Programming Languages
Lecture05 – Top-Down Parsing

남 범 석

bnam@skku.edu

Parsing Techniques

- ***Top-down parsers***
  - *LL = **L**eft-to-right input scan, **L**eftmost derivation*
  - Start at the root of the parse tree and grow toward leaves
  - Pick a production & try to match the input
  - Bad "pick" ⇒ may need to backtrack
  - Some grammars are backtrack-free *(predictive parsing)*

- ***Bottom-up parsers***
  - *LR = **L**eft-to-right input scan, **R**ightmost derivation*
  - Start at the leaves and grow toward root
  - As input is consumed, encode possibilities in an internal state
  - Start in a state valid for legal first tokens
  - Bottom-up parsers handle a large class of grammars

Top-down Parser

- Problems in Top-down parser
  - Backtrack $\Rightarrow$ predictive parser
  - Left-recursion $\Rightarrow$ may result in infinite loop

- Predictive Top-down parser
  - LL(1)
    - 1 means "predict based on k tokens of lookahead"
  - Left factoring transforms some non-LL(1) to LL(1)

CFG with precedence

1	*Goal*	→	*Expr*
2	*Expr*	→	*Expr* + *Term*
3		|	*Expr* - *Term*
4		|	*Term*
5	*Term*	→	*Term* * *Factor*
6		|	*Term* / *Factor*
7		|	*Factor*
8	*Factor*	→	number
9		|	id

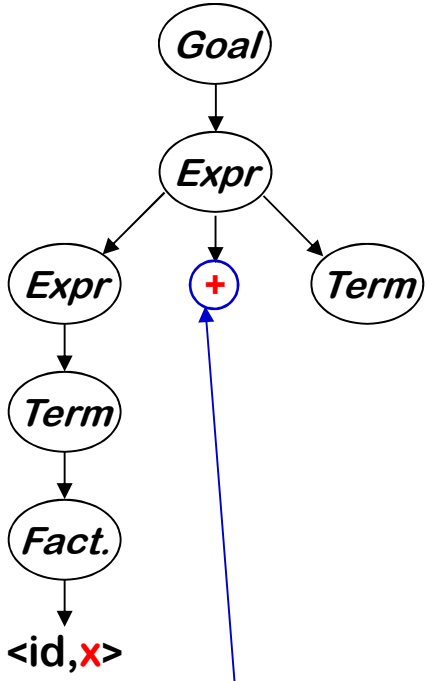
And the input x – 2 * y

Top-down Parser – backtrack (1)

- Let's try x - 2 * y :

Leftmost derivation, choose productions in an order that exposes problems

Rule	Sentential Form	Input
—	Goal	↑x - 2 * y
1	Expr	↑x - 2 * y
2	Expr + Term	↑x - 2 * y
4	Term + Term	↑x - 2 * y
7	Factor + Term	↑x - 2 * y
9	<id,x> + Term	↑x - 2 * y
—	<id,x> + Term	x ↑- 2 * y

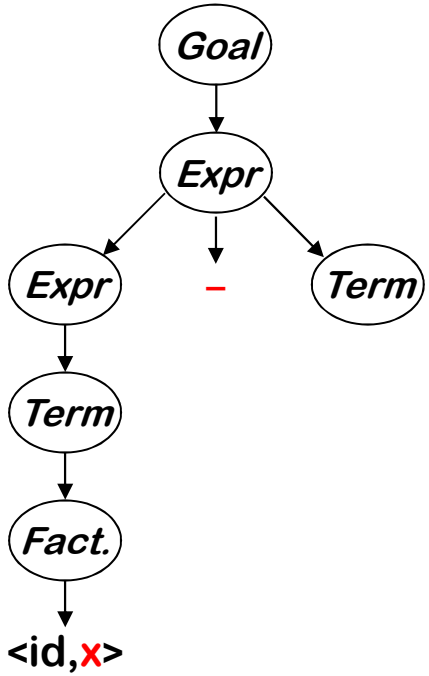


- This worked well, except that "-" doesn't match "+"
- The parser must backtrack to here

Top-down Parser – backtrack (2)

- Continuing with x – 2 * y :

Rule	Sentential Form	Input
—	*Goal*	↑x – 2 * y
1	*Expr*	↑x – 2 * y
3	*Expr* – *Term*	↑x – 2 * y
4	*Term* – *Term*	↑x – 2 * y
7	*Factor* – *Term*	↑x – 2 * y
9	<id,x> – *Term*	↑x – 2 * y
9	<id,x> – *Term*	x ↑– 2 * y
—	<id,x> – *Term*	x – ↑2 * y



This time, "–" and "–" matched

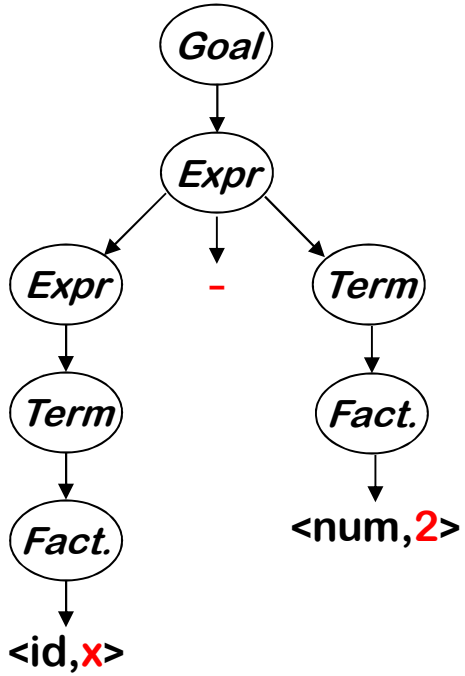
We can advance past "–" to look at "2"

⇒ Now, we need to expand *Term* - the last *NT* on the fringe

Top-down Parser – backtrack (3)

- Trying to match the "2" in x – 2 * y :

Rule	Sentential Form	Input
—	<id,x> – Term	x – ↑2 * y
7	<id,x> – Factor	x – ↑2 * y
9	<id,x> – <num,2>	x – ↑2 * y
—	<id,x> – <num,2>	x – 2 ↑* y



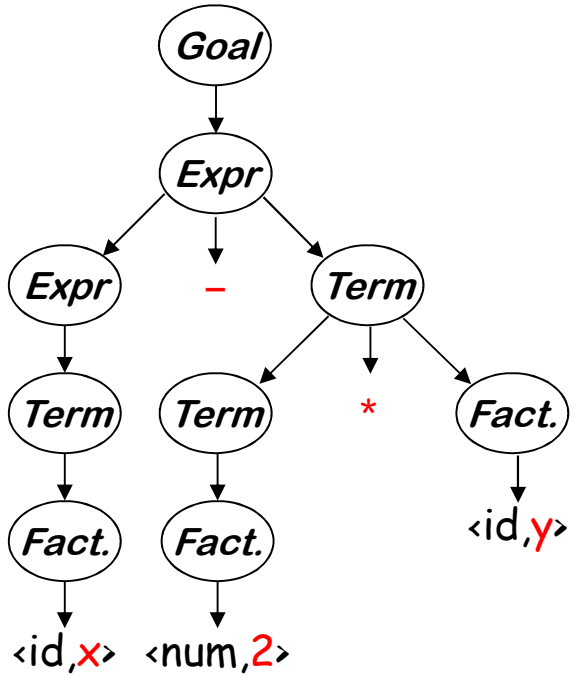
- Where are we?
  - "2" matches "2"
  - We have more input, but no *NT*s left to expand
  - The expansion terminated too soon

  ⇒Need to backtrack

Top-down Parser – backtrack (4)

- Trying again with "2" in x - 2 * y :

Rule	Sentential Form	Input
—	<id,x> - Term	x - ↑2 * y
5	<id,x> - Term * Factor	x - ↑2 * y
7	<id,x> - Factor * Factor	x - ↑2 * y
8	<id,x> - <num,2> * Factor	x - ↑2 * y
—	<id,x> - <num,2> * Factor	x - 2 ↑* y
—	<id,x> - <num,2> * Factor	x - 2 * ↑y
9	<id,x> - <num,2> * <id,y>	x - 2 * ↑y
—	<id,x> - <num,2> * <id,y>	x - 2 * y↑



- This time, we matched & consumed all the input

⇒Success!

Left Recursion

- Top-down parsers cannot handle left-recursive grammars
  - $S \rightarrow S\alpha \mid \beta$
  - Formally,

    A grammar is left recursive if $\exists\, A \in NT$ such that

    $\exists$ a derivation $A \Rightarrow^{+} A\alpha$, for some string $\alpha \in (NT \cup T)^{+}$

- Our expression grammar is left recursive
  - This can lead to non-termination in a top-down parser
  - We would like to convert the left recursion to right recursion

Eliminating Left Recursion

- Remove left recursion
  - Original grammar

    $F \rightarrow F\ \alpha$

    $\quad\quad\ \mid\ \beta$

    where neither α nor β starts with *F*
  - Rewrite the above as

    $F \rightarrow \beta\ P$

    $P \rightarrow \alpha\ P$

    $\quad\quad\ \mid\ \varepsilon$

    where *P* is a new non-terminal
  - *Accepts the same language, but uses only right recursion*

Eliminating Left Recursion

- The expression grammar contains two cases of left recursion

Expr	→	*Expr* + *Term*	*Term*	→	*Term* * *Factor*
	|	*Expr* - *Term*		|	*Term* / Factor
	|	*Term*		|	*Factor*

- Applying the transformation yields

Expr	→	*Term Expr′*	*Term*	→	*Factor Term′*
Expr′	→	+ *Term Expr′*	*Term′*	→	* *Factor Term′*
	|	- *Term Expr′*		|	/ *Factor Term′*
	|	ε		|	ε

  - These fragments use only right recursion
  - They retain the original left associativity (evaluate left to right)

Predictive Top-down Parsing

- Basic idea
  - Given A $\rightarrow \alpha \mid \beta$,
    the parser must be able to choose between α or β
  - First and Follow sets
  - Simple recursive descent parsers
  - Table-driven LL(1) parsers

- LL(1) Parser
  - L = scan input left to right
  - L = Leftmost derivation
  - 1 = lookahead is enough to pick right production rule to use
    - Given production rules
      $S \rightarrow \alpha \mid \beta$
      the parser should be able to choose between α or β using one lookahead
  - No Backtracking
  - No Left Recursion

First Sets

- For some rhs $\alpha \in G$,
  FIRST(α) is the set of terminals that appear as the first symbol in some string that derives from α
  - x $\in$ FIRST(α) iff $\alpha \Rightarrow^* x\,\gamma$, for some γ
  - Some number of derivations gets us x at the beginning

- Goal → SheepNoise
- SheepNoise → SheepNoise baa | baa

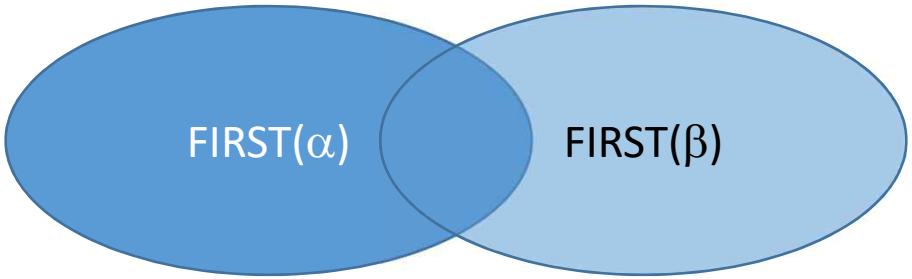
```
For SheepNoise:
      FIRST(Goal) = { baa }
      FIRST(SheepNoise) ={baa}
      FIRST(baa) = { baa }
```

The LL(1) Property (first version)

- If $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like

$$FIRST(\alpha) \cap FIRST(\beta) = \varnothing$$

- This would allow the parser to make a correct choice with a lookahead of exactly one symbol !



Does not have LL(1) Property

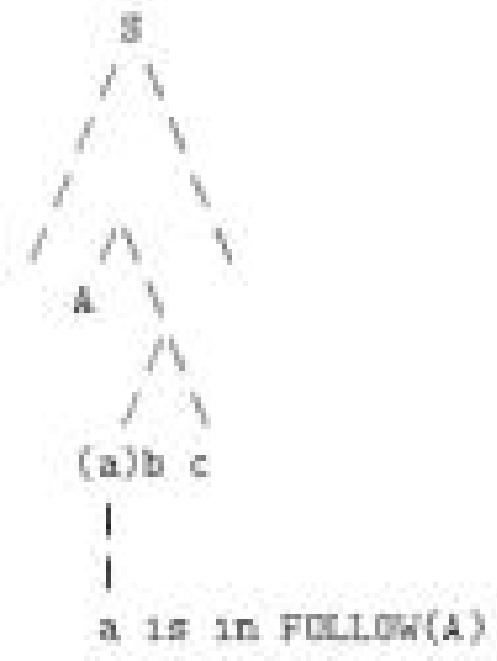
What about ε-productions?

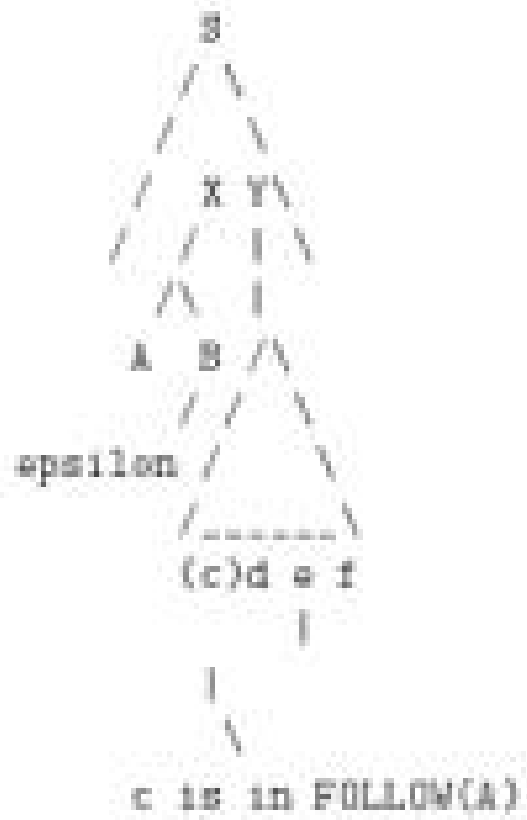
- If $A \rightarrow \alpha$ and $A \rightarrow \beta$ and $\varepsilon \in$ FIRST(α), then we need to ensure

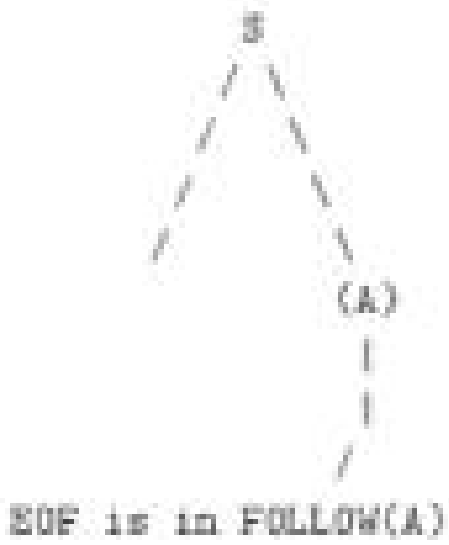
  $$\text{FIRST}(\beta) \cap \text{FOLLOW}(A) = \varnothing$$

  where,
  FOLLOW(A) = the set of terminal symbols that can immediately follow A in a sentential form
- Formally,
  FOLLOW(A) = {t | (t is a terminal and $G \Rightarrow^* \alpha A t \beta$)
  or (t is eof and $G \Rightarrow^* \alpha A$)}

Follow Sets Intuition







FIRST$^+$ sets

- Definition of FIRST$^+$(A $\rightarrow \alpha$)
  if $\varepsilon \in$ FIRST(α)

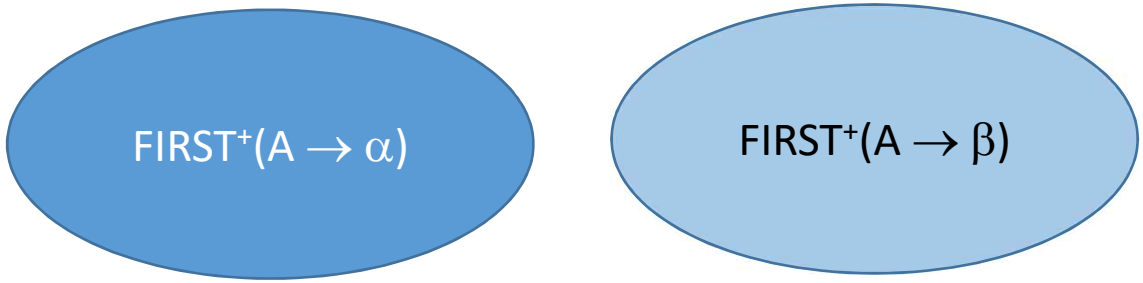
  $$\text{FIRST}^+(A \rightarrow \alpha) = \text{FIRST}(\alpha) \cup \text{FOLLOW}(A)$$

  else

  $$\text{FIRST}^+(A \rightarrow \alpha) = \text{FIRST}(\alpha)$$

- Grammar is LL(1) iff A $\rightarrow \alpha$ and A $\rightarrow \beta$ implies

  $$\text{FIRST}^+(A \rightarrow \alpha) \cap \text{FIRST}^+(A \rightarrow \beta) = \varnothing$$



Example:

Grammar

E→TE'

E'→+TE' | ε

T→ FT'

T'→*FT' | ε

F→(E) | id

First and Follow

Symbol	FIRST	FOLLOW
E	(,id	$,)
E’	+, ε	$,)
T	(,id	+,$,)
T’	*, ε	+,$,)
F	(,id	*,+,$,)

What if my grammar is not LL(1)?

- Can we transform a non-LL(1) grammar into an LL(1) grammar?
- In general, the answer is no
- In some cases, however, the answer is yes

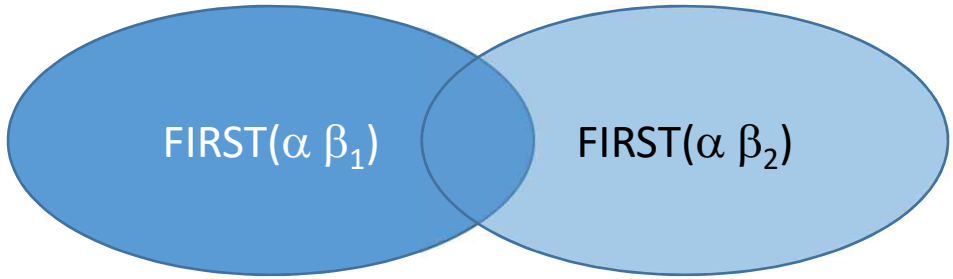
- Perform:
  - Eliminate left-recursion (Slide #10)
  - Perform left factoring

What if my grammar is not LL(1)?

- Given grammar G with productions

$$A \rightarrow \alpha\ \beta_1$$
$$A \rightarrow \alpha\ \beta_2$$

if α derives anything other than epsilon and
$FIRST+(A \rightarrow \alpha\ \beta_1) \cap FIRST+(A \rightarrow \alpha\ \beta_2) \neq \varnothing$



This grammar is not LL(1)

Left Factoring

- If we pull the common prefix, a, into a separate production, we may make the grammar LL(1).

$$A \rightarrow \alpha\, A'$$
$$A' \rightarrow \beta_1$$
$$\mid \beta_2$$

Now, if $FIRST^+(A' \rightarrow \beta_1) \cap FIRST^+(A' \rightarrow \beta_2) = \varnothing$, G may be LL(1)

Left Factoring

$\forall A \in NT$,

find the longest prefix α that occurs in two or more right-hand sides of A

if $\alpha \neq \varepsilon$ then replace all of the A productions,

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma \ ,$$

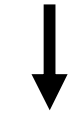
with

$$A \rightarrow \alpha\, Z \mid \gamma$$
$$Z \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

where Z is a new element of NT

Repeat until no common prefixes remain

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$$



$$A \rightarrow \alpha\, Z$$
$$Z \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$$

Left Factoring (An example)

- Consider the following fragment of the expression grammar

Factor	→	Identifier
	|	Identifier [*ExprList*]
	|	Identifier (*ExprList*)

FIRST⁺(rhs_1) = { Identifier }
FIRST⁺(rhs_2) = { Identifier }
FIRST⁺(rhs_3) = { Identifier }

- After left factoring, it becomes

Factor	→	**Identifier *Arguments***
Arguments	→	**[*ExprList*]**
	|	**(*ExprList*)**
	|	ε

FIRST⁺(rhs_1) = { [}
FIRST⁺(rhs_2) = { (}
FIRST⁺(rhs_3) = { ε } U FOLLOW(*Factor*)
⇒ It has the *LL(1)* property

- This form has the same syntax, with the *LL(1)* property

Left Recursion & Left Factoring (Generality)

Question

By *eliminating left recursion* and *left factoring*, can we transform an arbitrary CFG to a form where it meets the *LL(1)* condition? (and can be parsed predictively with a single token lookahead?)

Answer

Given a CFG that doesn't meet the *LL(1)* condition, it is undecidable whether or not an equivalent *LL(1)* grammar exists.

Language that Cannot Be LL(1)

Example that has no LL(1) grammar

$\{a^n\ 0\ b^n \mid n \geq 1\} \cup \{a^n\ 1\ b^{2n} \mid n \geq 1\}$

$G \rightarrow a\ A\ b$
$\quad\quad \mid a\ B\ bb$

$A \rightarrow a\ A\ b$
$\quad\quad \mid\ 0$

$B \rightarrow a\ B\ bb$
$\quad\quad \mid 1$

Problem: need an unbounded number of a characters before you can determine whether you are in the A group or the B group.

Automate Predictive Parsing

- Given a grammar that has the *LL(1)* property
  - Can write a simple routine to recognize each *lhs*
  - Code is both simple & fast
- Consider $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$, with
  - $\text{FIRST}^+(\beta_1) \cap \text{FIRST}^+(\beta_2) = \varnothing$
  - $\text{FIRST}^+(\beta_2) \cap \text{FIRST}^+(\beta_3) = \varnothing$
  - $\text{FIRST}^+(\beta_1) \cap \text{FIRST}^+(\beta_3) = \varnothing$

```
/* find an A */
if (current_token ∈ FIRST+(β1))
   find a β1 and return true
else if (current_token ∈ FIRST+(β2))
   find a β2 and return true
else if (current_token ∈ FIRST+(β3))
   find a β3 and return true
else
   report an error and return false
```

Grammars with the *LL(1)* property are called *predictive grammars* because the parser can "predict" the correct expansion at each point in the parse.

Parsers that capitalize on the *LL(1)* property are called *predictive parsers*.

One kind of predictive parser is the *recursive descent* parser.

Predictive Parsing Example

- Expression grammar, after transformation

1	*Goal*	→	*Expr*
2	*Expr*	→	*Term Expr′*
3	*Expr′*	→	+ *Term Expr′*
4		|	- *Term Expr′*
5		|	ε
6	*Term*	→	*Factor Term′*
7	*Term′*	→	* *Factor Term′*
8		|	/ *Factor Term′*
9		|	ε
10	*Factor*	→	id
11		|	number

This produces a parser with six *mutually recursive* routines:

- *Goal*
- *Expr*
- *EPrime*
- *Term*
- *TPrime*
- *Factor*

Each recognizes one *NT* or *T*

The term *descent* refers to the direction in which the parse tree is built.

Recursive Descent Parser

- A couple of routines from the expression parser

```
Goal( )
    token ← next_token( );
    if (Expr( ) = true & token = EOF)
       then next compilation step;
       else
           report syntax error;
           return false;

Expr( )
   if (Term( ) = false)
      then return false;
      else return Eprime( );
```

looking for EOF, found other token

```
Factor( )
   if (token = Number) then
       token ← next_token( );
       return true;
   else if (token = Identifier) then
        token ← next_token( );
        return true;
   else
       report syntax error;
       return false;
```

global variable

EPrime, Term, & TPrime follow the same basic lines

looking for Number or Identifier, found other token instead

Parse Tree - Recursive Descent Parser

- To build a parse tree:
  - Augment parsing routines to build nodes
  - Pass nodes between routines using a stack
  - Node for each symbol on *rhs*
  - Action is to pop *rhs* nodes, make them children of *lhs* node, and push this subtree
- To build an Abstract Syntax Tree (AST)
  - Non-terminals are not present in the AST
  - Build fewer nodes
  - Put them together in a different order

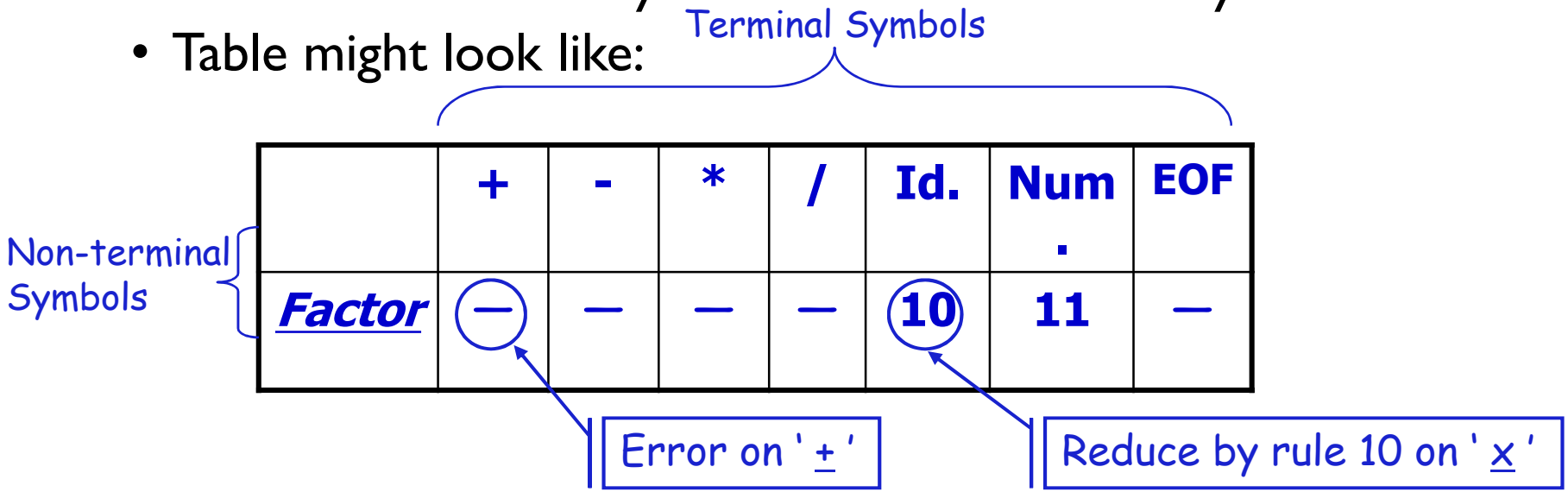
Expr → *Term Expr'*

```
Expr( )
    result ← true;
    if (Term( ) = false)
      then return false;
      else if (EPrime( ) = false)
              then result ← false;
              else
                 build an Expr node
                 pop EPrime node
                 pop Term node
                 make EPrime & Term
                    children of Expr
                 push Expr node
    return result;
```

Success ⇒ build a piece of the parse tree

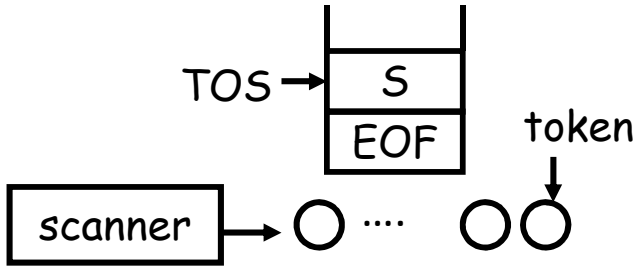
Building Table-driven Parser

- Strategy
  - Encode knowledge in a table
  - Use a standard "skeleton" parser to interpret the table
- Example
  - The non-terminal Factor has two expansions
    - Identifier or Number
  - Need a row for every NT & a column for every T
  - Table might look like:



LL(1) Skeleton Parser

```
token ← next_token()
push EOF onto Stack
push the start symbol, S, onto Stack
TOS ← top of Stack
loop forever
   if TOS = EOF and token = EOF then
     break & report success
   else if TOS is a terminal then
     if TOS matches token then
        pop Stack
        token ← next_token()
     else report error looking for TOS
   else
     if TABLE[TOS,token] is A→ B1B2…Bk then
        pop Stack
        push Bk, Bk-1, …, B1
     else report error expanding TOS
   TOS ← top of Stack
```



exit on success

// recognized TOS

// TOS is a non-terminal

// get rid of A
// in that order

Building LL(1) table

- Building the complete table for LL(1)
  - Need a row for every NT & a column for every T
  - Need an algorithm to build the table

- Filling in TABLE[X,y], $X \in NT$, $y \in T \cup \{EOF\}$
  - entry is the rule $X \rightarrow \beta$, if $y \in FIRST+(\beta)$
  - entry is error, otherwise

  - If any entry is defined multiple times, G is not LL(1)

Example:

Grammar

E→TE'

E'→+TE' | ε

T→ FT'

T'→*FT' | ε

F→(E) | id

Symbol	FIRST	FOLLOW
E	(,id	$,)
E’	+, ε	$,)
T	(,id	+,$,)
T’	*, ε	+,$,)
F	(,id	*,+,$,)

Building the table

	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)		

Input=id+id*id

Stack	Input buffer
$E	*id+id*id$*
$E'T	*Id+id*id$*
$E'T'F	*Id+id*id$*
$E'T'id	*Id+id*id$*
$E'T'	*+id*id$*
$E'	*+id*id$*
$E'T+	*+id*id$*
$E'T	*id*id$*

Input=id+id*id (Cont'd)

Stack	Input Buffer
$E'T'F	*id*id$*
$E'T'id	*id*id$*
$E'T'	**id$*
*$E'T'F**	**id$*
$E'T'F	*id$*
$E'T'id	*id$*
$E'T'	*$*
$E'	*$*
$	Accepted

Grammar that requires LL(k) parsing

Grammar:

S→iEtSS' | a

S'→eS | ε

E→b

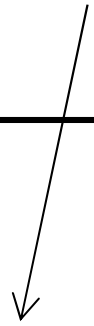
	FIRST	FOLLOW
S	a,i	e, $
S'	e, ε	e, $
E	b	t

Note that this is

If then else statement

Parse Table

	a	b	e	i	t	$
S	S→a			S→iEtSS′		
S′			S′→ε S′→eS			S′→ε
E		E→b				



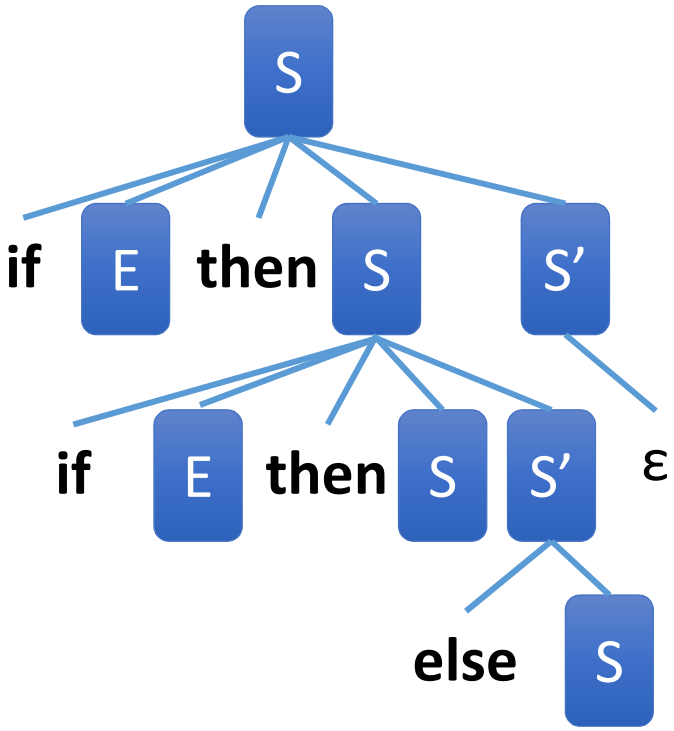
Ambiguity

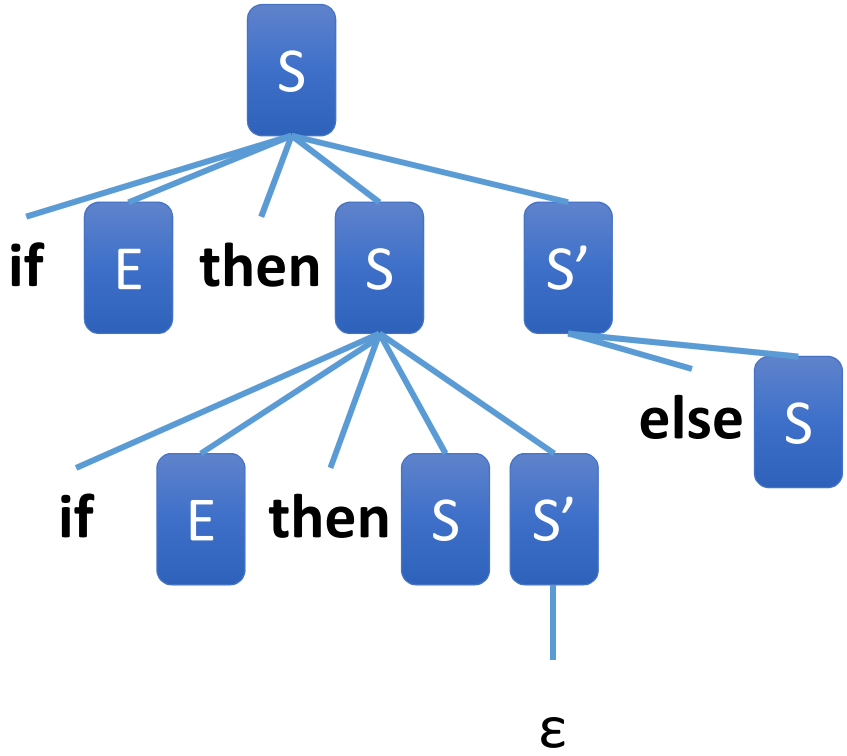
Ambiguous grammar

S→iEtSS' | a

S'→eS | ε

E→b





Ambiguous grammar and LL(2) Parsing

- The grammar is ambiguous and it is evident by the fact that we have two entries corresponding to M[S',e] containing S →ε and S' →eS. This ambiguity can be resolved if we choose S'→eS i.e associating the else's with the closest previous "then".

- Note that the ambiguity will be solved if we use LL(2) parser, i.e. always see for the two input symbols. How?
  - When input is 'e' then it looks at next input. Depending on the next input we choose the appropriate rule.

S→iES' | a
S'→tS | tSeS
E→b

Summary

- Top-down parser
  - Use leftmost derivation
  - Bad pick of rewrite rule results in backtrack
- Left recursion removal
  - Avoid non-terminating top-down parser
- Predictive parsing
  - LL(1) property ensures only one production rule is chosen by looking ahead one terminal symbol.
- Left factoring
  - Transform some non-LL(1) to LL(1)
- Automatic top-down parser generation
  - Recursive decent parser
  - Building LL(1) table: $f(X,y) \rightarrow P$ (where $X \in NT$, $y \in T$)