语法分析

1 语法分析的任务

语法分析的主要任务是在词法分析结果的基础上，进一步得到解析树（parsing tree）。例如，下面两个算术表达式：

```
1 + x * y
```

```
(1 + x) * y
```

它们经过词法分析结果如下：

```
TOK_NAT(1) TOK_PLUS TOK_IDENT(x) TOK_MUL TOK_IDENT(y)
```

```
TOK_LEFT_PAREN TOK_NAT(1) TOK_PLUS TOK_IDENT(x)
TOK_RIGHT_PAREN TOK_MUL TOK_IDENT(y)
```

期望的语法分析结果如下：

```
      *                  *
     / \                / \
   ( )   y             +   y
    |                 / \
    +                1   x
   / \
  1   x
```

2 上下文无关语法与解析树

下面是一套上下文无关语法（context-free grammar，CFG）的例子：

```
S -> S ; S          E -> ID          L -> E
S -> ID := E        E -> NAT         L -> L , E
S -> PRINT ( L )    E -> E + E
                    E -> ( E )
```

下面是这套上下文无关语法的一个派生（derivation）：

```
S -> S; S
  -> ID := E; S
  -> ID := E + E; S
  -> ID := ID + E; S
  -> ID := ID + NAT; S
  -> ID := ID + NAT; PRINT(L)
  -> ID := ID + NAT; PRINT(L, E)
  -> ID := ID + NAT; PRINT(E, E)
  -> ID := ID + NAT; PRINT(ID, E)
  -> ID := ID + NAT; PRINT(ID, ID)
```

在上面例子中，`S`表示语句，`E`表示表达式，`L`表示表达式列表。在这个语言中，`PRINT`指令可以带多个参数，表达式中只允许出现加法运算，多条语句只允许顺序执行，没有条件分支与循环。

一套上下文无关语法包含以下几个组成部分：

- 一个初始符号，例如：`S`；
- 一个终结符（terminal symbols）集合，例如：`ID NAT , ; ( ) + :=`，当词法分析器和语法分析器结合使用的时候，这个终结符集合一般就是词法分析中的标记集合；
- 一个非终结符（nonterminal symbols）集合，例如：`S E L`；
- 一系列产生式（production），每个产生式的左边是一个非终结符，每个产生式的右边是一列（可以为空）终结符或非终结符。

将初始符号依据产生式不断展开最后得到一个终结符序列的过程称为派生（derivable）。

下面是这套上下文无关语法的一棵解析树（parsing tree）：

```
          E                          E
        / | \                        -> E * E
       E  *  E                       -> E + E * E
     / | \   |                       -> ID + E * E
    E  +  E  ID                      -> ID + ID * E
    |     |                          -> ID + ID * ID
   ID     ID
```

解析树具有下面性质：

- 根节点为上下文无关语法的初始符号；
- 每个叶子节点是一个终结符，每个内部节点是一个非终结符；
- 每一个父节点和他的子节点构成一条上下文无关语法中的产生式；

3 歧义与歧义的消除

下面上下文无关语法有歧义：

```
E -> ID       E -> E + E      E -> E * E      E -> ( E )
```

同一标记串，有两种解析树：

```
          E                          E
        / | \                      / | \
       E  +  E                    E  *  E
       |     / | \                / | \    |
      ID     E  *  E              E    E    ID
             |     |              |    |
            ID     ID            ID   ID
```

.

修正上下文无关语法，消除加号与乘号优先级有关的歧义

```
E -> F       E -> E + E      F -> F * F
F -> ( E )   F -> ID
```

```
          E                          E
        / | \                        -> E + E
       E  +  E                       -> E + F
       |     |                       -> E + F * F
       F     F                       -> E + F * ID
       |     / | \                   -> E + ID * ID
      ID     F  *  F                 -> F + ID * ID
             |     |                 -> ID + ID * ID
            ID     ID
```

.

4 派生与规约

最左派生与最右派生：

- 一个派生中，如果每次都展开最左侧的非终结符，那么这个派生就称为一个最左派生（left-most derivation）；
- 一个派生中，如果每次都展开最右侧的非终结符，那么这个派生就称为一个最右派生（right-most derivation）；
- 同一棵解析树能够唯一确定一种最左派生，同一棵解析树能够唯一确定一种最右派生；
- 如果一串标记串没有歧义，那么只有唯一的最左派生可以生成这一标记串，也只有唯一的最右派生可以生成这一标记串；

规约是派生反向过程：

```
   ID + ID + ID                  E -> E + F
-> G + ID + ID                     -> E + G
-> F + ID + ID                     -> E + ID
-> E + ID + ID                     -> E + F + ID
-> E + G + ID                      -> E + G + ID
-> E + F + ID                      -> E + ID + ID
-> E + ID                          -> F + ID + ID
-> E + G                           -> G + ID + ID
-> E + F                           -> ID + ID + ID
-> E
```

请大家注意，上图中的派生是最右派生，但是他对应的规约却是每次尽可能地进行左侧规约。这是因为派生与规约的方向是相反的。

5 移入规约分析

下面将要介绍的移入规约分析，是一个计算生成最左规约（或者说最右派生）的过程。移入规约分析的主要思想是：从左向右扫描标记串，从左向右进行规约。

- 共有两类操作：移入、规约；
- 扫描线的右侧全部都是终结符；
- 规约操作都发生在扫描线的左侧紧贴扫描线的区域内。

例子

```
   | ID + ID + ID
-> ID | + ID + ID
-> G | + ID + ID
-> F | + ID + ID
-> E | + ID + ID
-> E + | ID + ID
-> E + ID | + ID
-> E + G | + ID
-> E + F | + ID
-> E | + ID
-> E + | ID
-> E + ID |
-> E + G |
-> E + F |
-> E |
```

移入与规约的选择问题：

- 既能移入，又能规约的情形，应当选择移入还是规约？
- 有多种规约方案的情形，应当如何选择？

例子：

```
   | ID + ID + ID
-> ID | + ID + ID
-> G | + ID + ID          E + E | + ID
-> F | + ID + ID
-> E | + ID + ID          E | + ID
-> E + | ID + ID
-> E + ID | + ID          E + F + | ID
-> E + G | + ID
-> E + F | + ID
```

6 已移入部分的结构

已移入部分的结构

- 已移入规约的部分可以分为 n 段；
- 每段对应一条产生式；
- 第 i 段拼接上第 $i+1$ 条产生式的左侧非终结符是第 i 条产生式右侧符号串的一个前缀。

例子

- `F * (E) |` 中已移入规约的部分可以分为：`F *` `( E )`
- 分别对应产生式：`F -> F * . G` `G -> ( E ) .`
- `F * (E + | ID )` 中已移入规约的部分可以分为：`F *` `(` `E +`
- 分别对应产生式：`F -> F * . G` `G -> ( . E )` `E -> E + . F`

移入与规约对应的扫描线左侧结构变化：
原操作：移入

```
| ID + ID + ID     ->     ID | + ID + ID
```

带结构的操作：

```
(START -> . E)     ->     (START -> . E)
                          (E -> . E + F)
                          (E -> . E + F)
                          (E -> . F)
                          (F -> . G)
                          (G -> ID .)
                                                    .
```

原操作：规约

```
ID | + ID + ID    ->    G | + ID + ID
```

带结构的操作：

```
(START -> . E)     ->     (START -> . E)
(E -> . E + F)            (E -> . E + F)
(E -> . E + F)            (E -> . E + F)
(E -> . F)                (E -> . F)
(F -> . G)                (F -> G .)
(G -> ID .)
                                                    .
```

原操作：规约

```
G | + ID + ID      ->     F | + ID + ID
```

带结构的操作：

```
(START -> . E)     ->     (START -> . E)
(E -> . E + F)            (E -> . E + F)
(E -> . E + F)            (E -> . E + F)
(E -> . F)                (E -> F .)
(F -> G .)


                                                    .
```

原操作：规约

```
F | + ID + ID      ->     E | + ID + ID
```

增加结构：

```
(START -> . E)      ->      (START -> . E)
(E -> . E + F)              (E -> . E + F)
(E -> . E + F)              (E -> E . + F)
(E -> F .)
```

.

原操作：移入

```
E | + ID + ID       ->      E + | ID + ID
```

增加结构：

```
(START -> . E)      ->      (START -> . E)
(E -> . E + F)              (E -> . E + F)
(E -> E . + F)              (E -> E + . F)
```

.

原操作：移入

```
E + | ID + ID       ->      E + ID | + ID
```

增加结构：

```
(START -> . E)      ->      (START -> . E)
(E -> . E + F)              (E -> . E + F)
(E -> E + . F)              (E -> E + . F)
                            (F -> .G)
                            (G -> ID .)
```

.

7 已移入部分的结构判定

以扫描线左侧的最右段为状态，构建 NFA。新增一段带来的变化对应一条 ϵ 边，其他加入符号带来的变化对应一条普通边。该 NFA 中的所有状态都是终止状态，要判断一串符号串是否是可行的扫描线左侧结构，只需判断这个符号串能否被这个 NFA 接受。

ϵ: `START -> . E` 变为 `E -> . F`

ϵ: `START -> . E` 变为 `E -> . E + F`

E: `START -> . E` 变为 `START -> E .`

ϵ: `E -> . F` 变为 `F -> . F * G`

ϵ: `E -> . F` 变为 `F -> . G`

F: `E -> . F` 变为 `E -> F .`

ϵ: `E -> . E + F` 变为 `E -> . F`

ϵ: `E -> . E + F` 变为 `E -> . E + F`

E: `E -> . E + F` 变为 `E -> E . + F`

+: `E -> E . + F` 变为 `E -> E + . F`

ϵ: `E -> E + . F` 变为 `F -> . F * G`

ϵ: `E -> E + . F` 变为 `F -> . G`

...

判断`E + F + | ...`是否是可行的扫描线左侧结构：

```
E + F + | ...

START -> . E
    E -> . F
    E -> . E + F
    F -> . G
    F -> . F * G
    G -> . ( E )
    G -> . ID                                   .
```

```
E + F + | ...

START -> E .
    E -> E . + F
                                                .
```

```
E + F + | ...

    E -> E + . F
    F -> . G
    F -> . F * G
    G -> . ID
    G -> . ( E )
                                                .
```

```
E + F + | ...

    E -> E + F .
    F -> F . * G
                                                .
```

```
E + F + | ...

    ( No possible state )
                                                .
```

8 基于 follow 集合的判定法

定义

- `X` 是 Follow(`Y`) 的元素当且仅当 `... Y X ...` 是可能被完全规约的。
- `X` 是 First(`Y`) 的元素当且仅当 `X ...` 是可能被规约为 `Y` 的。
- 上述计算只考虑 `X` 是终结符的情形。

First 集合的计算

- 对任意终结符 `X`，`X` 都是 First(`X`) 的元素。
- 对任意产生式 `Y -> Z ...`，First(`Z`) 的元素都是 First(`Y`) 的元素。

Follow 集合的计算

- 对任意产生式 `U -> ... Y Z ...`，First(`Z`) 是 Follow(`Y`) 的子集。
- 对任意产生式 `Z -> ... Y`，Follow(`Z`) 是 Follow(`Y`) 的子集。

9 移入规约分析完整过程实例

`ID * (ID + ID)` 的移入规约分析：

- 初始：`| ID * ( ID + ID )`
- 移入：`ID | * ( ID + ID )`
- 规约：`G | * ( ID + ID )`，因为 `ID * | ...` 不可行
- 规约：`F | * ( ID + ID )`，因为 `G * | ...` 不可行
- 移入：`F * | ( ID + ID )`，因为 `*` 不是 Follow(`E`) 中的元素
- 移入：`F * ( | ID + ID )`
- 移入：`F * ( ID | + ID )`
- 规约：`F * ( G | + ID )`，因为 `F * ( ID + | ...` 不可行
- 规约：`F * ( F | + ID )`，因为 `F * ( G + | ...` 不可行
- 规约：`F * ( E | + ID )`，因为 `F * ( F + | ...` 不可行
- 移入：`F * ( E + | ID )`
- 移入：`F * ( E + ID | )`
- 规约：`F * ( E + G | )`，因为 `F * ( E + ID ) | ...` 不可行
- 规约：`F * ( E + F | )`，因为 `F * ( E + G ) | ...` 不可行
- 规约：`F * ( E | )`，因为另外两种方案扫描线左侧的结构都不可行
- 移入：`F * ( E ) |`
- 规约：`F * G |`

- 规约：`F |`
- 规约：`E |`，亦可看做`E | EOF`
- 移入：`E EOF |`
- 规约：`START |`
- 语法分析结束