

第二章 一个简单的编译器



学习内容

- 构造一个以**语法分析器**为核心的编译器——将中缀表达式转换为后缀表示形式
- 利用**上下文无关文法描述**源语言的语法结构
- 利用**语法制导翻译**方法进行表达式转换
- 构造**预测分析器**，进行**语法分析**，并同时
进行语法制导翻译



学习内容（续）

- 构造更复杂的**词法分析器**——消除单字符单词的限制，支持变量
- 符号表的简单实现方法



2.1 概述

- 语法描述

上下文无关文法, context-free grammar

巴科斯-瑙尔范式, Backus-Naur Form,

BNF

- 语义描述: 非形式化描述

- 辅助代码生成:

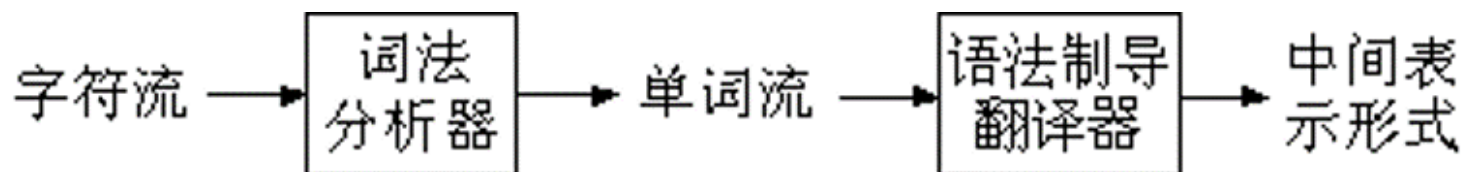
语法制导翻译, syntax-directed translation

构造一个简单的编译器

- 目标：表达式中缀表示 \square 后缀表示

$9-5+2 \square 95-2+$

- 过程：



- 语法制导翻译器：语法分析 + 中间代码生成



2.2 语法定义

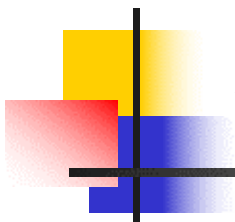
- 上下文无关文法：描述语言的语法结构
- **if** (expression) statement **else** statement

对应的文法规则

$stmt \sqsubseteq \mathbf{if} (expr) stmt \mathbf{else} stmt$

可 具有形式为...

- 产生式, production
- **if**, **else**——单词, 终结符号, terminal
- *expr*, *stmt*——单词序列, 非终结符号, nonterminal



上下文无关文法

○ 四个部分组成

1. 一组**终结符号**，单词，基本符号
2. 一组**非终结符号**（语法变量），语法范畴，
语法概念
3. 一组**产生式**，定义语法范畴

产生式： $A \rightarrow \alpha$

A—一个非终结符，**左部**

α —终结符或/与非终结符串，**右部**

4. 一个特定的非终结符——**开始符号**，**start**

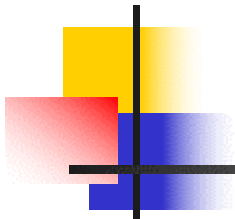
symbol



形式化定义

- 几个概念
- Σ : 有穷**字母表**, 元素——符号
- **符号串**: Σ 中符号构成的有穷序列
- **空字**: 不含任何符号的序列, ε
- Σ^* : 符号串全体, 包括空字
- \varnothing : 空集 $\{\}$, 区分 ε , $\{\}$, $\{\varepsilon\}$
- Σ^* 的子集 U 、 V 的**积** (**连接**)

$$UV = \{\alpha\beta \mid \alpha \in U \text{ 且 } \beta \in V\}$$



几个概念（续）

- $UV \neq VU$, $(UV)W = U(VW)$
- V 自身的 n 次积（连接）记为 V^n
- $V^0 = \{\epsilon\}$
- V 的**闭包**（**closure**）

$$V^* = V^0 \cup V^1 \cup V^2 \cup V^3 \cup \dots$$

每个符号串，都是 V 中符号串有限次连接

- **正则闭包**, $V^+ = VV^*$



四元式定义上下文无法文法

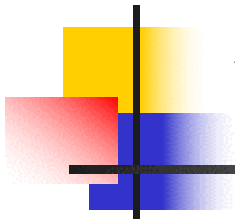
- (V_T, V_N, S, P)
- V_T : 非空有限集, 终结符号集合
- V_N : 非空有限集, 非终结符号集合
- S : 开始符号
- P : 产生式集合 (有限集)

每个产生式形式 $A \rightarrow \alpha$, 其中

$$A \in V_N, \alpha \in (V_T \cup V_N)^*$$

关于 A 的产生式

S 至少在某个产生式左部出现一次



例2.1 符号约定

$expr \sqsupseteq expr + digit$

$expr \sqsupseteq expr - digit$

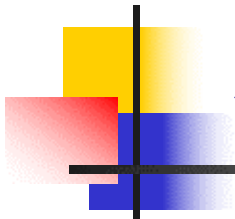
$expr \sqsupseteq digit$

$digit \sqsupseteq 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

- 数字、运算符、**黑体字符串**——终结符
- *斜体字符串*——非终结符
- 左部相同可合并，‘|’ —— “或” 的意思

$expr \sqsupseteq expr + digit \mid expr - digit \mid digit$

候选式



产生式设计练习

- 首先是“人会做”

- 我们自己先把语法概念“什么模样”搞清楚

- 然后是“让计算机做”——符号化

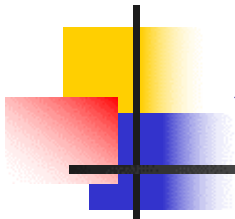
- 为这个语法概念起个名字，“模样”中的其他语法

- 概念、单词也都有相应的名字

- 将语法概念放在产生式左部

- “模样”放在产生式右部

- 都是用名字替换掉语法概念和单词——



产生式设计练习

- **while** (expression) statement

对应的产生式

$stmt \sqsubset \text{while} (expr) stmt$

- **for** (expression₁; expression₂; expression₃)

statement

对应的产生式

$stmt \sqsubset \text{for} (expr ; expr ; expr) stmt$

- **int(float, double, char)** id₁, id₂, ...;

$type \sqsubset \text{int} \mid \text{float} \mid \text{double} \mid \text{char}$

$idlist \sqsubset idlist , id \mid id$

$decl \sqsubset type idlist ;$



推导

- 单词串(string): 0个或多个单词构成的序列

- 推导(derive)

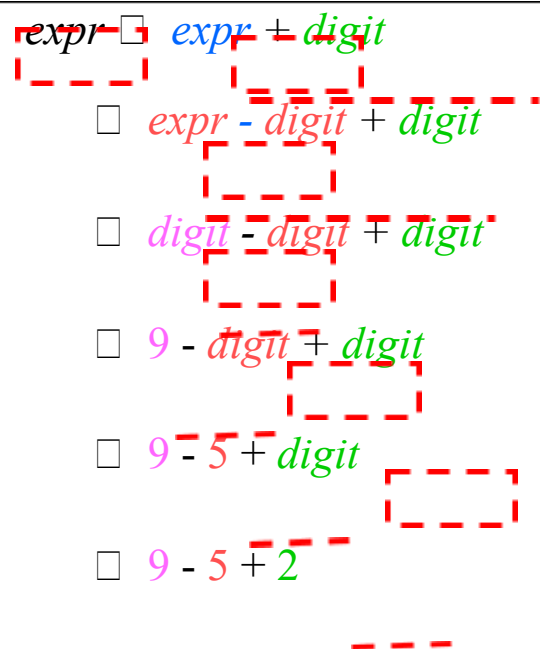
- 由开始符号作为推导起点
- 用产生式右部替换左部非终结符
- 反复替换, 最终得到单词串

- 语言(language)

语法所定义的语言——可由开始符号推导出的所有单词串的集合

例2.2

前面定义的表达式的CFG，可按如下步骤推导出表达式：9 - 5 + 2



P1 : $expr \rightarrow expr + digit$

P2 : $expr \rightarrow expr - digit$

P3 : $expr \rightarrow digit$

P4 : $digit \rightarrow 9$

P4 : $digit \rightarrow 5$

P4 : $digit \rightarrow 2$

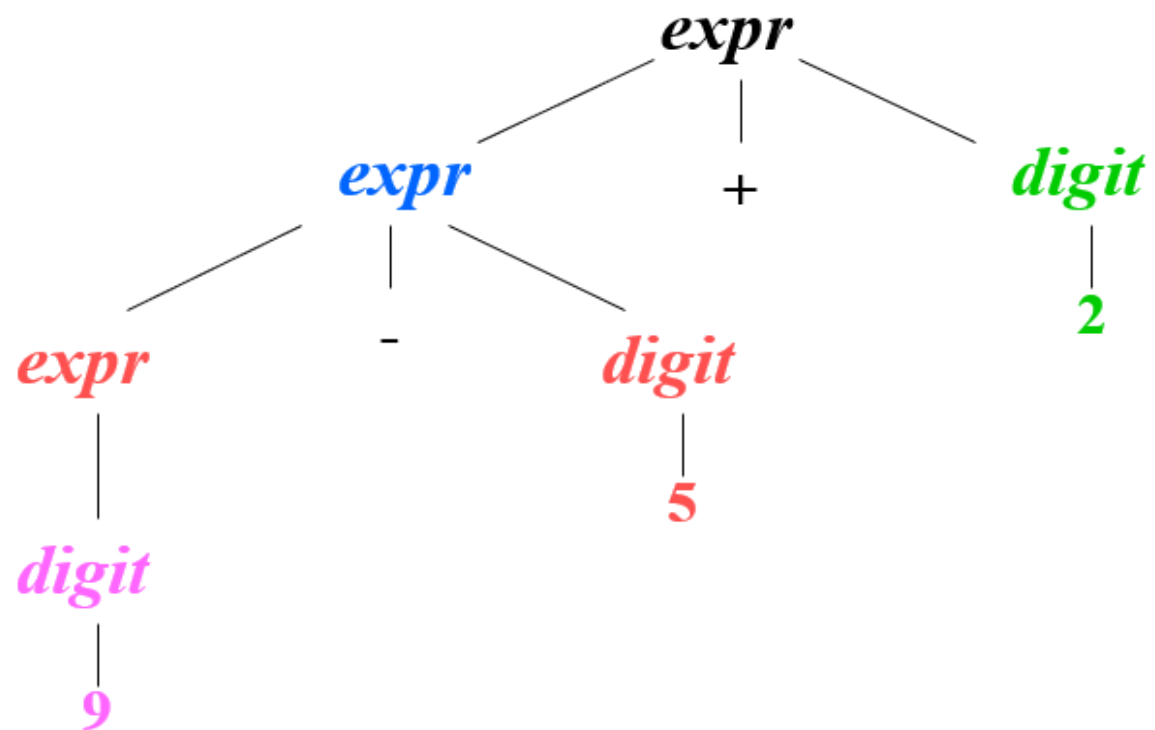
$expr \rightarrow expr + digit$

$expr \rightarrow expr - digit$

$expr \rightarrow digit$

$digit \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$

例2.2（续）

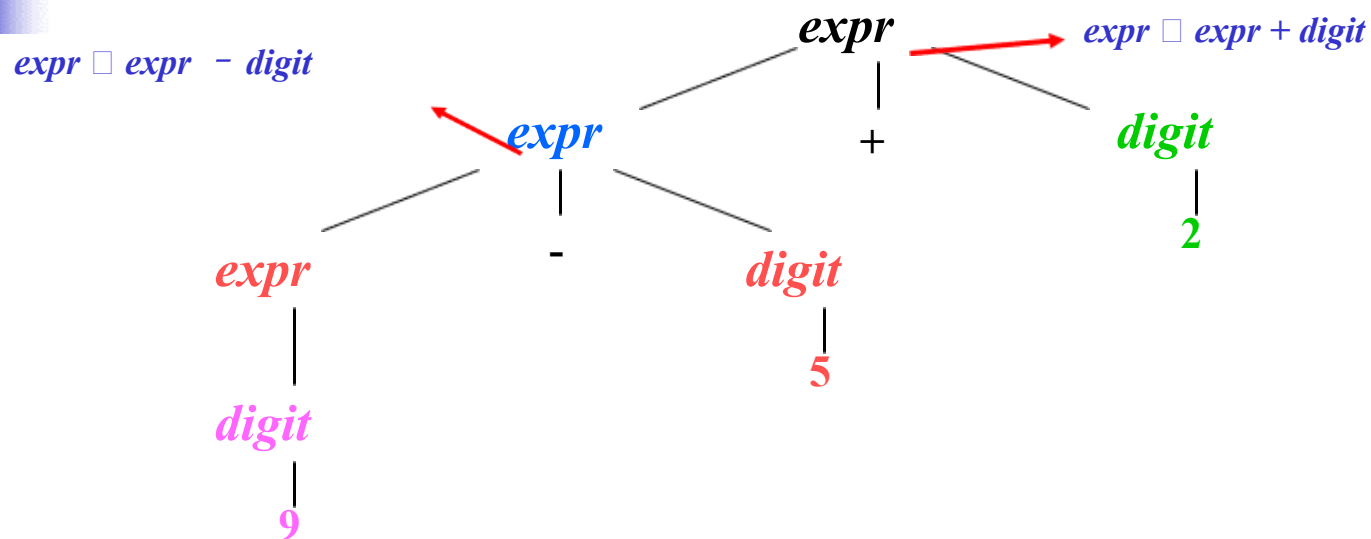




2.2.1 语法分析树(parse tree)

- 图示推导过程，推导□语法树：
 - 根节点标记为开始符号
 - 每个叶节点用一个T或 ε 标记
 - 每个内部节点用一个NT标记
 - 节点A，孩子节点 X_1, X_2, \dots, X_n
 - 节点应用了产生式 $A \rightarrow X_1 X_2 \dots X_n$
 - 对 $A \rightarrow \varepsilon$ ，节点A只有一个孩子节点 ε

例2.4



- 叶节点从左至右构成树的**输出 (yield)**
 - 开始符号**推导出 (生成)**的单词串
- **语言**: 语法分析树生成的单词串集合

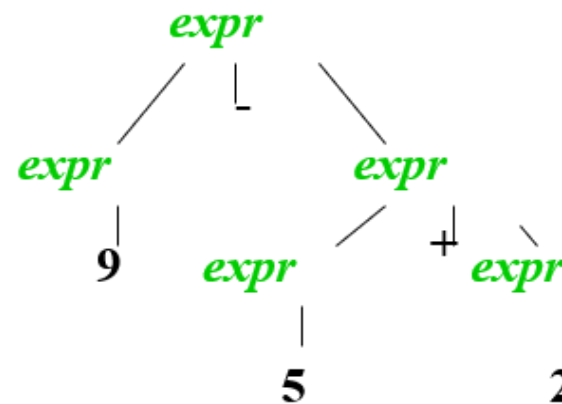
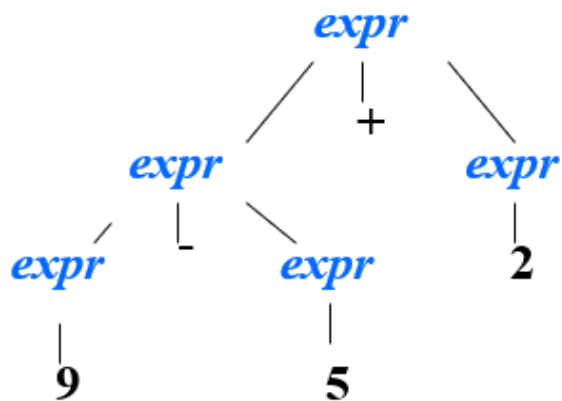


2.2.2 二义性 (ambiguity)

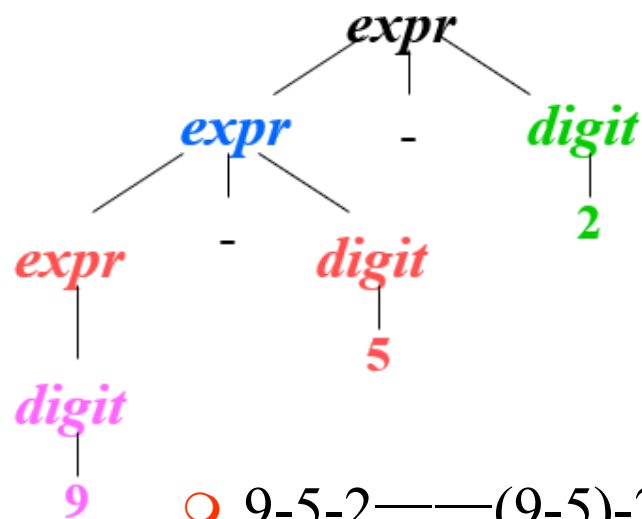
- 多个语法分析树生成相同的单词串
——多个意义
- 定义无歧义语法或用附加规则消除歧义

例2.5

$expr \square expr + expr \mid expr - expr \mid 0 \mid 1 \mid \cdots \mid 9$



2.2.3 运算符的结合率



$expr \square expr + digit$

$expr \square expr - digit$

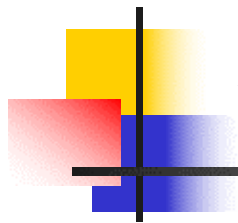
$expr \square digit$

$digit \square 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$

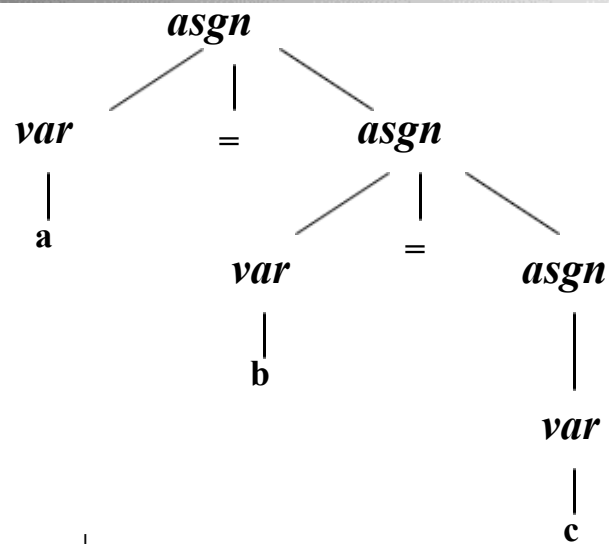
○ 9-5-2——(9-5)-2 还是 9-(5-2)?

○ 左结合 (left associative) : +, -, *, /

○ 右结合 (right associative) : ^, =



右结合例



$asgn \sqsupseteq var = asgn \mid var$

$var \sqsupseteq a \mid b \mid c \mid \cdots \mid z$



2.2.4 运算符优先级

- $9+5*2$ —— $(9+5)*2$ 还是 $9+(5*2)$?
- 运算符优先级: $*$ 的优先级比 $+$ 高
- 典型的优先级

$\left\{ \begin{array}{l} () \\ * / \text{——左结合} \\ + - \text{——左结合} \end{array} \right.$



结合优先级的表达式文法

$expr \square expr + term \mid expr - term \mid term$

$term \square term * factor \mid term / factor \mid factor$

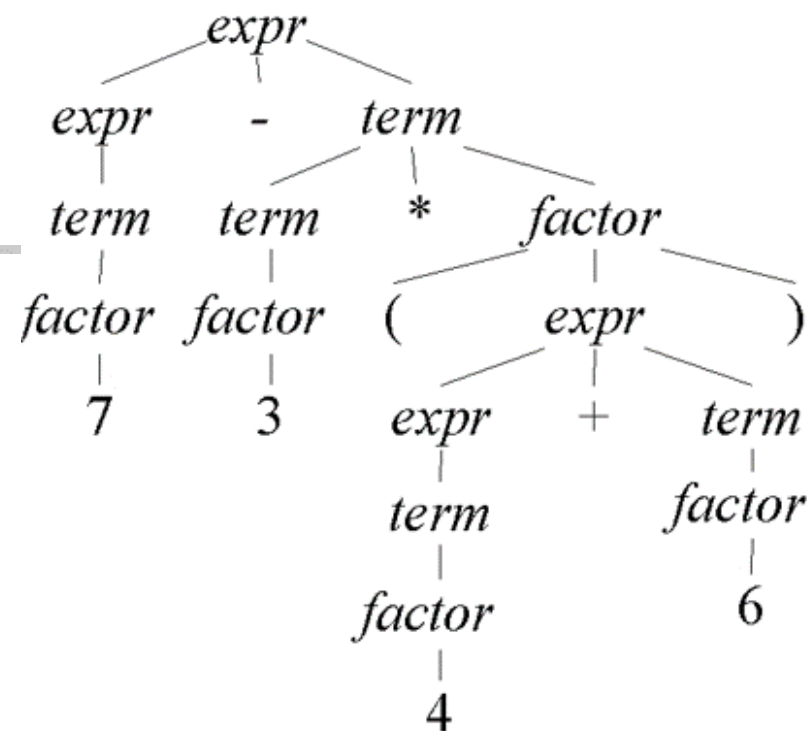
$factor \square \mathbf{digit} \mid (expr)$

$digit \square 0 \mid 1 \mid 2 \mid 3 \mid \cdots \mid 9$

○ $expr$ 、 $term$ 、 $factor$ ——不同的优先级

推导练习

○ $7 - 3 * (4 + 6)$



$\square \text{expr} \rightarrow \text{expr} - \text{term}$

$\square \text{term} \rightarrow \text{term}$

$\square \text{factor} \rightarrow \text{term}$

$\square 7 \rightarrow \text{term}$

$\square 7 \rightarrow \text{term} * \text{factor}$

$\square 7 \rightarrow \text{factor} * \text{factor}$

$\square 7 \rightarrow 3 * \text{factor}$

$\square 7 \rightarrow 3 * (\text{expr})$

$\square 7 \rightarrow 3 * (\text{expr} + \text{term})$

$\square 7 \rightarrow 3 * (\text{term} + \text{term})$

$\square 7 \rightarrow 3 * (\text{factor} + \text{term})$

$\square 7 \rightarrow 3 * (4 + \text{term})$

$\square 7 \rightarrow 3 * (4 + \text{factor})$

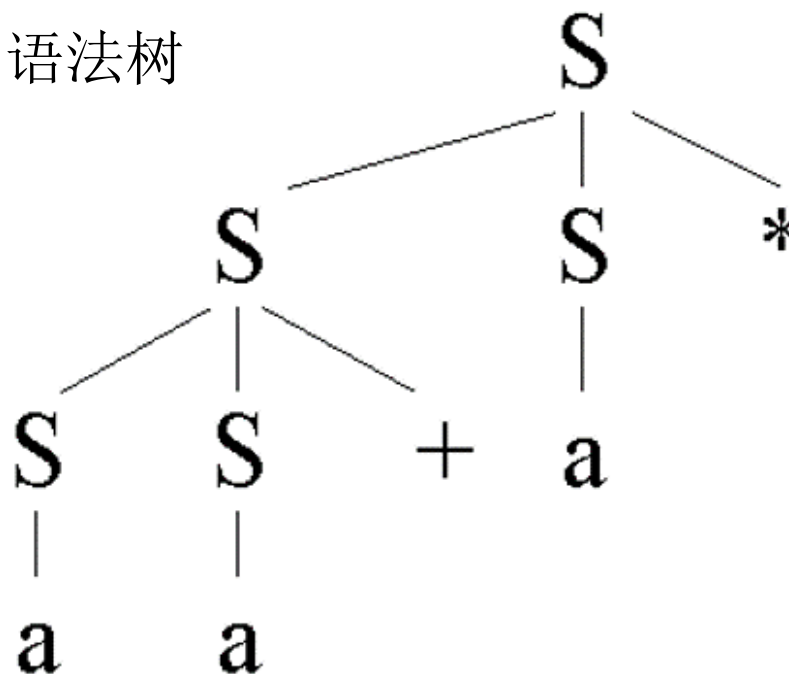
$\square 7 \rightarrow 3 * (4 + 6)$

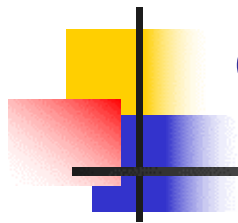
推导练习

○ $S \rightarrow S S + \mid S S * \mid a$

给出 $aa+aa^*$ 的推导和语法树

$$\begin{aligned} S &\rightarrow S S * \\ &\rightarrow S S + S * \\ &\rightarrow a S + S * \\ &\rightarrow a a + S * \\ &\rightarrow a a + a * \end{aligned}$$





Grammar-based compression

$x = 1001110001000111000111111000 \square$

$s_0 \square s_1 s_3 s_2 s_3 s_4 s_4 s_3$

$s_1 \square 100$

$s_2 \square s_1 0$

$s_3 \square s_4 s_2$

$s_4 \square 11$

解压——推导

$s_0 \square s_1 s_3 s_2 s_3 s_4 s_4 s_3 \square s_1 s_4 s_2 s_2 s_4 s_2 s_4 s_4 s_4 s_2$

$\square s_1 s_4 s_1 0 s_1 0 s_4 s_1 0 s_4 s_4 s_4 s_1 0$

$\square 100 s_4 10001000 s_4 1000 s_4 s_4 s_4 1000$

$\square 1001110001000111000111111000$



2.3 语法制导翻译

- 翻译：为生成代码，需保存语言结构的类型、代码位置、代码数量等
- 属性(attribute)：类型、串、内存位置等
- 语法制导翻译

syntax-directed translation

- 语法制导定义

syntax-directed definition

属性与语法结构相关联□指明翻译方法

- 翻译模式, translation scheme



怎么设计语法制导翻译程序

- 首先还是“人会做”

- 设计文法

- 程序员自己先理清翻译的方法——每个语法结构如何翻译

- 然后“让计算机做”

- 设计文法符号属性，表示翻译结果

- 把翻译方法转换为语义规则（语义动作）——属性的运算，将其附着于产生式

- 手工编写或利用自动生成工具形成语法分析程序，语义规则（语义动作）嵌入到程序的恰当位置，恰好实现了翻译方法



2.3.1 表达式的后缀表示法

○ 表达式E的后缀形式Postfix(E)如何生成:

1. E为变量或常量: $\text{Postfix}(E) = E$

2. $E = E_1 \text{ op } E_2$, **op**—二元运算符, E_1 、 E_2 —子表达式:

$$\begin{aligned}\text{Postfix}(E) &= \text{Postfix}(E_1 \text{ op } E_2) \\ &= \text{Postfix}(E_1) \text{ Postfix}(E_2) \text{ op}\end{aligned}$$

3. $E = (E_1)$:

$$\text{Postfix}(E) = \text{Postfix}(E_1)$$

○ $(9 - 5) + 2 \square 9 \ 5 - 2 +$

$9 - (5 + 2) \square 9 \ 5 \ 2 + -$



2.3.2 语法制导定义

- 基于语言的上下文无关文法
- 语法符号——^{关联}组属性
- 产生式——^{关联}组**语义规则**(semantic rule)
 - 属性值计算规则
- CFG+语义规则 □ 语法制导定义



语法制导翻译的基本过程

- 翻译——输入□输出映射过程
 1. 输入单词串 x □语法分析树
 2. 节点 n 标记为 X , $X.a$ —— X 的属性
 3. 计算节点 n 的 $X.a$ 的值——利用 X 产生式的语义规则□
 4. “注释语法分析树” (**annotated parse tree**)

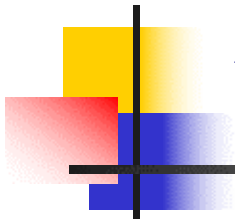


2.3.3 综合属性

- synthesized attributes:

节点属性值由其孩子节点属性值所决定

- 自底向上（bottom-up）计算

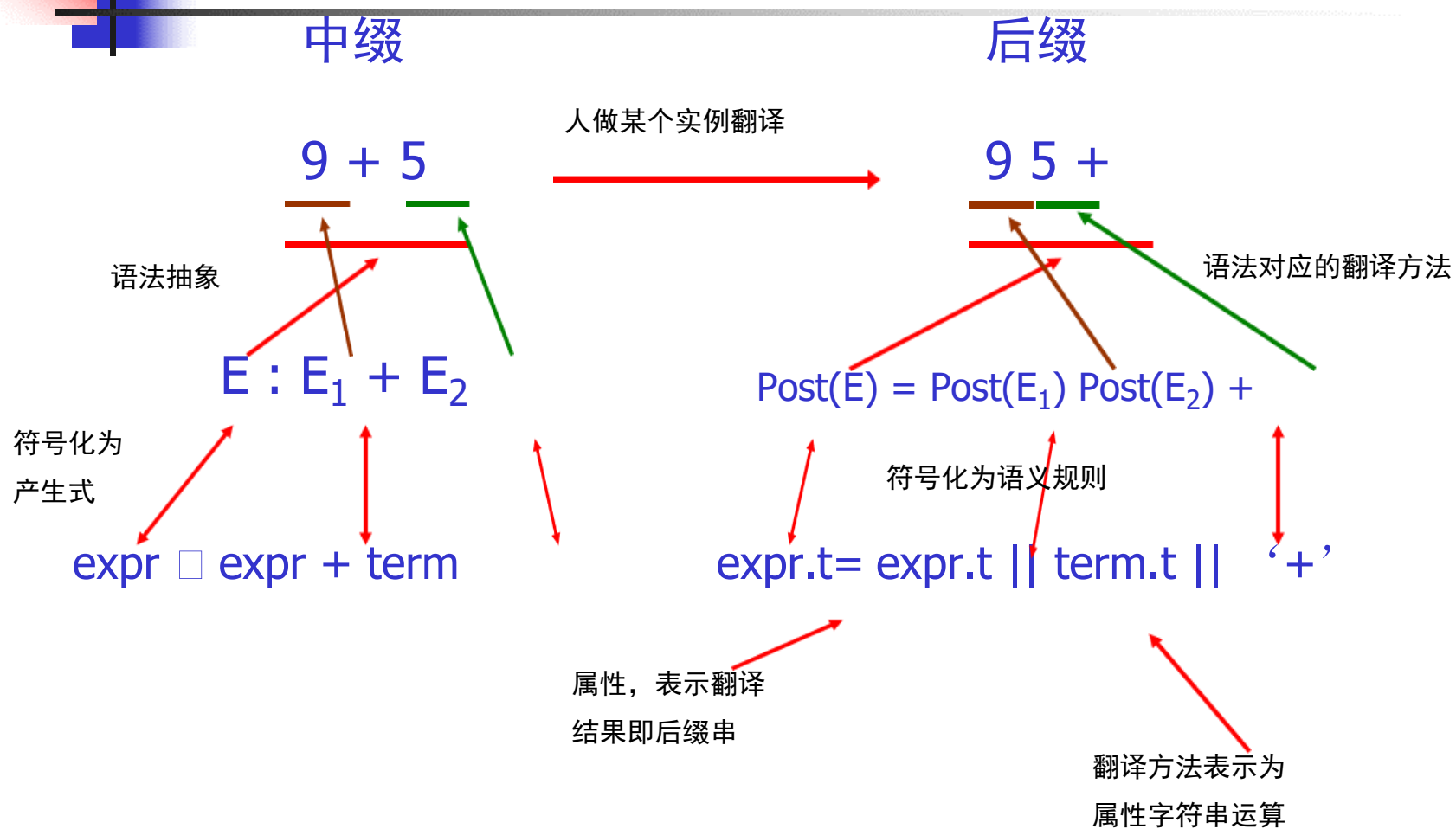


例2.6

$$\begin{aligned} \text{expr} &\rightarrow \text{expr} - \text{term} \mid \text{expr} + \text{term} \mid \text{term} \\ \text{term} &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid \dots \mid 9 \end{aligned}$$

expr、*term*都设置属性t——字符串型，
表示表达式的后缀表示形式

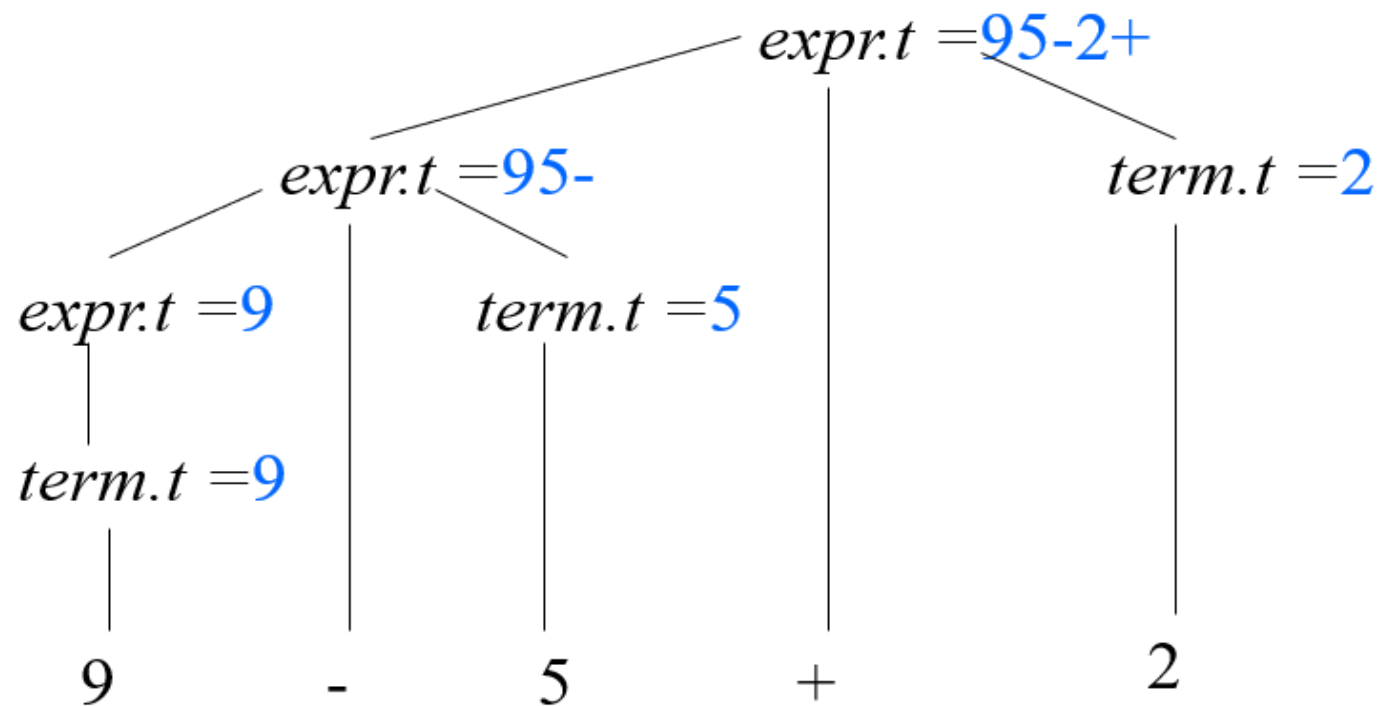
翻译方法□语义规则

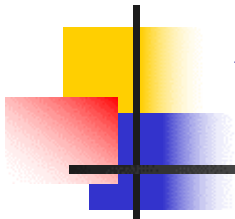


例2.6（续）

Production	Semantic Rule
$expr \sqsupseteq expr_1 + term$	$expr.t = expr_1.t \parallel term.t \parallel '+'$
$expr \sqsupseteq expr_1 - term$	$expr.t = expr_1.t \parallel term.t \parallel '-'$
$expr \sqsupseteq term$	$expr.t = term.t$
$term \sqsupseteq 0$	$term.t = '0'$
$term \sqsupseteq 1$	$term.t = '1'$
\dots	\dots
$term \sqsupseteq 9$	$term.t = '9'$

例2.6（续）注释语法分析树





例2.7：机器人移动

$seq \rightarrow seq\ instr \mid \mathbf{begin}$

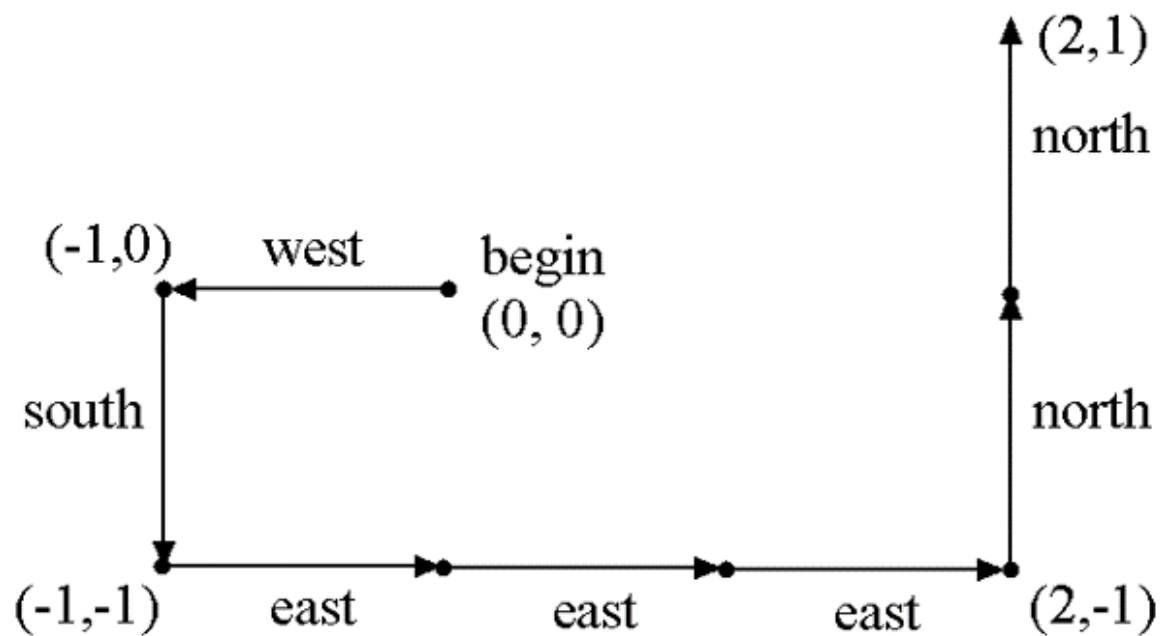
$instr \rightarrow \mathbf{east} \mid \mathbf{north} \mid \mathbf{west} \mid \mathbf{south}$

- 描述指挥机器人行动的指令序列，每个指令指挥机器人向一个方向前进一个距离单位

例2.7：机器人移动

○ 命令序列：

begin west south east east east north north



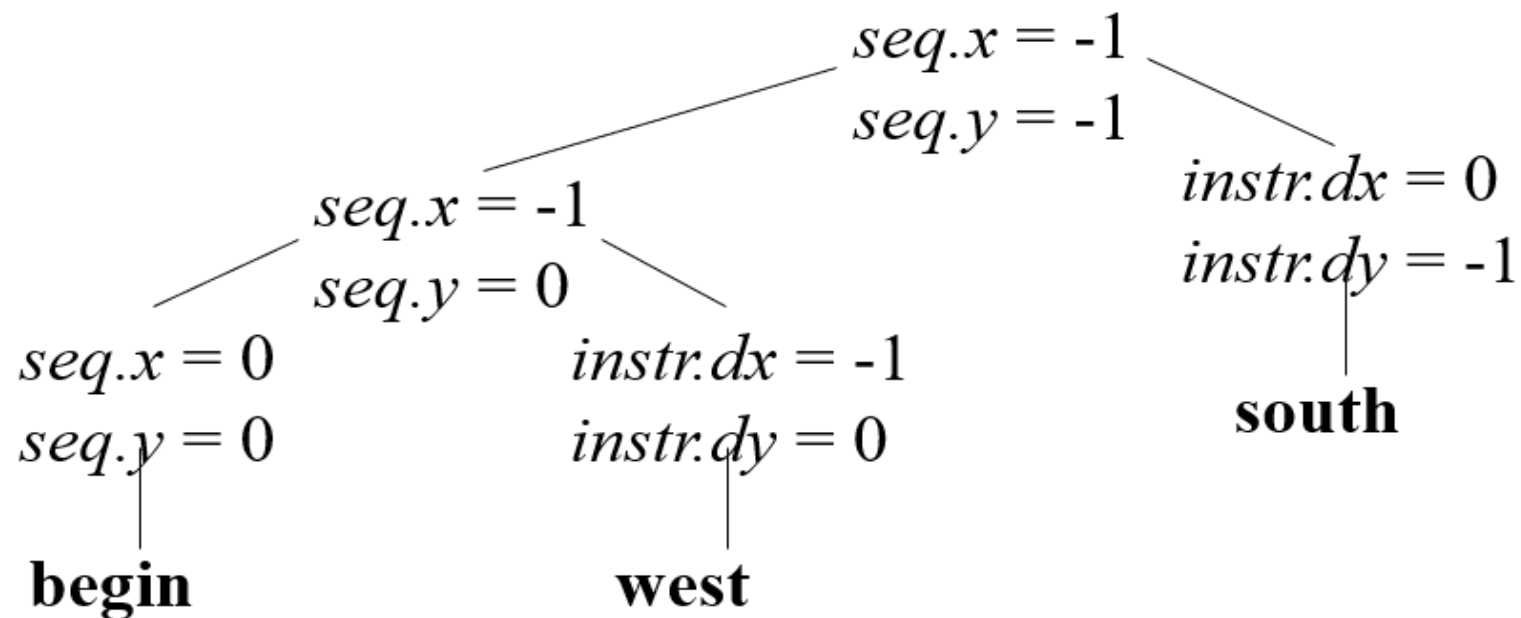


机器人移动的语法制导定义

Production	Semantic Rule
$seq \rightarrow \mathbf{begin}$	$seq.x := 0 \quad seq.y := 0$
$seq \rightarrow seq_1 \mathbf{instr}$	$seq.x := seq_1.x + instr.dx$ $seq.y := seq_1.y + instr.dy$
$instr \rightarrow \mathbf{east}$	$instr.dx := 1 \quad instr.dy := 0$
$instr \rightarrow \mathbf{north}$	$instr.dx := 0 \quad instr.dy := 1$
$instr \rightarrow \mathbf{west}$	$instr.dx := -1 \quad instr.dy := 0$
$instr \rightarrow \mathbf{south}$	$instr.dx := 0 \quad instr.dy := -1$

语法分析树

命令序列: **begin west south**





2.3.4 语法制导定义的实现

- 树的遍历：计算完所有孩子节点的属性，父节点才能计算自身属性
- 后序遍历，深度优先

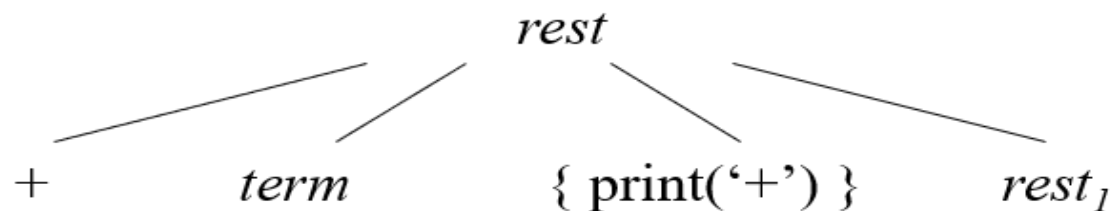
```
void visit(node n)
{
    for n的每个孩子m（从左至右的顺序）
        visit(m);
    利用n的语义规则计算其属性值
}
```

2.3.5 翻译模式

- translation scheme
- 同样基于上下文无关文法
- 语义动作（semantic action，程序片断）嵌入产

生式的右部

$rest \rightarrow + term \{ \text{print('+')} \} rest_1$



- 语法分析树添加额外节点
- 指明了语义动作执行顺序



2.3.6 执行翻译模式

- 语义动作的执行顺序非常重要
- 语法制导定义的特性——简单性

左部的翻译（后缀字符串）= 右部终结符的翻译按产生式中顺序连接

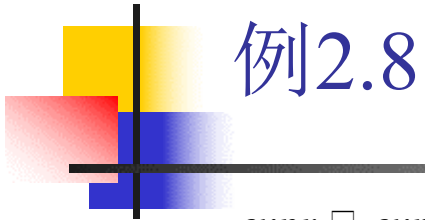
- 用翻译模式实现语法制导定义

- 语义动作——打印额外字符串

- 按照字符串在语法制导定义中的顺序

- $rest \sqsubseteq + term\ rest_1 \quad rest.t := term.t \parallel '+' \parallel rest_1.t \sqsubseteq$

- $rest \sqsubseteq + term \{ \text{print}('+') \} rest_1$



例2.8

$expr \sqsupset expr + term \quad \{print('+')\}$

$\sqsupset expr - term \quad \{print('-')\}$

$\sqsupset term$

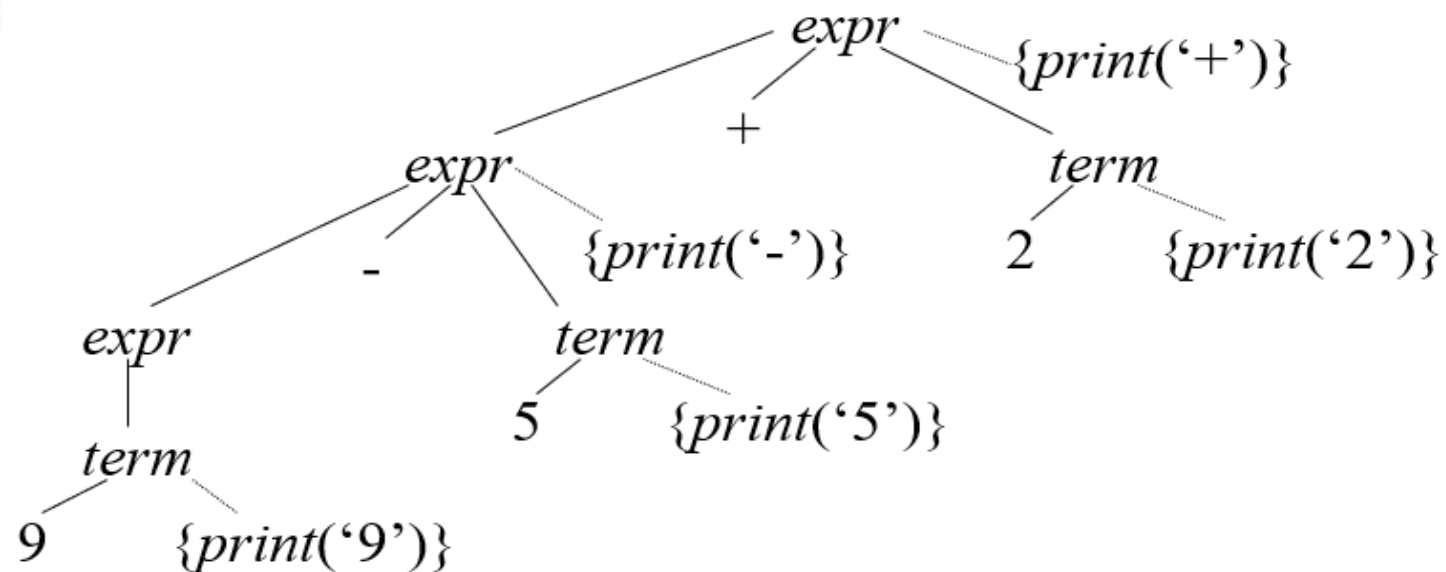
$term \sqsupset 0 \quad \{print('0')\}$

$term \sqsupset 1 \quad \{print('1')\}$

...

$term \sqsupset 9 \quad \{print('9')\}$

带语义动作的语法分析树



- “由左至右” 执行语义动作
- 边进行语法分析，边执行语义动作

语法制导定义练习

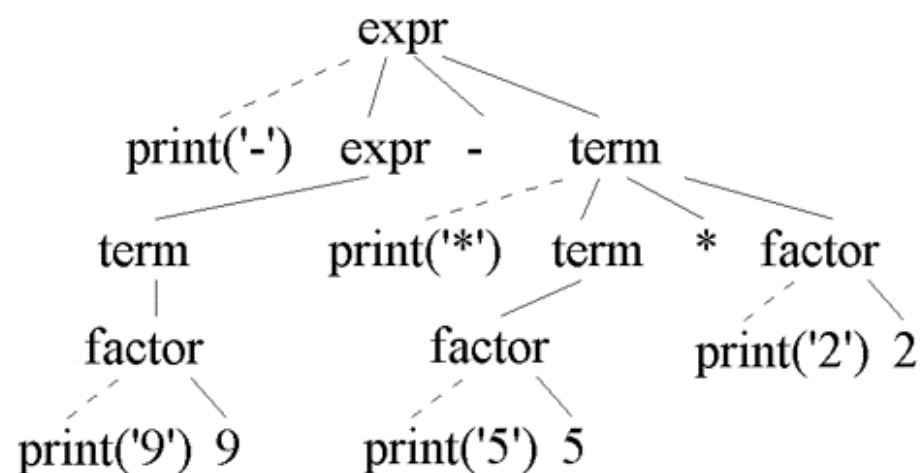
构造翻译模式，中缀→前缀

构造9-5*2的注释语法分析树

$\text{expr} \rightarrow \{ \text{print(' + '); } \} \text{expr} + \text{term}$
 $| \{ \text{print(' - '); } \} \text{expr} - \text{term}$
 $| \text{term}$

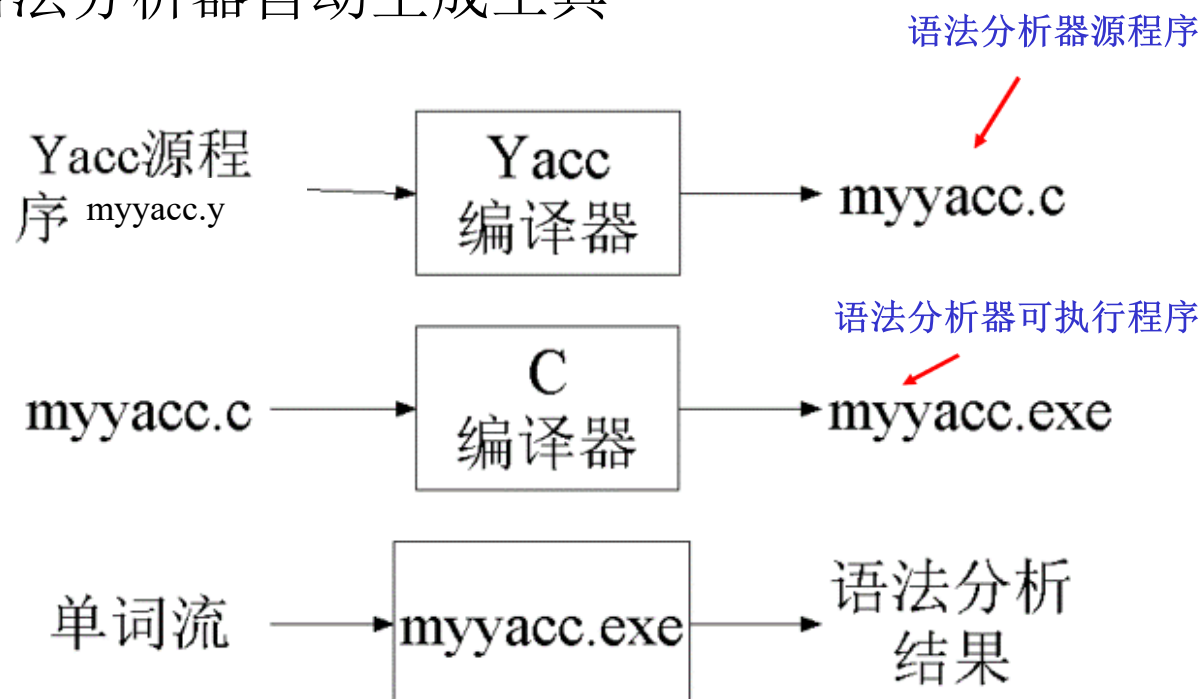
$\text{term} \rightarrow \{ \text{print(' * '); } \} \text{term} * \text{factor}$
 $| \{ \text{print(' / '); } \} \text{term} / \text{factor}$
 $| \text{factor}$

$\text{factor} \rightarrow \{ \text{print(num.value); } \} \text{num}$
 $| (\text{expr})$





○ 语法分析器自动生成工具





Yacc程序结构

定义段:

```
%{
```

C语言代码

```
%}
```

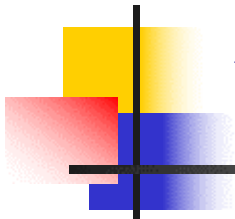
定义

```
%%
```

规则段: 语法规则、翻译模式

```
%%
```

用户子程序段



例：简单表达式计算

```
%{  
/*****  
  
*****  
  
expr.y  
  
ParserWizard generated YACC file.  
  
Date: 2005年8月22日  
  
*****  
  
*****/  
  
#include <ctype.h>  
#include <stdio.h>  
  
%}
```

→ 直接复制到语法分析程序中

例：简单表达式计算（续）

```
%include {  
#ifndef YYSTYPE  
#define YYSTYPE double  
#endif  
}
```

复制到头文件中

```
//%token NUMBER  
%left '+' '-'  
%left '*' '/'  
%right UMINUS  
  
%%
```

我们要翻译的结果是
什么？表达式值！

所以类型是**double**

YYSTYPE是翻译结

果的类型

运算符优先级


前□后，低□高



例：简单表达式计算（续）

```
lines  :    lines expr '\n' { printf("%g\n", $2); }
          |    lines '\n'
          |
          ;

expr   :    expr '+' expr  { $$ = $1 + $3; }
          |    expr '-' expr  { $$ = $1 - $3; }
          |    expr '*' expr  { $$ = $1 * $3; }
          |    expr '/' expr  { $$ = $1 / $3; }
          |    '(' expr ')'   { $$ = $2; }
          |    '-' expr %prec UMINUS  { $$ = -$2; }
          |    NUMBER
          ;
```



表达式的上下文无关文法/翻译模式



例：简单表达式计算（续）

```
NUMBER :      '0'                { $$ = 0.0; }
        |      '1'                { $$ = 1.0; }
        |      '2'                { $$ = 2.0; }
        |      '3'                { $$ = 3.0; }
        |      '4'                { $$ = 4.0; }
        |      '5'                { $$ = 5.0; }
        |      '6'                { $$ = 6.0; }
        |      '7'                { $$ = 7.0; }
        |      '8'                { $$ = 8.0; }
        |      '9'                { $$ = 9.0; }
        ;
```

%%



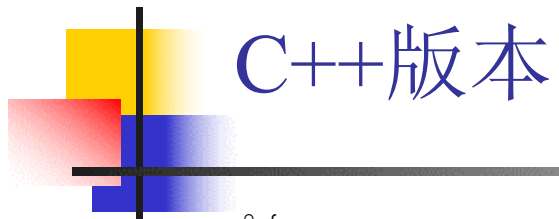
例：简单表达式计算（续）

```
int yygettoken(void)
{
    return getchar();
}

int main(void)
{
    return yyparse();
}
```

还没有完整的词法
分析器，简单地令
每个字符是一个单
词

调用语法分析（翻
译）器



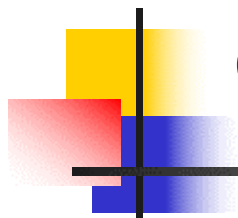
C++版本

```
%{  
/  
*****  
  
*****  
  
expr.y  
  
ParserWizard generated YACC file.  
  
Date: 2016年10月18日  
  
*****  
  
*****/  
  
#include <iostream>  
  
#include <cctype>  
  
using namespace std;  
  
%}
```



C++版本（续）

```
%include {  
    #ifndef YYSTYPE  
    #define YYSTYPE double  
    #endif  
}  
  
// parser name  
%name expr  
  
// class definition  
{  
    // place any extra class members here  
    virtual int yygettoken();  
}
```

C++版本（续）

```
// constructor
{
    // place any extra initialisation code here
}

// destructor
{
    // place any extra cleanup code here
}

// place any declarations here

//%token NUMBER

%left '+' '-'

%left '*' '/'

%right UMINUS
```



C++版本（续）

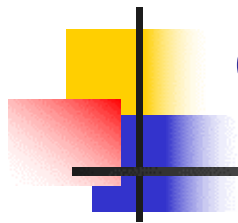
```
lines    :      lines expr '\n' { cout << $2 << endl; }  
          |      lines '\n'  
          |  
          ;
```

```
expr     :      expr '+' expr   { $$ = $1 + $3; }  
          |      expr '-' expr   { $$ = $1 - $3; }  
          |      expr '*' expr   { $$ = $1 * $3; }  
          |      expr '/' expr   { $$ = $1 / $3; }  
          |      '(' expr ')'    { $$ = $2; }  
          |      '-' expr %prec UMINUS { $$ = -$2; }  
          |      NUMBER  
          ;
```



C++版本（续）

```
NUMBER : '0'           { $$ = 0.0; }
        | '1'           { $$ = 1.0; }
        | '2'           { $$ = 2.0; }
        | '3'           { $$ = 3.0; }
        | '4'           { $$ = 4.0; }
        | '5'           { $$ = 5.0; }
        | '6'           { $$ = 6.0; }
        | '7'           { $$ = 7.0; }
        | '8'           { $$ = 8.0; }
        | '9'           { $$ = 9.0; }
        ;
```



C++版本（续）

```
int YYPARSENAME::yygettoken()
{
    // place your token retrieving code here
    return getchar();
}

int main(void)
{
    int n = 1;
    expr parser;
    if (parser.yycreate()) {
        n = parser.yyparse();
    }
    return n;
}
```



2.4 语法分析

- 确定一个单词串是否可由一个文法生成
- 构造语法分析树
- 时间复杂度 $O(n^3)$ \square $O(n)$
- 自顶向下分析方法, **top-down**
 - 语法树构造——由根向叶
 - 适合手工编写语法分析器
- 自底向上分析方法, **bottom-up**
 - 语法树构造——由叶向根
 - 适用更多文法, 自动生成工具



2.4.1 自顶向下分析方法

- 从根节点（标记为开始符号）开始构造语法树，不断重复以下步骤
 1. 对标记为NT A的节点n
 - 选择一个关于A的产生式
 - 利用产生式右部构造n的孩子节点
 2. 选择下一个没有扩展（构造孩子节点）的节点，对它执行1



例：Pascal的类型

type □ *simple*

| □ **id**

| **array** [*simple*] **of** *type*

simple □ **integer**

| **char**

| **num** **dotdot** **num**

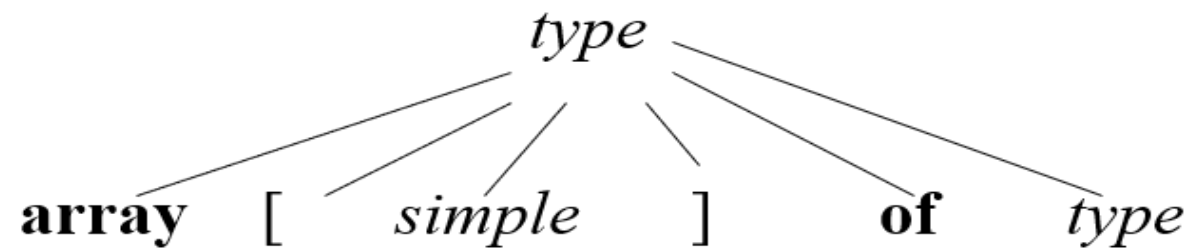
○ dotdot, ” ..”



- 输入: **array [num dotdot num] of integer**

type

type

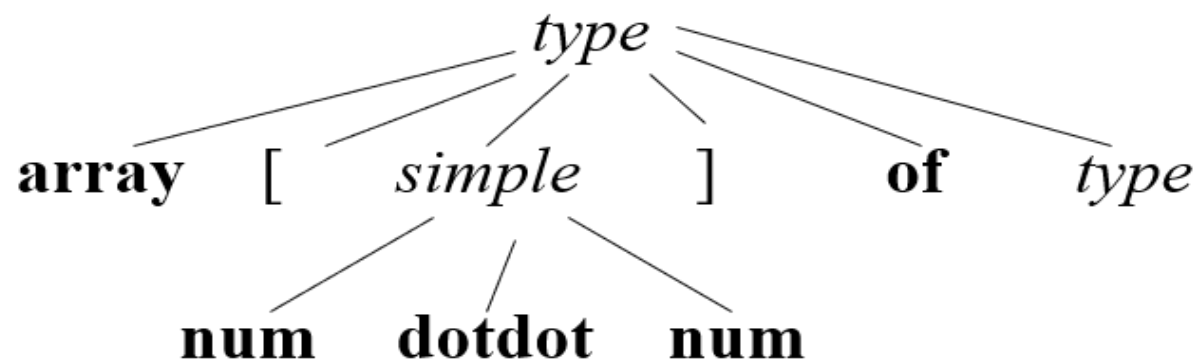




语法分析树构造过程

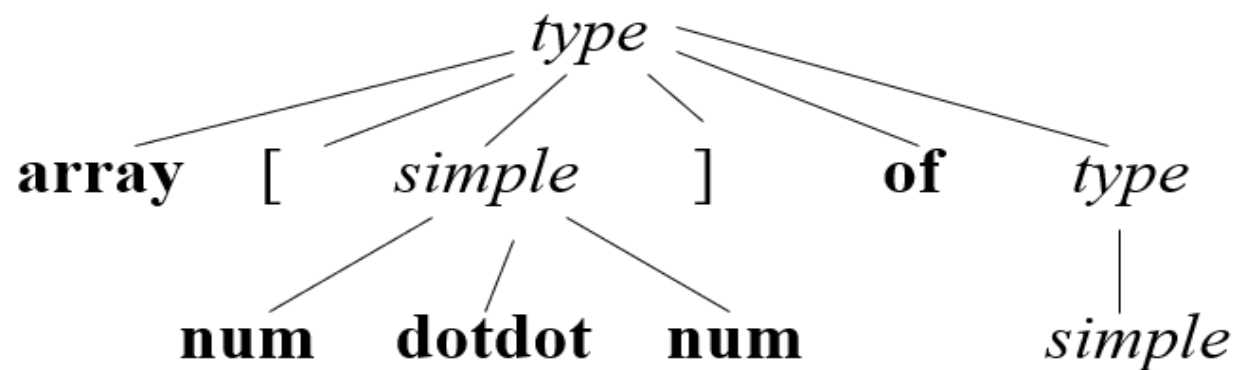
○ 输入: **array [num dotdot num] of integer**

(c)



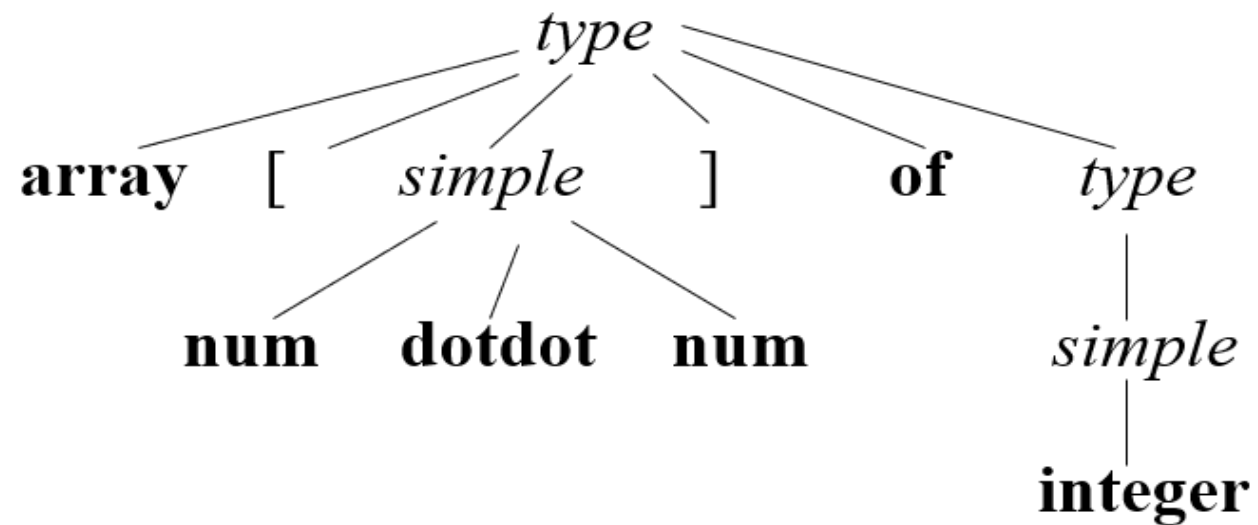
语法分析树构造过程（续）

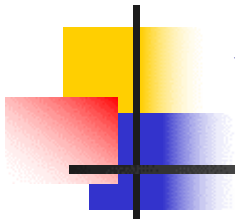
(d)



语法分析树构造过程（续）

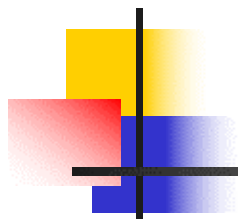
(e)





平凡算法

1. 初始状态，只有一个根节点，标记为开始符号，输入指针指向第一个单词
2. 对于NT节点
 - a. 选择产生式（尝试、回溯）构造孩子节点
 - b. 对孩子节点从左至右继续分析
1. 对于T节点
 - a. 与当前输入单词进行比较
 - b. 若匹配，输入指针前移，处理下一个节点
 - c. 不匹配，可能需要回溯或报告错误



扫描输入分析过程

语法树

type



输入缓冲 **array [num dotdot num] of integer**



语法树

type



simple



输入缓冲 **array [num dotdot num] of integer**



type □ *simple*

| □ **id**

| **array** [*simple*] **of** *type*

simple □ **integer**

| **char**

| **num dotdot num**

扫描输入分析过程（续）

语法树

type

simple

integer 不匹配, 回溯

输入缓冲 **array [num dotdot num] of integer**

语法树

type

simple

char 不匹配, 回溯

输入缓冲 **array [num dotdot num] of integer**

type □ *simple*
| □ **id**

| **array** [*simple*] **of** *type*

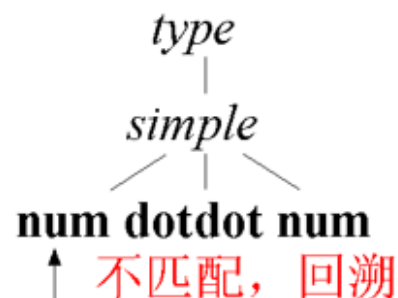
simple □ **integer**

| **char**

| **num dotdot num**

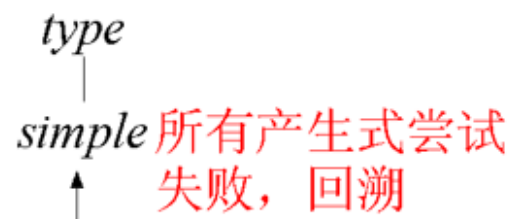
扫描输入分析过程（续）

语法树



输入缓冲 **array [num dotdot num] of integer**

语法树



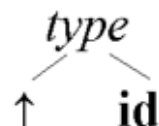
输入缓冲 **array [num dotdot num] of integer**

type □ *simple*
 | □ **id**
 | **array** [*simple*] **of** *type*

simple □ **integer**
 | **char**
 | **num dotdot num**

扫描输入分析过程（续）

语法树

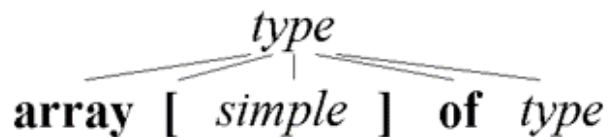


↑ 不匹配，回溯

输入缓冲 **array [num dotdot num] of integer**



语法树



输入缓冲 **array [num dotdot num] of integer**



type □ *simple*

| □ **id**

| **array** [*simple*] **of** *type*

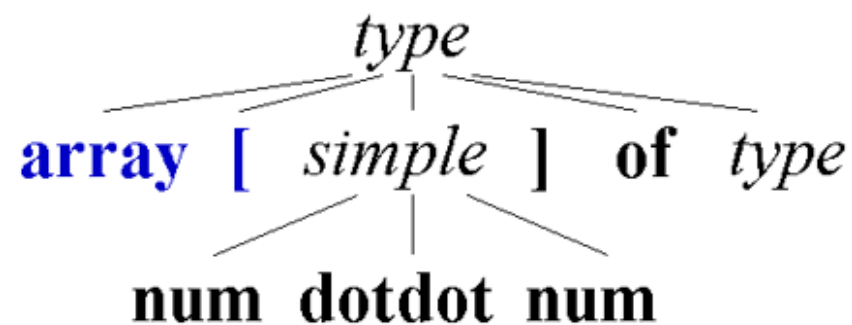
simple □ **integer**

| **char**

| **num dotdot num**

扫描输入分析过程（续）

语法树



输入缓冲 **array** [**num** dotdot **num**] of integer





2.4.2 预测分析

- 递归下降分析

recursive-descent parsing

递归函数 □ □ 非终结符

- 预测分析

predictive parsing

只需一个超前单词——^{唯一确定}产生式



预测分析程序

对终结符的处理——匹配输入，指针移动

```
procedure match ( t : token ) ;  
  
begin  
    if lookahead = t then  
        lookahead := nexttoken  
    else error  
end ;
```

预测分析程序（续）

```
procedure type ;
begin
    if lookahead is in { integer, char, num } then
        simple
    else if lookahead = '□' then begin
        match ( '□' ); match(id)
    end
    else if lookahead = array then begin
        match( array ); match( '[' );
        simple;
        match( ']' ); match(of);
        type
    end
    else error
end ;
```

超前单词 首单词集合

选择产生式

分析孩子节点

处理T

处理NT，递归

$type \sqsupset simple$
| $\sqsupset id$
| array [simple] of type
 $simple \sqsupset integer$
| char
| num dotdot num



预测分析程序（续）

```
procedure simple ;  
begin  
    if lookahead = integer then  
        match ( integer );  
    else if lookahead = char then  
        match ( char );  
    else if lookahead = num then begin  
        match (num); match (dotdot); match (num)  
    end  
    else error  
end ;
```



2.4.4 设计预测分析器

- 一个超前单词——^{唯一确定}产生式 $\square\square$

推导出的单词串的第一个单词不同

- $A\square\square$

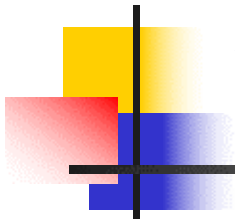
- \square $\text{FIRST}(\square)$: \square 推导出的单词串的第一个单词的集合,
可包含 ϵ

- \square $\text{FIRST}(\text{simple}) = \{ \text{integer}, \text{char}, \text{num} \}$

- $\text{FIRST}(\square \text{id}) = \{ \square \}$

- $\text{FIRST}(\text{array } [\text{simple}] \text{ of type}) = \{ \text{array} \}$

- $A\square\square, A\square\beta, \text{FIRST}(\square)$ 和 $\text{FIRST}(\beta)$ 不能相交

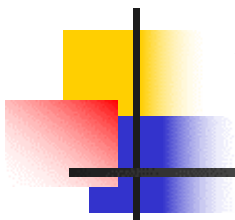


预测分析器算法

- 为每个NT A构造一个递归函数，完成如下工作：

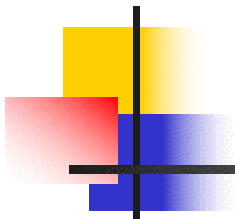
1. 产生式选择

- a. 对 $A \rightarrow \alpha$ ，超前符号在 $\text{FIRST}(\alpha)$ 中 α 选择它
- b. 若有冲突 \rightarrow 预测分析法不适用
- c. $A \rightarrow \epsilon$ ，且超前单词不在其他任何产生式右部的 FIRST 集中 \rightarrow 选择它



预测分析器算法

2. 产生式的使用——依次处理右部符号
 - a. 对NT，调用对应递归函数
 - b. 对T，与超前单词比较，若匹配读取下一单词，否则错误



结合翻译模式

- 简单方法

1. 首先构造预测分析器

2. 实现语义动作

- a. 将语义动作程序段复制到分析器代码中

- b. 位置?

- I. 语义动作位于语法符号X之后

程序段中将之放在紧接在处理X的代码之后

- II. 位于产生式开始，复制到函数最前面

2.4.5 左递归

○ left recursion, 导致无限循环

○ 解决方法: 改写文法

□ $A \rightarrow A \mid \epsilon$ 改写为

□ $A \rightarrow R$

$R \rightarrow R \mid \epsilon$

○ 例

□ $expr \rightarrow expr + term \mid term$

□ $expr \rightarrow term\ rest$

$rest \rightarrow + term\ rest \mid \epsilon$

void idlist()

{

...

if (lookahead == ...) {

idlist();

...

}

}

一直不变

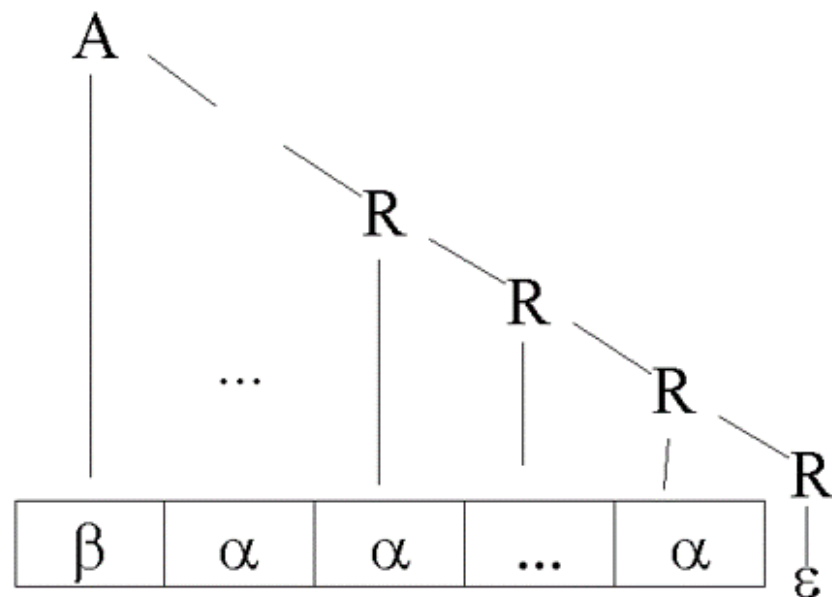
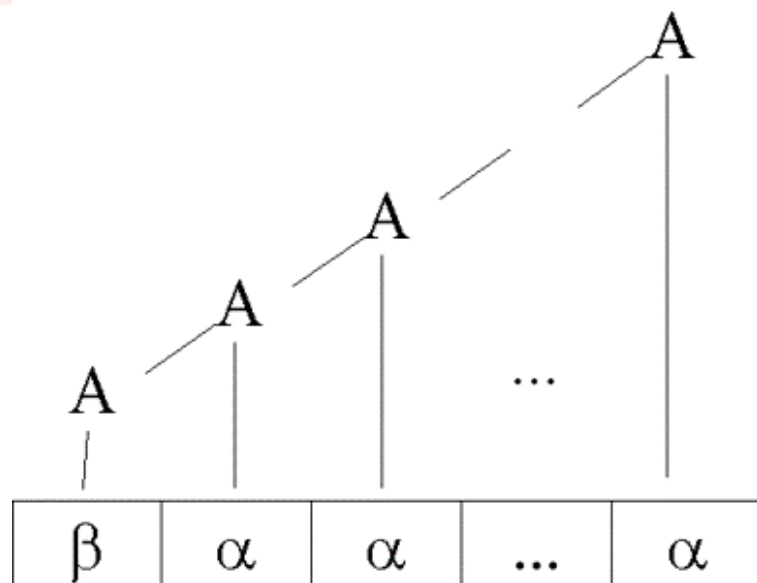
id, id, ...

□

α

β

左递归 vs. 右递归语法树差异





2.5 构造表达式转换的编译器

○ 原始语法：需改写

$expr \sqsupseteq expr + term \quad \{print('+')\}$

$expr \sqsupseteq expr - term \quad \{print('-')\}$

$expr \sqsupseteq term$

$term \sqsupseteq 0 \quad \{print('0')\}$

$term \sqsupseteq 1 \quad \{print('1')\}$

...

$term \sqsupseteq 9 \quad \{print('9')\}$



消除左递归、加入翻译模式

$expr \sqsupseteq term\ rest$

$rest \sqsupseteq +\ term\ \{ print('+') \}\ rest \mid$

$- \ term\ \{ print('-') \}\ rest \mid \varepsilon$

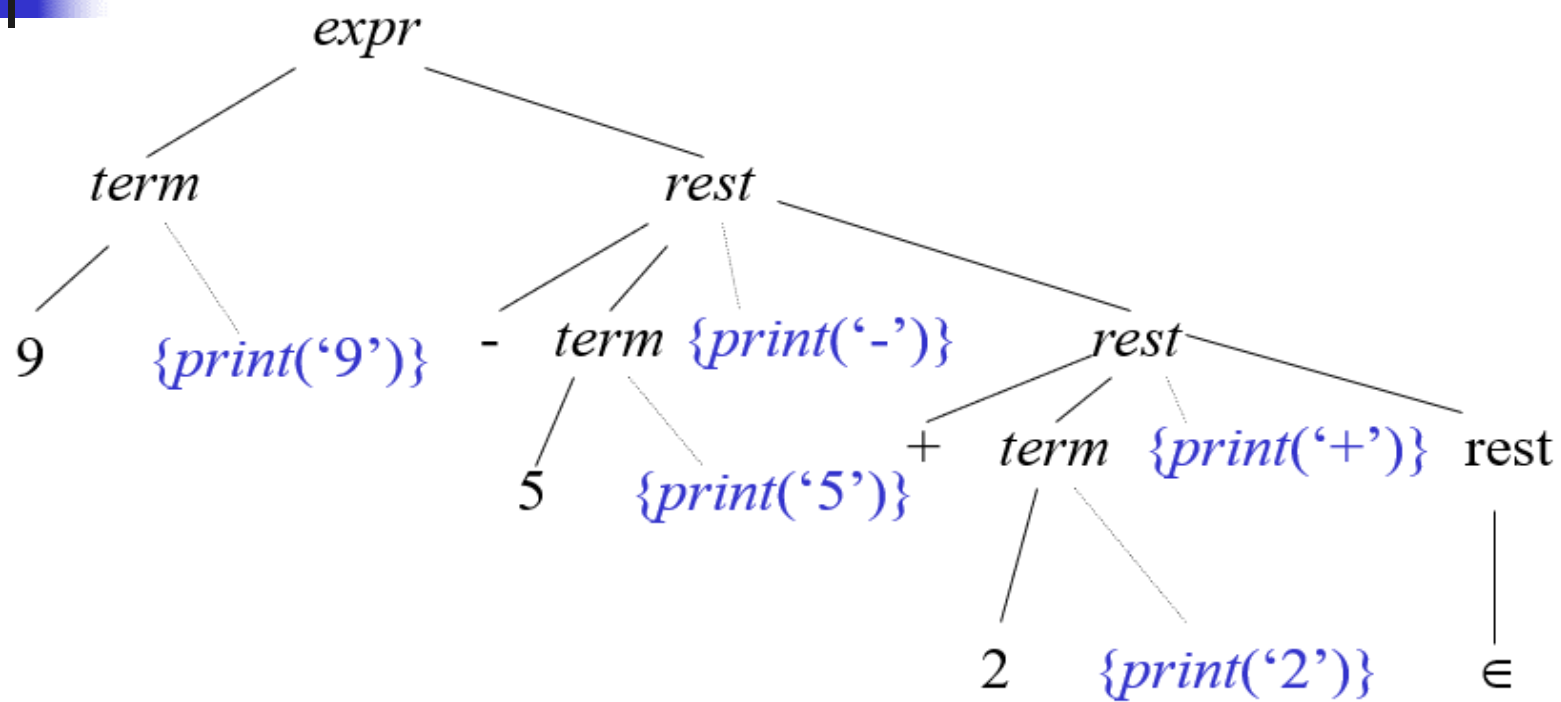
$term \sqsupseteq 0\ \{ print('0') \}$

$term \sqsupseteq 1\ \{ print('1') \}$

...

$term \sqsupseteq 9\ \{ print('9') \}$

9-5+2翻译为95-2+





2.5.3 非终结符实现代码

非终结符的分析函数——`expr`

```
void expr ()
```

```
{
```

```
    term(); rest();
```

```
}
```

expr \square *term rest*



非终结符的分析函数——rest

```
void rest ()
{
    if (lookahead == '+' ) {
        match( '+' ); term(); putchar( '+' ); rest();
    }
    else if (lookahead == '-' ) {
        match( '-' ); term(); putchar( '-' ); rest();
    }
}
```

$rest \sqsupseteq + term \{ print('+') \} rest \mid$
 $- term \{ print('-') \} rest \mid \epsilon$



非终结符的分析函数——term

```
void term ()
{
    if (isdigit(lookahead)) {
        putchar(lookahead); match(lookahead);
    }
    else error();
}
```

term □ 0 {print('0')}

term □ 1 {print('1')}

...

term □ 9 {print('9')}



2.5.4 优化——消除rest尾递归

```
void rest ()
{
    L:  if (lookahead == '+' ) {
            match( '+' ); term(); putchar( '+' ); goto L;
        }
    else if (lookahead == '-' ) {
            match( '-' ); term(); putchar( '-' ); goto L;
        }
}
```



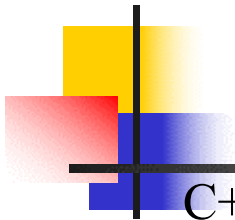
优化——rest并入expr

```
void expr ()
{
    term();
    while (1)
        if (lookahead == '+' ) {
            match( '+' ); term(); putchar( '+' );
        }
        else if (lookahead == '-' ) {
            match( '-' ); term(); putchar( '-' );
        }
        else break;
}
```



2.5.5 完整程序

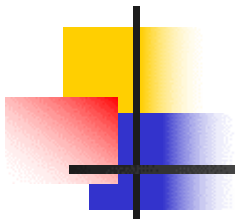
```
#include <ctype.h>
int lookahead;
main()
{
    lookahead = getchar();
    expr();
    putchar( '\n' );
}
void match(int t)
{
    if (lookahead == t)
        lookahead = getchar();
    else error();
}
```



C++编译器检查相容类型计算是否合规是在
_____阶段

- ☐ A 词法分析
- ☐ B 语法分析
- ☒ C 语义分析
- ☐ D 代码生成

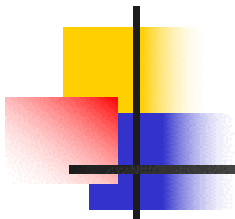
提交



编译器识别出标识符是在_____阶段

- ☒ A 词法分析
- ☐ B 语法分析
- ☐ C 语义分析
- ☐ D 代码生成

提交

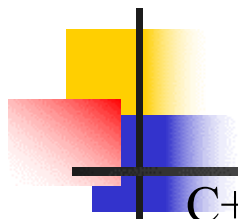


C++编译器过滤注释是在_____阶段。

- ☒ A 词法分析
- ☐ B 语法分析
- ☐ C 代码生成
- ☐ D 以上皆错

提交

单选题 1分



C++编译器检查数组下标越界是在_____阶段

。

- ☐ A 词法分析
- ☐ B 语法分析
- ☐ C 代码生成
- ☒ D 以上皆错

提交



Intel的深度学习编译器nGraph支持TensorFlow、MXNet等深度学习框架，令用户可在Intel CPU、GPU等不同硬件平台上高效运行这些框架编写的程序，它的实现是一种_____方式。

- ☐ A 单前端单后端
- ☐ B 单前端多后端
- ☐ C 多前端单后端
- ☒ D 多前端多后端

提交



LaTeX工具将.tex文件转换为PDF文件，因此它也是一种广义的编译器。这种说法是

_____。

A 正确的

B 错误的

提交

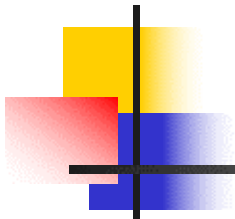
单选题 1分



浏览器渲染过程相当于编译器的_____阶段。

- ☐ A 分析
- ☒ B 综合
- ☐ C 前端
- ☐ D 后端

提交



符号表是在_____阶段创建的。

- ☒ A 词法分析
- ☐ B 语法分析
- ☐ C 语义分析
- ☐ D 代码生成

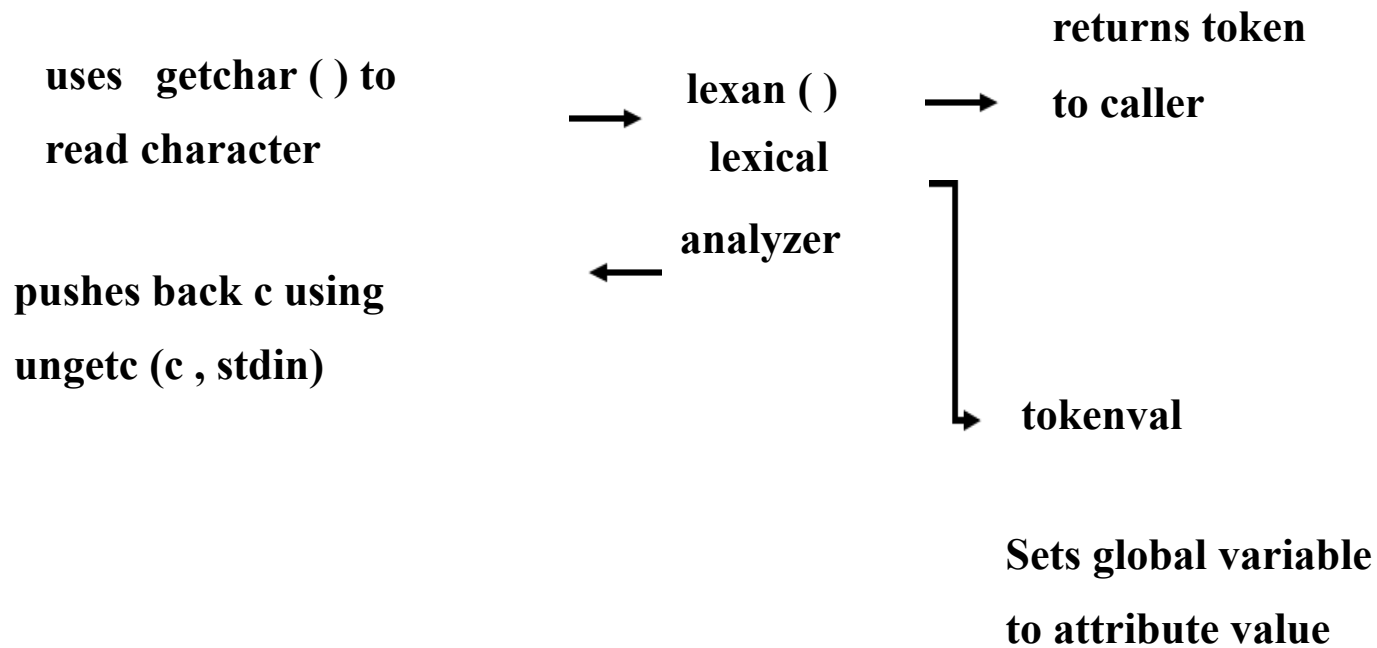
提交



2.6 词法分析

- 2.6.1 去除空白符和注释
- 2.6.2 常量: $\langle \text{num}, 31 \rangle$, $\langle \text{num}, 28 \rangle$, ...
- 2.6.3 标识符与关键字识别
 - $\text{count} = \text{count} + \text{increment}$ □ $\text{id} = \text{id} + \text{id}$
 - 单词id不同实例count, increment的区分
 - $\langle \text{id}, \text{符号表项指针} \rangle$
 - 关键字 (**keyword**) , 保留字 (**reserved**)
 - 运算符, $<$, \leq , \triangleleft , 可以每个运算符作为一类单词

2.6.4 词法分析器接口





2.6.5 词法分析器程序

```
#include <stdio.h>
#include <ctype.h>

#define NUM 256

int lineno = 1;
int tokenval = NONE;

int lexan()
{
    int t;
    while (1) {
        t = getchar();
        if (t == ' ' || t == '\t' )
            ;
    }
}
```



词法分析器程序（续）

```
    else if (t == '\n' )
        lineno++;
    else if (isdigit(t)) {
        tokenval = 0;
        while (isdigit(t)) {
            tokenval = tokenval * 10 + t - '0' ;
            t = getchar();
        }
        ungetc(t, stdin);
        return NUM;
    }
    else { tokenval = NONE; return t; }
}
```




2.7 符号表

- 不同阶段保存（使用）单词的不同信息

- 2.7.1 接口

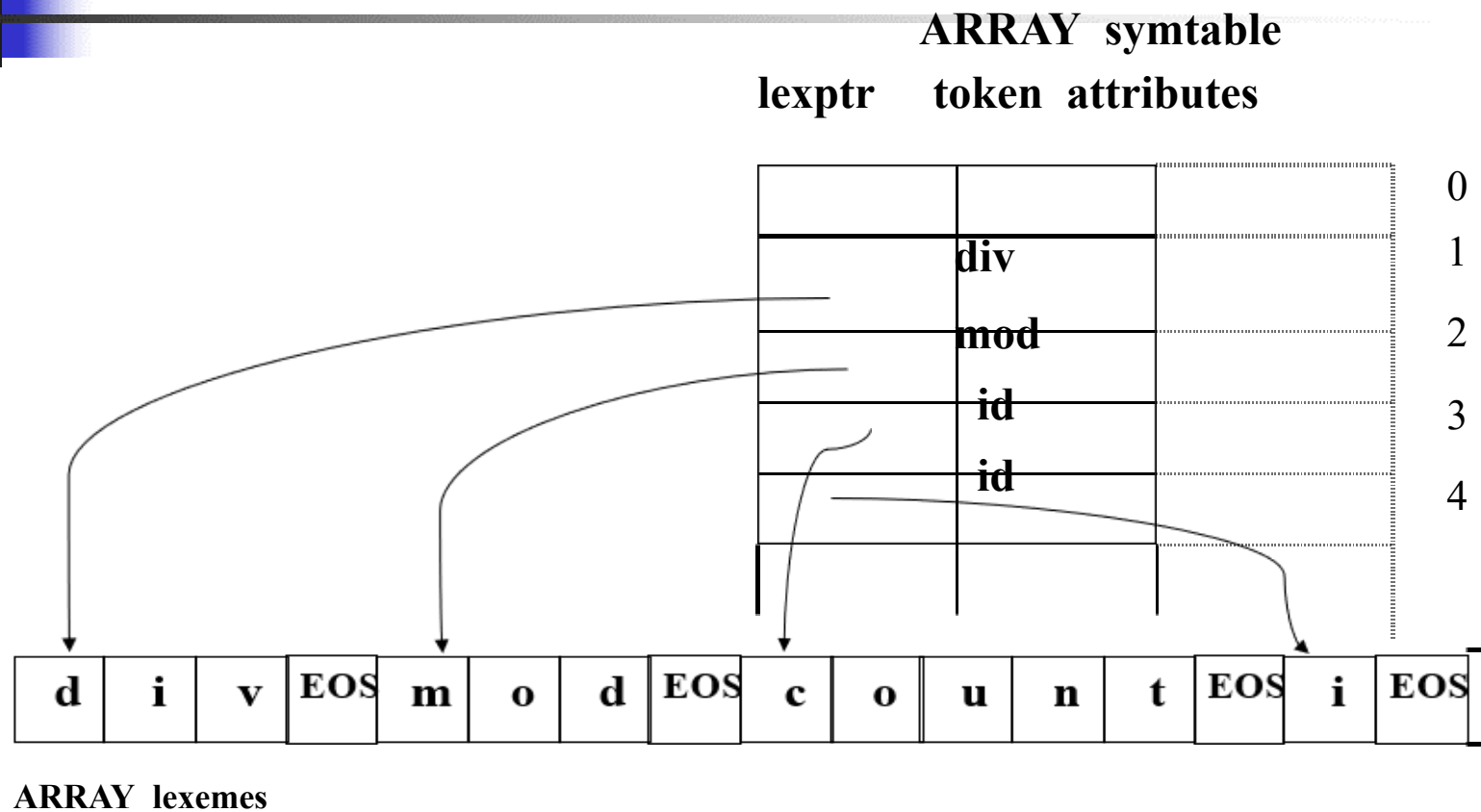
- `insert(s,t)`: 单词`t`的实例`s`（字符串）插入符号表，
返回其索引

- `lookup(s)`: 返回字符串`s`对应表项的索引，若未找到，
返回0

- 2.7.2 保留字可存入符号表

`insert(“div”, div)` `insert(“mod”, mod)`

2.7.3 一种实现方式

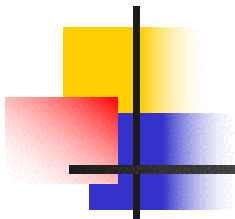




带有符号表功能的词法分析器

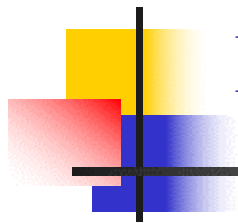
```
int lexan()
{
    char lexbuf[100];
    char c;

    while (1) {
        从输入字符序列读取一个字符，保存到c;
        if (c == ' ' || c == '\t' );
        else if (c == '\n' ) lineno++;
        else if (isdigit(c)) {
            tokenval = c及后续数字组成的数值;
            return NUM;
        }
    }
}
```



带有符号表功能的词法分析器

```
else if (isalpha(c)) {  
    将c及后续字母、数字放入lexbuf;  
    p = lookup(lexbuf);  
    if (p != 0)  
        p = insert(lexbuf, ID);  
    tokenval = p;  
    return 符号表项p的token值;  
} else {  
    tokenval = NONE;  
    return 字符c对应的整数;  
}  
}  
}
```

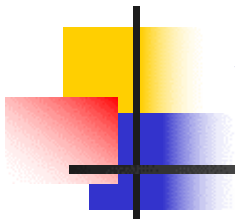


Lcc的符号表

○ 字符串的管理

```
//string.c
static struct string {
    char *str;
    int len;
    struct string *link;
} *buckets[1024];

char *stringn(const char *str, int len) {
    int i;
    unsigned int h;
    const char *end;
    struct string *p;
    assert(str);
    for (h = 0, i = len, end = str; i > 0; i--)    // Hash
        h = (h<<1) + scatter[*(unsigned char *)end++];
    h &= NELEMS(buckets)-1;
```



字符串管理

```
for (p = buckets[h]; p; p = p->link)           //搜索
    if (len == p->len) {
        const char *s1 = str;
        char *s2 = p->str;
        do {
            if (s1 == end)
                return p->str;
        } while (*s1++ == *s2++);
    }
{
    static char *next, *strlimit;                //创建
    if (len + 1 >= strlimit - next) {            //现有空间不够
        int n = len + 4*1024;
        next = allocate(n, PERM);               //分配空间
        strlimit = next + n;
    }
}
```



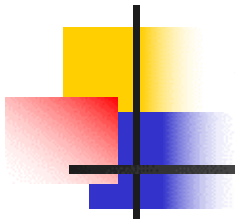
字符串管理

```
p->len = len;
for (p->str = next; str < end; )
    *next++ = *str++;           //复制字符串
    *next++ = 0;
p->link = buckets[h];          //放入Hash表
buckets[h] = p;
return p->str;
    }
}
```



符号表

```
//sym.c
struct table {
    int level;
    Table previous;
    struct entry {
        struct symbol sym;
        struct entry *link;
    } *buckets[256];
    Symbol all;
};
```

符号表

```
Symbol lookup(const char *name, Table tp) {  
    struct entry *p;  
    unsigned h = (unsigned long)name&(HASHSIZE-1);  
  
    assert(tp);  
    do  
        for (p = tp->buckets[h]; p; p = p->link)  
            if (name == p->sym.name)  
                return &p->sym;  
    while ((tp = tp->previous) != NULL);  
    return NULL;  
}
```

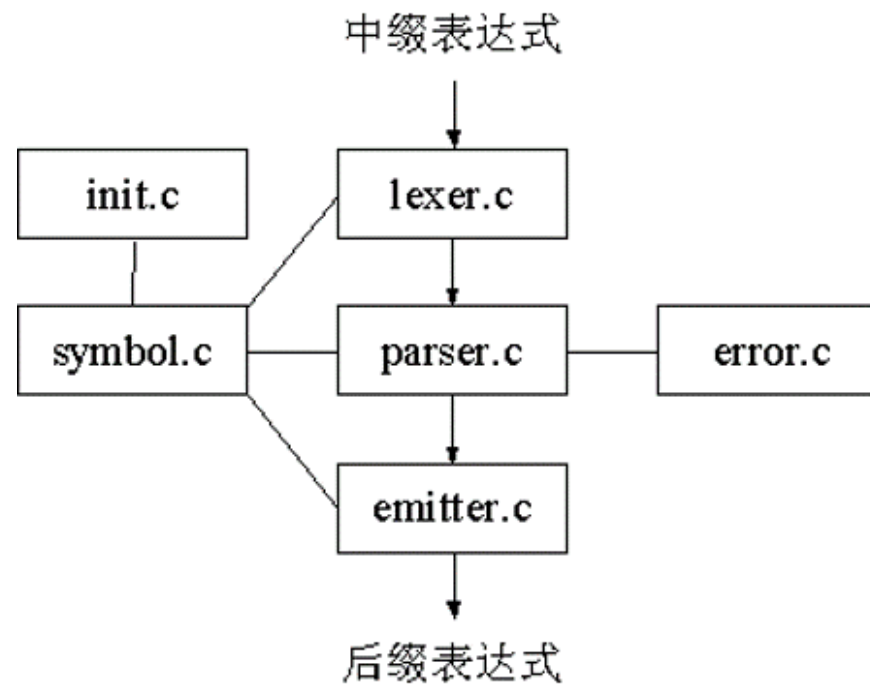


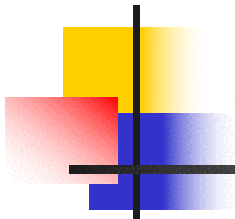
符号表

```
Symbol install(const char *name, Table *tpp, int level, int arena) {  
    Table tp = *tpp;  
    struct entry *p;  
    unsigned h = (unsigned long)name%(HASHSIZE-1);  
  
    assert(level == 0 || level >= tp->level);  
    if (level > 0 && tp->level < level)  
        tp = *tpp = table(tp, level);    //新作用域  
    NEW0(p, arena);  
    p->sym.name = (char *)name;  
    p->sym.scope = level;  
    p->sym.up = tp->all;  
    tp->all = &p->sym;  
    p->link = tp->buckets[h];    //放入Hash表  
    tp->buckets[h] = p;  
    return &p->sym;  
}
```

2.9 完整的编译器

- 结合前几节介绍的技术





完整的文法

start \square *list eof*

list \square *expr ; list* $|\epsilon$

expr \square *term moreterms*

moreterms \square *+* *term* { *print*('+') } *moreterms*

$|\text{- term \{ print('-') \} moreterms} |\epsilon$

term \square *factor morefactors*

morefactors \square *** *factor* { *print*('*') } *morefactors*

$|\text{/ factor \{ print('/') \} morefactors}$

$|\text{div factor \{ print('DIV') \} morefactors}$

$|\text{mod factor \{ print('MOD') \} morefactors} |\epsilon$

factor \square (*expr*)

$|\text{id \{ print(id.lexeme) \}}$

$|\text{num \{ print(num.value) \}}$



总结

- 本章概览了整个编译过程
- 介绍了上下文无关文法（CFG），展示了它与编译理论的关系
- 介绍了如何利用语法分析树，辅助分析和翻译
- 后续章节依次介绍编译各个阶段涉及的原理、技术
- 首先是词法分析：正规式、有限自动机

