Class 11
Review Top-down Parsing

Quiz 3: Tuesday, Sept 22

1. 1 pt What part of this grammar will cause a problem For a top-down parser?

F -> F * T | T
T -> a | ε

2. 1 pt What will the symptoms be for the parser?
3. 2 pts How about this grammar? Problem? Symptoms?

F -> a F | c F | d
T -> a | b

3. 2 pt Why does recursive-descent parsing do a leftmost derivation?

Extra Credit: What does the acronym ACM expand to in our field?

Curious - Who looked this up from last week? Hint: no "of"

Top Down Parsing Challenges

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

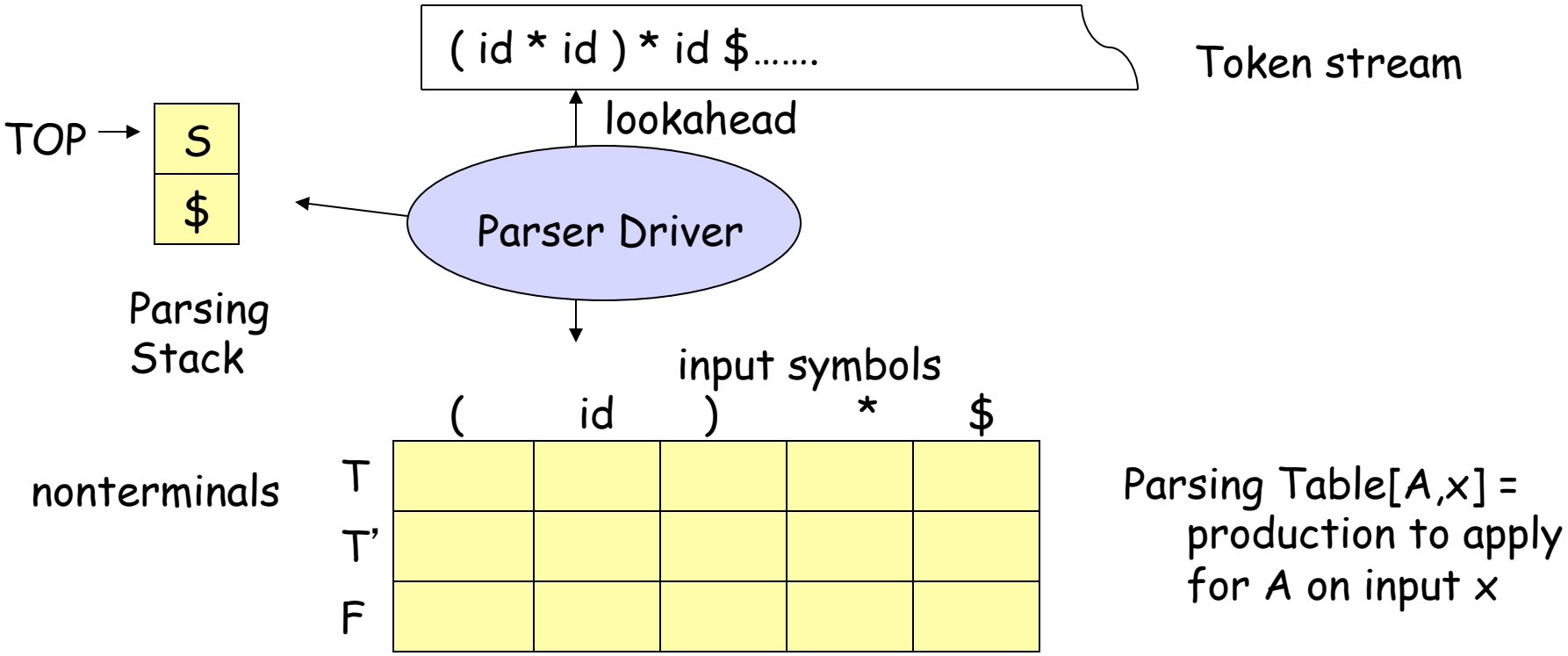
where param list => param : type; param : type;…param:type
param => var id, id, …, id
var is optional

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);

Table-driven Top Down Parsing



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
$

Any Questions?

- how the top-down parsing works?

- what you need to do to the grammar to use a top-down parser that is predictive (non-backtracking)

On to how to build that parse table…

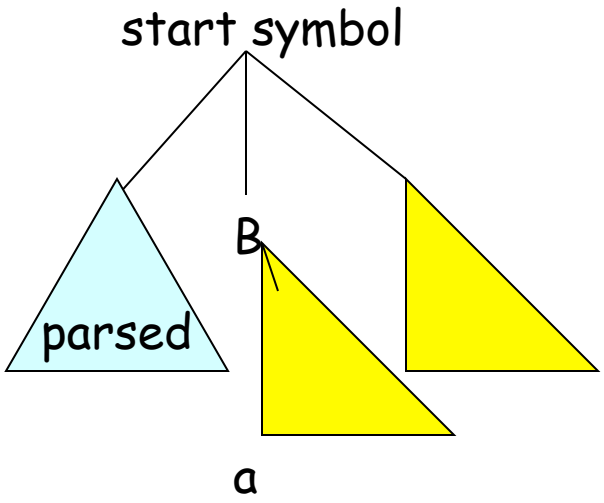
Predictive LL(1) Parse Table Build

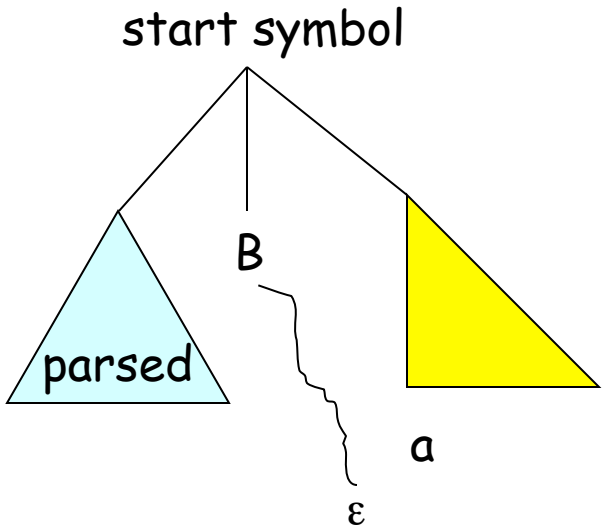
Key Insight:

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

FIRST(α) = { a | α $\Rightarrow^*$ aβ for some β }

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

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

FOLLOW(N):

For A -> α N β, Add FIRST(β), except ε, to FOLLOW(N)

For A -> α N β and FIRST(β) has ε, or A -> αN,
Add FOLLOW(A) to FOLLOW(N)

Add $ to FOLLOW(START SYMBOL)

Let's look at some grammars…

Example 1:

S -> ABC
A -> a | Cb | ε
B -> c | dA | ε
C -> e | f

Example 2:

S -> uBDz
B -> Bv | w
D -> E F
E -> y | ε
F -> x | ε

LL(1) Parse Table Construction

Look at each production, A -> α, and FIRST(α):

- If terminal a in FIRST(α) ==>

a
A | A -> α

- If ε in FIRST(α), then
  If terminal b in FOLLOW(A) ==>

b
A | A -> α

  If $ in FOLLOW(A) ==>

$
A | A -> α

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 -> α | β in G,
all of the following conditions hold true:

* FIRST(α) $\cap$FIRST(β) = 0

* At most 1 of α and β derive the empty string

* If β derives the empty string, then
  FIRST(α) $\cap$ 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