

# 编译原理语法分析实验

---

19335286 郑有为

## 编译原理语法分析实验

- I. 作业要求
- II. 实现说明
- III. 词法语法定义
  - 3.1 词法定义
  - 3.2 语法定义
- IV. 程序说明
  - 4.1 树的定义 `df.h`
  - 4.2 Lex 词法程序 `la.l`
  - 4.3 Yacc 语法程序 `ga.y`
- V. 运行说明
- VI. 运行测试
  - 4.1 全模块测试
  - 4.2 求解Fibonacci数代码测试
  - 4.3 错误检测测试 I - 缺少分号
  - 4.4 错误检测测试 II - 赋值符号写成等号
  - 4.5 错误检测测试 III - WHILE格式错误
- VII. 参考

## I. 作业要求

---

- 实验目的：为扩展语言TINY+构造语法分析程序，从而掌握语法分析程序的构造方法。
- 实验内容：用EBNF描述TINY+的语法，构造TINY+语法分析器。
- 实验要求：将TOKEN序列转换成语法分析树，并能检查移动的语法错误。

## II. 实现说明

---

在之前的基于 **Lex** 的TINY+词法分析程序的基础上实现语法分析，使用 **Yacc** 来帮助代码的生成。

修改词法分析定义和部分代码，返回TOKEN（跟之前的代码有较大的不同），但词法定义几乎不变，只有一下两点更改：

- **区分字符和字符串**，分别用 `string` 和 `char` 表示，`string` 用双引号包住，`char` 用单引号包住
- 增加一个**取非标识符**：**not**，优先级高于与运算

## III. 词法语法定义

---

### 3.1 词法定义

参考C语言、老师的资料和华南理工的TINY+实验，以下给出TINY+的词法定义：

**关键字定义**如下，区分大小写，共 20 个。



- EBNF描述语法 (Yacc格式) :

| 编号 | 产生式   | 注释   |
|----|---|--|
| 1  | <b>program:</b> declarations stmt   stmt                                    | <b>程序 (program)</b><br>由 <b>声明部分 (declarations)</b> 和 <b>语句部分 (stmt)</b> 组成, 变量声明需要在语句部分之前完成。                    |
| 2  | <b>declarations:</b> declaration ';'   declaration ';' declarations         | <b>声明部分 (declarations)</b><br>由若干条 <b>声明 (declaration)</b> 组成, 声明部分可以为空。   |
| 3  | <b>declaration:</b> TYPE varlist  | <b>声明 (declaration)</b><br>由 <b>变量类型 (TYPE)</b> 和 <b>变量列表 (varlist)</b> 组成。                                      |
| 4  | <b>varlist:</b> ID   ID ',' varlist   | <b>变量列表 (variable_list)</b><br>由若干个 <b>标识符 (ID)</b> 组成, 标识符之间由 <b>逗号 (,)</b> 隔开。                                 |
| 5  | <b>stmt:</b> xstmt ';'   xstmt ';' stmt                                     | <b>语句序列 (stmt)</b><br>由若干 <b>语句块 (xstmt)</b> 组成, 标识符之间由 <b>分号 (;)</b> 隔开。  |
| 6  | <b>xstmt:</b> WHILE boolexp DO stmt END                                     | <b>循环语句块 (while-stmt)</b><br>有固定格式, 包含关键字 <b>while</b> , 条件判断表达式和关键字 <b>do</b> 、 <b>end</b>                      |
| 7  | <b>xstmt:</b> IF boolexp THEN stmt ELSE stmt END   IF boolexp THEN stmt END | <b>条件判断语句块 (if-stmt)</b><br>有固定格式, 包含关键字 <b>if</b> , 条件判断表达式和关键字 <b>then</b> 、 <b>end</b> , 其中 <b>else</b> 是可选项。 |
| 8  | <b>xstmt:</b> REPEAT stmt UNTIL boolexp                                     | <b>重复语句块 (repeat-stmt)</b><br>有固定格式, 包含关键字 <b>repeat</b> , <b>until</b> 和条件判断表达式, 逻辑上类似于C语言的 <b>do while</b> 。   |

| 编号 | 产生式   | 注释  |
|----|---|---|
| 9  | <b>xstmt</b> : ID ASSIGN exp  | <b>赋值环语句块 (assign-stmt)</b><br>由标识符 (ID)、赋值符号 (:=)、表达式 (exp) 组成。    |
| 10 | <b>xstmt</b> : READ ID  | <b>读入语句块 (read-stmt)</b><br>从某个地方读入一个标识符 (ID)                       |
| 11 | <b>xstmt</b> : WRITE exp  | <b>写入语句块 (write-stmt)</b><br>写入一个表达式 (exp)                          |
| 12 | <b>exp</b> : arithmeticexp   boolexp   strexp   | <b>表达式 (exp)</b><br>有三种不同的类型 ( <b>x-exp</b> )，包括算术表达式、布尔表达式和字符串表达式。 |
| 13 | <b>arithmeticexp</b> : INT   FLOAT   ID   '(' arithmeticexp ')'   arithmeticexp '+' arithmeticexp   arithmeticexp '-' arithmeticexp   arithmeticexp '*' arithmeticexp   arithmeticexp '/' arithmeticexp | <b>算术表达式 (arithmetic_exp)</b><br>可以是整形、浮点数、标识符，也可以是加减乘除运算。          |
| 14 | <b>boolexp</b> : BOOL   comparison   '(' boolexp ')'   NOT boolexp   boolexp AND boolexp   boolexp OR boolexp   | <b>布尔表达式 (bool_exp)</b><br>定义为比较表达式，布尔型遍历或逻辑与或非运算。                  |
| 15 | <b>comparison</b> : arithmeticexp '>' arithmeticexp   arithmeticexp '<' arithmeticexp   arithmeticexp '=' arithmeticexp   arithmeticexp GE arithmeticexp   arithmeticexp LE arithmeticexp               | <b>比较表达式 (comparison)</b><br>含小于、等于、大于、小于等于、大于等于。                   |
| 16 | <b>strexp</b> : CHAR   STRING   | <b>字符表达式 (strexp)</b><br>为字符变量或字符串变量                                |

## IV. 程序说明

### 4.1 树的定义 df.h

- **属性**：定义语法树的结点，结点包含该节点对应的字符串，子节点的数目和指向其子节点的指针数组。
- **方法**：（实现于 `ga.y` 文件中）
  - `Node* genNode(char* content)` 生成一个新的空节点；
  - `void addChild(Node* p, Node* child)`；为一个节点 `p` 添加一个子节点 `child`；
  - `void freeNode(Node* p)`；释放所申请的空间；
  - `void showNode(Node* p, int d = 0, int i = 0)` 输出语法树，其中 `d` 是节点的深度，`i` 是节点标识父节点的第几个子节点。

- 代码:

```

1  /* Declarations of the syntax tree */
2  #ifndef TREE
3  typedef struct Node
4  {
5      char* content;
6      int cnum;
7      struct Node* children[10];
8  } Node;
9
10 Node* genNode(char* content);
11 void addChild(Node* p, Node* child);
12 void freeNode(Node* p);
13 void showNode(Node* p, int d, int i);
14 #endif

```

## 4.2 Lex 词法程序 `la.l`

- **以变量类型 TYPE 为例**: 在识别后调用 `genNode` 生成叶子节点, 并返回类型 TYPE 供语法分析使用, 在此之前, 我们通过 `#define YYSTYPE Node*` 将 `YYSTYPE` 的类型修改为 `Node*`, 这样变量的内容就能方便地在词法分析和语法分析之间传递了。

```

1  int      |
2  float    |
3  bool     |
4  char     |
5  string   { yylval = genNode(yytext); return TYPE; }

```

- **重写 `yyerror` 函数**: 实现自己的错误输出 (返回错误信息和代码行数)

```

1  void yyerror(char *s) {
2      fprintf(stderr, "Error at Line %02d: %s\n", Line, s);
3      exit(1);
4  }

```

## 4.3 Yacc 语法程序 `ga.y`

- **语法定义部分**: 以第一条规则 `**program**: declarations stmt \| stmt` 为例, 说明规约时的操作。

```

1  program: declarations stmt
2      {
3          $$ = genNode("program"); /* 生成节点 */
4          addChild($$, $1); /* 加入子节点, 也就是 declarations 生成的节点 */
5          addChild($$, $2); /* 加入子节点, 也就是 stmt 生成的节点 */
6
7          printf("\nSyntax Tree:\n"); /* 由于 program 是根节点, 故输出语法分析树 */
8          showNode($$, 0, 1); /* 输出语法分析树 */
9          freeNode($$); /* 销毁树, 释放内存 */
10         exit(0); /* 安全退出 */
11     }

```

```

12 | stmt
13 | {
14 |     $$ = genNode("program");    /* 定义部分为空的情况 */
15 |     addChild($$, $1);          /* 加入子节点，也就是 stmt 生成的节点 */
16 |
17 |     printf("\n语法树的前序遍历:\n0: ");
18 |     showNode($$, 0, 1);
19 |     freeNode($$);
20 |     exit(0);
21 | }

```

- 语法树的实现:

```

1  Node* genNode(char* content){
2
3      printf("[%s] ", content); /* 输出规约的符号以便于观察结果 */
4
5      Node* p = NULL;
6      if ((p = malloc(sizeof(Node))) == NULL){
7          yyerror("out of memory");
8      }
9      p->content = strdup(content);
10     p->cnum = 0;
11     for(int i = 0; i < 10; i++){
12         p->children[i] = NULL;
13     }
14     return p;
15 }
16
17 void addChild(Node* p, Node* child){
18     p->children[p->cnum] = child;
19     p->cnum++;
20 }
21
22 void freeNode(Node* p){
23     if(p == NULL){
24         return;
25     }
26     for(int i = 0; i < p->cnum; i++){
27         freeNode(p->children[i]);
28     }
29     free(p->content);
30     free(p);
31 }
32
33 void showNode(Node* p, int d, int i){
34     if(p == NULL){
35         return;
36     }
37     for(int i = 0; i < d; i++){
38         printf(" ");
39     }
40
41     printf("(%d,%d) %s \n", d, i, p->content);
42     for(int i = 0; i < p->cnum; i++){
43         showNode(p->children[i], d+1, i+1);
44     }

```

- 错误检测：使用 Yacc 工具帮助检测错误，遇到错误直接返回错误并终止。

```
1 | %error-verbose
```

## V. 运行说明

- 运行脚本 `co.sh` 编译，生成可运行文件 `ga.out`，脚本代码：

```
1 | lex la.l
2 | yacc -d ga.y
3 | gcc df.h lex.yy.c y.tab.c -o ga.out
```

- 运行可执行文件，输入文件地址即可读取文件中的代码并构建语法分析树。

```
1 | ./ga.out
```

## VI. 运行测试

### 4.1 全模块测试

- **测试输入：** `in/t1.txt`，包含变量测试、赋值测试、读写测试、运算优先级测试、布尔变量测试、语句块测试和嵌套测试。

```
1 | /* Test: 变量声明 */
2 | int a, b, c;
3 | float e, f;
4 |
5 | /* Test: 赋值 */
6 | a := 7;
7 | b := -10;
8 | c := 5;
9 | e := 0.15;
10 |
11 |
12 | /* Test: 读写 read / write */
13 | read f;
14 | write "hey";
15 |
16 |
17 | /* Test: 逻辑运算符 and or not */
18 | if not (a > 0 and not b > 0) then
19 |     write 'N';
20 | end;
21 |
22 | /* Test: 布尔变量 true false */
23 | if true or false or c >= 0 then
24 |     b := f * d;
25 | end;
26 |
27 |
28 | /* Test: 嵌套 */
29 | /* Test: if 语句 */
```



```

30  if a >= b and a < c then
31      /* Test: if-else 语句 */
32      if a >= 5 and a = c then
33          /* Test: while 语句 */
34          while a < 1000 do
35              a := a + 1;
36          end;
37      else
38          /* Test: repeat 语句 */
39          repeat
40              a := a + b * 2;
41          until a < 2000;
42      end;
43  end;
44
45  /* Test: 运算 */
46  a := a + b * (c - d);

```

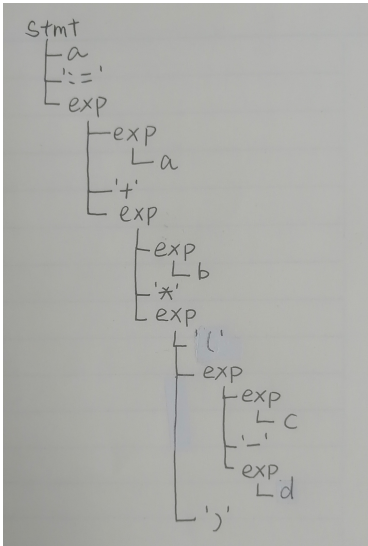
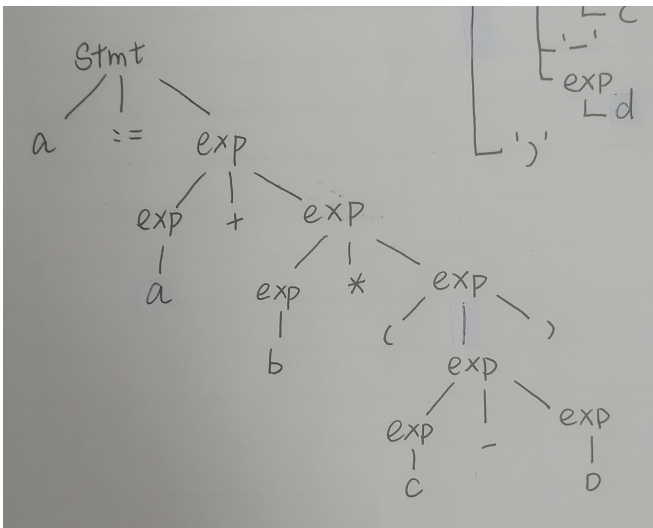
- **输出：**输出结果三百多行，位于 out/t1.txt，这里选出 `a := a + b * (c - d);` 的子树来进行分析。

```

1  (12,1) a
2  (12,2) :=
3  (12,3) arithexp
4      (13,1) arithexp
5          (14,1) a
6      (13,2) +
7      (13,3) arithexp
8          (14,1) arithexp
9              (15,1) b
10         (14,2) *
11         (14,3) arithexp
12             (15,1) (
13                 (15,2) arithexp
14                     (16,1) arithexp
15                         (17,1) c
16                 (16,2) -
17                     (16,3) arithexp
18                         (17,1) d
19             (15,3) )

```

- `(12,1) a` 表示节点上的值为 `a`，位于程序语法树的第12层节点上，是父节点的第一个节点，根据空格和节点深度和标号，我们可以得出出一棵语法树。语法树的形状类似于：

| 画出来   | 横过来  |
|---|--|
|  |  |

## 4.2 求解Fibonacci数代码测试

- 输入: `in/t2.txt`

```

1  /* 求解 Fibonacci 数列 */
2
3  int t1, t2, t3;
4  int ite; /* 迭代次数 */
5
6  t1 := 0;
7  t2 := 1;
8
9  read ite;
10
11 while ite > 0 do
12     ite := ite - 1;
13     t3 := t1 + t2;
14     t1 := t2;
15     t2 := t3;
16 end;
17
18 write "Result of Fibonacci is: ";
19 write t3;
```

- 输出: `out/t2.txt` 语法树有近100行, 故不在报告中展示。

## 4.3 错误检测测试 I - 缺少分号

- 输入: `in/t3.txt`

```

1  /* 求解 Fibonacci 数列 (第一行漏了分号) */
2
3  int t1, t2, t3
4  int ite; /* 迭代次数 */
5
6  t1 := 0;
7  t2 := 1;
8
```

```

9  read ite;
10
11 while ite > 0 do
12     ite := ite - 1;
13     t3 := t1 + t2;
14     t1 := t2;
15     t2 := t3;
16 end;
17
18 write "Result of Fibonacci is: ";
19 write t3;

```

- 输出: `out/t3.txt` 方框内显示每行依次规约的TOKEN

- 报错信息: **Error at Line 04: syntax error, unexpected TYPE, expecting ';'**

```

1  Input File:
2  in/t3.txt
3  Line 01:
4  Line 02:
5  Line 03: [int] [t1] [,] [t2] [,] [t3]
6  Error at Line 04: syntax error, unexpected TYPE, expecting ';'
7  Line 04: [int] [varlist] [varlist] [varlist] [declaration]

```

## 4.4 错误检测测试II - 赋值符号写成等号

- 输入: `in/t4.txt`

```

1  /* 求解 Fibonacci 数列(第六行赋值符号写成了等号) */
2
3  int t1, t2, t3;
4  int ite; /* 迭代次数 */
5
6  t1 = 0;
7  t2 := 1;
8
9  read ite;
10
11 while ite > 0 do
12     ite := ite - 1;
13     t3 := t1 + t2;
14     t1 := t2;
15     t2 := t3;
16 end;
17
18 write "Result of Fibonacci is: ";
19 write t3;

```

- 输出: `out/t4.txt`

- 报错信息: **Error at Line 06: syntax error, unexpected '=', expecting ASSIGN**

```

1 Input File:
2 in/t4.txt
3 Line 01:
4 Line 02:
5 Line 03: [int] [t1] [,] [t2] [,] [t3] [;] [varlist] [varlist] [varlist]
  [declaration]
6 Line 04: [int] [ite] [;] [varlist] [declaration]
7 Line 05:
8 Error at Line 06: syntax error, unexpected '=', expecting ASSIGN
9 Line 06: [t1] [declarations] [declarations] [=]

```

## 4.5 错误检测测试Ⅲ - WHILE格式错误

- 输入: in/t5.txt

```

1  /* 求解 Fibonacci 数列(第11行, while do 写成了 while then) */
2
3  int t1, t2, t3;
4  int ite; /* 迭代次数 */
5
6  t1 := 0;
7  t2 := 1;
8
9  read ite;
10
11 while ite > 0 then
12     ite := ite - 1;
13     t3 := t1 + t2;
14     t1 := t2;
15     t2 := t3;
16 end;
17
18 write "Result of Fibonacci is: ";
19 write t3;
20

```

- 输出: out/t5.txt
  - 报错信息: **Error at Line 11: syntax error, unexpected THEN, expecting DO or OR or AND**

```

1 Input File:
2 in/t5.txt
3 Line 01:
4 Line 02:
5 Line 03: [int] [t1] [,] [t2] [,] [t3] [;] [varlist] [varlist] [varlist]
  [declaration]
6 Line 04: [int] [ite] [;] [varlist] [declaration]
7 Line 05:
8 Line 06: [t1] [declarations] [declarations] [:=] [0] [arithexp] [;]
  [xstmt]
9 Line 07: [t2] [:=] [1] [arithexp] [;] [xstmt]
10 Line 08:
11 Line 09: [read] [ite] [xstmt] [;]
12 Line 10:

```

```
13 | Error at Line 11: syntax error, unexpected THEN, expecting DO or OR or
    | AND
14 | Line 11: [while] [ite] [arithexp] [>] [0] [arithexp] [then] [comparison]
    | [boolexp]
```

## VII. 参考

---

- %error-verbose 的使用: <https://www.thinbug.com/q/33430619>
- Bison 官方文档: [https://www.gnu.org/software/bison/manual/html\\_node/](https://www.gnu.org/software/bison/manual/html_node/)
- lex yacc 文档: ../../resource/lex yacc.pdf