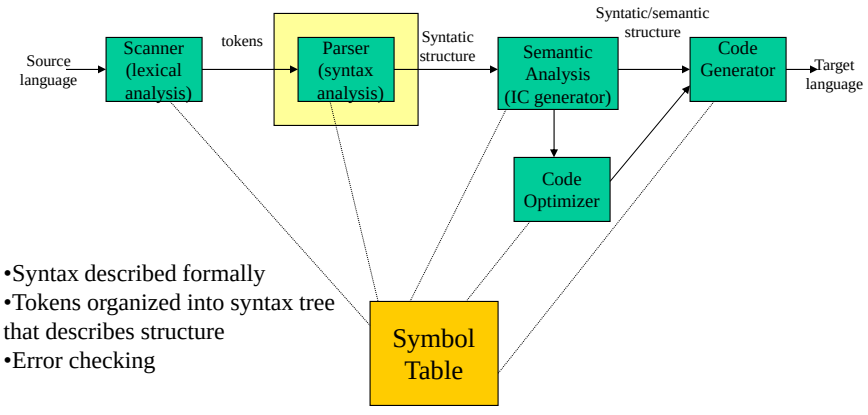
Lecture 4a: Parsing

COSC 4316

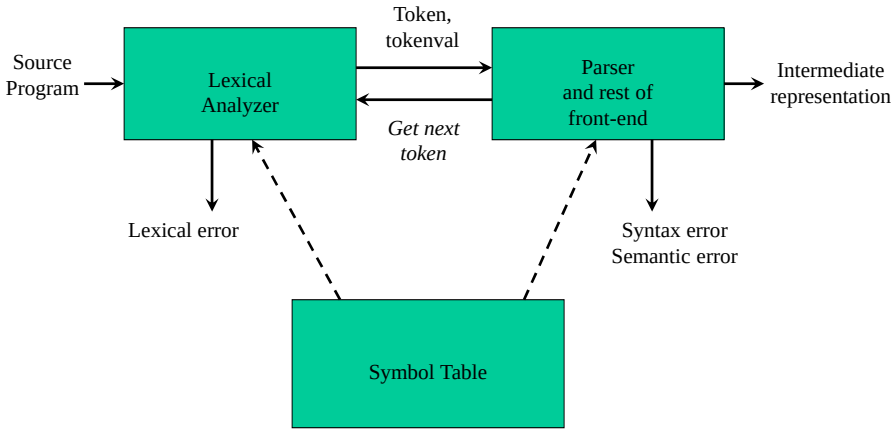
(grateful acknowledgement to Robert van Engelen and Elizabeth White for some of the material from which these slides have been adapted)

Syntax Analysis - Parsing



- Syntax described formally
- Tokens organized into syntax tree that describes structure
- Error checking

Position of a Parser in the Compiler Model



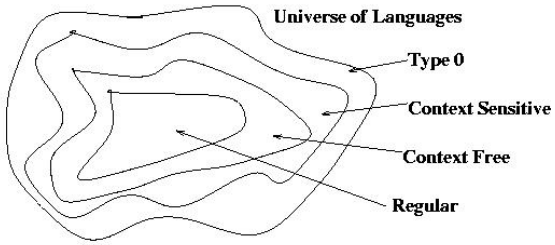
3

Syntax Described by CF Grammars (BNF)

- Advantages of CFGs
  - Precise, easily understood syntactic specification of a language
  - Automatic construction of parsers
  - Makes syntactic ambiguities more obvious
  - Allows extension of language to be done more readily

Static Analysis - Parsing

We can use context free grammars to specify the syntax of programming languages.



Role of the Parser

- Obtain a string of tokens from the lexical analyzer
- Verify that the string can be generated by the grammar for the source language
- Report syntax errors (intelligibly)
- Recover from some syntax errors

3 Types of Parsers

- Universal parsing algorithms
  - Examples: CYK algorithm or Earley's algorithm
    - May be used for any CF grammar
    - Too inefficient for practical use
- Top down parsers (often constructed by hand)
  - Build parse trees from the root down to the leaves
  - Works with certain classes of grammars (e.g. LL grammars – *later*)
- Bottom up parsers (often build with automated tools)
  - Build parsers from the leaves up to the root
  - Work with a broader class of grammars (LR)

Error Handling

- Errors **will** occur
- A good compiler should assist in identifying and locating errors
  - *Lexical errors*: important, compiler can easily recover and continue
  - *Syntax errors*: most important for compiler, can almost always recover
  - *Static semantic errors*: important, can sometimes recover
  - Goal: detection of an error *as soon as possible* without further consuming unnecessary input
  - How: detect an error as soon as the prefix of the input does not match a prefix of any string in the language (the *viable prefix property*)

Error Recovery Strategies

- *Bail out* (sudden death)
  - Stop when first error found
- *Panic mode*
  - Discard input until a token in a set of designated *synchronizing tokens* is found (e.g., a semicolon or a **}**)
- *Phrase-level recovery*
  - Perform local correction on the input to repair the error
  - Easy to do in predictive parsing because you know what is expected (**match**)
- *Error productions*
  - Augment grammar with productions for erroneous constructs
- *Global correction*
  - Choose a minimal sequence of changes to obtain a global least-cost correction

9

Context-Free Grammars (Recap)

- Context-free grammar is a 4-tuple $G = (N, T, P, S)$ where
  - T is a finite set of tokens (*terminal* symbols)
  - N is a finite set of *nonterminals*
  - P is a finite set of *productions* of the form
    $A \rightarrow \beta$
    where $A \in N$ and $\beta \in (N \cup T)^*$
  - $S \in N$ is a designated *start symbol*

10

Notational Conventions Used

- Terminals
  a,b,c,… $\in T$ (lowercase letters early in the alphabet)
  specific terminals: **0**, **1**, **id**, **+**
- Nonterminals (uppercase letters early in the alphabet)
  $A,B,C,\ldots \in N$
  specific nonterminals: *expr*, *term*, *stmt*
- Grammar symbols (uppercase letters late in the alphabet)
  $X,Y,Z \in (N \cup T)$
- Strings of terminals (lowercase letters late in the alphabet)
  $u,v,w,x,y,z \in T^*$
- Strings of grammar symbols (Greek letters)
  $\alpha,\beta,\gamma \in (N \cup T)^*$

11

Derivations

- The *one-step derivation* is defined by
  $\alpha A \beta \Rightarrow \alpha \gamma \beta$
  where $A \rightarrow \gamma$ is a production in the grammar
- In addition, we define
  - $\Rightarrow$ is *leftmost* $\Rightarrow_{lm}$ if α does not contain a nonterminal
  - $\Rightarrow$ is *rightmost* $\Rightarrow_{rm}$ if β does not contain a nonterminal
  - Transitive closure $\Rightarrow^*$ (zero or more steps)
  - Positive closure $\Rightarrow^+$ (one or more steps)

12

Derivations

- A derivation is an alternative to constructing a parse tree.
- We view a production as a *rewriting rule*.
- The sequence of replacements is called a *derivation.*
- If $S \Rightarrow^* \alpha$, then α is said to be a *sentential form* of the CFG, G
- The *language generated by G* is defined by
  $L(G) = \{w \in T^* \mid S \Rightarrow^+ w\}$
  and w is called a sentence in L(G)
  (a sentential form with no non-terminals)

13

Derivations

- When deriving a token sequence, if more than one nonterminal is present, we have a choice of which to replace next.
- One convention:
  - Leftmost derivation –
    - Choose the leftmost possible nonterminal at each step.
    - $\Rightarrow_{lm} \quad \Rightarrow^*_{lm} \quad \Rightarrow^+_{lm}$
- A sentential form produced via a leftmost derivation is called a ***left sentential form***

14

Derivation (Example)

S → P (S) | **var** R

P → **func** | ε

R → + S | ε

Expressions of variables and functions

- A leftmost derivation of **func** (**var** + **var**) is

  S $\Rightarrow_{lm}$ P (S) $\Rightarrow_{lm}$ **func** (S) $\Rightarrow_{lm}$ **func** (**var** R)
  $\Rightarrow_{lm}$ **func** (**var** + S) $\Rightarrow_{lm}$ **func** (**var** + **var** R)
  $\Rightarrow_{lm}$ **func** (**var** + **var**)

15

Derivations

- Analogous to leftmost derivations, we have ***rightmost derivations***.
- These seem less intuitive, but they correspond to a large class of parsers (the *bottom-up parsers*.)
- Leftmost derivations are usually associated with top-down parsing.
- Rightmost derivations are sometimes called ***canonical derivations***.

16

CFG Examples

Indicates a production

T = {+,-,0..9}, N = {L,D}, S = L

L → L + D | L – D | D

D → 0 | … | 9

Shorthand for multiple productions

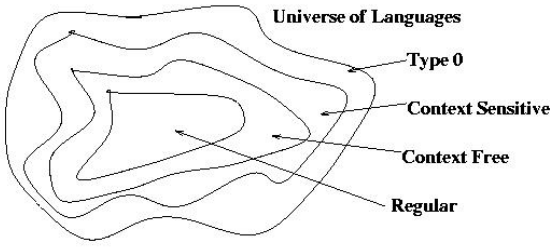
T={ (,) }, N = {L}, S = L

L → (L) L

L → ε

Languages

Regular	A → a B, C → ε
Context free	A → α
Context sensitive	αAβ → αγβ
Type 0	α → β



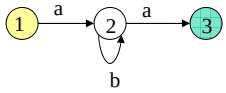
Any regular language can be expressed using a CFG

Starting with a NFA:

- For each state S_i in the NFA
  - Create non-terminal A_i
  - If transition $(S_i,a) = S_k$, create production $A_i \rightarrow a\ A_k$
  - If transition $(S_i,\varepsilon) = S_k$, create production $A_i \rightarrow A_k$
  - If S_i is a final state, create production $A_i \rightarrow \varepsilon$
  - If S_i is the NFA start state, $s = A_i$
- *What does the existence of this algorithm tell us about the relationship between regular and context free languages?*

NFA to CFG Example

ab*a



$A_1 \rightarrow a\ A_2$
$A_2 \rightarrow b\ A_2$
$A_2 \rightarrow a\ A_3$
$A_3 \rightarrow \varepsilon$

Writing Grammars

When writing a grammar (or RE) for some language, the following must be true:

1. All strings generated are in the language.
2. Your grammar produces all strings in the language.

Try these:

- Integers divisible by 2
- Legal postfix expressions
- Floating point numbers with no extra zeros
- Strings of 0,1 where there are more 0 than 1 (hard)

Regular Expressions vs. CFGs

- A grammar in which every production is of the form:

  $A \rightarrow w\ B$ ($w, x \in T^*$ and $A,B \in N$)

  or $A \rightarrow x$

  is called a ***right-linear grammar***. (*Left-linear grammar* defined analogously.)
- It can be proven that any right-linear grammar generates a regular language, and *vice versa*.

Regular Expressions vs. CFGs

- Why use regular expressions to denote the lexical syntax of a language if we could use CFGs instead?
  - Lexical rules are simple – don't use a chainsaw to prune a rose
  - Regular expressions are more concise and easier to understand than CFGs
  - Easier to generate a lexical analyzer from a regular expression than an ***arbitrary*** grammar.
  - Promotes modularity of the front end.

Parsing

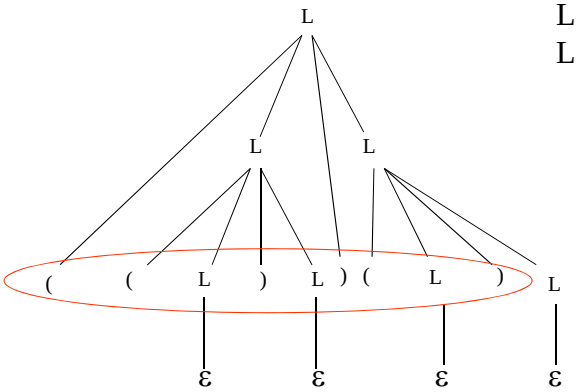
- The task of parsing is figuring out what the parse tree looks like for a given input and language.
- If a string is in the given language, a parse tree must exist.
- However, just because a parse tree exists for some string in a given language doesn't mean a given algorithm can find it.

Parse Trees

The parse tree for some string in a language that is defined by the grammar G as follows:

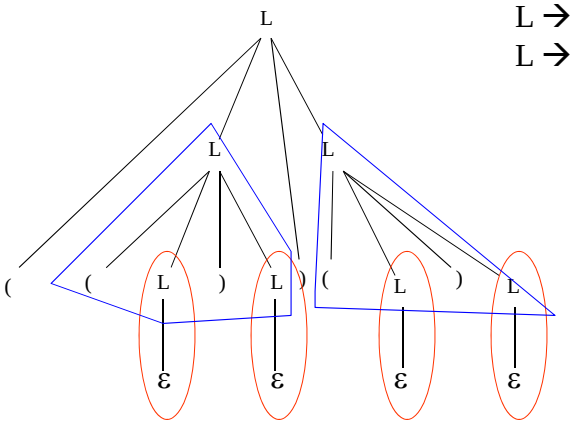
- The root is the start symbol of G
- The leaves are terminals or ε. When visited from left to right, the leaves form the input string
- The interior nodes are non-terminals of G
- For every non-terminal A in the tree with children $B_1 \ldots B_k$, there is some production $A \rightarrow B_1 \ldots B_k$

Parse Tree for (())()



L → (L) L
L → ε

Parse Tree for (())()



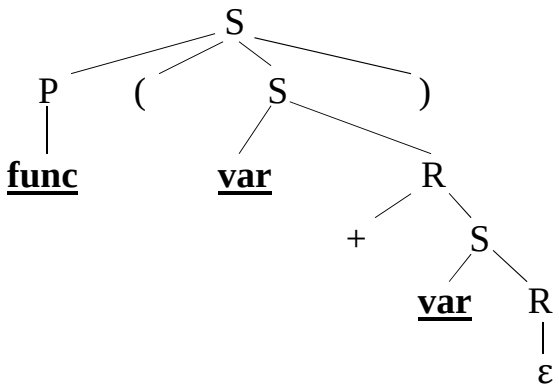
L → (L) L
L → ε

Parse Trees & Derivations

- A derivation is a *linear representation* of a parse tree.
- Equivalently, a parse tree is a *graphical representation* of a derivation.

For the grammar and the string **func** (**var** + **var**):
S → P (S) | **var** R
P → **func** | ε
R → + S | ε

S ⇒$_{lm}$ P (S) ⇒$_{lm}$ **func** (S) ⇒$_{lm}$ **func** (**var** R)
⇒$_{lm}$ **func** (**var** + S)
⇒$_{lm}$ **func** (**var** + **var** R)
⇒$_{lm}$ **func** (**var** + **var**)



Single Step Derivation

Definition: Given $\underline{\alpha\ A\ \beta}$ (with α,β in $(V_n \cup V_t)^*$) and a production $\underline{A \rightarrow \gamma}$,

$\alpha\ A\ \beta \Rightarrow \alpha\ \gamma\ \beta$ is a **single step derivation**.

Examples:

L + D ⇒ L – D + D	L → L - D
(L) (L) ⇒ ((L) L) (L)	L → (L) L

Greek letters (α,β,χ,…) denote a (possibly empty) sequence of terminals and non-terminals.

Derivations

Definition: A sequence of the form:

$w_0 \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n$

is a **derivation** of w_n from w_0 ($w_0 \Rightarrow^* w_n$)

L	production L → (L) L
⇒ (L) L	production L → ε
⇒ ( ) L	production L → ε
⇒ ( )	

$L \Rightarrow^* ()$

If w_i has non-terminal symbols, it is referred to as *sentential form*.

$L \Rightarrow^* (( )) ( )$

L	production L → (L) L
⇒ (L) **L**	production L → (L) L
⇒ (L) (L) **L**	production L → ε
⇒ (**L**) (L)	production L → (L) L
⇒ ((**L**) L) (L)	production L → ε
⇒ (( ) L) (**L**)	production L → ε
⇒ (( ) **L**) ( )	production L → ε
⇒ (( )) ( )	

- L(G), the language generated by grammar G is
  {w in T^*: S $\Rightarrow^*$ w, for start symbol S}
- Both **( )** and **(( )) ( )** are in L(G) for the following grammar.
  - L → **(** L **)** L
  - L → ε

Leftmost Derivations

- Recall that a leftmost derivation is one where the leftmost nonterminal is always chosen
- If a string is in a given language (i.e. a derivation exists), then a leftmost derivation *must* exist
- Rightmost derivation defined as you would expect

Leftmost Derivation for (())()

L	production L → (L) L
⇒ (**L**) L	production L → (L) L
⇒ ((**L**) L) L	production L → ε
⇒ (( ) **L**) L	production L → ε
⇒ (( )) **L**	production L →(L) L
⇒ (( )) (**L**) L	production L → ε
⇒ (( )) ( ) **L**	production L → ε
⇒ (( )) ( )	

L → **(** L **)** L
L → ε

Rightmost Derivation for (())()

L	production L → (L) L
⇒ (L) **L**	production L → (L) L
⇒ (L) (L) **L**	production L → ε
⇒ (L) (**L**)	production L → ε
⇒ (**L**) ( )	production L → (L) L
⇒ ((L) **L**) ( )	production L → ε
⇒ ((**L**)) ( )	production L → ε
⇒ (( )) ( )	

L → **(** L **)** L
L → ε

Ambiguity

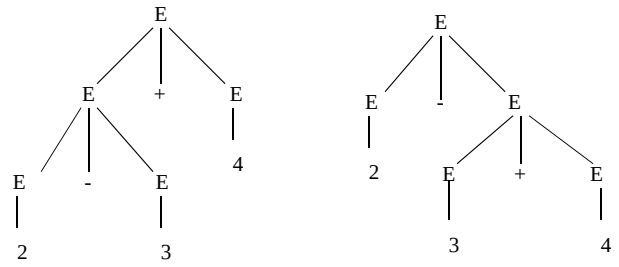
- An ***ambiguous*** grammar is one in which two (or more) parse trees or leftmost derivations exist for *some string in the language*

E → E + E

E → E – E

E → 0 | … | 9

2 – 3 + 4



- Two leftmost derivations

E	⇒	E + E	E	⇒	E – E
	⇒	E – E + E		⇒	2 – E
	⇒	2 – E + E		⇒	2 – E + E
	⇒	2 – 3 + E		⇒	2 – 3 + E
	⇒	2 – 3 + 4		⇒	2 – 3 + 4

- We must either write unambiguous grammars **or** have *disambiguating* rules.

- An ambiguous grammar can sometimes be made unambiguous:

  E → E + T | E – T | T

  T → 0 | … | 9

  enforces the correct associativity

- Precedence can be specified as well:

  E → E + T | E – T | T

  T → T * F | T / F | F

  F → (E) | 0 | …| 9

Another example of ambiguity

- **if-else** in Java

```
if (expr)
    if (expr)
      stmt;
    else
       stmt;
```

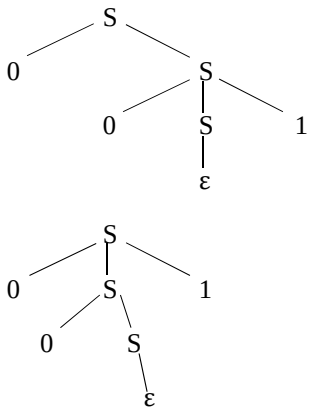
- Which **if** is the **else** associated with?
- The last one, but we can't specify that via the definition.
- Could fix this by requiring the use of an **endif** keyword

Yet another example of ambiguity

- **L = { $0^i 1^j$ | $i \geq j \geq 0$}** (0 ≡ if, 1 ≡ else)
- Expressed by the grammar

  S → 0 S | 0 S 1 | ε

Parse trees for **001**

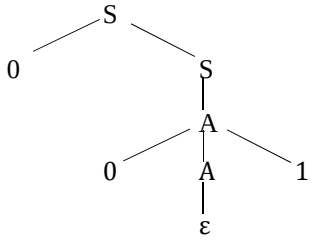


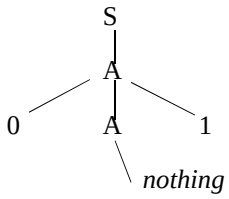
Disambiguating rule: match each 1 with the closest unmatched 0

Disambiguating rule incorporated into the grammar

- S → 0 S | A
- A → 0 A 1 | ε

Parse trees for **001**





Disambiguating rule: match each 1 with the closest unmatched 0

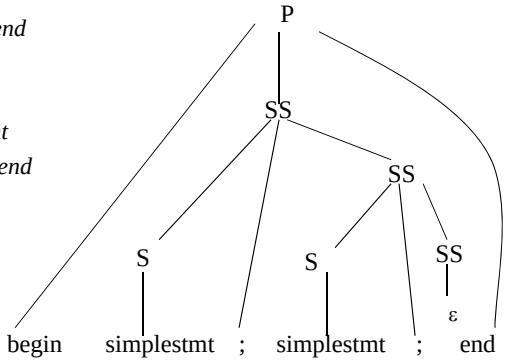
Grammar still has issues, because we can't use a predictive parser – cant predict whether the 0 is matched or unmatched.

The dangling else is really a language design issue.

Conclusion: If you ever design a programming language, you need to know the issues involved in parsing that language!

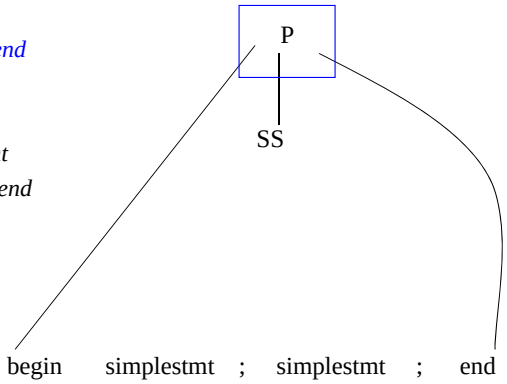
Input: **`begin simplestmt; simplestmt; end`**

P → *begin* SS *end*
SS → S *;* SS
SS → ε
S → *simplestmt*
S → *begin* SS *end*



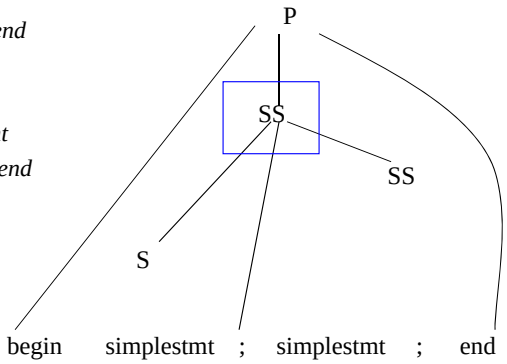
Top Down (LL) Parsing

P → *begin* SS *end*
SS → S *;* SS
SS → ε
S → *simplestmt*
S → *begin* SS *end*



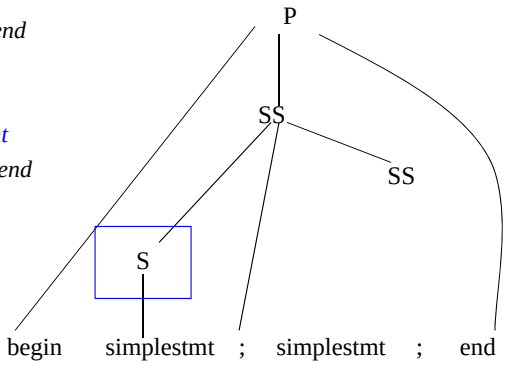
Top Down (LL) Parsing

P → *begin* SS *end*
SS → S *;* SS
SS → ε
S → *simplestmt*
S → *begin* SS *end*



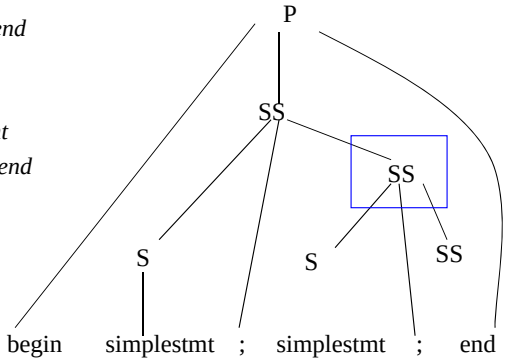
Top Down (LL) Parsing

P → *begin* SS *end*
SS → S *;* SS
SS → ε
S → *simplestmt*
S → *begin* SS *end*



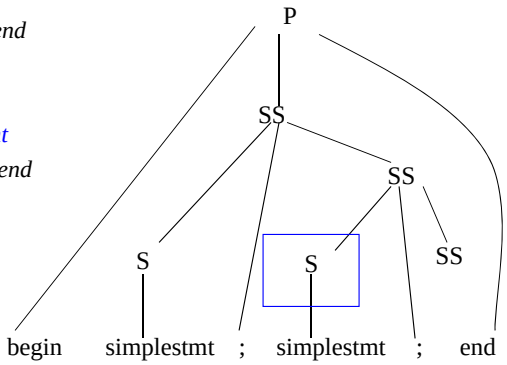
Top Down (LL) Parsing

P→ *begin* SS *end*
SS → S *;* SS
SS → ε
S → *simplestmt*
S → *begin* SS *end*



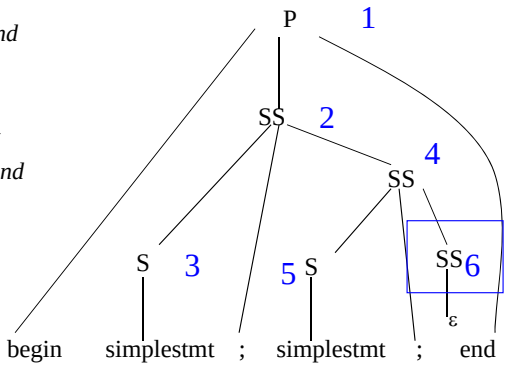
Top Down (LL) Parsing

P→ *begin* SS *end*
SS → S *;* SS
SS → ε
S → *simplestmt*
S → *begin* SS *end*



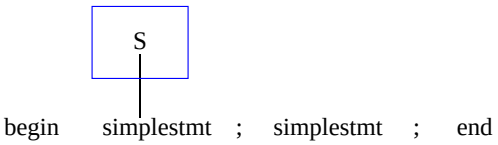
Top Down (LL) Parsing

P→ *begin* SS *end*
SS → S *;* SS
SS → ε
S → *simplestmt*
S → *begin* SS *end*



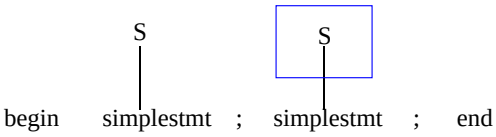
Bottomup (LR) Parsing

P→ *begin* SS *end*
SS → S *;* SS
SS → ε
S → *simplestmt*
S → *begin* SS *end*



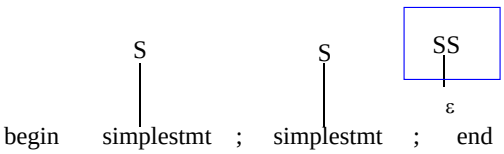
Bottomup (LR) Parsing

P→ *begin* SS *end*
SS → S *;* SS
SS → ε
S → *simplestmt*
S → *begin* SS *end*



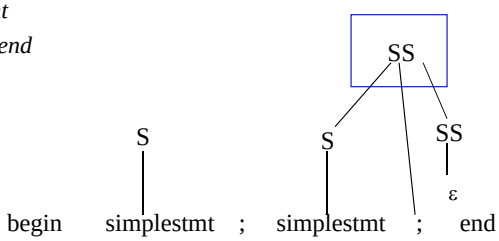
Bottomup (LR) Parsing

P→ *begin* SS *end*
SS → S *;* SS
SS → ε
S → *simplestmt*
S → *begin* SS *end*



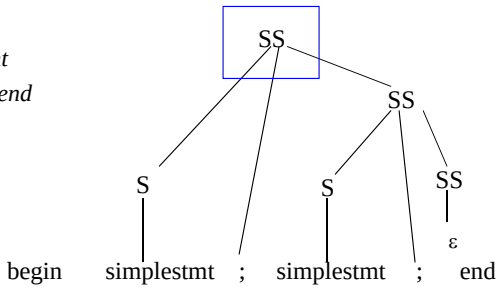
Bottomup (LR) Parsing

P→ *begin* SS *end*
SS → S ; SS
SS → ε
S → *simplestmt*
S → *begin* SS *end*



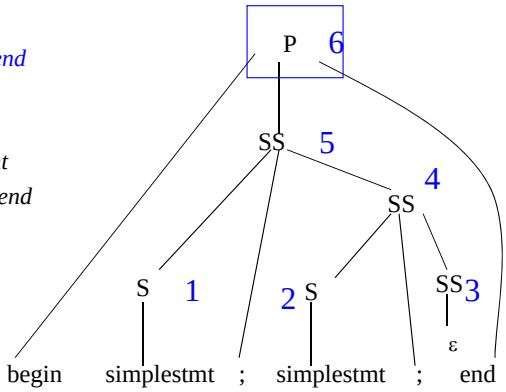
Bottomup (LR) Parsing

P→ *begin* SS *end*
SS → S ; SS
SS → ε
S → *simplestmt*
S → *begin* SS *end*



Bottomup (LR) Parsing

P→ *begin* SS *end*
SS → S *;* SS
SS → ε
S → *simplestmt*
S → *begin* SS *end*



Left Recursion (Recap)

- Productions of the form
  $A \rightarrow A\ \alpha$
  $\quad\quad | \ \beta$
  are left recursive
- When one of the productions in a grammar is left recursive then a predictive parser loops forever on certain inputs

Left Recursion (Recap)

- Replace

  $A \rightarrow A\ \alpha$

  $\quad\quad | \beta$

  with

  $A \rightarrow \beta\ R$

  $R \rightarrow \alpha\ R \mid \varepsilon$

57

Indirect Left Recursion

- What about indirect left recursion?

  $B \rightarrow D\ \alpha$

  $D \rightarrow B\ \beta$

58

General Left Recursion Elimination Method

Arrange the nonterminals in some order $A_1, A_2, \ldots A_n$
for i = 1, …, n **do**
    for j = 1, …, i-1 **do**
        for each production $A_i \rightarrow A_j \alpha$ and $A_j \rightarrow \beta$ **do**
            Replace $A_i \rightarrow A_j \alpha$ with $A_i \rightarrow \beta \alpha$
        endfor
    endfor
    eliminate any direct *left recursion* among A_i
endfor

Example Left Recursion Elimination

$A \rightarrow B$ **a** | **b** C
$B \rightarrow B$ **c** | A **d**
$C \rightarrow$ **e** | **f**

} Choose arrangement: $A = A_1$, $B = A_2$, $C = A_3$

A unchanged
$B \rightarrow B$ **c** | B **a d** | **b** C **d**
$B \rightarrow$ **b** C **d** D *(eliminating direct left recursion)*
$D \rightarrow$ **c** D | **a d** D | ε
C unchanged

Result:
$A \rightarrow B$ **a** | **b** C
$B \rightarrow$ **b** C **d** D
$C \rightarrow$ **e** | **f**
$D \rightarrow$ **c** D | **a d** D | ε

Left Factoring

- Most problems with predictive parsing are either (a) left recursion (which we have already dealt with) or (b) common prefixes
- Example:
  - *stmt* → **if** *expr* **then** *seq-of-stmts* **endif**
  - *stmt* → **if** *expr* **then** *seq-of-stmts* **else** *seq-of-stmts* **endif**
- Solution: Rewrite the production to defer the decision until we have enough information to make the right choice.

61

Left Factoring

- Replace productions
  $A \rightarrow \alpha\ \beta \mid \alpha\ \gamma$
  with
  $A \rightarrow \alpha\ A_R$
  $A_R \rightarrow \beta \mid \gamma$
- e.g.
  - *stmt* → **if** *expr* **then** *seq-of-stmts opt_end*
  - *opt_end* → **endif** | **else** *seq-of-stmts* **endif**

62

Non-context-free Language Constructs

- Not all syntactic rules are expressible using CFGs.
  - *e.g.*, "variables must be declared before they are used" cannot be exptessed in a CFG.
  - *or*, "the number of formal parameters for a function must equal the number of actual parameters."
- In practice, syntactic details that cannot be represented in a CFG are considered part of the ***static semantics*** and deferred to the semantic analysis phase.

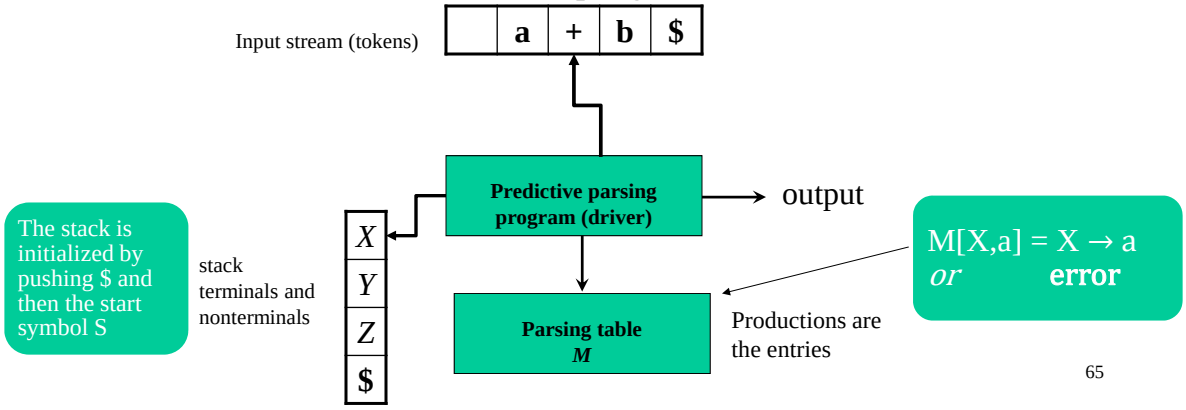
Top-Down Parsing

An LL(1) grammar can always be parsed top-down without backtracking.

- General Algorithms:
  - LL (top down)
    - (We looked at elementary recursive descent parsing in module 2)
  - LR (bottom up)
- Both algorithms are driven by the input grammar and the input to be parsed
- LL(1) grammars are those suitable for predictive parsing
- "LL(1)"≡ scans input from **L**eft to right, **L**eftmost derivation, **1** token lookahead.

Non-Recursive Predictive Parsing: Table-Driven Parsing

- Given an LL(1) grammar $G = (N, T, P, S)$ construct a table $M[A,a]$ for $A \in N$, $a \in T$ and use a *driver program* with a *stack*



65

Predictive Parsing Algorithm

- If X = **a** = $ ⋯ parser halts. Success!
- If X= **a** ≠ $
  - Pop X off stack
  - Advance input pointer
- If X is a non-terminal, look up M[X,**a**]
  - If, for example M[X,**a**] = $X \rightarrow UVW$, push WVU on the stack (U on top)

Predictive Parsing Program (Driver)

```
push($)
push(S)
a := lookahead
repeat
        X := pop()
        if X is a terminal or X = $ then
                match(X)                // moves to next token and a := lookahead
        else if M[X,a] = X → Y1Y2…Yk then
                push(Yk, Yk-1, …, Y2, Y1)        // such that Y1 is on top
                … invoke actions and/or produce IR output …
        else    error()
        endif
until X = $
```


Example Table

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow \mathbf{(}\ E\ \mathbf{)} \mid \mathbf{id}$

Eliminate left recursion

$E \rightarrow T\ E_R$
$E_R \rightarrow +\ T\ E_R \mid \varepsilon$
$T \rightarrow F\ T_R$
$T_R \rightarrow *\ F\ T_R \mid \varepsilon$
$F \rightarrow \mathbf{(}\ E\ \mathbf{)} \mid \mathbf{id}$

Nevermind how the parse table is built just yet

	id	**+**	*****	**(**	**)**	**$**
E	$E \rightarrow T\ E_R$			$E \rightarrow T\ E_R$		
E_R		$E_R \rightarrow +\ T\ E_R$			$E_R \rightarrow \varepsilon$	$E_R \rightarrow \varepsilon$
T	$T \rightarrow F\ T_R$			$T \rightarrow F\ T_R$		
T_R		$T_R \rightarrow \varepsilon$	$T_R \rightarrow *\ F\ T_R$		$T_R \rightarrow \varepsilon$	$T_R \rightarrow \varepsilon$
F	$F \rightarrow$ **id**			$F \rightarrow$ **(** E **)**		

Parsing Example

	id	+	*	(	)	$
E	$E \rightarrow T\ E_R$			$E \rightarrow T\ E_R$		
E_R		$E_R \rightarrow +\ T\ E_R$			$E_R \rightarrow \varepsilon$	$E_R \rightarrow \varepsilon$
T	$T \rightarrow F\ T_R$			$T \rightarrow F\ T_R$		
T_R		$T_R \rightarrow \varepsilon$	$T_R \rightarrow *\ F\ T_R$		$T_R \rightarrow \varepsilon$	$T_R \rightarrow \varepsilon$
F	$F \rightarrow$ **id**			$F \rightarrow (\ E\ )$		

Stack	Input	Production applied
$E	**id**+**id***id$	$E \rightarrow T\ E_R$
$E_R T	**id**+**id***id$	$T \rightarrow F\ T_R$
$E_R T_R F	**id**+**id***id$	$F \rightarrow$ **id**
$E_R T_R **id**	**id**+**id***id$	
$E_R T_R	+**id***id$	$T_R \rightarrow \varepsilon$
$E_R	+**id***id$	$E_R \rightarrow +\ T\ E_R$
$E_R T +	+**id***id$	
$E_R T	**id***id$	$T \rightarrow F\ T_R$
$E_R T_R F	**id***id$	$F \rightarrow$ **id**
$E_R T_R **id**	**id***id$	
$E_R T_R	***id$	$T_R \rightarrow *\ F\ T_R$
$E_R T_R F *	***id$	
$E_R T_R F	**id**$	$F \rightarrow$ **id**
$E_R T_R **id**	**id**$	
$E_R T_R	$	$T_R \rightarrow \varepsilon$
$E_R	$	$E_R \rightarrow \varepsilon$
$	$	

- So, the big question:
  How do we compute the parse tables?

- **We're going to need some more theory, folks, so hang on to your hats.**

FIRST Sets

FIRST(α) is the set of all terminal symbols that can begin some sentential form in a derivation that starts with α

$$\alpha \Rightarrow \ldots \Rightarrow a\,\beta$$

(also include ε if $\alpha \Rightarrow^* \varepsilon$

- FIRST(α) = {**a** $\in$ T | $\alpha \Rightarrow^*$ **a**β } $\cup$ { ε if $\alpha \Rightarrow^* \varepsilon$ }
- Example:

  simple $\rightarrow$ **integer** | **char** | **num dotdot num**

FIRST(*simple*) = { **integer**, **char**, **num** }

Computing FIRST

- To compute FIRST(X) for any single grammar symbol X:
  1. If X is a terminal, FIRST(X) = {X}
  2. If X $\rightarrow \varepsilon$ is a production, add ε to FIRST(X)
  3. If X is a nonterminal and X $\rightarrow Y_1\ Y_2 \ldots Y_n$ is a production, add **a** to FIRST(X) **if** a $\in$ FIRST(Y_j) **and** $\varepsilon \in$ FIRST(Y_1) $\cap$FIRST(Y_2) $\cap \ldots \cap$FIRST(Y_{i-1})

(i.e, $Y_1\ Y_2 \ldots Y_{i-1} \Rightarrow^* \varepsilon$)

Computing FIRST

- To compute FIRST(α) where $\alpha = X_1 X_2 \ldots X_n$:
  1. Add non-ε symbols of FIRST(X_1)
  2. If ε ϵ FIRST(X_1) then add non-ε symbols of FIRST(X_2).
  3. As long as ε ϵ FIRST(X_{i-1}) then add non-ε symbols of FIRST(X_i).
  4. If ε is a member of ***all*** the FIRST(X_i) sets then add ε to FIRST(α)

Example 1

- S → a S e
- S → B
- B → b B e
- B → C
- C → c C e
- C → d

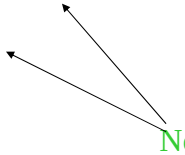
- FIRST(C) = {c,d}
- FIRST(B) =
- FIRST(S) =

Start with the 'simplest' non-terminal

Example 1

- S → a S e
- S → B
- B → b B e
- B → C
- C → c C e
- C → d

- FIRST(C) = {c,d}
- FIRST(B) = {b,c,d}
- FIRST(S) =



Now that we know FIRST(C) …

Example 1

- S → a S e
- S → B
- B → b B e
- B → C
- C → c C e
- C → d

- FIRST(C) = {c,d}
- FIRST(B) = {b,c,d}
- FIRST(S) = {a,b,c,d}

Example 2

- P → i | c | n T S
- Q → P | a S | d S c S T
- R → b | ε
- S → e | R n | ε
- T → R S q

- FIRST(P) = {i,c,n}
- FIRST(Q) =
- FIRST(R) = {b,ε}
- FIRST(S) =
- FIRST(T) =

Example 2

- P → i | c | n T S
- Q → P | a S | d S c S T
- R → b | ε
- S → e | R n | ε
- T → R S q

- FIRST(P) = {i,c,n}
- FIRST(Q) = {i,c,n,a,d}
- FIRST(R) = {b,ε}
- FIRST(S) =
- FIRST(T) =

Example 2

- P → i | c | n T S
- Q → P | a S | d S c S T
- R → b | ε
- S → e | R n | ε
- T → R S q

- FIRST(P) = {i,c,n}
- FIRST(Q) = {i,c,n,a,d}
- FIRST(R) = {b,ε}
- FIRST(S) = {e,b,n,ε}
- FIRST(T) =

Note:
S ⇒ R n ⇒ n because R ⇒* ε

Example 2

- P → i | c | n T S
- Q → P | a S | d S c S T
- R → b | ε
- S → e | R n | ε
- T → R S q

- FIRST(P) = {i,c,n}
- FIRST(Q) = {i,c,n,a,d}
- FIRST(R) = {b,ε}
- FIRST(S) = {e,b,n,ε}
- FIRST(T) = {b,c,n,q}

Note:
T ⇒ R S q ⇒ S q ⇒ q
because both R and S ⇒* ε

Example 3

- S → a S e | S T S
- T → R S e | Q
- R → r S r | ε
- Q → S T | ε

- FIRST(S) =
- FIRST(R) =
- FIRST(T) =
- FIRST(Q) =

Example 3

- S → a S e | S T S
- T → R S e | Q
- R → r S r | ε
- Q → S T | ε

- FIRST(S) = {a}
- FIRST(R) = {r, ε}
- FIRST(T) = {r,a, ε}
- FIRST(Q) = {a, ε}

FOLLOW Sets

- FOLLOW(A) is the set of terminals (including end of file - $) that may follow non-terminal A in some sentential form.
- FOLLOW(A) = $\{a \in T \mid S \Rightarrow^+ \alpha A a \beta\} \cup \{\$\}$ if $S \Rightarrow^+ \gamma A$
- For example, consider $L \Rightarrow^+ (())(L)L$
  Both ‘)’ and end of file can follow L
- NOTE: ε is ***never*** in FOLLOW sets

Computing FOLLOW(A)

1. If S is the start symbol, put $ in FOLLOW(S)
2. Productions of the form $B \rightarrow \alpha A \beta$,
   Add FIRST(β) – {ε} to FOLLOW(A)
   INTUITION: Suppose $B \rightarrow AX$ and FIRST(X) = {c}
   $S \Rightarrow^+ \alpha B \beta \Rightarrow \alpha A X \beta \Rightarrow^+ \alpha A c \delta \beta$
   = FIRST(X)

3. Productions of the form B → α A or

B → α A β where β $\Rightarrow^*$ ε (i.e, ε ϵ FIRST(β))

Add everything in FOLLOW(B) to FOLLOW(A)

INTUITION:

- Suppose B → Y A

  S $\Rightarrow^+$ α B β $\Rightarrow$ α Y A β

  FOLLOW(B)

- Suppose B → A X and X $\Rightarrow^*$ ε

  S $\Rightarrow^+$ α B β $\Rightarrow$ α A X β $\Rightarrow^*$ α A β

  FOLLOW(B)

Assume the first non-terminal is the start symbol

Example 4

- S → a S e | B
- B → b B C f | C
- C → c C g | d | ε

- FIRST(C) = {c,d,ε}
- FIRST(B) = {b,c,d,ε}
- FIRST(S) = {a,b,c,d,ε}

- FOLLOW(C) =
- FOLLOW(B) =
- FOLLOW(S) = {$}

Using rule #1

Example 4

- S → a S e | B
- B → b B C f | C
- C → c C g | d | ε

- FIRST(C) = {c,d,ε}
- FIRST(B) = {b,c,d,ε}
- FIRST(S) = {a,b,c,d,ε}

- FOLLOW(C) = {f,g}
- FOLLOW(B) = {c,d,f}
- FOLLOW(S) = {$,e}

Using rule #2

Example 4

- S → a S e | B
- B → b B C f | C
- C → c C g | d | ε

- FIRST(C) = {c,d,ε}
- FIRST(B) = {b,c,d,ε}
- FIRST(S) = {a,b,c,d,ε}

- FOLLOW(C) = {f,g} ∪ FOLLOW(B) = {c,d,e,f,g,$}
- FOLLOW(B) = {c,d,f} ∪ FOLLOW(S) = {c,d,e,f,$}
- FOLLOW(S) = {$, e }

Using rule #3

Example 5

- S → A B C | A D
- A → ε | a A
- B → b | c | ε
- C → D d C
- D → e b | f c

- FIRST(D) = {e,f}
- FIRST(C) = {e,f}
- FIRST(B) = {b,c,ε}
- FIRST(A) = {a, ε}
- FIRST(S) = {a,b,c,e,f}

- FOLLOW(S) =
- FOLLOW(A) =
- FOLLOW(B) =
- FOLLOW(C) =
- FOLLOW(D) =

Example 5

- S → A B C | A D
- A → ε | a A
- B → b | c | ε
- C → D d C
- D → e b | f c

- FIRST(D) = {e,f}
- FIRST(C) = {e,f}
- FIRST(B) = {b,c,ε}
- FIRST(A) = {a, ε}
- FIRST(S) = {a,b,c,e,f}

- FOLLOW(S) = {$}
- FOLLOW(A) = {b,c,e,f}
- FOLLOW(B) = {e,f}
- FOLLOW(C) = {$}
- FOLLOW(D) = {$}

Example 6

- S → (A) | ε
- A → T E
- E → & T E | ε
- T → (A) | a | b | c

- FIRST(T) = {(,a,b,c}
- FIRST(E) = {&, ε }
- FIRST(A) = {(,a,b,c}
- FIRST(S) = {(, ε}

- FOLLOW(S) =
- FOLLOW(A) =
- FOLLOW(E) =
- FOLLOW(T) =

Example 6

- S → (A) | ε
- A → T E
- E → & T E | ε
- T → (A) | a | b | c

- FIRST(T) = {(,a,b,c}
- FIRST(E) = {&, ε }
- FIRST(A) = {(,a,b,c}
- FIRST(S) = {(, ε}

- FOLLOW(S) = {$}
- FOLLOW(A) = {) }
- FOLLOW(E) =

FOLLOW(A) = {) }

- FOLLOW(T) =

FIRST(E) ∪ FOLLOW(A) ∪ FOLLOW(E) = {&,)}

Example 7

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

- FOLLOW(E) =
- FOLLOW(E') =
- FOLLOW(T) =
- FOLLOW(T') =
- FOLLOW(F) =

- FIRST(F) = FIRST(T) = FIRST(E) = {(,id}
- FIRST(T') = {*,ε}
- FIRST(E') = {+,ε}

Example 7

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

- FOLLOW(E) = {$,)}
- FOLLOW(E') = FOLLOW(E) = {$,)}
- FOLLOW(T) = FIRST(E') ∪ FOLLOW(E) ∪ FOLLOW(E') = {+,$,)}
- FOLLOW(T') = FOLLOW(T) = {+,$,)}
- FOLLOW(F) = FIRST(T') ∪ FOLLOW(T) ∪ FOLLOW(T') = {*,+,$,)}

- FIRST(F) = FIRST(T) = FIRST(E) = {(,id}
- FIRST(T') = {*,ε}
- FIRST(E') = {+,ε}

Using FIRST and FOLLOW to Write a Recursive Descent Parser

expr → *term rest*
rest → **+** *term rest*
| **-** *term rest*
| ε
term → **id**

```
procedure rest();
begin
   if lookahead in FIRST(+ term rest) then
      match('+'); term(); rest()
   else if lookahead in FIRST(- term rest) then
      match('-'); term(); rest()
   else if lookahead in FOLLOW(rest) then
      return
   else error()
end;
```

FIRST(+ *term rest*) = { + }
FIRST(- *term rest*) = { - }
FOLLOW(*rest*) = { **$** }