编译原理

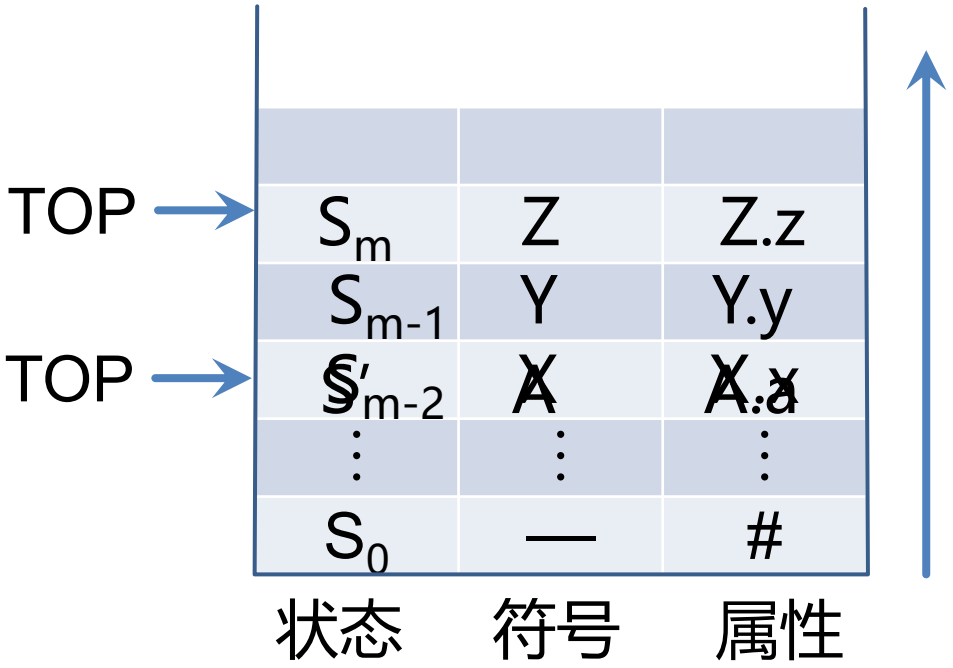
S-属性文法

S-属性文法的自下而上计算

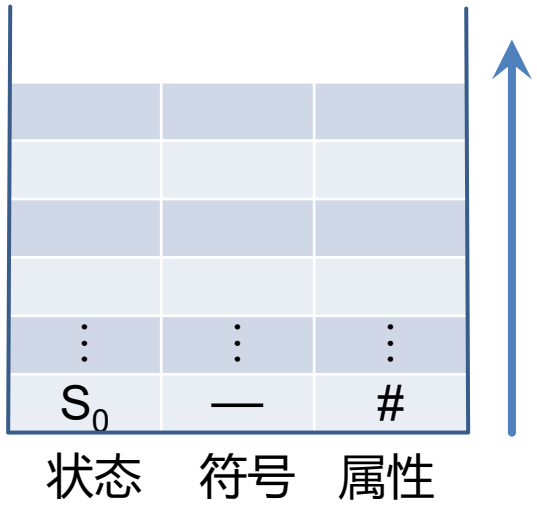
- S-属性文法：只含有综合属性
- 在自下而上的分析器分析输入符号串的同时计算综合属性
  - 分析栈中保存语法符号和有关的综合属性值
  - 每当进行归约时，新的语法符号的属性值就由栈中正在归约的产生式右边符号的属性值来计算

S-属性文法的自下而上计算

- 在分析栈中增加附加域存放综合属性值
- 假设产生式A→XYZ对应的语义规则为 a:=f(X.x,Y.y,Z.z)



S-属性文法的自下



产生式	语 义 规 则
L→En	print(E.val)
$E→E_1+T$	E.val := E_1.val+T.val
E→T	E.val :=T.val
$T→T_1*F$	T.val :=T_1.val* F.val
T→F	T.val :=F.val
F→ (E)	F.val :=E.val
F→digit	F.val :=digit.lexval

产生式	代 码 段
L→En	print(val[top])
$E→E_1+T$	val[ntop] := val[top-2]+val[top]
E→T	
$T→T_1*F$	val[ntop] := val[top-2]*val[top]
T→F	
F→ (E)	val[ntop] :=val[top-1]
F→digit	

S-属性文法的分析过程

- 句子3*5+4n

步骤	状态栈	符号栈	属性(val)	输入串

产生式	代 码 段
L→En	print(val[top])
$E \rightarrow E_1+T$	val[ntop] := val[top-2]+val[top]
E→T	
$T \rightarrow T_1*F$	val[ntop] := val[top-2]*val[top]
T→F	
F→ (E)	val[ntop] :=val[top-1]
F→digit	

编译原理

L-属性文法

编译原理

L-属性文法定义

一遍扫描的处理方法

- 在语法分析的同时计算属性值
  - 所采用的语法分析方法
  - 属性的计算次序
- S - 属性文法适合一遍扫描的自下而上分析
- L - 属性文法适合一遍扫描的自上而下分析

L-属性文法和自顶向下翻译

- 按照深度优先遍历语法树，计算所有属性值
- 与LL(1) 自上而下分析方法结合
  - 深度优先建立语法树
  - 按照语义规则计算属性

L-属性文法

- 一个属性文法称为L-属性文法，如果对于每个产生式$A \rightarrow X_1 X_2 \ldots X_n$，其每个语义规则中的每个属性或者是综合属性，或者是$X_i (1 \le i \le n)$的一个继承属性且这个继承属性仅依赖于：
  (1) 产生式中X_i左边符号X_1，X_2，…，X_{i-1}的属性
  (2) A的继承属性
- S-属性文法一定是L-属性文法

测试：L-属性文法

- 某属性文法包含下面的定义，该文法是L-属性文法吗

A. 是

B. 否

产 生 式	语 义 规 则
A→LM	L.i := g(A.i)
	M.i :=m(L.s)
A→QR	R.i := r(A.i)
	Q.i :=q(R.s)
	A.s :=f(Q.s)

编译原理

翻译模式

翻译模式

- 语义规则：给出了属性计算的定义，没有属性计算的次序等实现细节
- 翻译模式：给出使用语义规则进行计算的次序，把实现细节表示出来
- 在翻译模式中，和文法符号相关的属性和语义规则（也称语义动作），用花括号{ }括起来，插入到产生式右部的合适位置上

产生式	语义规则
$E \rightarrow TR$	
$R \rightarrow addop\ T\ R_1 \mid \varepsilon$	print(addop.lexeme)
$T \rightarrow num$	print(num.val)

$E \rightarrow TR$

$R \rightarrow addop\ T$ {print(addop.lexeme)} $R_1 \mid \varepsilon$

$T \rightarrow num$ {print(num.val)}

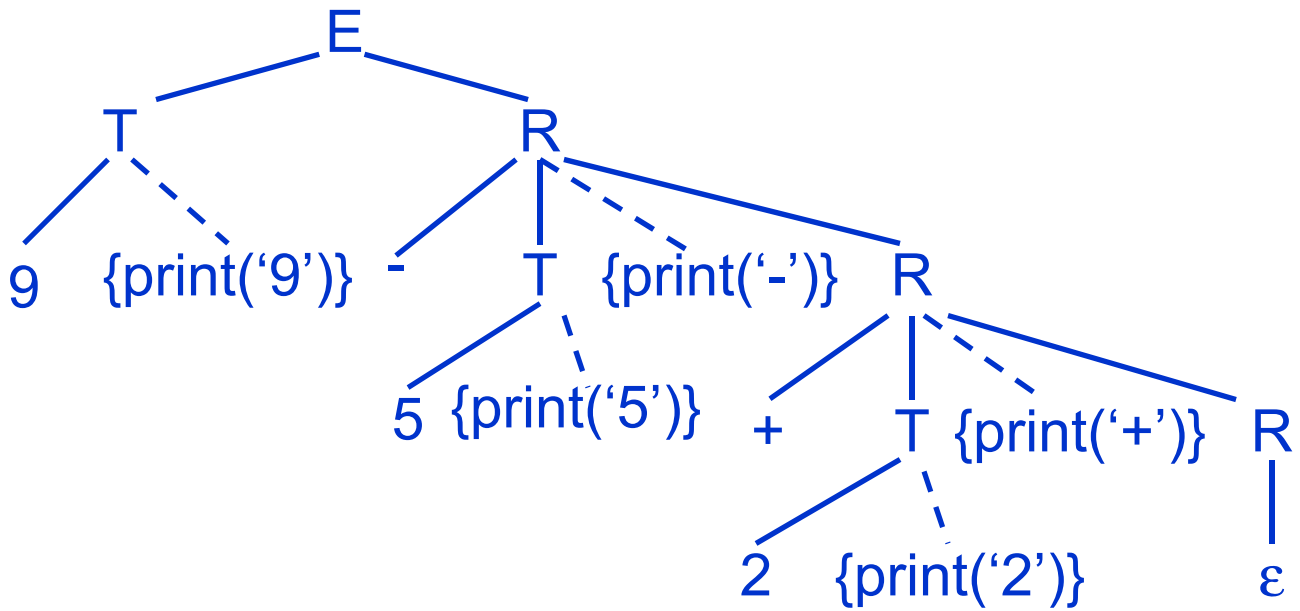
翻译模式示例

$E \rightarrow TR$

$R \rightarrow addop\ T\ \{print(addop.lexeme)\}\ R_1 \mid \varepsilon$

$T \rightarrow num\ \ \{print(num.val)\}$

- 把带加号和减号的中缀表达式翻译成相应的后缀表达式
- 考虑输入串：9-5+2



设计翻译模式的原则

- 设计翻译模式时，必须保证当某个动作引用一个属性时它必须是有定义的
- L-属性文法本身就能确保每个动作不会引用尚未计算出来的属性

建立翻译模式

- 当只需要综合属性时：为每一个语义规则建立一个包含赋值的动作，并把这个动作放在相应的产生式右边的末尾

产生式　　语义规则

$T \rightarrow T_1 * F$　　$T.val := T_1.val \times F.val$

设计产生式和语义动作：

$T \rightarrow T_1 * F$ {$T.val := T_1.val \times F.val$}

建立翻译模式

- 如果既有综合属性又有继承属性，在建立翻译模式时就必须保证：

1. 产生式右边的符号的继承属性必须在这个符号以前的动作中计算出来
2. 一个动作不能引用这个动作右边的符号的综合属性
3. 产生式左边非终结符的综合属性只有在它所引用的所有属性都计算出来以后才能计算。计算这种属性的动作通常可放在产生式右端的末尾

$S \rightarrow A_1 A_2 \{A_1.in:=1; A_2.in:=2\}$

$A \rightarrow a \{print(A.in)\}$

$S \rightarrow \{A_1.in:=1\} A_1 \{A_2.in:=2\} A_2$

$A \rightarrow a \{print(A.in)\}$

编译原理

翻译模式示例

翻译模式示例

- 排版软件 TeX、LaTeX

Donald Ervin Knuth

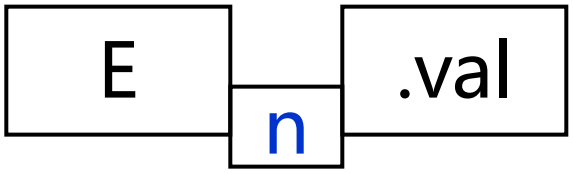
The quadratic formula is $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

```
The quadratic formula is ${-b\pm\sqrt{b^2-4ac} \over {2a}}$
```

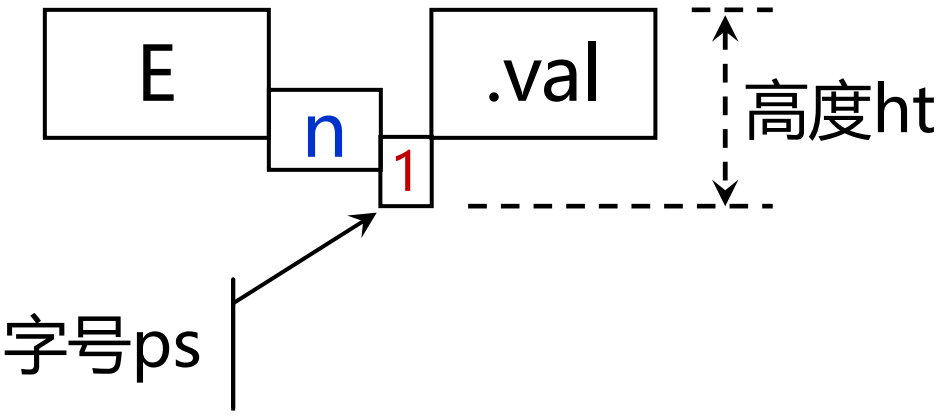
数学格式语言EQN

- 给定输入

E sub n .val



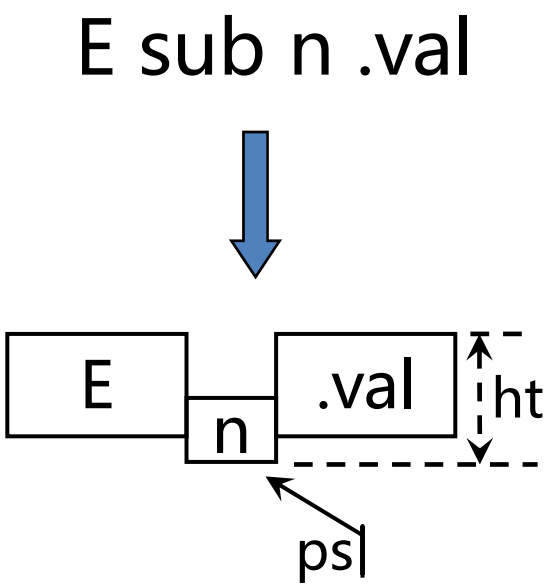
E sub n sub 1.val



数学格式语言EQN

- 识别输入并进行格式转换的L-属性文法

产生式	语义规则
S→B	B.ps :=10
	S.ht :=B.ht
$B \rightarrow B_1 B_2$	B_1.ps :=B.ps
	B_2.ps :=B.ps
	B.ht := max(B_1.ht, B_2.ht)
$B \rightarrow B_1$ sub B_2	B_1.ps :=B.ps
	B_2.ps :=shrink(B.ps)
	B.ht :=disp(B_1.ht, B_2.ht)
B→text	B.ht :=text.h×B.ps



翻译模式

S → { B.ps:=10 } B
{ S.ht:=B.ht }

B → { B_1.ps:=B.ps } B_1
{ B_2.ps:=B.ps } B_2
{ B.ht:=max(B_1.ht,B_2.ht) }

B → { B_1.ps:=B.ps } B_1
sub
{ B_2.ps:=shrink(B.ps) } B_2
{ B.ht:=disp(B_1.ht,B_2.ht) }

B → text { B.ht:=text.h×B.ps }

产生式	语义规则
S→B	B.ps :=10 S.ht :=B.ht
$B→B_1B_2$	B_1.ps :=B.ps B_2.ps :=B.ps B.ht := max(B_1.ht, B_2.ht)
$B→B_1$ sub B_2	B_1.ps :=B.ps B_2.ps :=shrink(B.ps) B.ht :=disp(B_1.ht,B_2.ht)
B→text	B.ht :=text.h×B.ps

1. 产生式右边符号的继承属性必须在该符号以前的动作中计算出来
2. 一个动作不能引用该动作右边符号综合属性
3. 产生式左边非终结符的综合属性只有在它所引用的所有属性都计算出来后才能计算

编译原理

语义动作执行时机统一

建立翻译模式

$E \rightarrow T\ R$

$R \rightarrow +T\ \{ \text{print}(\ '+'\ )\ \}\ R$

$\quad\ \mid -T\ \{ \text{print}(\ '-'\ )\ \}\ R$

$\quad\ \mid \varepsilon$

$T \rightarrow \text{num}\ \{ \text{print}(\text{num.val})\ \}$

建立翻译模式

- 把所有的语义动作都放在产生式的末尾
  - 语义动作的执行时机统一
- 转换方法
  - 加入新产生式M→ε
  - 把嵌入在产生式中的每个语义动作用不同的非终结符M代替，并把这个动作放在产生式M→ε的末尾

```
E → T R
R → ＋T { print ( ‘＋’ ) } R
    | －T { print ( ‘－’ ) } R
    | ε
T → num { print (num.val) }
```

```
E → TR
R → ＋T M R | －T N R | ε
T → num { print (num.val) }
M → ε { print ( ‘＋’ ) }
N → ε { print ( ‘－’ ) }
```

编译原理

消除翻译模式中的左递归

消除翻译模式中的左递归

- 语义动作是在相同位置上的符号被展开（匹配成功）时执行的
- 为了构造不带回溯的自顶向下语法分析，必须消除文法中的左递归

$E \rightarrow E_1+T$	$\{E.val:=E_1.val+T.val\}$
$E \rightarrow E_1-T$	$\{E.val:=E_1.val-T.val\}$
$E \rightarrow T$	$\{E.val:=T.val\}$
$T \rightarrow (E)$	$\{T.val:=E.val\}$
$T \rightarrow num$	$\{T.val:=num.val\}$

$E \rightarrow T\ R$
$R \rightarrow +\ T\ R_1$
$R \rightarrow -\ T\ R_1$
$R \rightarrow \varepsilon$
$T \rightarrow (\ E\ )$
$T \rightarrow num$

消除翻译模式中的左递归

- 语义动作是在相同位置上的符号被展开（匹配成功）时执行的
- 为了构造不带回溯的自顶向下语法分析，必须消除文法中的左递归
- 当消除一个翻译模式的基本文法的左递归时同时考虑属性计算
  - 适合带综合属性的翻译模式

消除翻译模式中的左递归

- 消除左递归，构造新的翻译

$E \rightarrow E_1+T$	{E.val:=E_1.val+T.val}
$E \rightarrow E_1-T$	{E.val:=E_1.val-T.val}
$E \rightarrow T$	{E.val:=T.val}
$T \rightarrow (E)$	{T.val:=E.val}
$T \rightarrow num$	{T.val:=num.val}

$E \rightarrow$	T {R.i:=T.val}
	R {E.val:=R.s}
$R \rightarrow$	+
	T {R_1.i:=R.i+T.val}
	R_1 {R.s:=R_1.s}
$R \rightarrow$	-
	T {R_1.i:=R.i-T.val}
	R_1 {R.s:=R_1.s}
$R \rightarrow$	ε {R.s:=R.i}
$T \rightarrow$	(E) {T.val:=E.val}
$T \rightarrow$	num {T.val:=num.val}

$E \rightarrow T\ R$

$R \rightarrow +TR_1$

$R \rightarrow -TR_1$

$R \rightarrow \varepsilon$

$T \rightarrow (\ E\ )$

$T \rightarrow num$

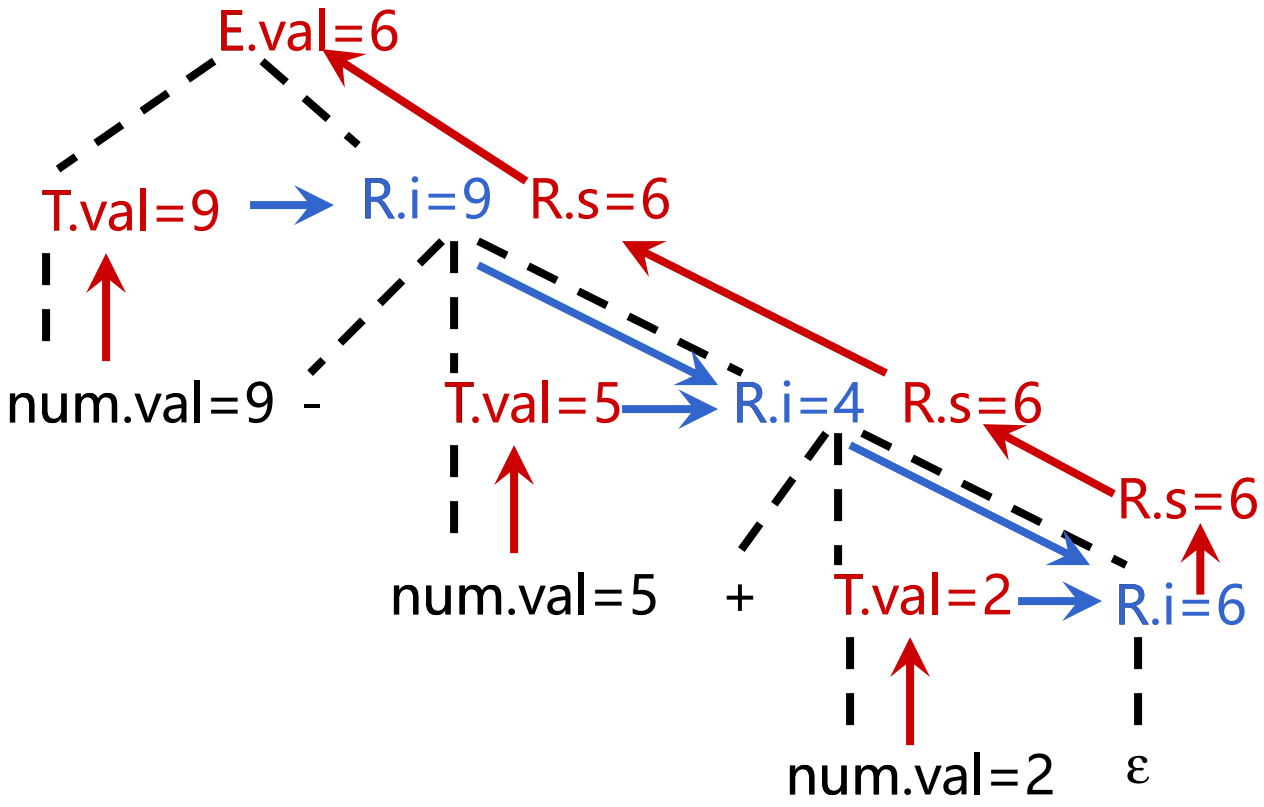
R.i: R前面子表达式的值

R.s: 分析完R时子表达式的值

消除翻译模式中的左递归

- 计算表达式9 - 5 + 2

E	→ T	{R.i:=T.val}
	R	{E.val:=R.s}
R	→ +	
	T	{R_1.i:=R.i+T.val}
	R_1	{R.s:=R_1.s}
R	→ -	
	T	{R_1.i:=R.i-T.val}
	R_1	{R.s:=R_1.s}
R	→ ε	{R.s:=R.i}
T	→ (E)	{T.val:=E.val}
T	→ num	{T.val:=num.val}



消除翻译模式中的左递归

$E \rightarrow T\ R$
$R \rightarrow +TR_1$
$R \rightarrow -TR_1$
$R \rightarrow \varepsilon$
$T \rightarrow (\ E\ )$
$T \rightarrow num$
R.i: R前面子表达式的值
R.s: 分析完R时子表达式的值

- 假设有翻译模式：

  $A \rightarrow A_1 Y \ \{A.a:=g(A_1.a, Y.y)\}$

  $A \rightarrow X \{A.a:=f(X.x)\}$

它的每个文法符号都有一个综合属性，用小写字母表示，g和f是任意函数。

基础文法消除左递归：

$A \rightarrow XR$

$R \rightarrow YR \mid \varepsilon$

翻译模式消除左递归：

$A \rightarrow X \quad \{R.i:=f(X.x)\}$

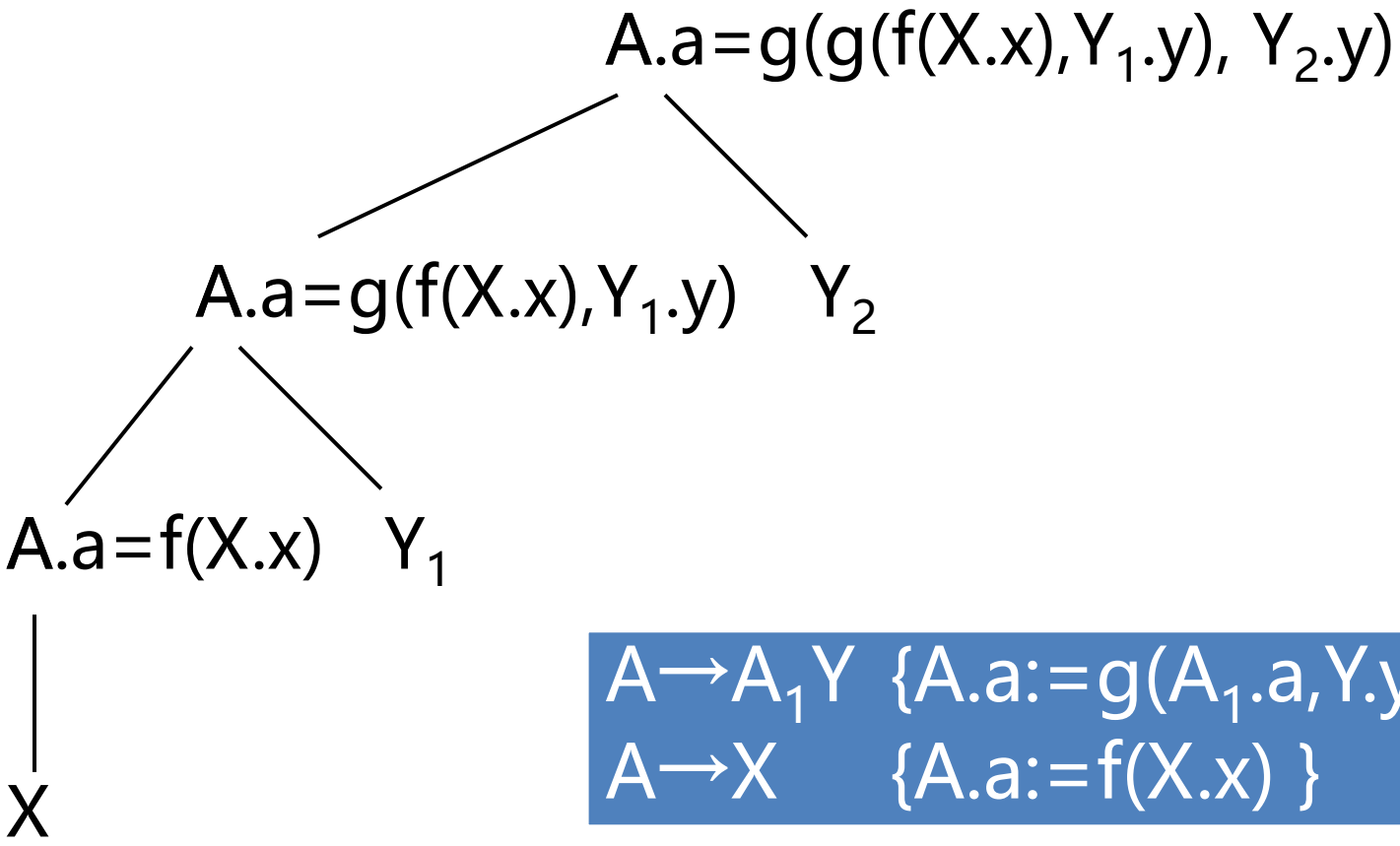
$\quad\quad R \quad \{A.a:=R.s\}$

$R \rightarrow Y \{R_1.i:=g(R.i, Y.y)\}\ R_1 \{R.s:=R_1.s\}$

$R \rightarrow \varepsilon \{R.s:=R.i\}$

消除翻译模式中的左递归

- XYY



$A \rightarrow A_1 Y$ $\{A.a := g(A_1.a, Y.y)\}$

$A \rightarrow X$ $\{A.a := f(X.x)\}$

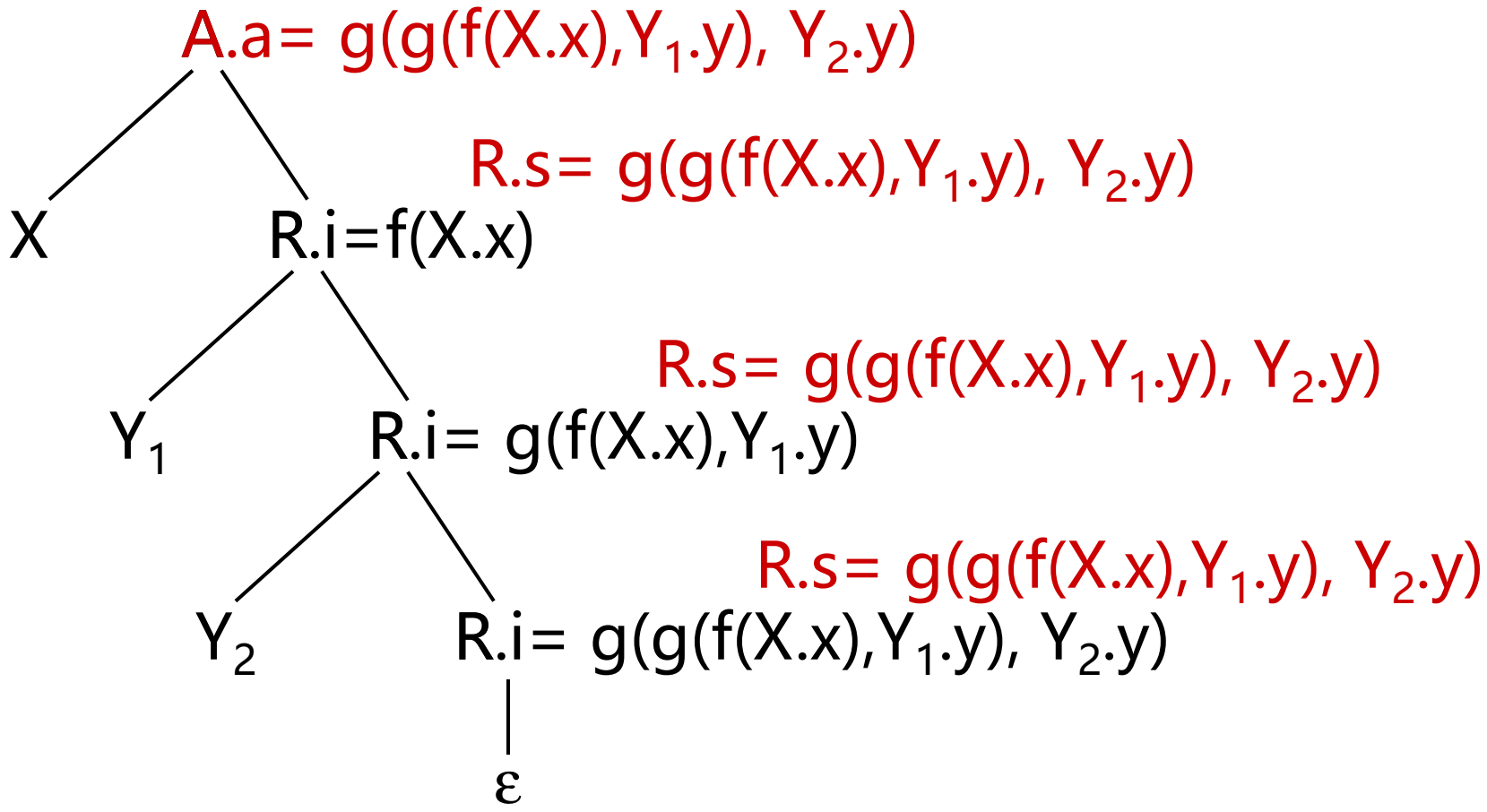
消除翻译模式中的左递归

$A \rightarrow X \{R.i:=f(X.x)\}\ R \{A.a:=R.s\}$

$R \rightarrow Y \{R_1.i:=g(R.i, Y.y)\}\ R_1 \{R.s:=R_1.s\}$

$R \rightarrow \varepsilon \{R.s:=R.i\}$

- XYY



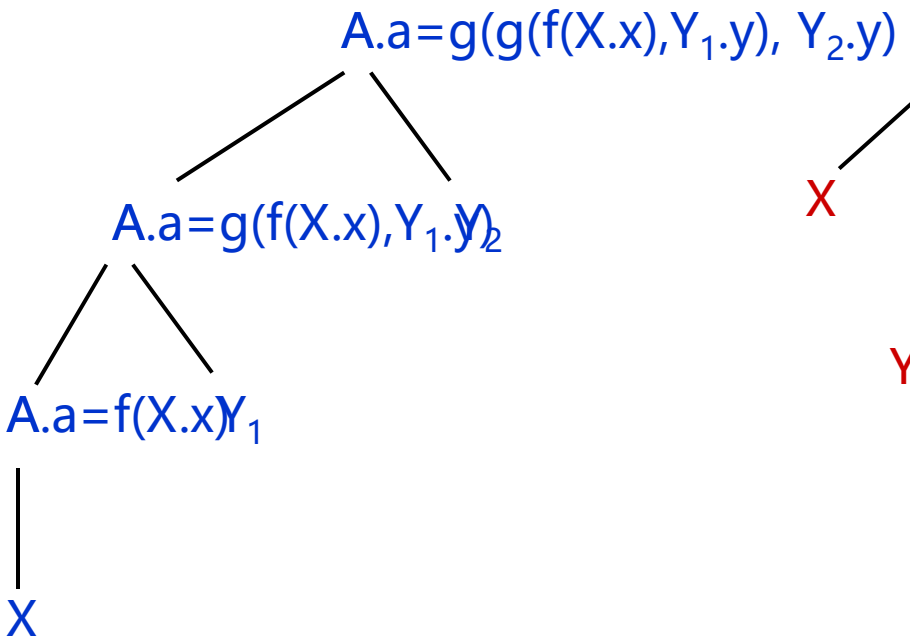
消除翻译模式中的左递归

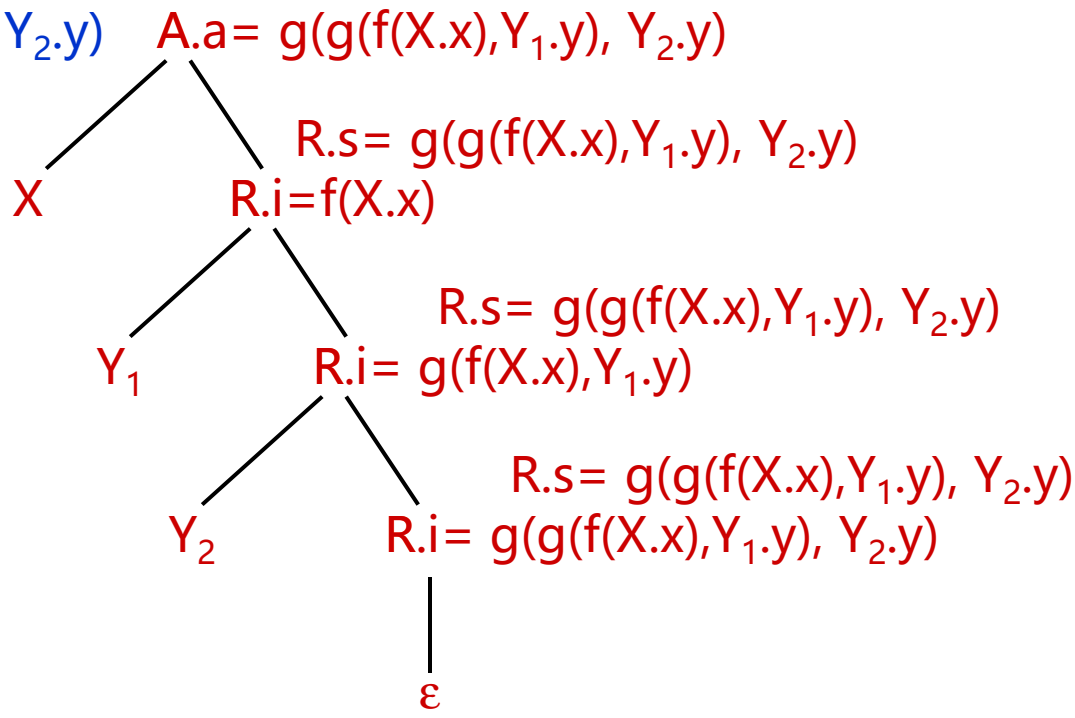
$A \rightarrow X \{R.i := f(X.x)\}\ R \{A.a := R.s\}$

$R \rightarrow Y \{R_1.i := g(R.i, Y.y)\}\ R_1 \{R.s := R_1.s\}$

$R \rightarrow \varepsilon \{R.s := R.i\}$

- XYY





$A \rightarrow A_1 Y \{A.a := g(A_1.a, Y.y)\}$

$A \rightarrow X \quad \{A.a := f(X.x)\}$

$A \rightarrow A_1 Y$ {A.a:=g(A_1.a,Y.y) }
$A \rightarrow X$ {A.a:=f(X.x) }

消除翻译模式中的左递

A → X {R.i:=f (X.x) } R {A.a:=R.s}
R → Y {R_1.i:=g(R.i, Y.y)} R_1{R.s:=R_1.s}
R → ε {R.s:=R.i}

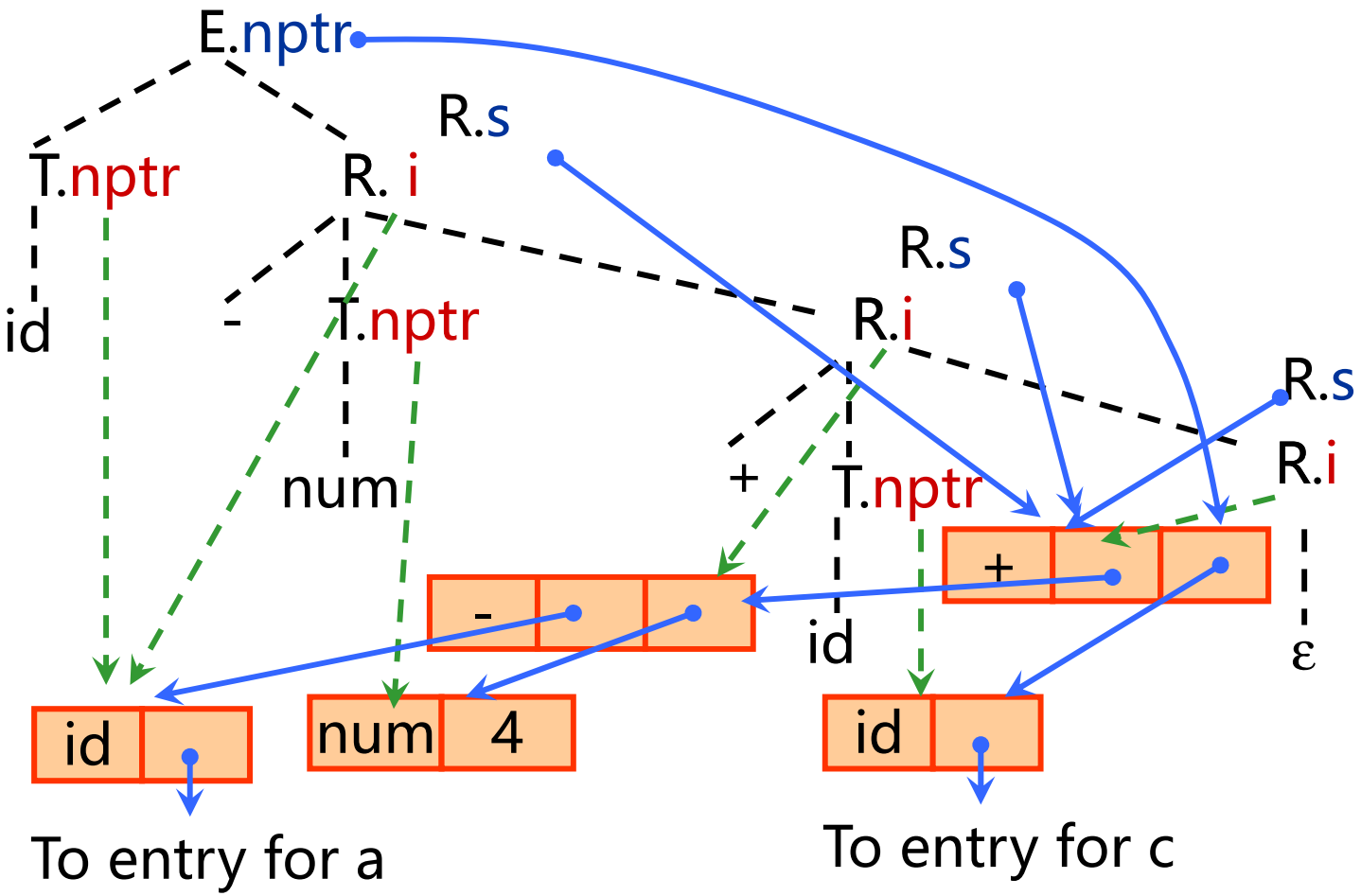
- 抽象语法树的翻译模式

$E \rightarrow E_1$+T {E.nptr:=mknode('+',E_1.nptr,T.nptr)}
$E \rightarrow E_1$-T {E.nptr:=mknode('-',E_1.nptr,T.nptr)}
E→T {E.nptr:=T.nptr}

E → T { R.i:=T.nptr } R { E.nptr:=R.s }
R → + T { R_1.i:=mknode('+', R.i, T.nptr) } R_1 { R.s:=R_1.s }
R → - T { R_1.i:=mknode('—', R.i, T.nptr) } R_1 { R.s:=R_1.s }
R →ε { R.s:=R.i }
T → (E) { T.nptr:=E.nptr }
T → id { T.nptr:=mkleaf(id, id.entry) }
T → num { T.nptr:=mkleaf(num, num.val) }

- 构造a - 4 + c 的抽象语法树

$E \rightarrow T\ \{R.i:=T.nptr\}\ \ R\ \{E.nptr:=R.s\}$
$R \rightarrow +\ \ T\ \{R_1.i:=mknode(\ '+'\ ,R.i,T.nptr)\}\ R_1\ \{R.s:=R_1.s\}$
$R \rightarrow -\ T\ \{R_1.i:=mknode(\ '-'\ ,R.i,T.nptr)\}\ R_1\ \{R.s:=R_1.s\}$
$R \rightarrow \varepsilon\ \{R.s:=R.i\}$
$T \rightarrow (\ E\ )\ \{T.nptr:=E.nptr\}$
$T \rightarrow id\ \{T.nptr:=mkleaf(id,id.entry)\}$
$T \rightarrow num\ \{T.nptr:=mkleaf(num,num.val)\}$



编译原理

递归下降翻译器的设计

递归下降翻译器的设计

- 递归下降分析器的设计
  - 分析程序由一组递归子程序(函数)组成，每个非终结符对应一个子程序(函数)

递归下降翻译器的设计

- R→addop TR|ε的递归下降分析过程

```
procedure R;
begin
     if sym=addop then begin
          advance;T; R
     end
     else begin /*do nothing*/
     end
end;
```

R → addop
T {R_1.i:= mknode(addop.lexme, R.i, T.nptr)}
R_1 {R.s:=R_1.s}
R→ ε {R.s:=R.i}

递归下降翻译器的设计

- 对每个非终结符A构造一个函数过程
- A的属性实现为参数和变量
  - 继承属性：对A的每个继承属性设置为函数的一个形式参数
  - 综合属性：实现为函数的返回值
    - 若有多个综合属性，打包成作为结构或记录记录返回
    - 为了简单，我们假设每个非终结只有一个综合属性
  - A的产生式中的每一个文法符号的每一个属性：实现为A对应的函数过程中的局部变量

递归下降翻译器的设计

- 函数的功能
  - 非终结符A对应的函数过程中，根据当前的输入符号决定使用哪个产生式候选

递归下降翻译器的设计

- 按照产生式右部从左到右的，对于单词符号（终结符）、非终结符和语义动作，分别实现
  - 对于带有综合属性x的终结符X，把x的值存入为X.x设置的变量中。然后产生一个匹配X的调用，并继续读入一个输入符号。
  - 对于每个非终结符B，产生一个右边带有函数调用的赋值语句c=B(b_1,b_2,...,b_k)，其中，b_1,b_2,...,b_k是为B的继承属性设置的变量，c是为B的综合属性设置的变量。
  - 对于语义动作，把动作的代码抄进分析器中，用代表属性的变量来代替对属性的每一次引用。

递归下降翻译器的设计

- 抽象语法树的翻译模式

$E \rightarrow T \{ R.i := T.nptr \}\ R \{ E.nptr := R.s \}$

$R \rightarrow + T \{ R_1.i := mknode(\text{‘+’}, R.i, T.nptr) \}\ R_1 \{ R.s := R_1.s \}$

$R \rightarrow - T \{ R_1.i := mknode(\text{‘—’}, R.i, T.nptr) \}\ R_1 \{ R.s := R_1.s \}$

$R \rightarrow \varepsilon \{ R.s := R.i \}$

$T \rightarrow (E) \{ T.nptr := E.nptr \}$

$T \rightarrow id \{ T.nptr := mkleaf(id, id.entry) \}$

$T \rightarrow num \{ T.nptr := mkleaf(num, num.val) \}$

递归下降翻译器的设计

- 非终结符E、R、T的函数及其参数类型

  function E:↑AST-node;

  function R(in:↑AST-node): ↑AST-node;

  function T: ↑AST-node;

- 用addop代表 + 和 -

R → addop

T {R_1.i := mknode(addop.lexme, R.i,T.nptr) }

R_1 {R.s:=R_1.s}

R → ε { R.s:=R.i }

递归下降翻译器的设计

- R→addop TR|ε的递归下降分析过程

```
procedure R;
begin
     if sym=addop then begin
          advance;T; R
     end
     else begin /*do nothing*/
     end
end;
```

R → addop
T {R_1.i:= mknode(addop.lexme, R.i, T.nptr)}
R_1 {R.s:=R_1.s}
R→ ε {R.s:=R.i}

递归下降

$R \rightarrow addop\ T\ \{R_1.i := mknode(addop.lexme,\ R.i,\ T.nptr)\}\ R_1\ \{R.s := R_1.s\}$

$R \rightarrow \varepsilon\ \{R.s := R.i\}$

- R→addop TR|ε的过程

```
procedure R;
begin
      if sym=addop then begin
            advance;T; R
      end
      else begin /*do nothing*/
      end
end;
```

```
function R (in:↑AST-node): ↑AST-node;
      var nptr, i1,s1,s: ↑AST-node;
      addoplexeme: char;
begin
      if sym=addop then begin/*产生式 R→addop TR */
            addoplexeme:=lexval;
            advance;
            nptr:=T;
            i1:=mknode (addoplexeme, in, nptr);
            s1:=R (i1)
            s:=s1
      end
      else  s:= in;
      return  s
end;
```

小结

- S-属性文法
- L-属性文法
- 翻译模式
- 翻译模式的改造——消除左递归
- 自顶向下翻译——构造递归下降翻译器