# 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 -> T * F | F
F -> ( E ) | id

=> T -> F T'
T' -> * F T' | ε
F -> ( E ) | id

```
T( )
{

}

T' ( )
{

}

F( )
{

}
```

Consider input string: a * b * c

# Table-driven Top Down Parsing

( id * id ) * id $.......

Token stream

lookahead

TOP → S $

Parser Driver

Parsing Stack

input symbols

nonterminals

| | ( | id | ) | * | $ |
|---|---|---|---|---|---|
| T | | | | | |
| T' | | | | | |
| F | | | | | |

Parsing Table[A,x] = production to apply for A on input x

TOP = lookahead = $ -> ACCEPT!
TOP = lookahead <> $ -> POP stack; ADVANCE lookahead;
TOP is terminal<> lookahead -> error
TOP is nonterminal -> Lookup(Table[TOP,lookahead])
For production X -> Y1 Y2 … Yn
POP X; PUSH Yn Yn-1… Y2 Y1

# Predictive Parser (Top Down)

Nonterminal Input Token

| | id | + | * | ( | ) | $ |
|---|---|---|---|---|---|---|
| E | E->TE' | | | E -> TE' | | |
| E' | | E' ->+TE' | | | E' ->ε | E' -> ε |
| T | T->FT' | | | T->FT' | | |
| T' | | T' -> ε | T' ->*FT' | | T' -> ε | T' -> ε |
| F | F->id | | | F->(E) | | |

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

Initial Stack:

| E |
|---|
| $ |

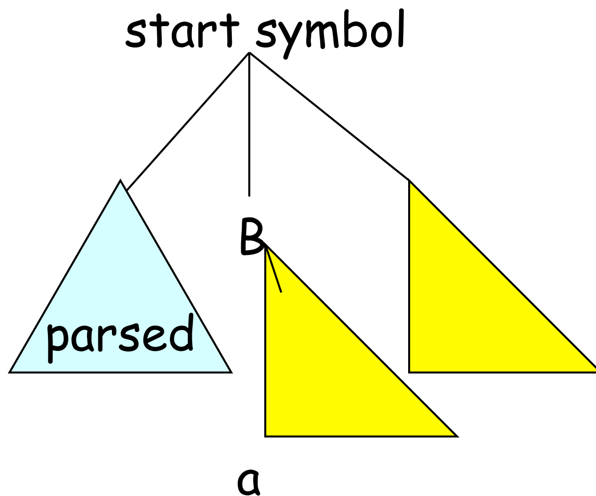
# Predictive LL(1) Parse Table Build

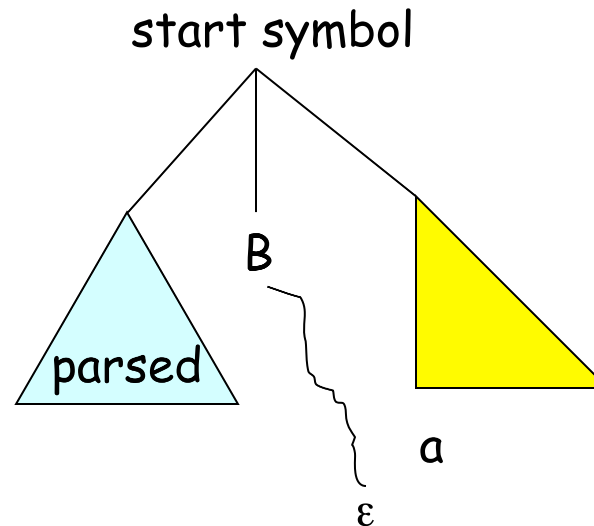
Key Insight:

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

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

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



# Computing FIRST and FOLLOW

FIRST($\alpha$) = set of terminals that can begin strings derived by $\alpha$

FIRST($\alpha$) = { a | $\alpha \overset{*}{\Rightarrow} a\beta$ for some $\beta$ }

E -> T E'
E' -> + T E' | $\varepsilon$
T -> FT'
T' -> *FT' | $\varepsilon$
F -> ( E ) | id

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

FOLLOW(N):

For A -> $\alpha$ N $\beta$, Add FIRST($\beta$), except $\varepsilon$, to FOLLOW(N)
For A -> $\alpha$ N $\beta$ and FIRST($\beta$) has $\varepsilon$, or A -> $\alpha$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 FIRST($\alpha$):

- If terminal a in FIRST($\alpha$) ==>

| | a |
|---|---|
| A | $A \rightarrow \alpha$ |

- If $\varepsilon$ in FIRST($\alpha$), then

  If terminal b in FOLLOW(A) ==>

| | b |
|---|---|
| A | $A \rightarrow \alpha$ |

  If $ in FOLLOW(A) ==>

| | $ |
|---|---|
| A | $A \rightarrow \alpha$ |

| 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:

* $FIRST(\alpha) \cap FIRST(\beta) = 0$

* At most 1 of $\alpha$ and $\beta$ derive the empty string

* If $\beta$ derives the empty string, then

  $FIRST(\alpha) \cap FOLLOW(A) = 0$

# 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…

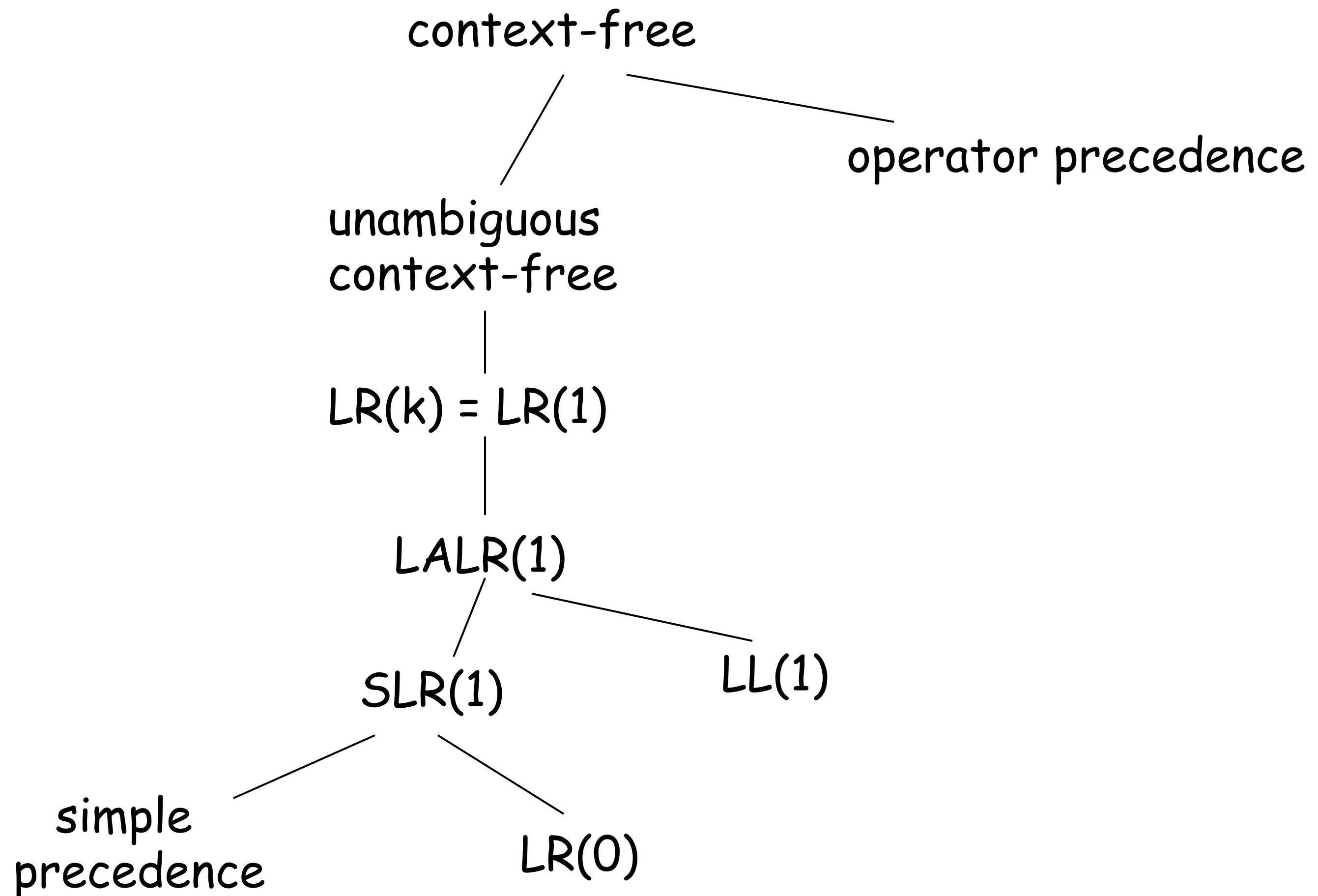
Example 1:

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

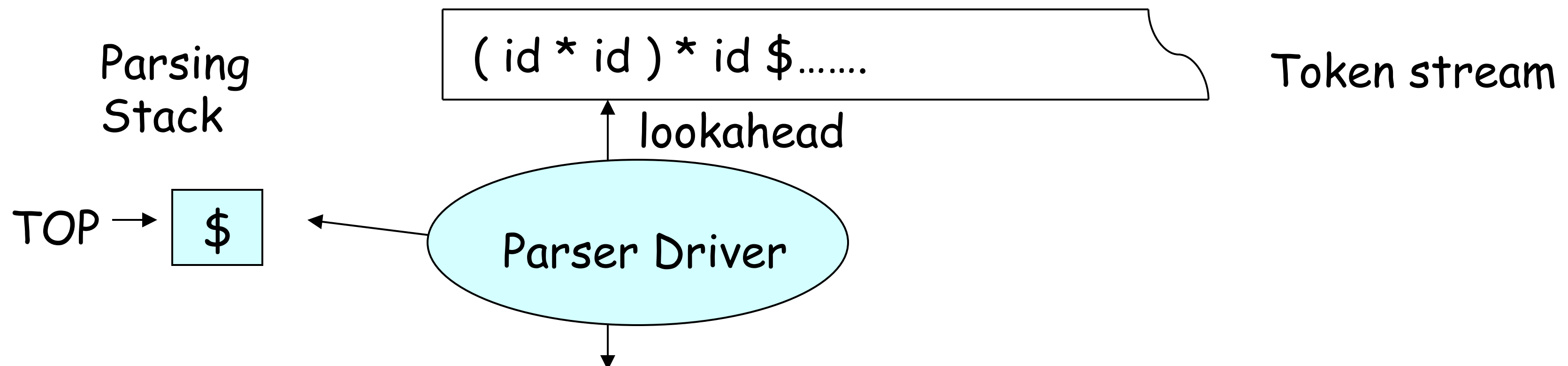
Example 2:

S -> iEtSS' | a
S' -> eS | ε
E -> b

# 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 -> $\beta$
  pop rhs from stack; push A; push next state
  given by Goto[exposed state,A]
- accept
- error

# LR Parsing Example

1: P -> b S e
2: S -> a ; S
3: S -> b S e ; S
4: S -> ε

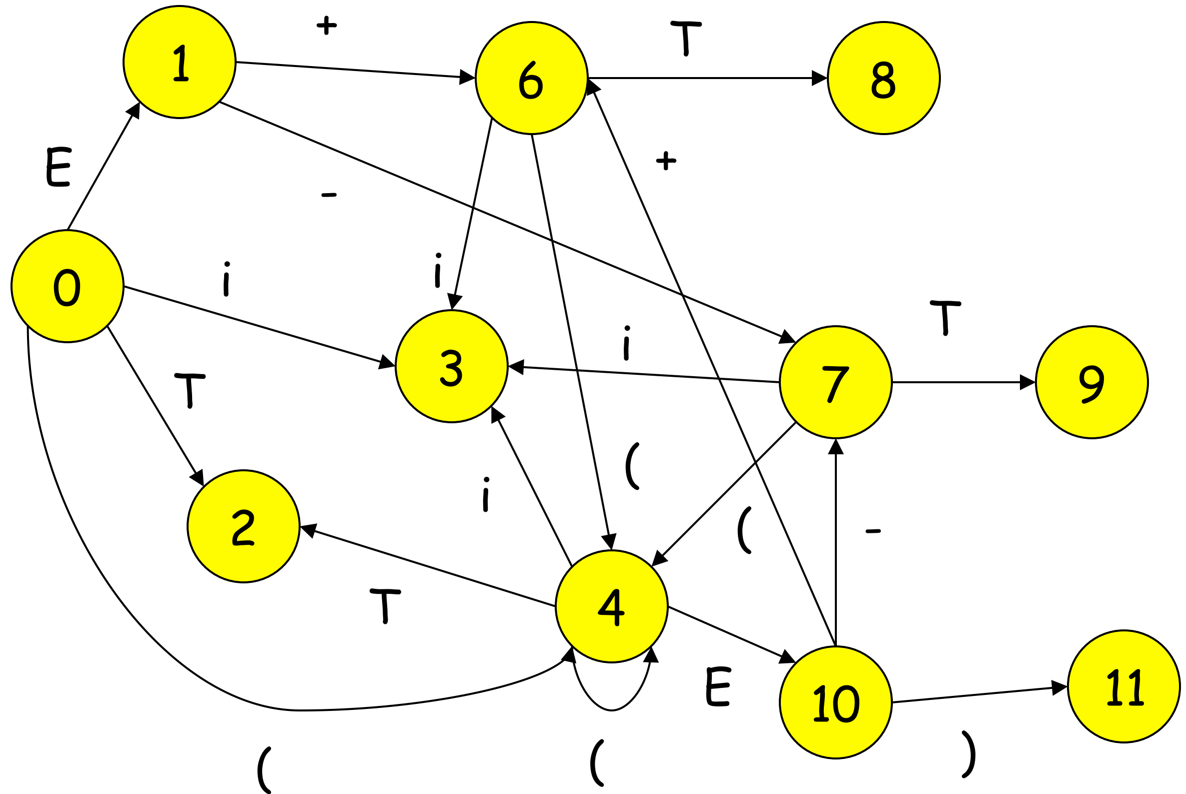
| Stack | Input |
|---|---|
| 0 | ba;a;e$ |
| 0b1 | a;a;e$ |
| 0b1a5 | ;a;e$ |
| 0b1a5;6 | a;e$ |
| 0b1a5;6a5 | ;e$ |
| 0b1a5;6a5;6 | e$ |
| 0b1a5;6a5;6S10 | e$ |
| 0b1a5;6S10 | e$ |
| 0b1S2 | e$ |
| 0b1S2e3 | $ |
| accept! | |

Parse Table

| state | b | e | a | ; | $ | P | S |
|---|---|---|---|---|---|---|---|
| 0 | s1 | | | | | | |
| 1 | s4 | r4 | s5 | | | | 2 |
| 2 | | s3 | | | | | |
| 3 | | | | | accept | | |
| 4 | s4 | r4 | s5 | | | | 7 |
| 5 | | | | s6 | | | |
| 6 | s4 | r4 | s5 | | | | 10 |
| 7 | | s8 | | | | | |
| 8 | | | | s9 | | | |
| 9 | s4 | r4 | s5 | | | | 11 |
| 10 | | r2 | | | | | |
| 11 | | r3 | | | | | |

# DFA for parser

S -> E
E -> T | E + T | E - T
T -> I | ( E )




Reduce States:

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

| | state | semantic value |
|---|---|---|
| | | |
| num | S | 4 |
| | S | |

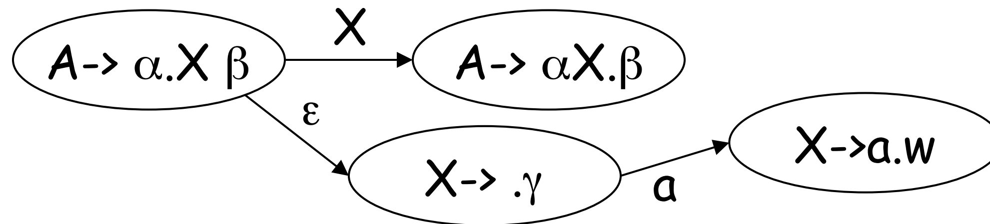
Parsing Stack

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

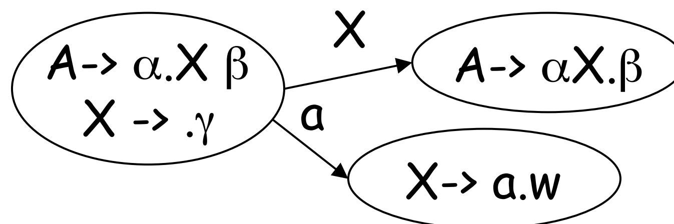
1. Augment grammar
   - Add a production S' -> S, where S is original start state
   - Causes one ACCEPT table entry when reduce S' ->S on $.
2. Create DFA from grammar

   item A -> $\alpha$ . $\beta$
   - just seen a string derivable from $\alpha$
   - expect to see a string derivable from $\beta$

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)