第四章 语法分析
Syntax Analysis

Part I

主要内容

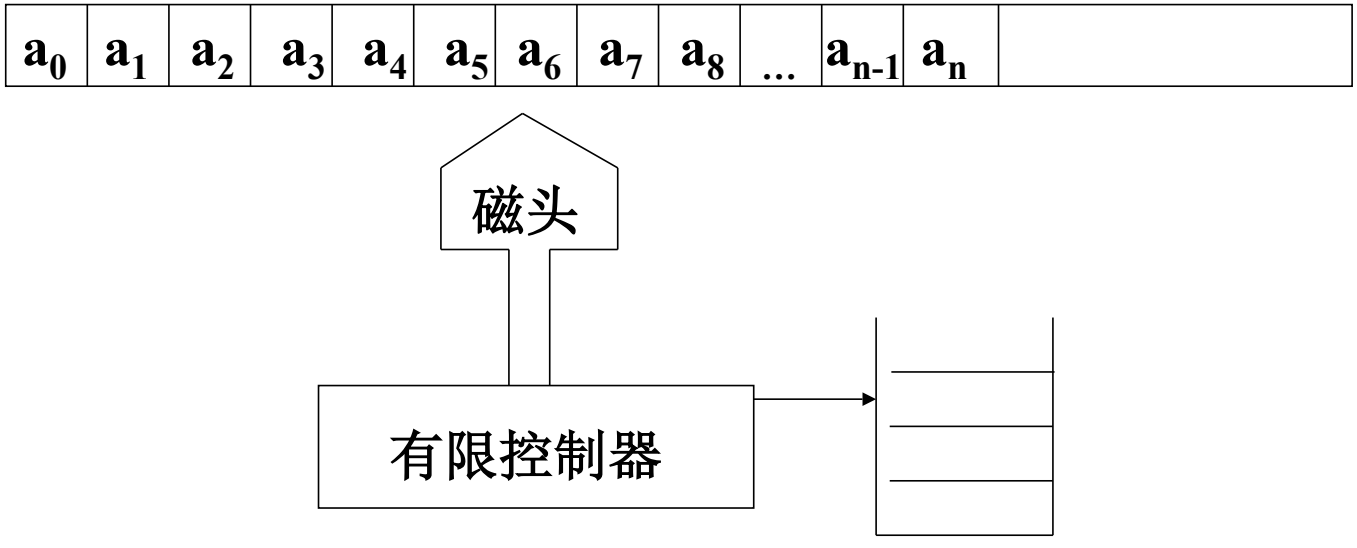
- PDA
- 语法分析概述
- 递归子程序法
- 常用终结符号集计算
- LL(1)分析方法
- LR分析方法

1、下推自动机（PDA）

- Push Down Automata
- M=(S, ∑, Γ, δ, K, x0, F)
- S：状态集合
- ∑：输入字母表
- Γ：下推字母表
- δ：$S \times (\Sigma \cup \{\}) \times \Gamma \rightarrow S \times \Gamma^*$
- K：初态集合
- x0：下推栈中的初始符号
- F：终态集合

- δ（Si, a, x_k) = (Sj, β)
- ε-转换：输入符号全部读完，但PDA的状态仍然可以进行转换

识别程序的数学模型下推自动机



例：

- $S=\{S_0\}$
- $\Sigma=\{(,)\}$
- $\Gamma=\{A,(\}$
- δ： $S \times (\Sigma \cup \{\}) \times \Gamma \rightarrow S \times \Gamma^*$
- $K=\{S_0\}$
- x0=A
- $F=\{S_0\}$

$\delta (S_0,(,A)=\{(S_0,(A)\}$

$\delta (S_0,(,( )=\{(S_0,(()\}$

$\delta (S_0,),( )=\{(S_0, \varepsilon)\}$

$\delta (S_0,\varepsilon,A)=\{(S_0, \varepsilon)\}$

对于输入串：(( ) ( ))

如何进行识别？

例：PDA P=({A,B,C),{a,b,c), {h,i}, f, A, i,{ })

f(A,a,i) = (B,h)　　f(B,a,h) = (B,hh)

f(C,b,h) = (C, ε)　f(A,c,i) = (A, ε)

f(B,c,h) = (C,h)

接受输入串aacbb的过程

(A,aacbb,i) 读a, pop i, push h, goto B
(B,acbb,h) 读a, pop h, push hh, goto B
(B,cbb,hh) 读c, pop h, push h , goto C
(C,bb,hh)　读b, pop h, push **ε** , goto C
(C, b, h)　　读b, pop h, push **ε** , goto C
(C, ε, ε)

PDA与语法分析

- PDA: NPDA, DPDA
- 2型文法-程序设计语言-PDA

终止和接受的条件：

1.到达输入串结尾时，处在F中的一个状态(终态)

或 2.某个动作序列导致栈空时

2、语法分析概述

- 语法分析任务
- 语法分析分类

2、语法分析概述

- 语法分析任务：根据语法规则逐一分析词法分析得到的属性字（单词序列），检查语法错误，若无错，则给出正确的语法结构；若有错，则报错。

2、语法分析概述

- 语法分析分类
  - 自顶向下top-down parsing
  - 自底向上bottom-up parsing

句型的分析算法分类

分析算法可分为：

自上而下分析法：

从文法的开始符号出发，反复使用文法的产生式，寻找与输入符号串匹配的推导。

自下而上分析法：

从输入符号串开始，逐步进行归约，直至归约到文法的开始符号。

自顶向下

- **G[Z]:**
  - **Z->aBd**
  - **B->d | c**
  - **B->bB**

  给定符号串abcd，如何Z*=>abcd?

PDA模拟

设下推栈#S，状态控制器中状态只有一个，整个分析过程是在语法分析程序控制下进行的：

1、若栈顶X为Vn,则查询语法分析表，找出一个以X为左部的产生式（语法规则），将X弹出栈，而把产生式右部的符号串以从右向左的次序进栈（推导）

2、若栈顶X为Vt,且读头所指向的输入符号也是X，则匹配。此时，X出栈，读头右移

3、ERROR：若栈顶X为Vt,且读头所指向的输入符号不是X，则说明前面推导时选错了规则，应退到上次规则之前（回溯）

4、重新选规则进行推导

5、若栈内只有栈底符号#,而读头也指到了输入的最后符号#，则分析成功

- 自顶向下的关键问题是：
  每次该选择哪条规则

自底向上

- **G[S]**
  - **S->aAcBe**
  - **A->Ab | b**
  - **B->d**
- **abbcde**

自底向上的栈式分析过程

- 从输入串依次读入输入符号，直到一个简单短语出现在分析栈的栈顶，然后将栈顶的简单短语归约成相应的Vn,重复上述过程，直到栈中只剩下开始符号，而输入串全部被处理完。

- 自底向上主要问题是**何时应该归约**，以及**选择哪条规则进行归约**。

句型分析的有关问题

1）在自上而下的分析方法中如何选择使用哪个产生式进行推导？

假定要被代换的最左非终结符号是B，且有n条规则：B→A1|A2|…|An，那么如何确定用哪个右部去替代B？

2）在自下而上的分析方法中如何识别可归约的串？

在分析程序工作的每一步，都是从当前串中选择一个子串，将它归约到某个非终结符号，该子串称为“可归约串”

刻画“可归约串”

文法G[S]

句型的短语

S =>* α A δ 且 A =>+ β，则称 β 是句型 α β δ 相对于非终结符A的短语

句型的直接短语

若有A ⇒ β，则称 β 是句型 α β δ 相对于非终结符A 的直接短语

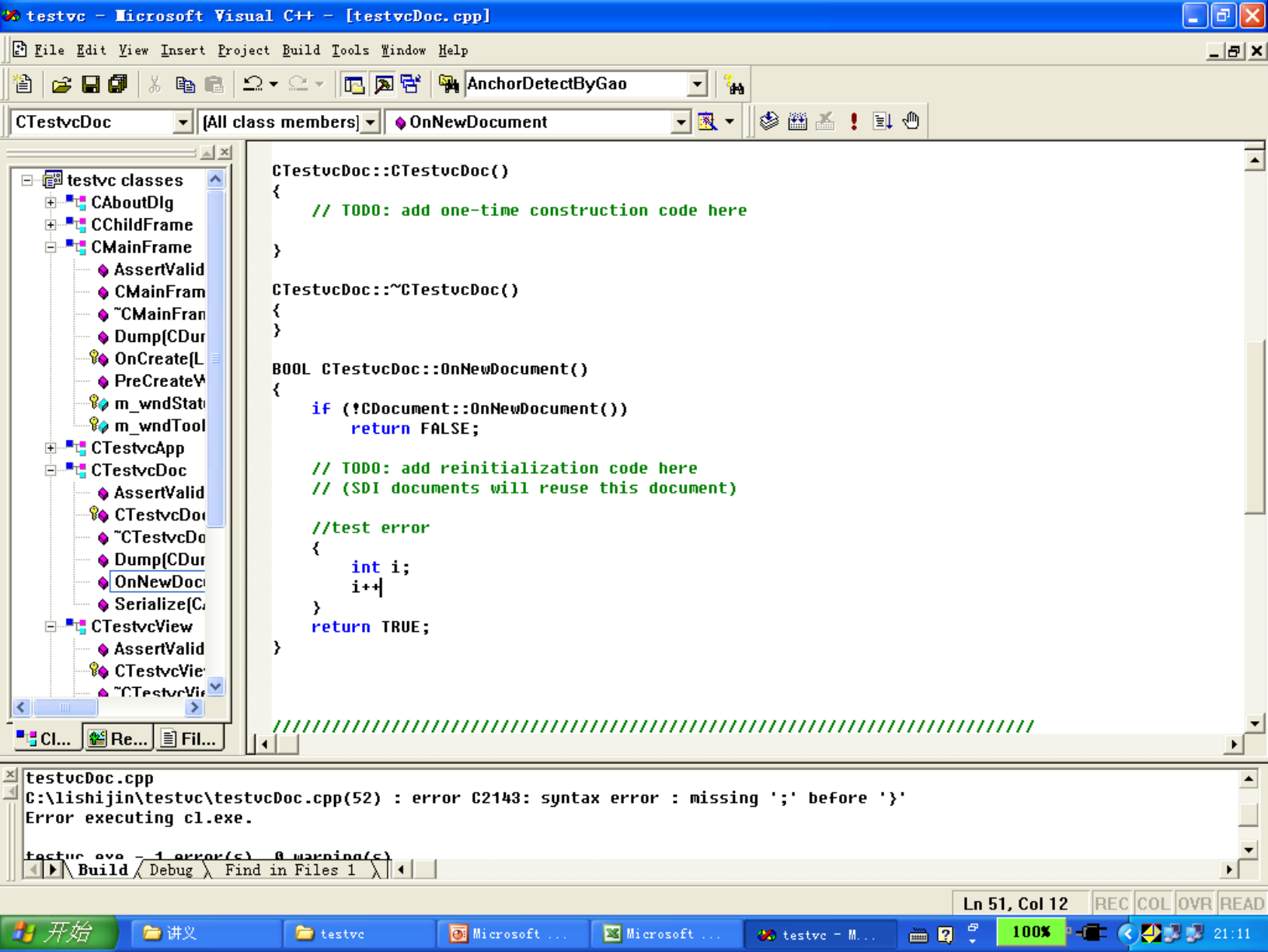
句型的句柄

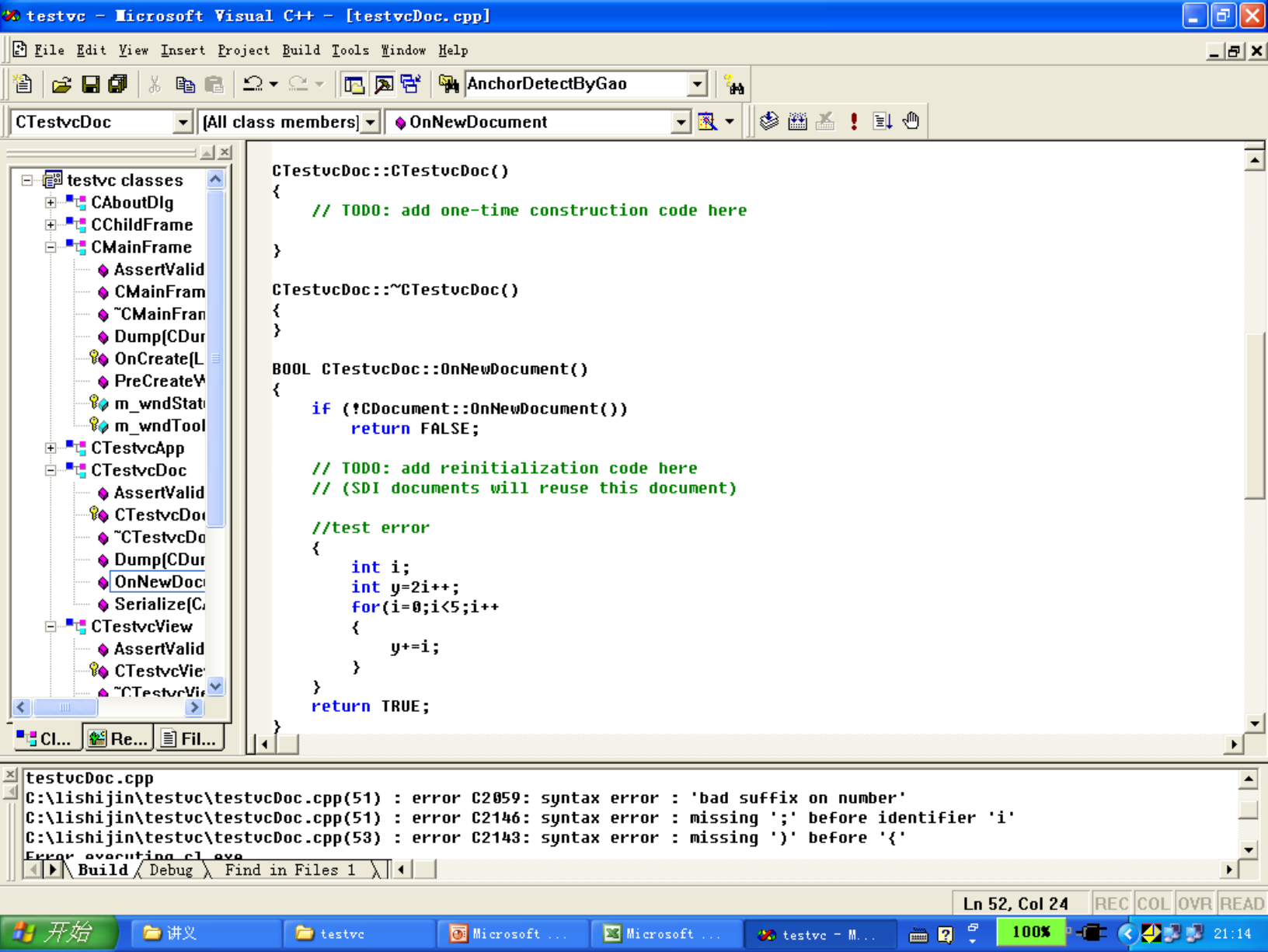
一个句型的最左直接短语称为该句型的句柄

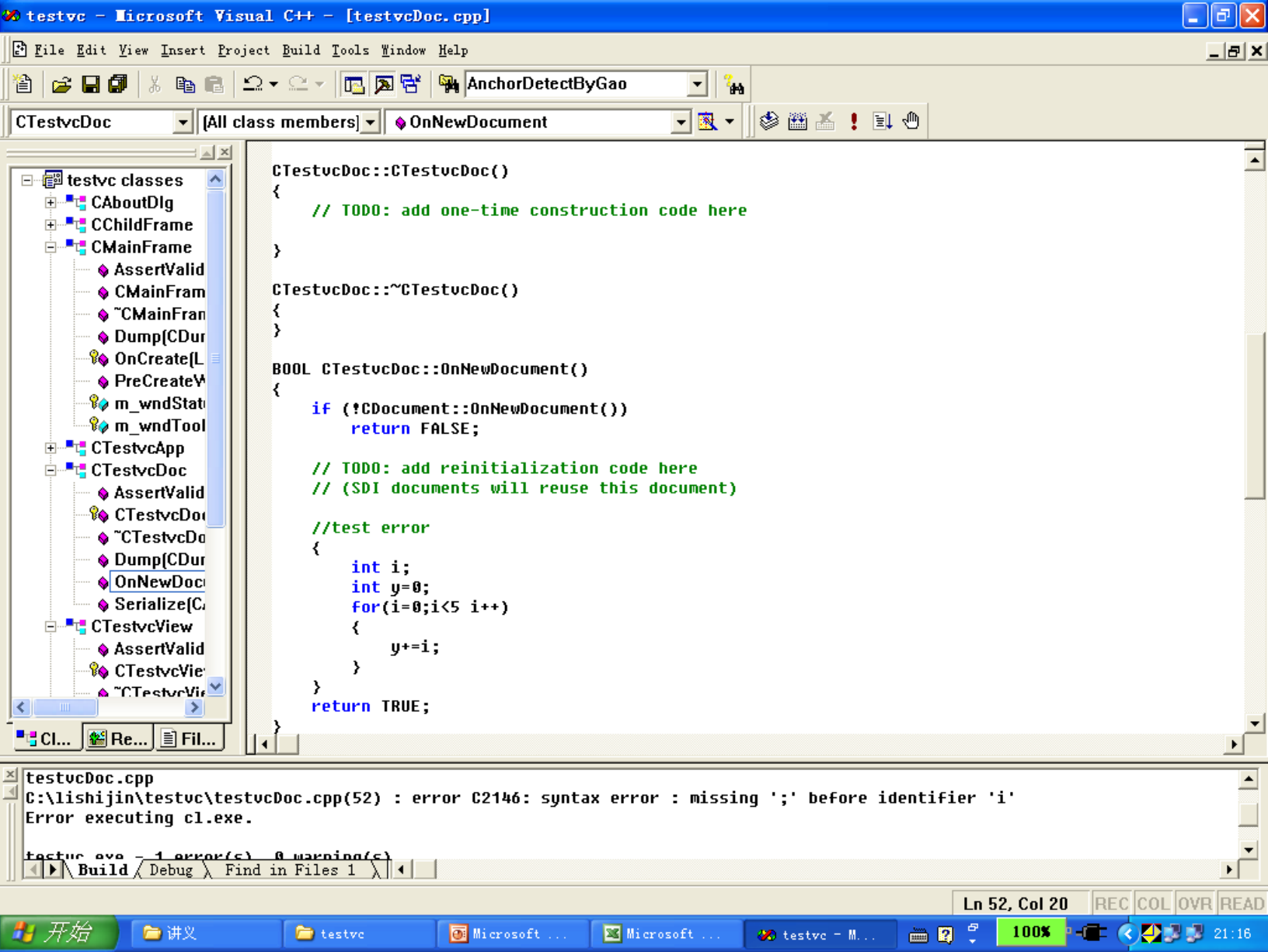
2、语法分析概述

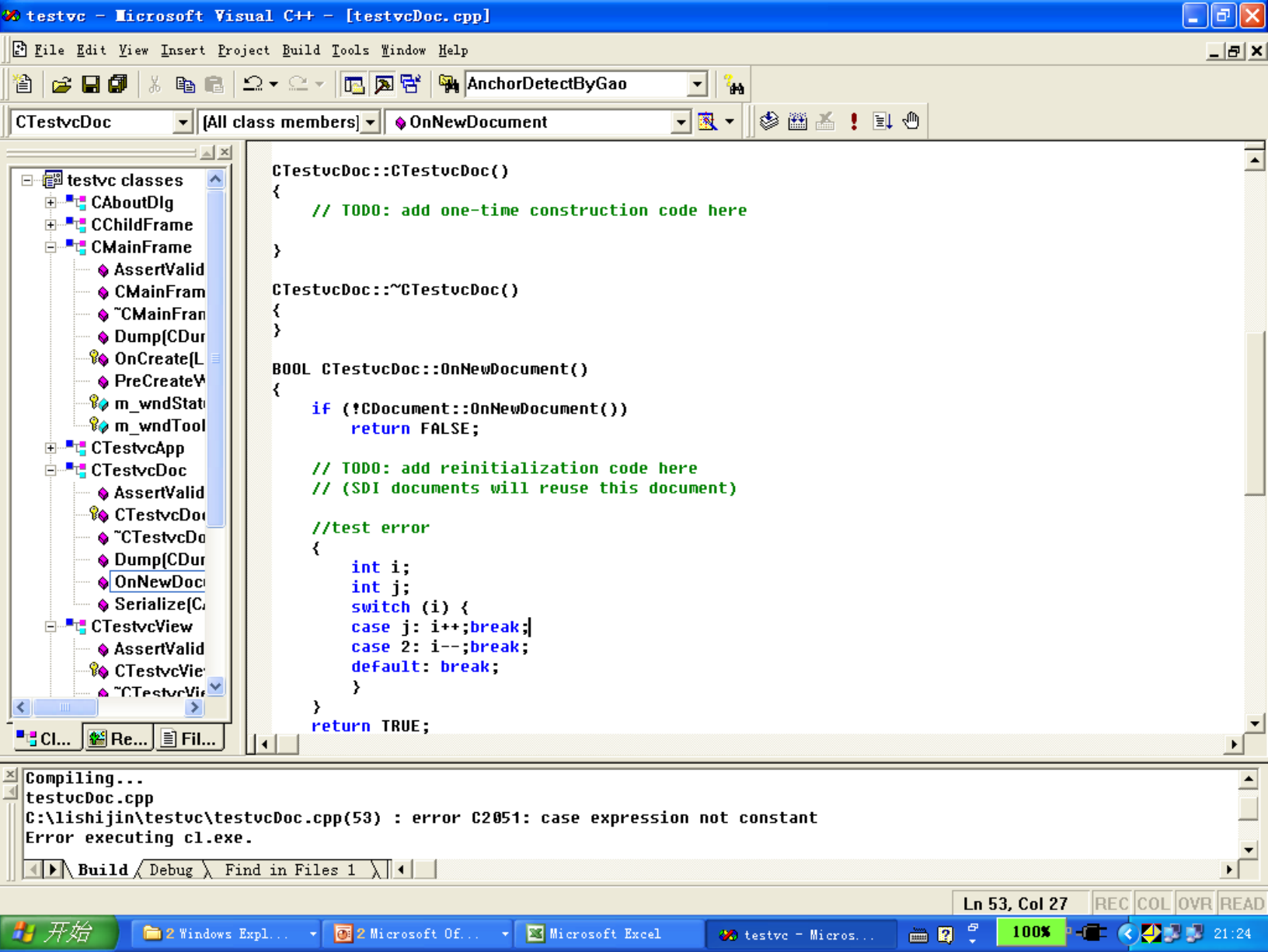
- 错误处理

  - 开始单词错
  - 后继单词错
  - 标识符和常量错
  - 括号类配对错
  - 分隔符错


testvc - Microsoft Visual C++ - [testvcDoc.cpp]
File Edit View Insert Project Build Tools Window Help
AnchorDetectByGao
CTestvcDoc
[All class members]
OnNewDocument
testvc classes
CAboutDlg
CChildFrame
CMainFrame
CTestvcApp
CTestvcDoc
CTestvcView
CTestvcDoc::CTestvcDoc()
{
// TODO: add one-time construction code here
}
CTestvcDoc::~CTestvcDoc()
{
}
BOOL CTestvcDoc::OnNewDocument()
{
if (!CDocument::OnNewDocument())
return FALSE;
// TODO: add reinitialization code here
// (SDI documents will reuse this document)
//test error
{
int i;
i++
}
return TRUE;
}
testvcDoc.cpp
C:\lishijin\testvc\testvcDoc.cpp(52) : error C2143: syntax error : missing ';' before '}'
Error executing cl.exe.
Build
Debug
Find in Files 1
Ln 51, Col 12
REC COL OVR READ
开始
讲义
testvc
Microsoft ...
Microsoft ...
testvc - M...
100%
21:11


testvc - Microsoft Visual C++ - [testvcDoc.cpp]
File Edit View Insert Project Build Tools Window Help
AnchorDetectByGao
CTestvcDoc
[All class members]
OnNewDocument
testvc classes
CAboutDlg
CChildFrame
CMainFrame
CTestvcApp
CTestvcDoc
CTestvcView
CTestvcDoc::CTestvcDoc()
{
// TODO: add one-time construction code here
}
CTestvcDoc::~CTestvcDoc()
{
}
BOOL CTestvcDoc::OnNewDocument()
{
if (!CDocument::OnNewDocument())
return FALSE;
// TODO: add reinitialization code here
// (SDI documents will reuse this document)
//test error
{
int i;
int y=2i++;
for(i=0;i<5;i++
{
y+=i;
}
}
return TRUE;
}
testvcDoc.cpp
C:\lishijin\testvc\testvcDoc.cpp(51) : error C2059: syntax error : 'bad suffix on number'
C:\lishijin\testvc\testvcDoc.cpp(51) : error C2146: syntax error : missing ';' before identifier 'i'
C:\lishijin\testvc\testvcDoc.cpp(53) : error C2143: syntax error : missing ')' before '{'
Build
Debug
Find in Files 1
Ln 52, Col 24
REC COL OVR READ
开始
讲义
testvc
Microsoft ...
Microsoft ...
testvc - M...
100%
21:14


testvc - Microsoft Visual C++ - [testvcDoc.cpp]
File Edit View Insert Project Build Tools Window Help
AnchorDetectByGao
CTestvcDoc
[All class members]
OnNewDocument
testvc classes
CAboutDlg
CChildFrame
CMainFrame
CTestvcApp
CTestvcDoc
CTestvcView
CTestvcDoc::CTestvcDoc()
{
// TODO: add one-time construction code here
}
CTestvcDoc::~CTestvcDoc()
{
}
BOOL CTestvcDoc::OnNewDocument()
{
if (!CDocument::OnNewDocument())
return FALSE;
// TODO: add reinitialization code here
// (SDI documents will reuse this document)
//test error
{
int i;
int y=0;
for(i=0;i<5 i++)
{
y+=i;
}
}
return TRUE;
}
testvcDoc.cpp
C:\lishijin\testvc\testvcDoc.cpp(52) : error C2146: syntax error : missing ';' before identifier 'i'
Error executing cl.exe.
Build
Debug
Find in Files 1
Ln 52, Col 20
REC COL OVR READ
开始
讲义
testvc
Microsoft ...
Microsoft ...
testvc - M...
100%
21:16


testvc - Microsoft Visual C++ - [testvcDoc.cpp]
File Edit View Insert Project Build Tools Window Help
AnchorDetectByGao
CTestvcDoc
[All class members]
OnNewDocument
testvc classes
CAboutDlg
CChildFrame
CMainFrame
CTestvcApp
CTestvcDoc
CTestvcView
CTestvcDoc::CTestvcDoc()
{
// TODO: add one-time construction code here
}
CTestvcDoc::~CTestvcDoc()
{
}
BOOL CTestvcDoc::OnNewDocument()
{
if (!CDocument::OnNewDocument())
return FALSE;
// TODO: add reinitialization code here
// (SDI documents will reuse this document)
//test error
{
int i;
int j;
switch (i) {
case j: i++;break;
case 2: i--;break;
default: break;
}
}
return TRUE;
Compiling...
testvcDoc.cpp
C:\lishijin\testvc\testvcDoc.cpp(53) : error C2051: case expression not constant
Error executing cl.exe.
Build
Debug
Find in Files 1
Ln 53, Col 27
REC COL OVR READ
开始
2 Windows Expl...
2 Microsoft Of...
Microsoft Excel
testvc - Micros...
100%
21:24

3、递归子程序法

- 基本思想：对源程序的每个语法成分编制一个处理子程序，从处理这个语法成分的子程序开始，在分析过程中调用一系列过程或函数，对源程序进行语法语义分析，直到整个源程序处理完毕。

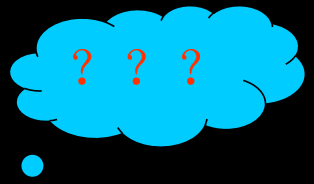
- 子程序
  - 简单子程序
  - 嵌套子程序
  - 递归子程序

- 子程序执行机制分析：进入子程序时要保存现场；退出子程序时要恢复现场。

递归子程序语法分析方法

- 语法规则预处理：
  - 消除左递归
  - 提取公因子

消除左递归



- 无法根据左递归文法进行自顶向下的分析

A　　$a_1a_2\cdots\cdots a_i\cdots\cdots a_n$

- 直接左递归
  - $A\Rightarrow A\alpha$

当前变量 (栈顶、最左变量)　　输入指针

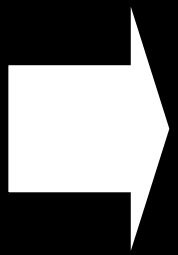
- 间接左递归
  - $A\Rightarrow^{+}A\alpha$
- 左递归的消除方法
  - 将$A\rightarrow A\alpha|\beta$替换为$A\rightarrow\beta A'$ 和 $A'\rightarrow\alpha A'\ |\ \varepsilon$

例：表达式文法直接左递归的消除

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$



$E \rightarrow T\ E'$

$E' \rightarrow + T\ E' \mid \varepsilon$

$T \rightarrow F\ T'$

$T' \rightarrow * F\ T' \mid \varepsilon$

$F \rightarrow (E) \mid id$

例：间接左递归的消除

S→Ac|c　　A→Bb|b　　B→Sa|a

将B的定义代入A产生式得：A→Sab|ab|b

将A的定义代入S产生式得：S→Sabc|abc|bc|c

消除直接左递归：　　S→abcS'|bcS'|cS'

S'→abcS'|ε

删除“多余的”产生式：A→Sab|ab|b和B→Sa|a

结果：　　S→abcS'|bcS'|cS'

S'→abcS'|ε

消除左递归的一般方法

- 用产生式组
  - $A \rightarrow \beta_1 B | \beta_2 B | \ldots | \beta_m B$
  - $B \rightarrow \alpha_1 B | \alpha_2 B | \ldots | \alpha_n B | \varepsilon$
- 替换产生式组
  - $A \rightarrow A\alpha_1 | A\alpha_2 | \ldots | A\alpha_n | \beta_1 | \beta_2 | \ldots | \beta_m$
  - 其中：B为新变量，相当于A’

提取左因子

- 例：if语句的原始文法
  - S→ if E then S
    | if E then S else S
    | other
- 存在左因子 if E then S
- 影响分析:遇到 if 时难以判断用哪一个产生式进行匹配（推导）

左因子提取方法

将形如

$A \rightarrow \alpha\beta_1|\alpha\beta_2|\ldots|\alpha\beta_n|\gamma_1|\gamma_2|\ldots|\gamma_m$

的规则改写为

$A \rightarrow \alpha A'|\gamma_1|\gamma_2|\ldots|\gamma_m$ 和 $A' \rightarrow \beta_1|\beta_2|\ldots|\beta_n$

- 结果：
  - $S \rightarrow$ if E then SS'|other
  - $S' \rightarrow \varepsilon$|else S

例:简单算术表达式的分析器

- E的子程序

E→ T E’
E’→ + T E’|ε

```
procedure E;
  begin
     T;                          T 的过程调用
   while lookhead='+' do
      begin                      当前符号等于＋时
         match(‘+’);     处理终结符＋
         T                T 的过程调用
   end
  end;                          lookahead：当前符号
```

T的子程序

T→ F T’
T’→* F T’ | ε

```
procedure T;
  begin
     F;                          F 的过程调用
   while lookhead='*' then
       begin                     当前符号等于＊时
     match(‘*’);                处理终结符＊
     F                           F 的递归调用
    end
  end;
```

F的子程序（F→(E)|id）

```
procedure F;
  begin
     if lookhead='(' then
        begin                          当前符号等于（
        match('(');                处理终结符（
        E;                         E 的递归调用
        match(')');                处理终结符）
     end
     else if lookhead=id then
                match(id)              处理终结符id
     else error                        出错处理
  end
```

主程序

```
begin
   lookhead:=nexttoken;          调词法分析程序
  E                              E 的过程调用
end
procedure match(t:token);
  begin
     if lookhead=t  then
          lookhead:=nexttoken
     else error                  出错处理程序
  end;
```

递归下降子程序

program –> function_list

function_list –> function function_list | ε

function –> FUNC identifier (parameter_list)
  statement

```
void ParseFunction()
{
   MatchToken(T_FUNC);
   ParseIdentifier();
   MatchToken(T_LPAREN);
   ParseParameterList();
   MatchToken(T_RPAREN);
   ParseStatement();
}
```

优缺点分析

- 优点：
  - 1）直观、简单、可读性好
  - 2）便于扩充
- 缺点：
  - 1) 递归算法的实现效率低
  - 2) 处理能力相对有限
  - 3) 通用性差，难以自动生成

第四章 语法分析
Syntax Analysis

Part II

4、LL(1)预测分析法

- 1) 常用终结符号集计算
- 2) LL(1)文法
- 3) LL(1)预测分析表
- 4) LL(1)预测分析方法

1) 常用终结符号集

- First(**α**)
- Follow(A)
- Select(**A→β**)

FIRST 和 FOLLOW 集

- 对于$\forall\alpha\in(V_T\cup V_N)^*$定义:α的首符号集
  - FIRST(α)={a|$\alpha\Rightarrow^*$a…，$a\in V_T$}
  - 若$\alpha\Rightarrow^*\varepsilon$则规定$\varepsilon\in$ FIRST(α)。

- 对于$\forall A\in V_N$定义A的后续符号集:
  - FOLLOW(A)={a|$S\Rightarrow^*$…Aa…，$a\in V_T$}

- if β *=> ε
  - Select(A → β) = First(β)\{ ε } ∪ Follow(A)
- else
  - Select(A → β) = First(β)

- S->pA
- S->qB
- A->cAd
- A->a

pccadd

- S->Ap
- S->Bq
- A->a
- A->cA
- B->b
- B->dB

ccap

ddbq

- S->aA
- S->d
- A->bAS
- A-> **ε**

abd

求FIRST(X) 的算法

1) 对$\forall X \in V_T$，FIRST(X)={X}；

2) 对$\forall X \in V_N$，设置FIRST(X)的初值：

$$FIRST(X)=\begin{cases}\{a|X\to a\ldots\in P\}; & X\to\varepsilon\notin P\\ \{a|X\to a\ldots\in P\}\cup\{\varepsilon\}; & X\to\varepsilon\in P\end{cases}$$

3)对$\forall X \in V_N$，重复如下过程，直到所有FIRST集不变

若 $X \to Y\ldots \in P$，且 $Y \in V_N$，

则 $FIRST(X) = FIRST(X) \cup (FIRST(Y) - \{\varepsilon\})$

若 $X \to Y_1 \ldots Y_k \in P$，且 $Y_1 \ldots Y_{i-1} \Rightarrow^* \varepsilon$，

则 对n=1到i-1:

$FIRST(X) = FIRST(X) \cup (FIRST(Y_n) - \{\varepsilon\})$

若 $Y_1 \ldots Y_k \Rightarrow^* \varepsilon$，

则 $FIRST(X) = FIRST(X) \cup \{\varepsilon\}$

例:表达式文法的语法符号的FIRST集

FIRST(F)={（,id}

FIRST(T)=FIRST(F)={（,id}

FIRST(E)=FIRST(T)={（,id}

FIRST(E‘)={+, ε}

FIRST(T‘)={*, ε}

FIRST(+)={+}, FIRST(*)={*}

FIRST(（)={（}

FIRST(）)={）}

FIRST(id)={id}

E→TE'

E'→+TE’|ε

T→FT'

T'→*FT’|ε

F→(E)|id

求FOLLOW(B) 的算法

- 1) 首先将 # 加入到 FOLLOW(S)
  - #为句子括号

对于所有非终结符, 重复进行以下计算

- 2) 若 $A \rightarrow \alpha B \beta$，
  - 则 $FOLLOW(B)=FOLLOW(B) \cup FIRST(\beta)-\{\varepsilon\}$
- 3) 如果$A \rightarrow \alpha B$或$A \rightarrow \alpha B \beta$，且$\beta \Rightarrow^{*} \varepsilon$，$A \neq B$，
  - 则 $FOLLOW(B)=FOLLOW(B) \cup FOLLOW(A)$

例 表达式文法的语法变量的 FOLLOW集

FOLLOW(E) = {), # }

FOLLOW(E') = FOLLOW(E) = {), # }

FOLLOW(T) = { +,), # }

FOLLOW(T') = FOLLOW(T) = { +,), # }

FOLLOW(F) = { *, +,), # }

Select(E→TE’)={(,id }

Select(E'→+TE’)={+}

Select(E'→ε)={),#}

Select(T→FT’)={ (,id }

Select(T'→ε)={+,),#}

T'→*FT’|ε　　F→(E)|id

Review

The *first set* of a sequence of symbols u, written as First(u) is the set of terminals which start all the sequences of symbols derivable from u. A bit more formally, consider all strings derivable from u by a leftmost derivation. If u =>* v , where v begins with some terminal, that terminal is in First(u). If u =>* ε , then ε is in First(u).

The *follow set* of a nonterminal A is the set of terminal symbols that can appear immediately to the right of A in a valid sentence. A bit more formally, for every valid sentence S =>*uAv , where v begins with some terminal, that terminal is in Follow(A).

To calculate First(u) where u has the form X1X2...Xn, do the following:

1. If X1 is a terminal, then add X1 to First(u), otherwise add First(X1) - ε to First(u) .
2. If X1 is a nullable nonterminal, i.e., X1 =>* ε , add First(X2) - ε to First(u). Furthermore,

   if X2 can also go to ε , then add First(X3) - ε and so on, through all Xn until the first nonnullable one.
3. If X1X2...Xn =>* ε , add ε to the First set.

Follow(.)

1. Place# in Follow(S) where S is the start symbol and # is the input's right endmarker. The endmarker might be end of file, it might be newline, it might be a special symbol, whatever is the expected end of input indication for this grammar. We will typically use # as the endmarker.
2. For every production A –> uBv where u and v are any string of grammar symbols and B is a nonterminal, everything in First(v) except ε is placed in Follow(B).
3. For every production A –> uB, or a production A –> uBv where First(v) contains ε (i.e. v is nullable), then everything in Follow(A) is added to Follow(B).

例:

- S->aAd
- A->BC
- B->b | ε
- C->c | ε

求First(A), Follow(B)

- **S → A B**
- **A → B a | ε**
- **B → D b | D**
- **D → d | ε**

Select(S → A B)=?

2) LL(1)文法

Top-Down Parsing

- 从左到右扫描输入串——寻找它的一个最左推导
- 对于 G 的每个非终结符 A 的任何两个不同的候选式 $A \rightarrow \alpha|\beta$

  1) $FIRST(\alpha) \cap FIRST(\beta) = \varphi$

  2) 如果$\beta \Rightarrow^{*} \varepsilon$，则 $FISRT(\alpha) \cap FOLLOW(A) = \varphi$
- ——文法G是 LL(1) 的充要条件

- LL(1)文法表示了自顶向下方法能够处理的一类文法
  - 第一个 L 表示从左向右扫描输入符号串
  - 第二个 L 表示生成最左推导
  - 1 表示读入一个符号可确定下一步推导

表达式文法是 LL(1) 文法

- **E → T E'**
- **E'→ + T E' | ε**
- **T → F T'**
- **T'→ * F T' | ε**
- **F → (E) | id**

考察:

- **E’: + 不在 FOLLOW(E’) = {), # }**
- **T’: * 不在 FOLLOW(T’) = { +,), # }**
- **F: (和 id 不同**

这个文法是LL(1)文法？

G[S]:

- S->AB | bC
- A->b | ε
- B->aD | ε
- C->AD | b
- D->aS | c

对所有LL(1)文法

希望有一种统一的控制算法

- 直接根据
  - 当前的语法变量
  - 输入符号
- 确定进行分析所需的候选产生式
- 由一个统一的算法实现分析

预测分析方法

- 系统维持一个分析表和一个分析栈，根据当前扫描到的符号，选择当前语法变量（处于栈顶）的候选式进行推导——希望找到相应的输入符号串的最左推导。
- 一个通用的控制算法
- 一个统一形式的分析表M
  - 不同语言使用内容不同的分析表
- 一个分析栈，#为栈底符号
- 一个输入缓冲区，#为输入串结束符

表达式文法的预测分析表

非终结符	输入符号					
	id	+	*	(	)	#
E	→TE'			→TE'		
E′		→+TE			→ ε	→ ε
T	→FT'			→FT'		
T′		→ ε	→*FT'		→ ε	→ ε
F	→id			→(E)		

3）LL(1)分析表的构造

LL(1)分析表的构造算法I

- 1) 对于每一产生式 A→α，执行2)
- 2) 对于 Select(A→α)中的每一终结符a, 将 A→α 填入 M[A,a]
- 3) 将所有无定义的 M[A,b] 标上错误标志

G[E]: (1) E –> TE’ (2) E’ –> +TE’ (3) E’ –> ε
(4) T –> FT’ (5) T’ –> *FT’ (6) T’ –> ε
(7) F –> (E) (8) F –> id

	id	+	*	(	)	#
E	–> TE’			–> TE’		
E’						
T						
T’						
F						

Select(E→TE’)={(,id }

G[E]: (1) E –> TE’ (2) E’ –> +TE’ (3) E’ –> ε
(4) T –> FT’ (5) T’ –> *FT’ (6) T’ –> ε
(7) F –> (E) (8) F –> id

	id	+	*	(	)	#
E	–> TE’			–> TE’		
E’		–> +TE’			–> ε	–> ε
T						
T’						
F						

Select(E'→+TE’)={+}

Select(E'→ε)={),#}

G[E]: (1) E –> TE’ (2) E’ –> +TE’ (3) E’ –> ε
(4) T –> FT’ (5) T’ –> *FT’ (6) T’ –> ε
(7) F –> (E) (8) F –> id

	id	+	*	(	)	#
E	–> TE’			–> TE’		
E’		–> +TE’			–> ε	–> ε
T	–> FT’			–> FT’		
T’						
F						

Select(T→FT’)={ (,id }

G[E]: (1) E –> TE’ (2) E’ –> +TE’ (3) E’ –> ε
(4) T –> FT’ (5) T’ –> *FT’ (6) T’ –> ε
(7) F –> (E) (8) F –> id

	id	+	*	(	)	#
E	–> TE’			–> TE’		
E’		–> +TE’			–> ε	–> ε
T	–> FT’			–> FT’		
T’		–> ε	–> *FT’		–> ε	–> ε
F	–> id			–>(E)		

Select(T'→ε)={+,),#}

G[E]: (1) E –> TE’ (2) E’ –> +TE’ (3) E’ –> ε
(4) T –> FT’ (5) T’ –> *FT’ (6) T’ –> ε
(7) F –> (E) (8) F –> id

	id	+	*	(	)	#
E	–> TE’			–> TE’		
E’		–> +TE’			–> ε	–> ε
T	–> FT’			–> FT’		
T’		–> ε	–>*FT’		–> ε	–> ε
F	–> id			–>(E)		

表达式文法的预测分析表

非终结符	输入符号					
	id	+	*	(	)	#
E	→TE'			→TE'		
E′		→+TE			→ ε	→ ε
T	→FT'			→FT'		
T′		→ ε	→*FT'		→ ε	→ ε
F	→id			→(E)		

预测分析表构造算法II

1.对文法G的每个产生式A→α执行第二步和第三步；

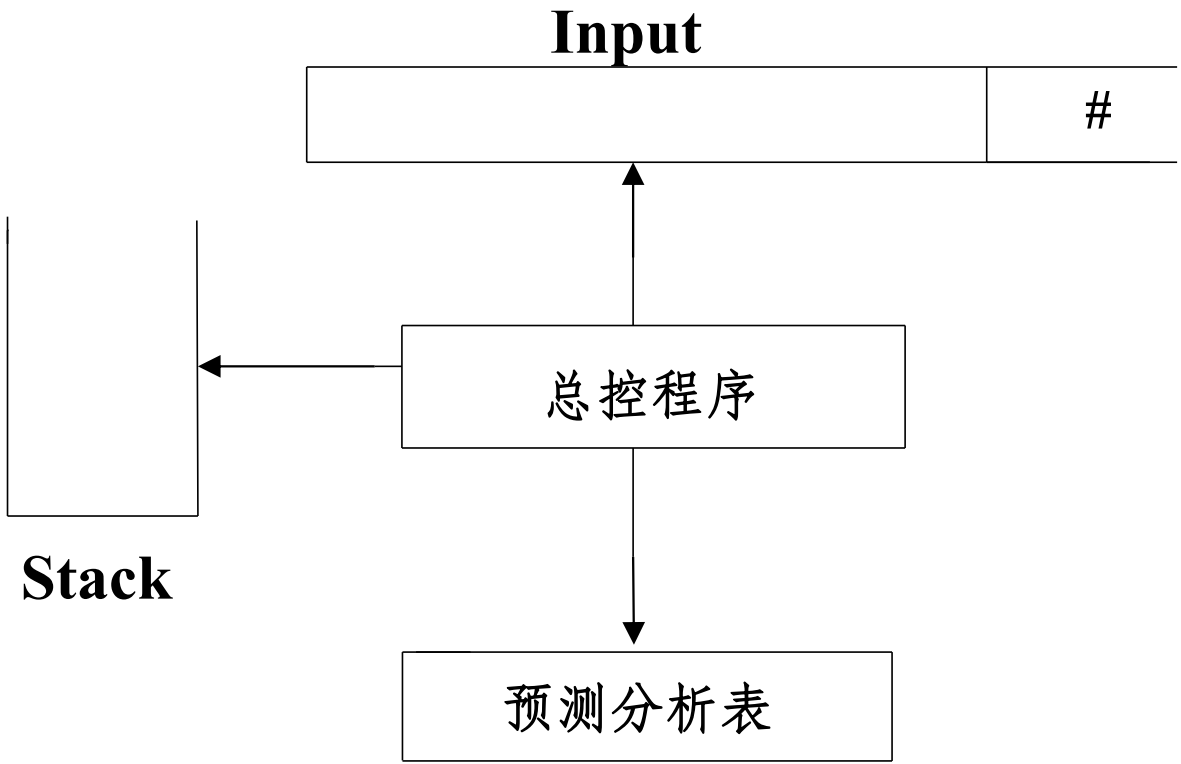
2.对每个终结符a∈FIRST(α)，把A→α加至M[A,a]中，

3.若ε∈ FIRST(α)，则对任何b∈FOLLOW(A)把A→α加至M[A,b]中，

4.把所有无定义的M[A,a]标上“出错标志”。

4) LL(1)分析法

表驱动预测分析程序模型



系统的执行与特点

- 在系统启动时，输入指针指向输入串的第一个字符，分析栈中存放着栈底符号#和文法的开始符号。
- 根据栈顶符号A和读入的符号a，查看分析表M,以决定相应的动作。
- 优点：
  - 1）效率高
  - 2）便于维护、自动生成
- 关键——分析表M的构造

LL(1)通用控制算法

repeat

X=当前栈顶符号;a=当前输入符号;

if $X \in V_T \cup \{\#\}$ then

if X=a then

if X≠# then {将X弹出,且前移输入指针}

else error

else

if $M[X,a]=Y_1Y_2\ldots Y_k$ then

{将X弹出;依次将$Y_k\ldots Y_2Y_1$压入栈;

输出产生式$X \rightarrow Y_1Y_2\ldots Y_k$ }

else error

until X=#

执行例：分析 id+id*id

栈	输入缓冲区	输出
#E	id+id*id#	
#E‘T	id+id*id#	E→TE'
#E‘T’F	id+id*id#	T→FT'
#E‘T’id	id+id*id#	F→id
#E‘T’	+id*id#	
#E‘	+id*id#	T'→ε
#E‘T+	+id*id#	E'→+TE'
#E‘T	id*id#	

#E‘T	id*id#	
#E‘T’F	id*id#	T→FT'
#E‘T’id	id*id#	F→id
#E‘T’	*id#	
#E‘T’F*	*id#	T'→*FT'
#E‘T’F	id#	
#E‘T’id	id#	F→id
#E‘T’	#	
#E‘	#	T'→ε
#	#	E'→ε

输出的产生式序列形成了最左推导对应的分析树

- #i*(i+i)#

栈	输入
#E	i*(i+i)#

LL(1)分析中的一种错误处理办法

发现错误：

1栈顶的终结符与当前输入符不匹配

2非终结符A于栈顶，面临的输入符为a，但分析表M的M[A,a]为空

“应急”恢复策略：

跳过输入串中的一些符号直至遇到“同步符号”为止

同步符号的选择：

1.把FOLLOW(A)中的所有符号作为A的同步符号。跳过输入串中的一些符号直至遇到这些“同步符号”，把A从栈中弹出，可使分析继续

2. 把FIRST(A)中的符号加到A的同步符号集，当FIRST(A)中的符号在输入中出现时，可根据A恢复分析

LL(1)预测分析法(小结)

- 消除左递归、提取左因子；
- 求每个候选式的FIRST集和非终结符号的FOLLOW集
- 检查是不是 LL(1) 文法
  - 若不是 LL(1),说明文法复杂性超过LL(1)分析法的分析能力
- 构造预测分析表
- 实现预测分析器

LL(1)预测分析法(小结)

存在问题:

- 对于某些语言现象，难以用 LL(1) 文法来描述
- 消除左递归和提取左因子影响文法的可读性，造成语义处理的困难

Review

predictive parser and LL(1)grammar

Predictive parser is a non-backtracking top-down parser. A predictive parser is characterized by its ability to choose the production to apply solely on the basis of the next input symbol and the current nonterminal being processed.

To enable this, the grammar must take a particular form. We call such a grammar *LL(1)*. The first “L” means we scan the input from left to right; the second “L” means we create a leftmost derivation; and the 1 means one input symbol of lookahead.

recursive-descent

The first technique for implementing a predictive parser is called *recursive-descent*.

A recursive descent parser consists of several small functions(procedures), one for each nonterminal in the grammar. As we parse a sentence, we call the functions (procedures) that correspond to the left side nonterminal of the productions we are applying. If these productions are recursive, we end up calling the functions recursively.

Table-driven LL(1) parsing

In a recursive-descent parser, the production information is embedded in the individual parse functions for each nonterminal and the run-time execution stack is keeping track of our progress through the parse. There is another method for implementing a predictive parser that uses a table to store that production along with an explicit stack to keep track of where we are in the parse.

How a table-driven predictive parser works

We push the start symbol on the stack and read the first input token. As the parser works through the input, there are the following possibilities for the top stack symbol X and the input token nonterminal a:

1. If X = a and a = end of input (#): parser halts and parse completed successfully
2. If X = a and a != #: successful match, pop X and advance to next input token. This is called a *match* action.
3. If X != a and X is a nonterminal, pop X and consult table at [X,a] to see which production applies, push right side of production on stack. This is called a *predict* action.
4. If none of the preceding cases applies or the table entry from step 3 is blank, there has been a parse error

Constructing the parse table

1. For each production A –> u of the grammar, do steps 2 and 3
2. For each terminal a in First(u), add A –> u to M[A,a]
3. If ε in First(u), (i.e. A is nullable) add A –> u to M[A,b] for each terminal b in Follow(A),

   If ε in First(u), and # is in Follow(A), add A –> u to M[A,#]
4. All undefined entries are errors

LL(1) grammar

A grammar G is LL(1) iff whenever A –> u | v are two distinct productions of G, the following conditions hold:

- for no terminal a do both u and v derive strings beginning with a (i.e. first sets are disjoint)
- at most one of u and v can derive the empty string
- if v =>* ε then v does not derive any string beginning with a terminal in Follow(A)

Error-reporting and recovery

An error is detected in predictive parsing when the terminal on top of the stack does not match the next input symbol or

when nonterminal A is on top of the stack, a is the next input symbol and the parsing table entry M[A,a] is empty.

习题

1、P.99　第1题 3),4)
2、P.100 第3题
3、P.101 第7题 1),3)