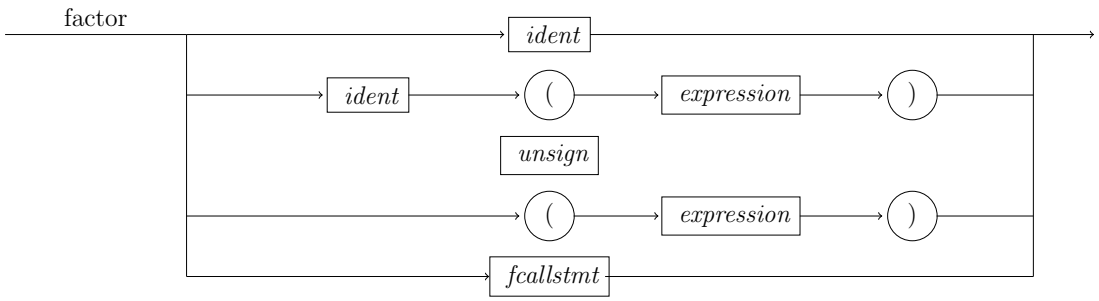

Figure 6: factor 语法图

6 其他一些杂项

ident ::= *letter* { *letter* | *digit* }

paralist ::= [**var**] *ident* {, *ident* } : *basictype* {; *paralist* }

funident ::= *ident*

arglist ::= *argument* {, *argument* }

argument ::= *expression*

addop ::= +|−

multop ::= ∗|/

condition ::= *expression* *relop* *expression*

relop ::= < | <= | > | >= | = | <>

杂项的说明:

1. 关于标识符的解析可以总结成一个状态机，见Figure 7
2. 形参列表需要给出参数类型。
3. 参数列表是由实参组成，之间使用逗号隔开。
4. 实参被定义成一个表达式。
5. 条件是基于一些比较，零表示假，非零表示真。

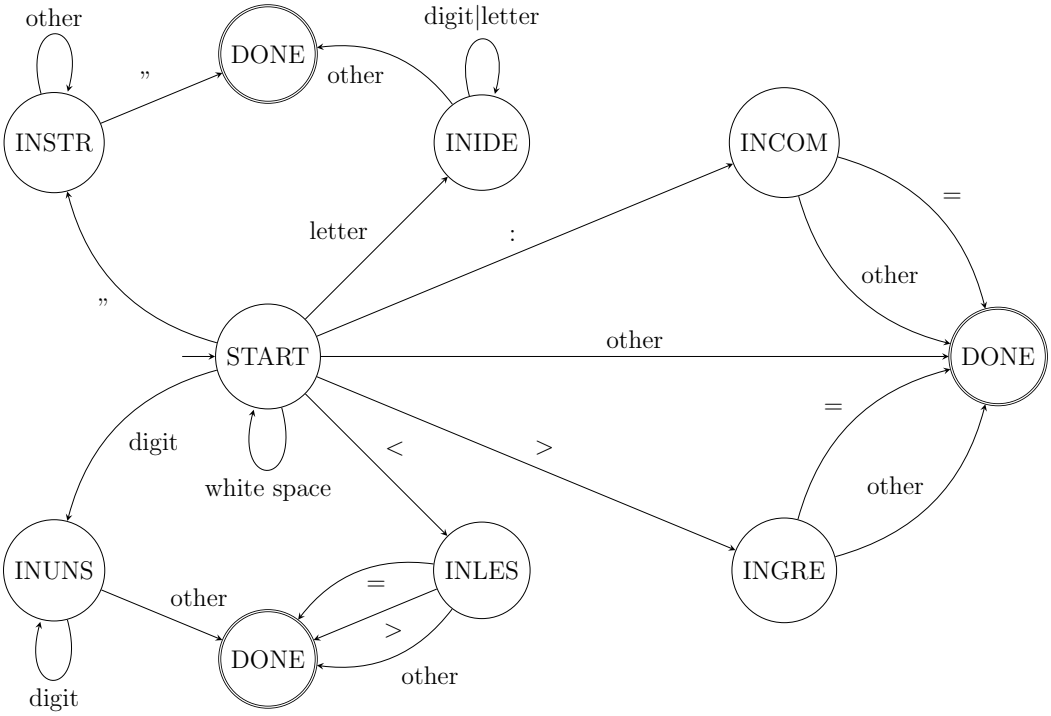

Figure 7: getToken 状态机