

Top Down Parsing Issues

Consider: `procedure id (param list) ;` param list is optional

where `param list => param : type; param : type;...param:type`
`param => var id, id, ..., id`
var is optional

Context-free Grammar:

```
S -> procedure id P ; | ε
P -> ( L ) | ε
L -> R : T | R : T ; L
R -> V D
V -> var | ε
D -> D , id | id
T -> int | real
```

String: `procedure print (var x,y,z: int; a,b: real);`

Recursive-descent Parsing

CFG: $T \rightarrow T * F \mid F$
 $F \rightarrow (E) \mid id$

$\Rightarrow T \rightarrow F T'$
 $T' \rightarrow * F T' \mid \epsilon$
 $F \rightarrow (E) \mid id$

T()
{

F()
{

}

T'()
{

}

}

Consider input string: $a * b * c$

Predictive Parser (Top Down)

Nonterminal	Input Token					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Let input token stream be: (id + id) * id \$

Initial Stack:

E
\$



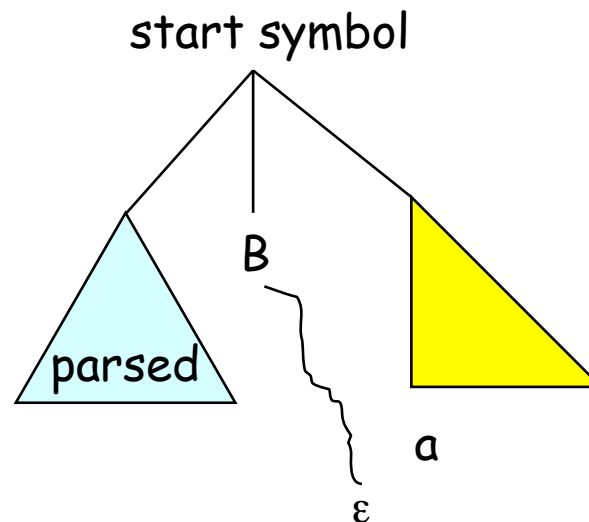
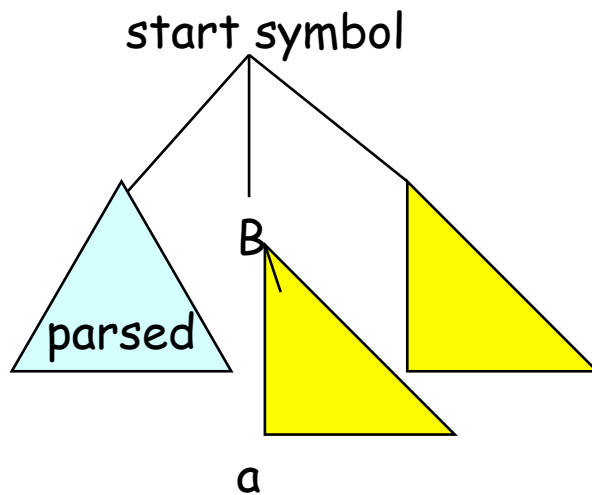
Predictive LL(1) Parse Table Build

Key Insight:

Given input "a" and nonterminal B to be expanded, which one of the alternatives

$$B \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$$

is the unique choice to derive a string starting with "a"?



Computing FIRST and FOLLOW

FIRST(α) = set of terminals that can begin strings derived by α

$$\text{FIRST}(\alpha) = \{ a \mid \alpha \Rightarrow a\beta \text{ for some } \beta \}$$

$E \rightarrow T E'$
 $E' \rightarrow + T E' \mid \varepsilon$
 $T \rightarrow F T'$
 $T' \rightarrow * F T' \mid \varepsilon$
 $F \rightarrow (E) \mid \text{id}$

FOLLOW(N) = set of terminals that can immediately follow N in right sentential form

FOLLOW(N):

For $A \rightarrow \alpha N \beta$, Add FIRST(β), except ε , to FOLLOW(N)

For $A \rightarrow \alpha N \beta$ and FIRST(β) has ε , or $A \rightarrow \alpha N$,

Add FOLLOW(A) to FOLLOW(N)

Add \$ to FOLLOW(START SYMBOL)

LL(1) Parse Table Construction

Look at each production, $A \rightarrow \alpha$, and $\text{FIRST}(\alpha)$:

- If terminal a in $\text{FIRST}(\alpha) \implies$

$$\frac{\quad}{A \mid A \rightarrow \alpha} \quad a$$

- If ϵ in $\text{FIRST}(\alpha)$, then

 If terminal b in $\text{FOLLOW}(A) \implies$

$$\frac{\quad}{A \mid A \rightarrow \alpha} \quad b$$

 If $\$$ in $\text{FOLLOW}(A) \implies$

$\$$

$$\frac{\quad}{A \mid A \rightarrow \alpha}$$

Nonterminal	id	+	*	()	\$
-------------	----	---	---	---	---	----

E
E'
T
T'
F

Is a Grammar LL(1)?

Method:

- * Construct table and look for multidefined entries.
If no multidefined entries, then LL(1) grammar
- * Look at FIRST and FOLLOW sets as follows:

A grammar G is LL(1) iff

whenever there exists $A \rightarrow \alpha \mid \beta$ in G ,
all of the following conditions hold true:

- * $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$
- * At most 1 of α and β derive the empty string
- * If β derives the empty string, then
 $\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \emptyset$

Summary: Top-down Parsing

- * To avoid backtracking:
 - no left recursion, no common prefixes, no ambiguity -> rewrite
- * Easy to write parser:
 - but sometimes difficult to structure grammar to be LL(1)
- * Error detection:
 - terminal on TOP not equal terminal on input
 - no table entry for [TOP, input]
- * Error Recovery:
 - panic mode:
 - skip input until input in FOLLOW(TOP) or FIRST(TOP)
 - pop terminal and pretend match
 - phrase level
 - error calls in table

Let's look at some grammars...

Example 1:

$$\begin{aligned} E &\rightarrow T E' \\ E' &\rightarrow + T E' \mid \varepsilon \\ T &\rightarrow F T \\ T' &\rightarrow * F T' \mid \varepsilon \\ F &\rightarrow (E) \mid id \end{aligned}$$

Example 2:

$$\begin{aligned} S &\rightarrow iEtSS' \mid a \\ S' &\rightarrow eS \mid \varepsilon \\ E &\rightarrow b \end{aligned}$$

Grammar Class Inclusion Tree

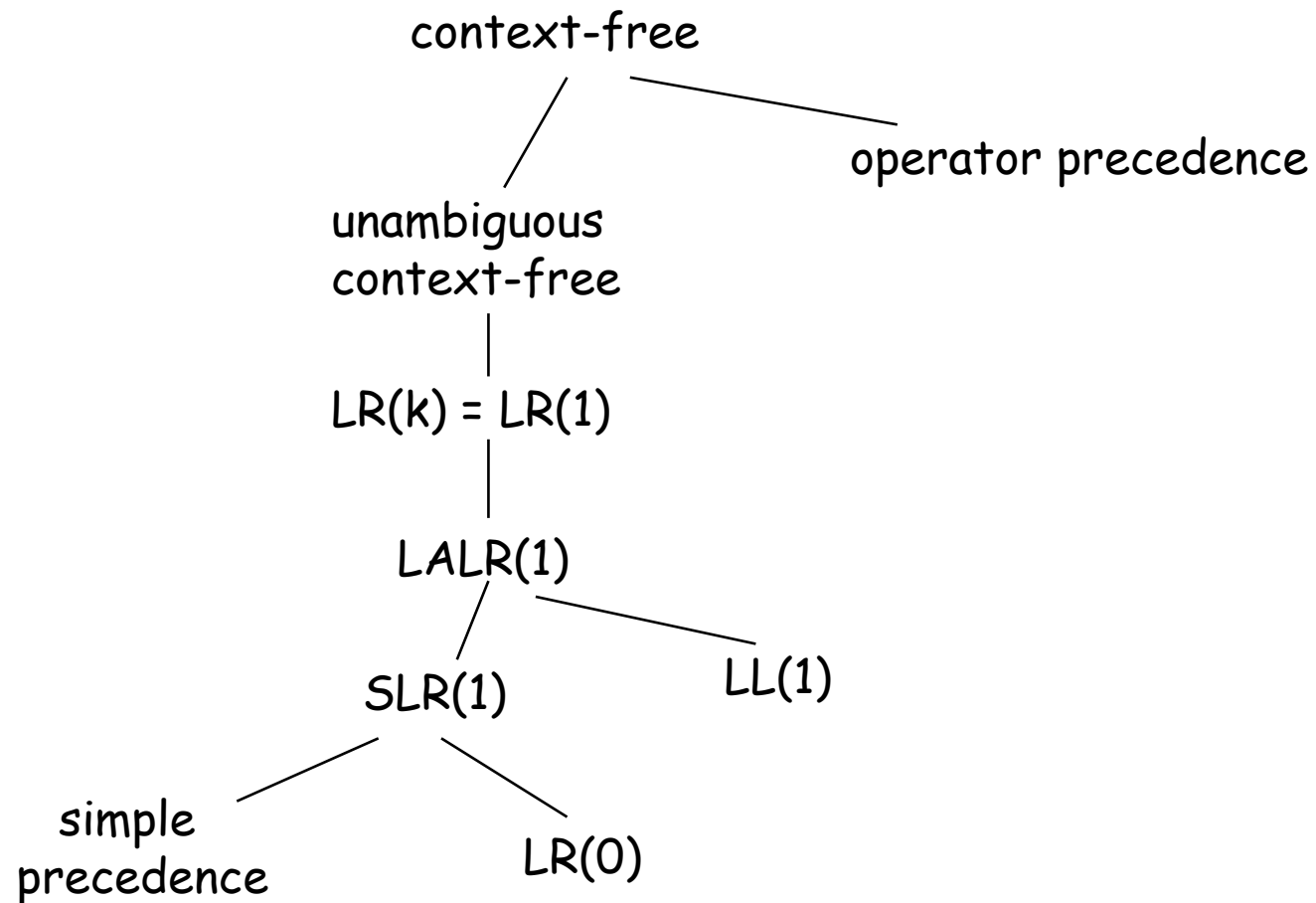
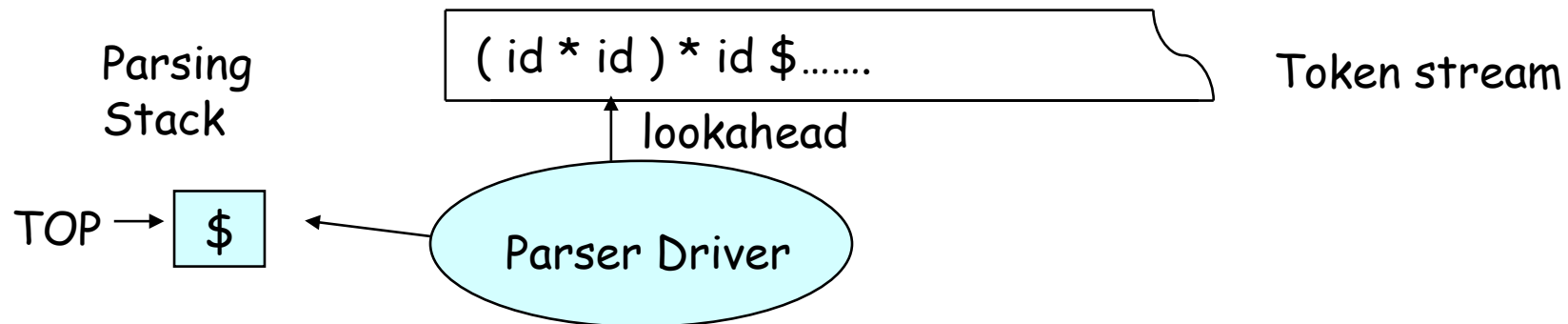


Table-driven Bottom-up Parsing



parse states	Action				Goto			
	(id)	*	\$	T	T'	F
0								
1								
2								

- Table[state,terminal] =
- shift token and state onto stack.
 - reduce by production $A \rightarrow \beta$
 pop rhs from stack; push A; push next state
 given by Goto[exposed state,A]
 - accept
 - error

LR Parsing Example

- 1: $P \rightarrow b S e$
- 2: $S \rightarrow a ; S$
- 3: $S \rightarrow b S e ; S$
- 4: $S \rightarrow \epsilon$

Parse Table

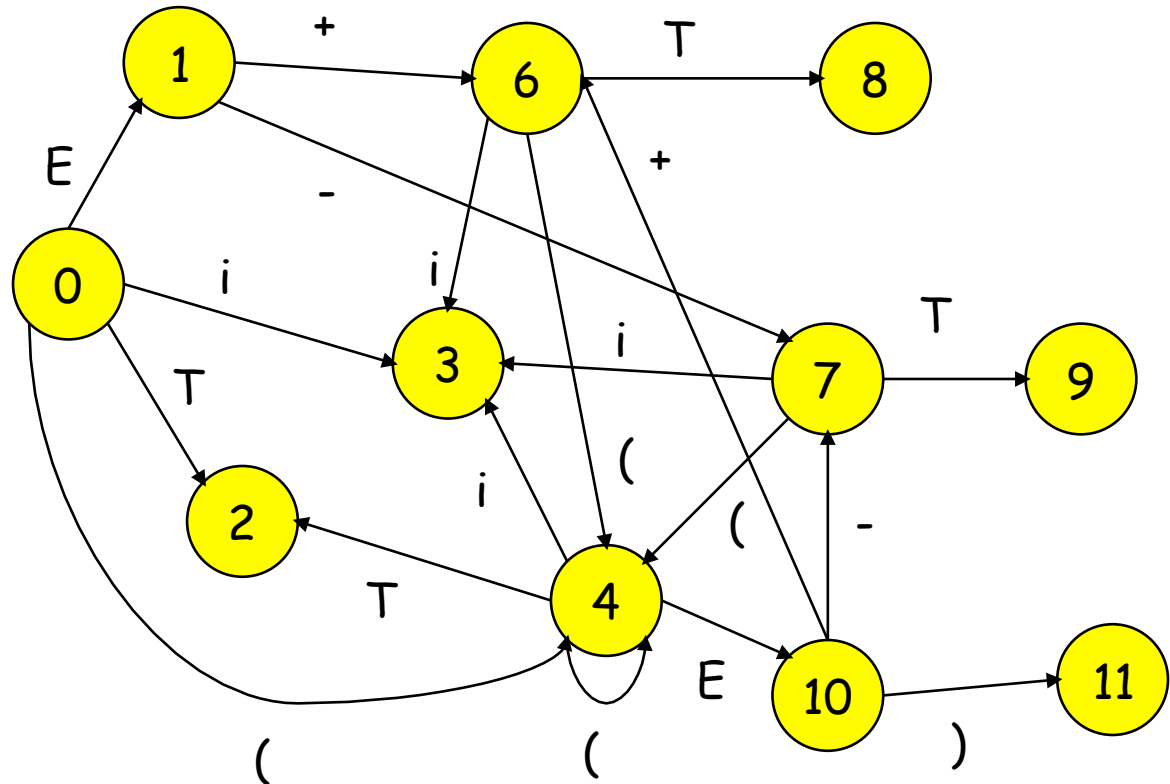
Stack	Input	state	b	e	a	;	\$	P	S
		0	s1						
0	ba;a;e\$	1	s4	r4	s5				2
0b1	a;a;e\$	2		s3					
0b1a5	;a;e\$	3					accept		
0b1a5;6	a;e\$	4	s4	r4	s5				7
0b1a5;6a5	;e\$	5				s6			
0b1a5;6a5;6	e\$	6	s4	r4	s5				10
0b1a5;6a5;6S10	e\$	7		s8					
0b1a5;6S10	e\$	8				s9			
0b1S2	e\$	9	s4	r4	s5				11
0b1S2e3	\$	10		r2					
accept!		11		r3					

DFA for parser

$S \rightarrow E$
 $E \rightarrow T \mid E + T \mid E - T$
 $T \rightarrow I \mid (E)$

Reduce States:

3: $T \rightarrow i$
 2: $E \rightarrow T$
 8: $E \rightarrow E + T$
 9: $E \rightarrow E - T$
 11: $T \rightarrow (E)$
 1: (on \$) $S \rightarrow E$



stack	input
0	i-(i+i)\$
0i3	-(i+i)\$
0T2	-(i+i)\$
...	

Semantic Actions during Parsing

```

S -> E           { $$ = $1; root = $$; }
E -> E + T       { $$ = makenode('+', $1, $3);} // E is $1, - is $2, T is $3
E -> E - T       { $$ = makenode('-', $1, $3);}
E -> T           { $$ = $1;} // $$ is top of stack
T -> ( E )       { $$ = $2;}
T -> id          { $$ = makeleaf('idnode', $1);}
T -> num         { $$ = makeleaf('numnode', $1);}
  
```

Consider parsing $4 + (x - y)$

num	S	4
	S	

state semantic value

Parsing Stack

Building LR(0) and SLR(1) Parse Tables

1. Augment grammar

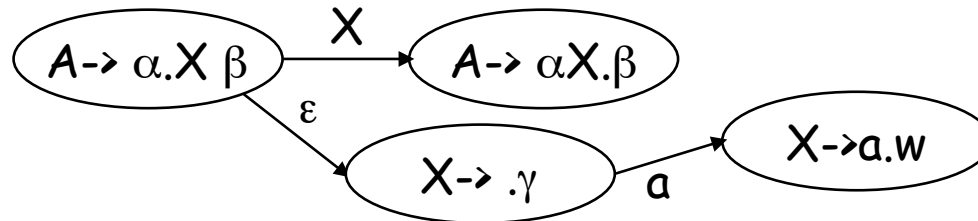
- Add a production $S' \rightarrow S$, where S is original start state
- Causes one ACCEPT table entry when reduce $S' \rightarrow S$ on \$.

2. Create DFA from grammar

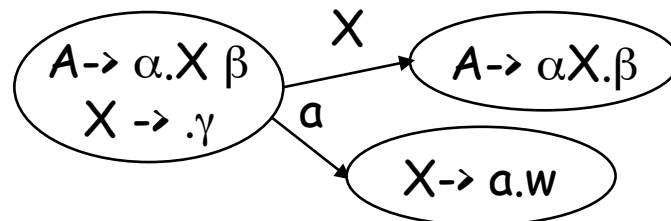
item $A \rightarrow \alpha . \beta$

- just seen a string derivable from α
- expect to see a string derivable from β

NFA : Each state represents a set of recognized viable prefixes
(kernel set of items)



DFA: Subset construction to go from NFA to DFA = closure(kernel)



LR(1) Parser (for same grammar)

