编译原理

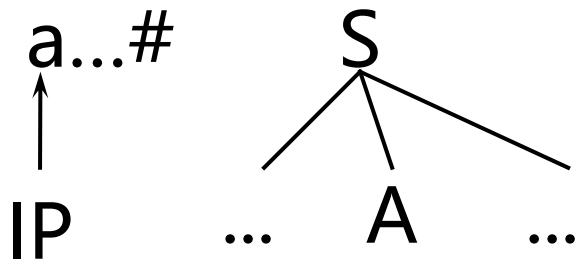
递归下降分析程序

编译原理

回顾LL(1)文法和LL(1)分析法

自上而下分析

- 基本思想
  - 从文法的开始符号出发，向下推导，推出句子
  - 针对输入串，试图用一切可能的办法，从文法开始符号(根结点)出发，自上而下地为输入串建立一棵语法树



自上而下分析

- 构造不带回溯的自上而下分析算法
  - 消除文法的左递归
  - 提取左公共因子，克服回溯
- 计算FIRST和FOLLOW集合
- LL(1)文法的条件

LL(1)文法

- 构造不带回溯的自上而下分析的文法条件

1. 文法不含左递归
2. 对于文法中每一个非终结符A的各个产生式的候选首符集两两不相交。即，若

$$A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$$

则 $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \phi \qquad (i \neq j)$

3. 对文法中的每个非终结符A，若它存在某个候选首符集包含ε，则

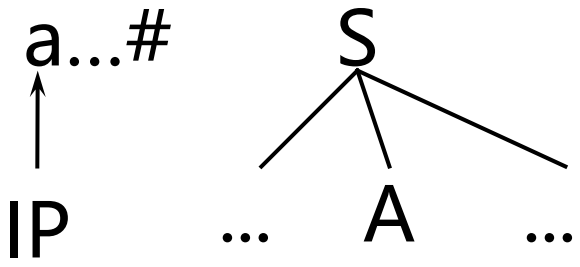
$$FIRST(\alpha_i) \cap FOLLOW(A) = \phi \qquad i = 1,2,\ldots,n$$

如果一个文法G满足以上条件，则称该文法G为LL(1)文法。

自上而下分析

- 构造不带回溯的自上而下分析算法
  - 消除文法的左递归
  - 提取左公共因子，克服回溯
- 计算FIRST和FOLLOW集合
- LL(1)文法的条件
- LL(1)分析法

LL(1)分析法



- 假设要用非终结符A进行匹配，面临的输入符号为a，A的所有产生式为

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

1. 若$a \in FIRST(\alpha_i)$，则指派α_i执行匹配任务；
2. 若a不属于任何一个候选首符集，则：
   (1) 若ε属于某个$FIRST(\alpha_i)$且 $a \in FOLLOW(A)$，则让A与ε自动匹配。
   (2) 否则，a的出现是一种语法错误。

编译原理

构造递归下降分析器

递归下降分析器

- 分析程序由一组子程序组成， 对每一语法单位(非终结符)构造一个相应的子程序，识别对应的语法单位
- 通过子程序间的相互调用实现对输入串的识别
  - 例如，A → B c D
- 文法的定义通常是递归的，通常具有递归结构

递归下降分析器

- 定义全局过程和变量
  - ADVANCE，把输入串指示器IP指向下一个输入符号，即读入一个单词符号
  - SYM，IP当前所指的输入符号
  - ERROR，出错处理子程序

递归下降子程序设计

- 文法G(E):
  E→TE′
  E′→+TE′ | ε
  T→FT′
  T′→*FT′ | ε
  F→(E) | i
- 每个非终结符有对应的子程序的定义，在分析过程中，当需要从某个非终结符出发进行展开(推导)时，就调用这个非终结符对应的子程序。

递归下降子程序设计

A→TE′ | BC | ε　对应的递归下降子程序为

```
PROCEDURE  A;
BEGIN
   IF  SYM ∈FIRST(TE') THEN
        BEGIN  T;  E′ END
   ELSE IF  SYM ∈FIRST(BC) THEN
        BEGIN  B; C  END
   ELSE IF  SYM ∉FOLLOW(A) THEN
        ERROR
END;
```

递归下降子程序设计

- 文法G(E):
  E→TE′
  E′→+TE′ | ε
  T→FT′
  T′→*FT′ | ε
  F→(E) | i
- 对应的递归下降子程序为

递归下降子程序设计

- 文法G(E):

  E→TE′

  E′→+TE′ | ε

  T→FT′

  T′→*FT′ | ε

  F→(E) | i

- 对应的递归下降子程

```
PROCEDURE  F;
  IF SYM='i' THEN ADVANCE
  ELSE
       IF SYM='(' THEN
       BEGIN
          ADVANCE;
          E;
          IF SYM=')' THEN ADVANCE
             ELSE ERROR
       END
       ELSE ERROR;
```

递归下降子程序设计

- 文法G(E):
  - **E→TE′**
  - **E′→+TE′ | ε**
  - T→FT′
  - T′→*FT′ | ε
  - F→(E) | i
- 对应的递归下降子程

E′实现了ε吗？
A. 实现了
B. 没实现

```
PROCEDURE  E′;
IF SYM=  ‘+’  THEN
BEGIN
    ADVANCE;
        T;  E′
END
```

FOLLOW(E′)={), # }

```
PROCEDURE  E;
BEGIN
        T;  E′
END;
```

```
PROCEDURE  E′;
IF SYM=‘+’ THEN
BEGIN
    ADVANCE;
        T;  E′
END
ELSE  IF SYM<>‘#’ AND SYM<>’)’
            THEN ERROR
```

E ′不考虑Follow集合有问题吗？
A. 有问题
B. 没有问题

- 文法G(E):
  E→TE′
  E′→+TE′ | ε
  T→FT′
  T′→*FT′ | ε
  F→(E) | i
- 对应的递归下降子程

```
PROCEDURE  E;
BEGIN
        T;  E′
END;
```

```
PROCEDURE  E′;
IF SYM=  ‘+’   THEN
BEGIN
    ADVANCE;
        T;  E′
END
```

FOLLOW(E′)={), # }

```
PROCEDURE  E′;
IF SYM=‘+’ THEN
BEGIN
    ADVANCE;
        T;  E′
END
ELSE  IF SYM<>‘#’ AND SYM<>’)’
                THEN ERROR
```

构造每个非终结符的FOLLOW集合

- 对于文法G的每个非终结符A构造FOLLOW(A)的办法是，连续使用下面的规则，直至每个FOLLOW不再增大为止：

1. 对于文法的开始符号S，置＃于FOLLOW(S)中；
2. 若$A\rightarrow\alpha B\beta$是一个产生式，则把FIRST(β)\{ε}加至FOLLOW(B)中；
3. 若$A\rightarrow\alpha B$是一个产生式，或$A\rightarrow\alpha B\beta$是一个产生式而$\beta\overset{*}{\Rightarrow}\varepsilon$ (即$\varepsilon\in$ FIRST(β))，则把FOLLOW(A)加至FOLLOW(B)中

递归下降子程序设计

- 文法G(E):
  E→TE′
  E′→+TE′ | ε
  T→FT′
  T′→*FT′ | ε
  F→(E) | i
- 对应的递归下降子程

```
PROCEDURE  E′;
IF SYM= ‘+’  THEN
BEGIN
    ADVANCE;
        T;  E′
END
```

FOLLOW(E′)={), # }

```
PROCEDURE  E;
BEGIN
        T;  E′
END;
```

```
PROCEDURE  E′;
IF SYM=‘+’ THEN
BEGIN
    ADVANCE;
        T;  E′
END
ELSE  IF SYM<>‘#’ AND SYM<>’)’
            THEN ERROR
```

递归下降子程序设计

- 文法G(E):

  E→TE′

  E′→+TE′ | ε

  T→FT′

  T′→*FT′ | ε

  F→(E) | i

- 对应的递归下降

```
PROCEDURE  T;
BEGIN
      F;  T′
END
```

```
PROCEDURE  T′;
IF SYM=  ‘*’   THEN
BEGIN
   ADVANCE;
   F;  T′
END;
```

```
PROCEDURE  T′;
IF SYM=‘*’ THEN
BEGIN
        ADVANCE;
        F;  T′
END
ELSE  IF SYM<>‘#’  AND
            SYM<>’)’ AND SYM<>’+’
        THEN  ERROR
```

递归下降子程序设计

- 文法G(E):

  E→TE′

  E′→+TE′ | ε

  T→FT′

  T′→*FT′ | ε

  F→(E) | i

- 对应的递归下降子程序

主程序:

```
PROGRAM PARSER;
BEGIN
    ADVANCE;
    E;
    IF SYM <>' #'  THEN
        ERROR
END;
```

编译原理

扩充的巴科斯范式和语法图

扩充的巴科斯范式

- 在元符号“→”或“::=”和“|”的基础上，扩充几个元语言符号：
  - 用花括号{α}表示闭包运算α*。
  - 用表示$\{\alpha\}_0^n$可任意重复0次至n次。
  - 用方括号[α]表示$\{\alpha\}_0^1$，即表示α的出现可有可无(等价于α|ε)。

扩充的巴科斯范式

- 例如，通常的“实数”可定义为：

  Decimal→[Sign]Integer.{digit}[Exponent]

  Exponent→E[Sign]Integer

  Integer→digit{digit}

  Sign→ + | -

- 用扩充的巴科斯范式来描述语法，直观易懂，便于表示左递归消去和因子提取。

扩充的巴科斯范式

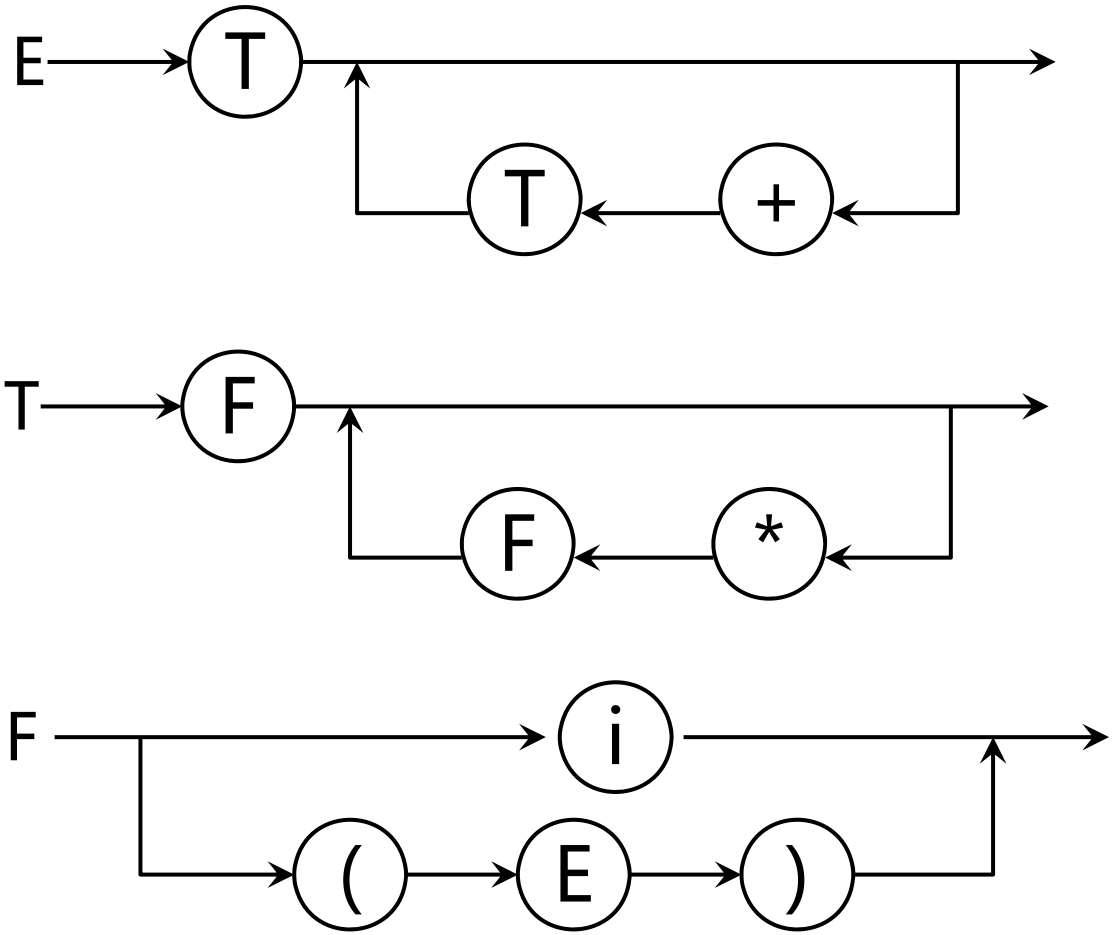
- 文法G(E)：
  E→T | E+T
  T→F | T*F
  F→i | (E)

可表示成

E→T{+T}

T→F{*F}

F→i | (E)



设计递归下降分析

- E→T{+T}

  T→F{*F}

  F→i | (E)
- 可构造一组递归下降

```
PROCEDURE  T;
BEGIN
     F;
     WHILE SYM=  '*'   DO
     BEGIN
          ADVANCE;
          F
      END
END;
```

```
PROCEDURE  E;
BEGIN
    T;
    WHILE SYM=  '+'   DO
    BEGIN
       ADVANCE;
       T
    END
END;
```

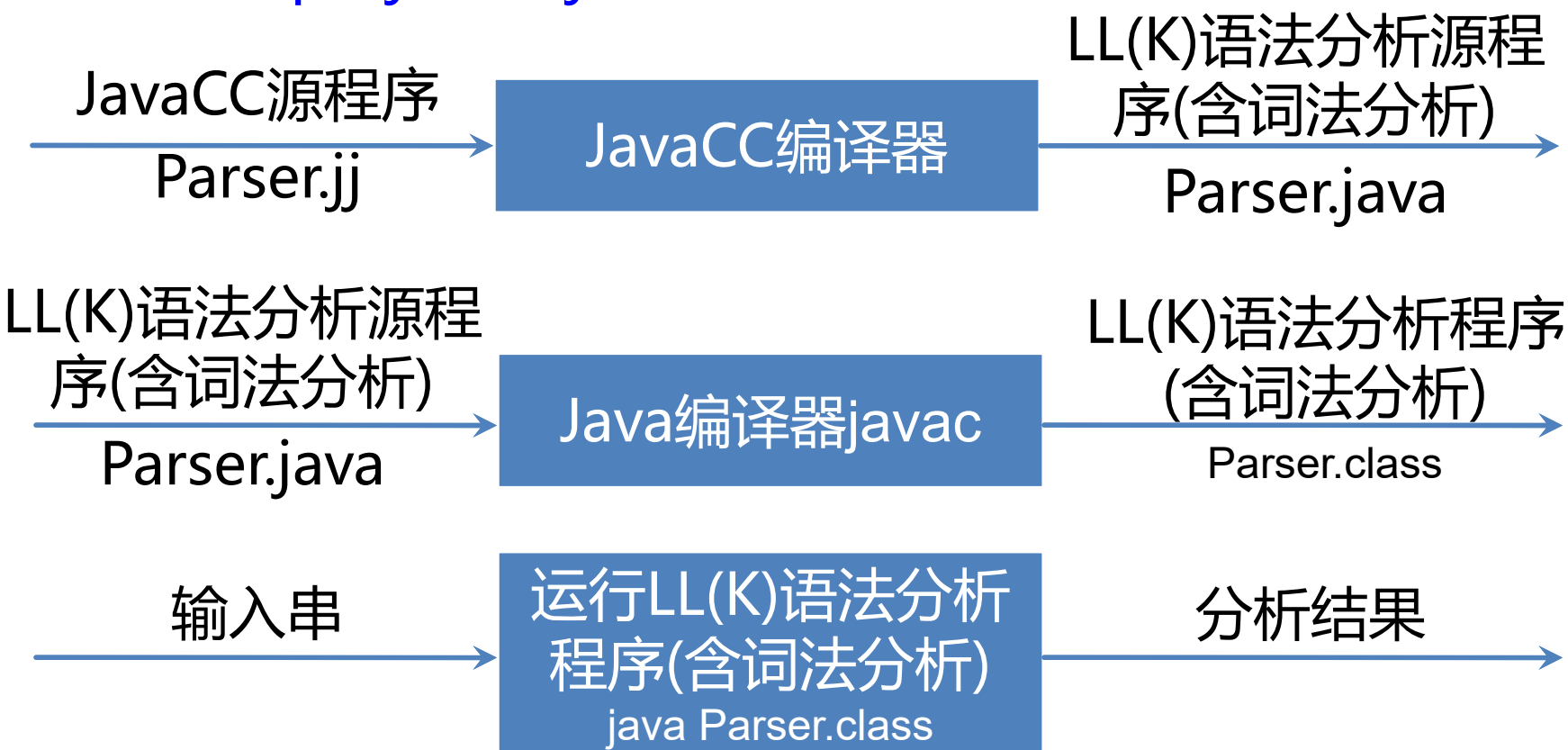
```
PROCEDURE  F;
  IF SYM=  'i'   THEN ADVANCE
  ELSE
       IF SYM=  '('   THEN
       BEGIN
          ADVANCE;
           E;
           IF SYM=  ')'   THEN
ADVANCE
                ELSE ERROR
       END
       ELSE ERROR;
```

编译原理

JavaCC简介

JavaCC

- Java Compiler Compiler (JavaCC) - The Java Parser Generator
  - http://javacc.java.net/



JavaCC

- <parser_name>.java
- <parser_name>Constants.java
- <parser_name>TokenManager.java
- ParseException.java
- SimpleCharStream.java
- Token.java
- TokenMgrError.java

小结

- LL(1)分析法——递归下降分析程序
  - 消除左递归，消除回溯，转换成LL(1)文法
  - 分析程序由一组递归过程组成， 对每一语法变量(非终结符)构造一个相应的子程序，识别对应的语法单位
  - 通过子程序间的相互调用实现对输入串的识别
- JavaCC
  - 递归下降分析程序自动生成