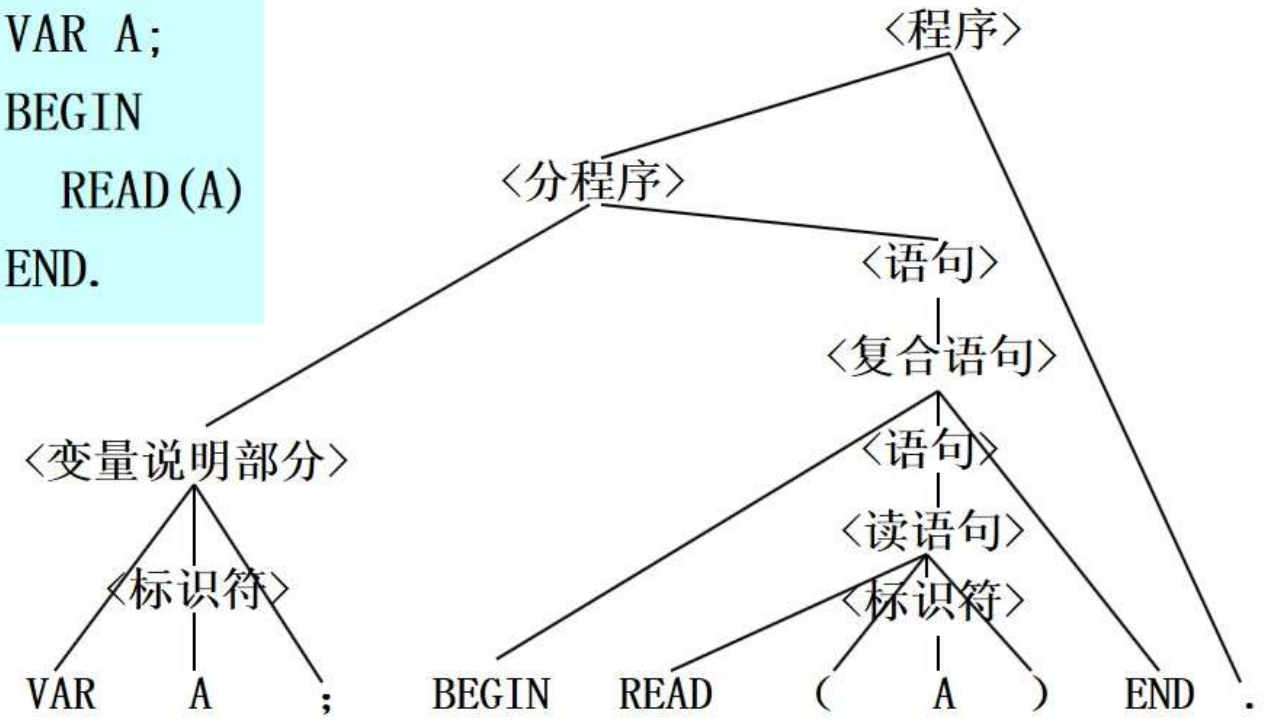
第5章　自下而上的语法分析

刘远兴

- 自下而上语法分析概述
- 简单优先分析
- 算符优先分析

自下而上语法分析概述

- VAR A;
- BEGIN
- READ(A)
- END.



3

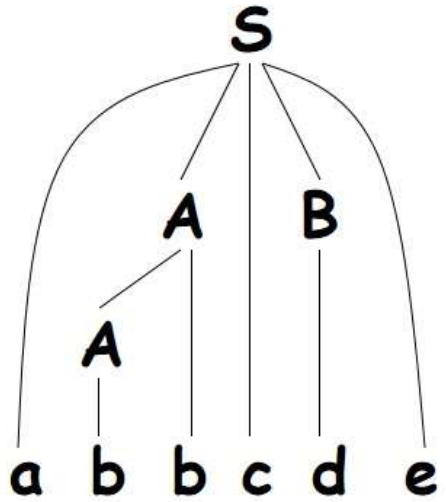
自下而上语法分析概述

- 如名字如示，自下而上语法分析试图将一个字符串反向归约至开始符号。
- 自下而上语法分析比自上而下语法分析更有效率，对语法的限制更少

4

文法G[S]:

(1) S → aAcBe
(2) A → b
(3) A → Ab
(4) B → d



步骤	符号栈	输入符号串	动作
1）	#	abbcde#	移进
2）	#a	bbcde#	移进
3）	#ab	bcde#	归约(A→b)
4）	#aA	bcde#	移进
5）	#aAb	cde#	归约(A→Ab)
6）	#aA	cde#	移进
7）	#aAc	de#	移进
8）	# aAcd	e#	归约(B→d)
9）	#aAcB	e#	移进
10）	#aAcBe	#	归约(S→aAcBe)
11）	#S	#	接受

对输入串abbcde#的移进-规约分析过程

分析符号串abbcde是否G[S]的句子

S⇒aAcBe⇒ aAcde⇒ aAbcde ⇒ abbcde

自下而上语法分析算法的一般描述

Let I = input string

repeat

pick a non-empty substring β of I

where X→ β is a production

if no such β, backtrack

replace one β by X in I

until I = "S" (the start symbol) or all possibilities are exhausted

6

算法应考虑的问题

- 算法是否能够终止？
- 算法是否快速？
- 算法是否能够处理所有的情况？
- 在每一步中如何选择子串进行归约？

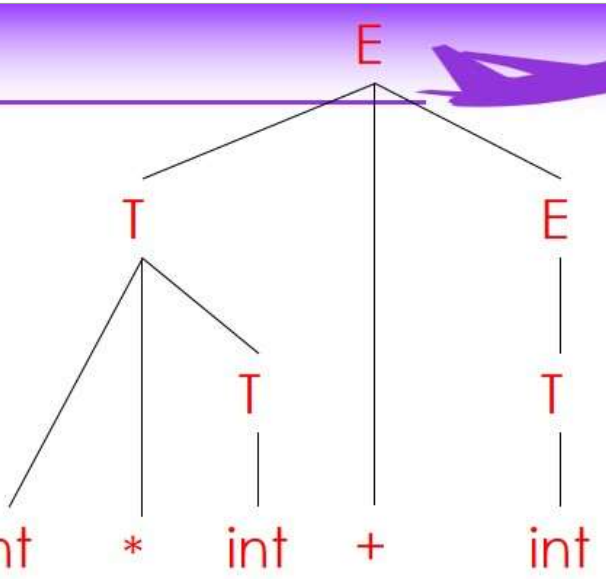
- 自下而上语法分析的策略：移进-规约分析；
- 移进就是将一个终结符推进栈
- 归约就是将0个或多个符号从栈中弹出，根据产生式将一个非终结符压入栈
- 移进-归约过程是自顶向下最右推导的逆过程（规范归约）

文法G[E]:

$E \rightarrow T + E \mid T$

$T \rightarrow int * T \mid int \mid (E)$

|int * int + int	移进
int | * int + int	移进
int * | int + int	移进
int * int | + int	规约 $T \rightarrow int$
int * T | + int	规约 $T \rightarrow int * T$
T | + int	移进
T + | int	移进
T + int |	规约 $T \rightarrow int$
T + T |	规约 $E \rightarrow T$
T + E |	规约 $E \rightarrow T + E$
E |	



A Shift-Reduce Parse in Detail (1)

|int * int + int

int * int + int

↑

A Shift-Reduce Parse in Detail (2)

|int * int + int

int | * int + int

int * int + int

↑

A Shift-Reduce Parse in Detail (3)

|int * int + int

int | * int + int

int * | int + int

int * int + int

↑

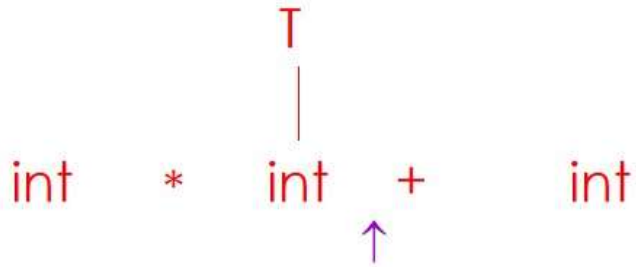
A Shift-Reduce Parse in Detail (4)

|int * int + int
int | * int + int
int * | int + int
int * int | + int

int * int + int
↑

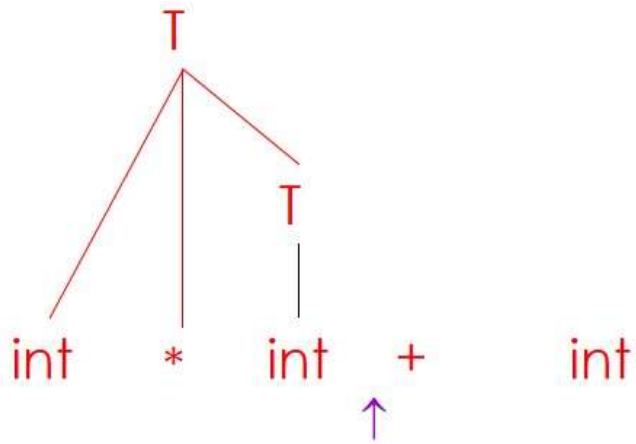
A Shift-Reduce Parse in Detail (5)

|int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int



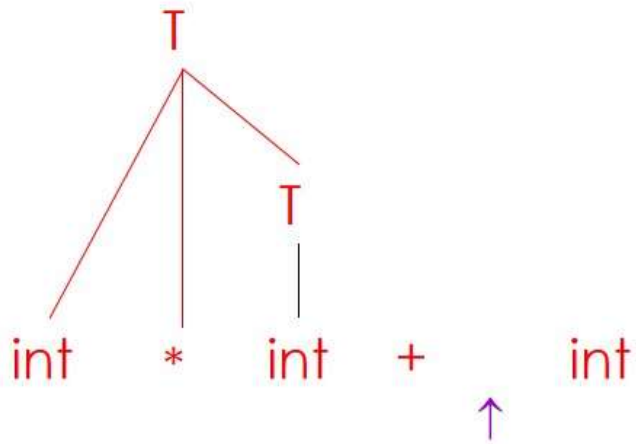
A Shift-Reduce Parse in Detail (6)

|int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int



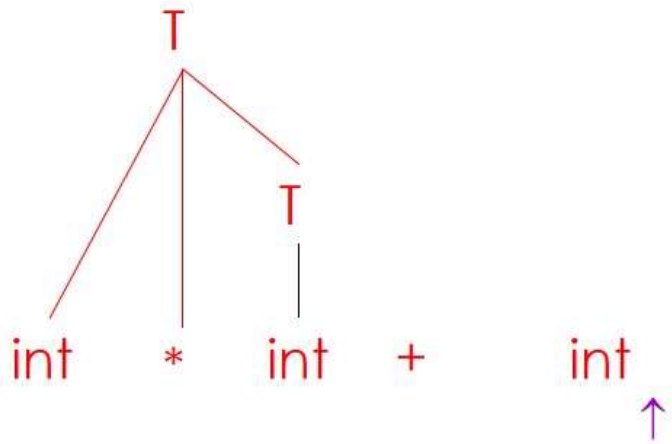
A Shift-Reduce Parse in Detail (7)

|int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int



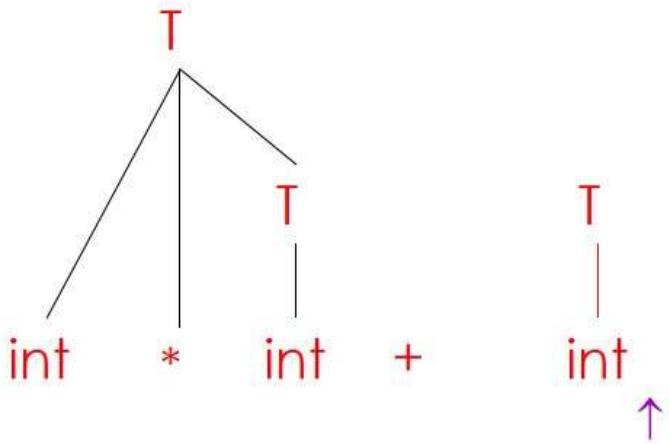
A Shift-Reduce Parse in Detail (8)

|int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |



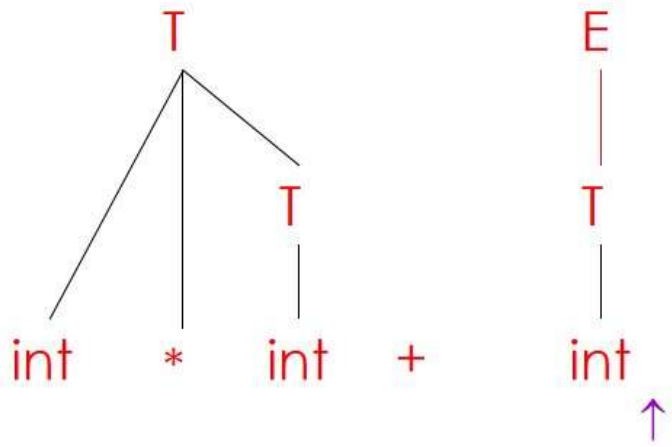
A Shift-Reduce Parse in Detail (9)

|int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |
T + T |



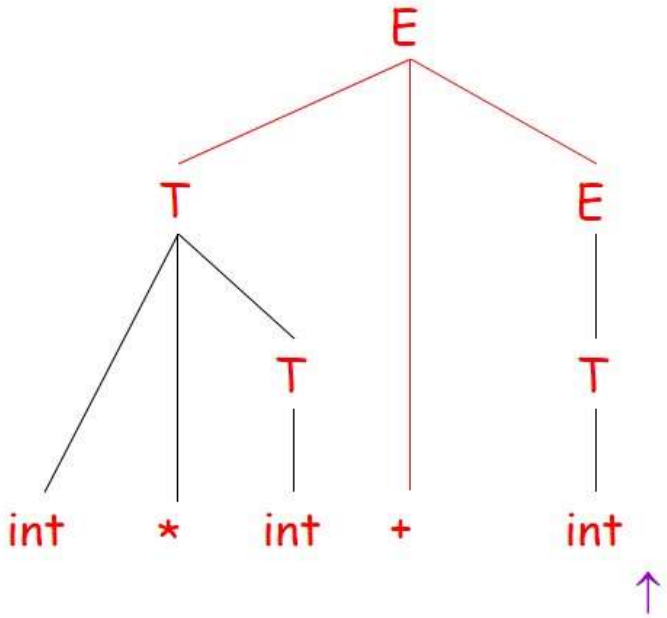
A Shift-Reduce Parse in Detail (10)

|int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |
T + T |
T + E |



A Shift-Reduce Parse in Detail (11)

|int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |
T + T |
T + E |
E |



- 我们如何决定什么时候移进，什么时候规约？
  - 考虑 int | * int + int
  - 我们可以用 T → int进行归约，而得到 T | * int + int
  - 致命错误：无法规约到开始符号 E
  - 直觉：Want to reduce only if the result can still be reduced to the start symbol
- 一般的移进-归约策略：
  - 若句柄在栈顶出现，则归约
  - 否则移进
- 句柄（Handles）：句型的最左直接短语

短语、直接短语、句柄的定义：

文法$G[S]$，

$S \overset{*}{\Rightarrow} \alpha A \delta$且$A \overset{+}{\Rightarrow} b$则称b是句型$\alpha b \delta$相对于非终结符A的短语。

若有$A \Rightarrow b$则称b是句型$\alpha b \delta$相对于该规则$A \rightarrow b$的直接短语。

一个句型的最左直接短语称为该句型的句柄。

Conflicts

- 实际应用中可能出现的’冲突‘
  - 移进与归约都合法，产生移进–归约冲突
  - 归约时可以适用两个不同的产生式，产生归约–归约冲突

Source of Conflicts

- 二义文法会导致‘冲突’
- 但应注意，许多的非二义文法同样会导致‘冲突’

Conflict Example

考虑下面的二义文法:

$$
\begin{array}{lcl}
E & \rightarrow & E + E \\
 & | & E * E \\
 & | & (E) \\
 & | & int
\end{array}
$$

One Shift-Reduce Parse

|int * int + int	shift
. . .	. . .
E * E | + int	reduce E → E * E
E | + int	shift
E + | int	shift
E + int|	reduce E → int
E + E |	reduce E → E + E
E |	

Another Shift-Reduce Parse

|int * int + int	shift
. . .	. . .
E * E | + int	shift
E * E + | int	shift
E * E + int |	reduce E → int
E * E + E|	reduce E → E + E
E * E |	reduce E → E * E
E |	

Conflict Solutions

- 改写文法
- 根据产生式出现的顺序来选择
- 根据算符的优先级

自下而上的分析算法

- 优先分析法
  - 简单优先分析法
  - 算符优先分析法
- LR分析

简单优先分析法

- 按照文法符号（包括终结符和非终结符）的优先关系确定句柄。
- 示例见下页

30

文法G[S]:

(1) S → bAb

(2) A → (B|a

(3) B → Aa)

	S	b	A	(	B	a	)	#
S								>
b			=	<		<		>
A		=				=		
(			<	<	=	<		
B		>				>		
a		>				>	=	
)		>				>		
#	<	<						

简单优先关系矩阵

步骤	符号栈	输入符号串	动作
1)	#	b(aa)b#	#<b,移进
2)	#b	(aa)b#	b<(,移进
3)	#b(	aa)b#	(<a,移进
4)	#b(a	a)b#	a>a,归约A→a
5)	#b(A	a)b#	A=a,移进
6)	#b(Aa	)b#	a=),移进
7)	#b(Aa)	b#	)>b,归约B→Aa)
8)	#b(B	b#	B>b,归约A→(B
9)	#bA	b#	A=b,移进
10)	#bAb	#	b>#,归约S→bAb
11)	#S	#	接受

对输入串b(aa)#的简单优先分析过程

优先关系

- 如何确定两个文法符号之间的优先关系？

- 优先关系
  - X=Y ⇔ 文法G中存在产生式A→...XY...
  - X<Y ⇔ 文法G中存在产生式A→...XB...，且 $B \overset{+}{\Rightarrow} Y...$
  - X>Y⇔ 文法G中存在产生式A→...BD...，且 $B \overset{+}{\Rightarrow} ...X$，$D \overset{*}{\Rightarrow} Y...$

简单优先文法的定义

满足以下条件的文法是简单优先文法

（1）在文法符号集V中，任意两个符号之间最多只有一种优先关系成立。

（2）在文法中任意两个产生式没有相同的右部

（3）不含空产生式