Programming Language Syntax

長庚大學資訊工程學系　陳仁暉　助理教授
Tel: (03) 211-8800 Ext: 5990
Email: jhchen@mail.cgu.edu.tw
URL: http://www.csie.cgu.edu.tw/~jhchen

Syntax of Programming Languages

- Formal Languages and Grammars
- Regular Grammars and Languages
- Context-free Grammars and Languages
- Context-sensitive Grammars and Languages
- Attribute Grammars

Formal Languages and Grammars

- Formal Languages
  - Programming languages are formal languages.
  - A formal language is a set of finite strings of symbols taken from some alphabet.
  - Example
    - Here are some languages over the alphabet {0,1}
      - $L_1 = \{\}$
      - $L_2 = \{0,1\}$
      - L_3 = the set of all binary strings ending in 10
        = {10, 010, 110, 0010, 0110, 1010, 1110, …}
      - L_4 = the set of all binary strings
        = { ε , 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, …}
    - Programming language C = the set of all syntactically correct C programs over C's alphabet

Grammars & Terminals

- Grammars
  - Grammars are used to define formal languages.
  - A grammar consists of four parts:
    - 1. A set of terminal symbols
    - 2. A set of nonterminal symbols
    - 3. A set of rewriting rules (or production rules) of the form

$$\alpha \rightarrow \beta$$

      where α and β are strings of terminals and nonterminals, and α contains at least one nonterminal.
  - Terminals are symbols of the language being defined.

    Nonterminals are symbols of the defining language.

    A rewriting rules specifies that the string α may produce or be rewritten as the string β.

    The rewriting process begins with the start symbol.

Grammars & Terminals

- A grammar G generates a language L(G) defined by
  - L(G) = the set of all strings of terminals, called sentences, that can be derived from the start symbol through a sequence of applications of the rewriting rules of G.
- Example
  - Let G_1 be a grammar that consists of a terminal a, two terminals S and A, the start symbol S, and the rules:

    $$S \rightarrow a$$

    $$S \rightarrow aA$$

    $$A \rightarrow aS$$

  - Conventions
    - Small letters are terminals, and capital letters are nonterminals
    - The nonterminal in the left-hand side of the first rule is the start symbol.
    - The first two rules are usually abbreviated as

      $$S \rightarrow a \mid aA$$

Grammars & Terminals

- Example (continued)
  - The string aaaaa is a sentence of the language generated by G.

    S ➔ aA ➔ aaS ➔ aaaA ➔ aaaaS ➔ aaaaa

    - This sequence is called a derivation of the sentence aaaaa.
    - The symbol ➔ means "derive in one step," whereas the symbol → means "produce".
  - $L(G_1)$ = the set of all strings containing odd number of a's = { a^n | n >= 1 is odd}
  - A language may be generated by many different grammars. For example, the language $L(G_1)$ may also be generated by the following grammars:

    G_2 S → a | aaS

    G_3 S → a | Saa

    G_4 S → a | aSa

Regular Grammars and Languages

- Regular grammars and languages
  - A regular grammar is a left- or right-linear grammar whose production rules are of the form

    $A \rightarrow \omega \mid B\omega$ ← left-linear

    or, $A \rightarrow \omega \mid \omega B$ ← right-linear

    where A and B are nonterminals, and ω is a (possibly empty) string of terminals.
  - Regular grammars generate regular languages.
  - Example
    - The language of all binary strings ending in 10 is regular.
    - Right-linear grammar

      $S \rightarrow 0S \mid 1S \mid 10$

      Right-linear grammars generate sentences from the left end

      $S \Rightarrow 0S \Rightarrow 01S \Rightarrow 0110$

Regular Grammars and Languages

- Example (Continued)
  - Left-linear grammar

    S → A10

    A → A0 | A1 | ε

    Left-linear grammars generate sentences from the right end

    S ➔ A10 ➔ A110 ➔ A0110 ➔ 0110
  - Here is an equivalent left-linear grammar without rules that produce the empty string ε.

    S → A10 | 10

    A → A0 | A1 | 0 | 1

Regular Grammars and Languages

- Lexical syntax
  - The lexical syntax of a programming language (i.e., the syntax of tokens, including keywords, identifiers, numeric constants, etc), can be described by regular grammars.
  - Example
    - The language of C's keywords is regular.

      S → if | while | for | int | long | double | …
    - The language of C's identifiers (a letter or _followed by any number of letters, digits, or _) is regular, too.

      Right-linear grammar

      S → aA | … | zA | \`A'A | … | \`Z'A | _A

      A → aA | … | zA | \`A'A | … | \`Z'A | _A

      0A | … | 9A | ε

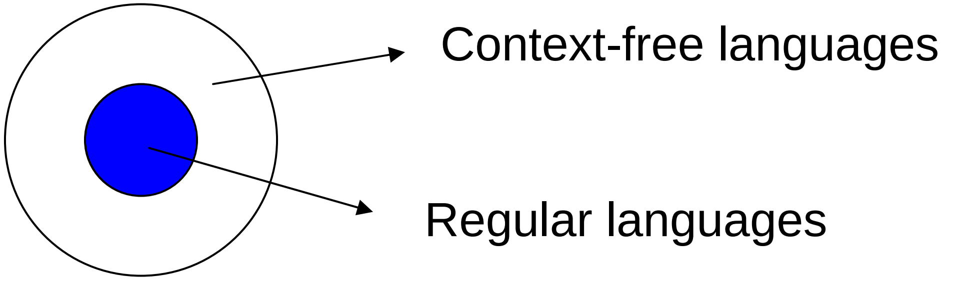
Context-free Grammars and Languages

- Context-free grammars and language
- A context-free grammar (CFG) has production rules of the form

  $A \rightarrow \alpha$

  where A is a nonterminal, α is a (possibly empty) string of terminals and nonterminals.

  - Context-free grammars generate context-free language (CFL).
  - Context-free grammars are so called because the rewriting of a nonterminal is independent of its context.
  - A regular grammar (language) is also a CFG (CFL) but a CFG (CFL) may not be a regular grammar (language).



Context-free Grammars and Languages

- Example (Continued)
  - Consider the following language

    L = the set all nested balanced parentheses

    $= \{\varepsilon, (\,), (\,(\,)\,), (\,(\,(\,)\,)\,), \ldots\}$

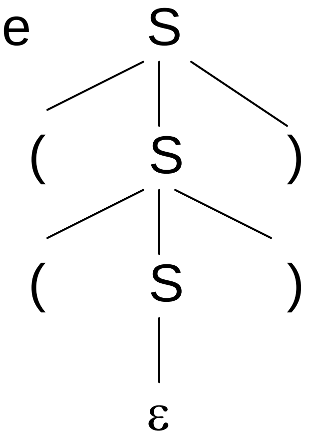
    $= \{ (^n\,)^n \mid n \geqq 0\}$
  - It can be shown that L is not a regular language, i.e., no regular grammars can generate L.
  - That L is not regular implies that programming languages are not regular, since nested balanced parentheses are parts of expression syntax, e.g., (((x+((2))))).

Context-free Grammars and Languages

- Parse tree (derivation tree)
  - A parse tree is a graphical representation of derivations.
  - Example
    - Let the CFG be

      $S \rightarrow (S) \mid \varepsilon$
    - Derivation

      $S \Rightarrow (\,S\,) \Rightarrow (\,(\,S\,)\,) \Rightarrow (\,(\,)\,)$
    - Parse tree



      - Leaves, from left to right, contain the sentence (( ))
    - Every sentence has a single derivation and a single parse tree.

Specifying Syntax

- Regular expression
  - Any set of strings that can be defined in terms of the first three rules (concatenation, alternation (choice among a finite set of alternatives), and so-called "Kleene closure" (repetition an arbitrary number of times)) is called a *regular set*, or sometimes a *regular language*.
- Context-Free Grammars
  - Any set of strings that can be defined if we add recursion is called a *context-free language* (CFL).

Tokens and Regular Expressions

- Tokens are the basic building blocks of programs.
  - Pascal, for example, has 64 kinds of tokens, including 21 symbols (+, -, ;, :=, .., etc.), 35 keywords (begin, end, div, record, while, etc.), integer literals (e.g., 137), real (floating-point) literals (e.g., 6.022e23), character/string literals (e.g., `snerk').
- To specify tokens, we use the notation of regular expressions. A regular expression is one of the following:
  - 1. a character
  - 2. the empty string, denoted ε
  - 3. two regular expressions next to each other, meaning any string generated by the first one followed by (concatenated with) any string generated by the second one
  - 4. two regular expressions separated by a vertical bar (|), meaning any string generated by the first one or any string generated by the second one
  - 5. a regular expression followed by a Kleene star, meaning the concatenation of zero or more strings generated by the expression in front of the star

Tokens and Regular Expressions

- digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- unsigned_integer → digit digit*
- unsigned_number → unsigned_integer ((. unsigned_integer) | ε) ((e (+ | - | ε) unsigned_integer) | ε)

Context-Free Grammar

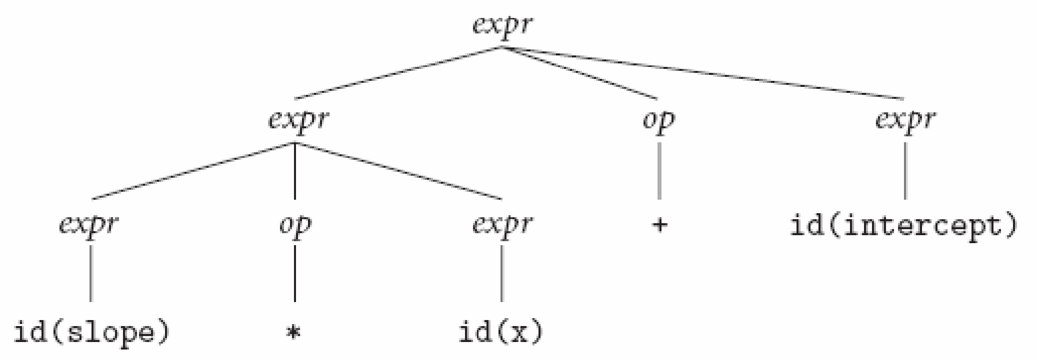
- CFG can help us to specify nested constructs, which are central to programming languages.
  - express → identifier | number | - expression | (expression) | expression operator expression
  - operator → + | - | * | /
  - Each of the rules in a context-free grammar is known as a *production*.
  - The symbols on the left-hand sides of the productions are known as variables, or *nonterminals*.
  - Symbols that are to make up the strings derived from the grammar are know as *terminals*.
  - One of the nonterminals, usually the one on the left-hand side of the first production, is called the *start symbol*.

Context-Free Grammar

- The notation for context-free grammars is sometimes called Backus-Naur Form (BNF), in honor of John Backus and Peter Naur, who devised it for the definition of the Algol-60 programming language [NBB+ 63].
- The vertical bar, Kleene star, and meta-level parentheses of regular expressions are not allowed in BNF.
- These extra operators, the notation is often called extended BNF (EBNF).

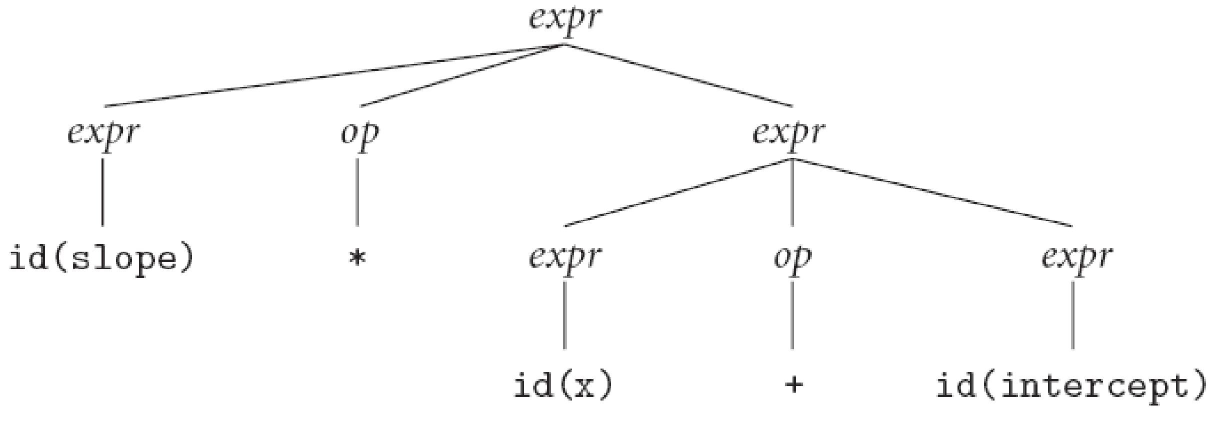
Derivations and Parse Trees

- Parsing the string "slope * x + intercept"
- expr → expr op expr
  - → expr op id
  - → expr + id
  - → expr op expr + id
  - → expr op id + id
  - → expr * id + id
  - → id * id + id



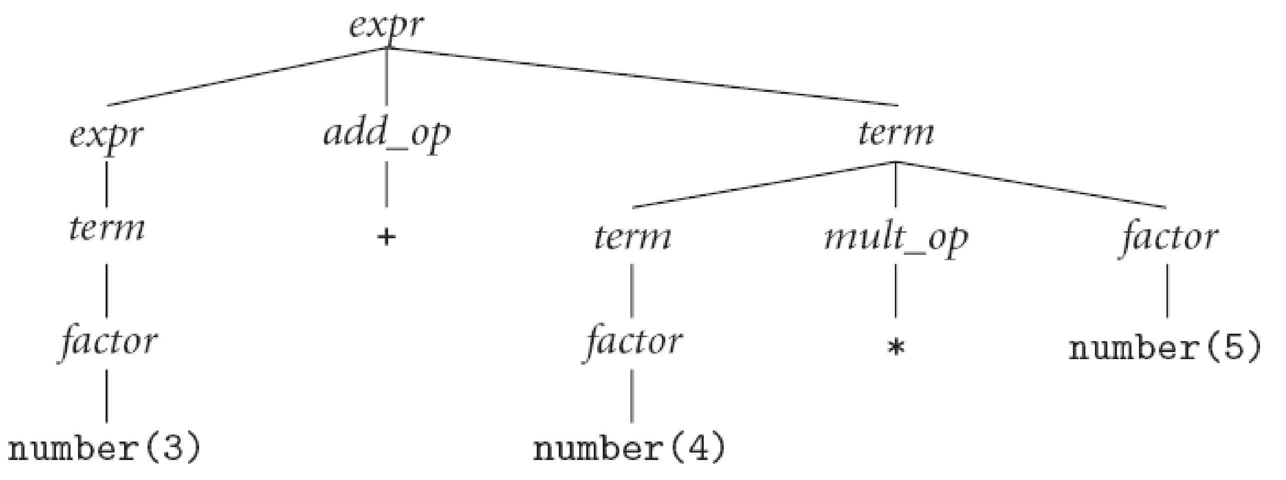
Ambiguous

- The above example chooses at each step to replace the right-most nonterminal with the *right-most* derivation, also called a *canonical* derivation.
- There are many other possible derivations, including *left-most* and options in-between.



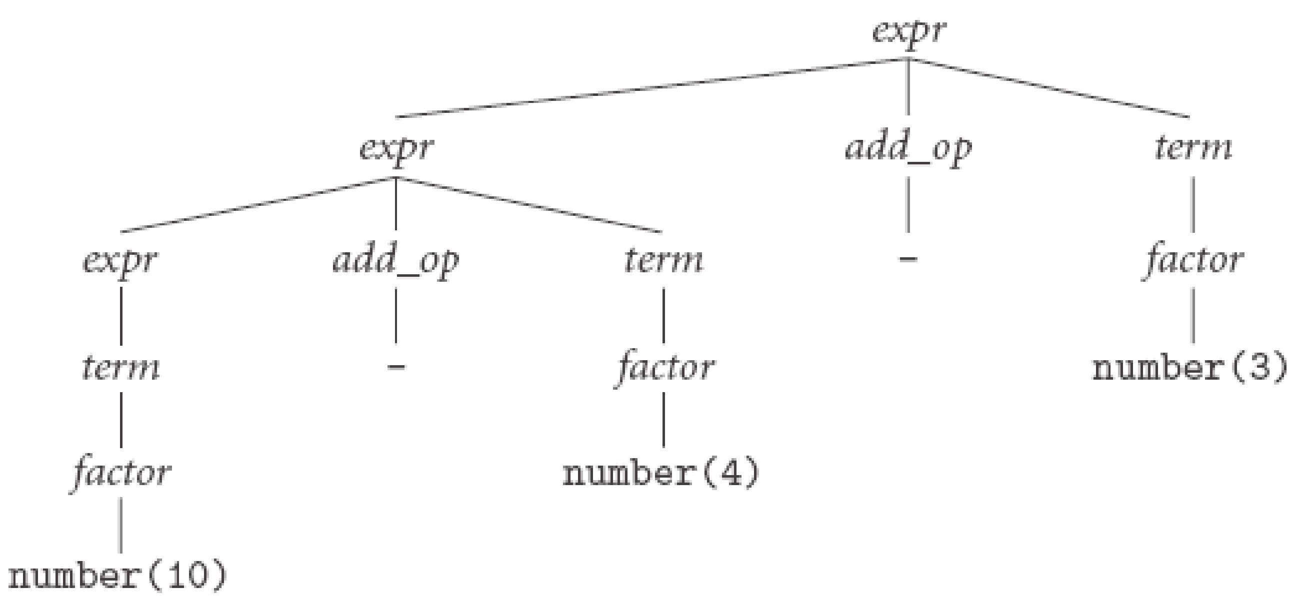
Parse Tree

- Parse tree for 3+4*5, with precedence.
  - expr → term | expr add_op term
  - term → factor | term mult_op factor
  - factor → id | number | - factor | (expr)
  - add_op → + | -
  - mult_op → * | /



Another Example of Parse Tree

- The subtraction groups more tightly to the left, so that 10 – 4 – 3 would evaluated to 3, rather than to 9. (Grammar shown in page 38)



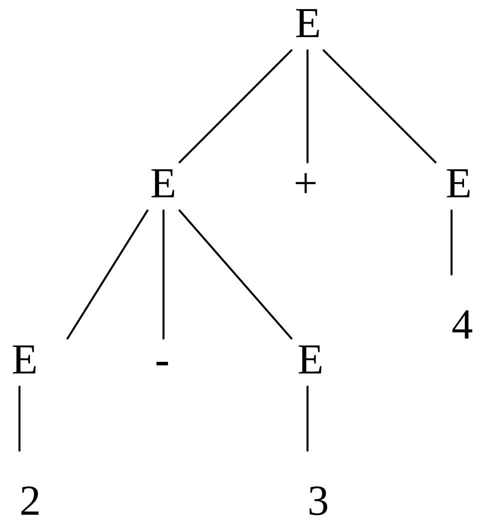
Another example of ambiguity

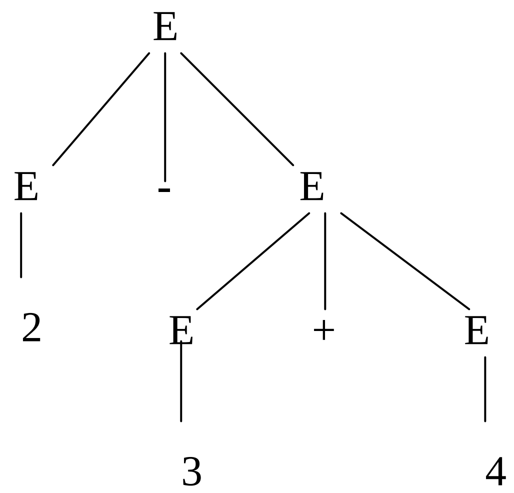
Two (or more) parse trees or leftmost derivations for the *same string*

$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow 0 \mid \ldots \mid 9$





- 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

- An ambiguous grammar can sometimes be made unambiguous

E → E + T | E – T | T

T → 0 | … | 9

Recognizing Syntax: Scanners & Parsers

- The scanner and parser for a programming language are responsible for discovering the syntactic structure of a given program.
- The parser is the heart of a typical compiler.
- Parser calls the scanner to obtain the tokens of the input program, assembles the tokens together into a parse tree, and passes the tree (perhaps one subroutine at a time) to the later phases of the compiler, which perform semantic analysis and code generation and improvement.

Recognizing Syntax: Scanners & Parsers

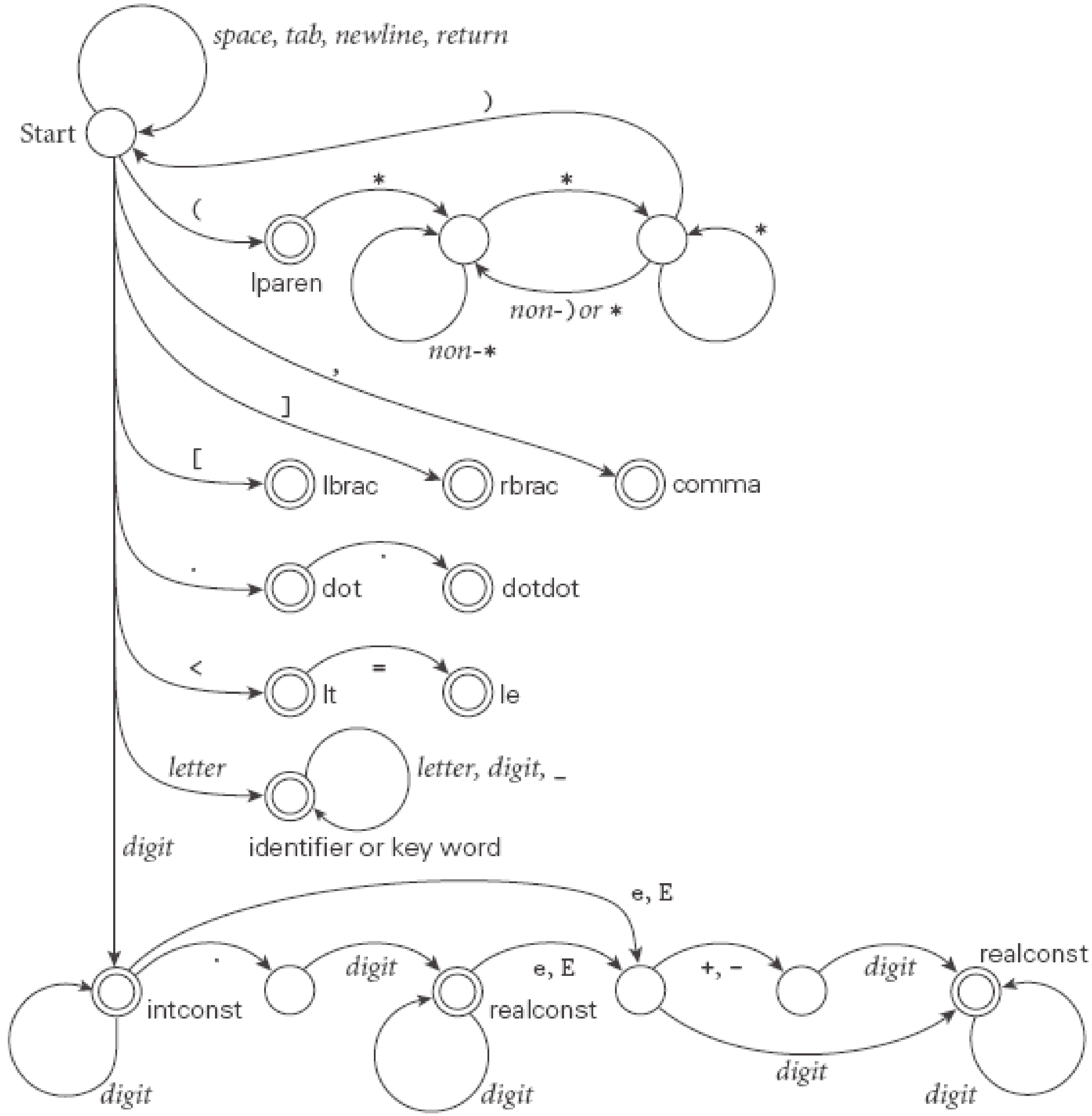
- Scanner
  - It dramatically reduces the number of individual items.
  - Typically remove comments (so the parser doesn't have to worry about them appearing throughout the context-free grammar).
- Scanners normally deal only with nonrecursive constructs, nested comments require special treatment.
- In theoretical parlance, a scanner is a deterministic finite automaton (DFA) that recognize the tokens of a programming language.
- A parser is a deterministic push-down automaton (PDA) that recognizes the language' context-free syntax.
- This task is performed by tools such as Unix's lex and yacc.
  - At many sites, lex and yacc have been superseded by the GNU flex and bison tools. These independently developed, noncommercial alternatives are available without charge from the Free Software Foundation at www.fsf.org/software.

Scanning

- Please refer to textbook on page 40.
- This algorithm is a pseudo code of scanner for Pascal.
- It is not difficult to flesh out the algorithm above by hand, to produce code in some programming language.
- We can write the code by hand (this option basically amounts to a highly stylized ad hoc scanner), or we can use a scanner *generator* (e.g., lex) to build it automatically from a set of regular expressions.

Scanning

- Pictorial representation of (part of) a Pascal scanner as a finite automaton.



Top-Down and Bottom-Up Parsing

- A context-free grammar (CFG) is a generator for a CF language.
- A parser is a language *recognizer*.
- LL stands for “Left-to-right, Left-most derivation.” LR parser is called “top-down,” or “predictive” parser.
- LR stands for “Left-to-right, Right-most derivation.” LR parser is called “bottom-up” parser.

Common Orderings

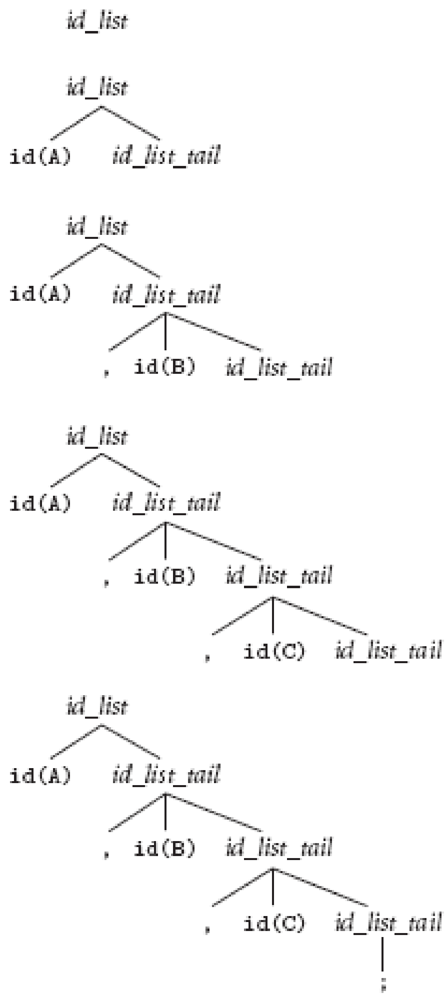
- Top-down
  - Start with the root
  - Traverse the parse tree depth-first, left-to-right (leftmost derivation)
  - LL(k)
- Bottom-up
  - Start at leaves and build up to the root
    - Effectively a rightmost derivation in reverse(!)
  - LR(k) and subsets (Look Ahead Left Recursive, LALR(k), Simple Left Recursive, SLR(k), etc.)

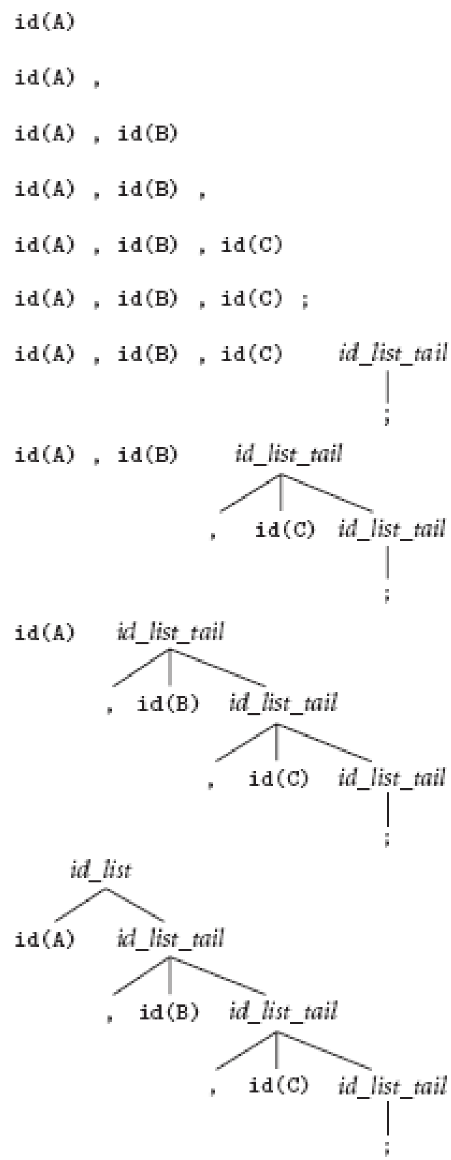
Top-down vs. bottom-up

- Consider the grammar (Scott, p. 49)
  - *id_list* → `id` *id_list_tail*
  - *id_list_tail* → `,  id` *id_list_tail*
  - *id_list_tail* → `;`

- And input text:
  - `A, B, C;`

Top-down vs. bottom-up Parsing





id_list ⟶ id *id_list_tail*

id_list_tail ⟶ , id *id_list_tail*

id_list_tail ⟶ ;

Disadvantage of Bottom-Up Parsing

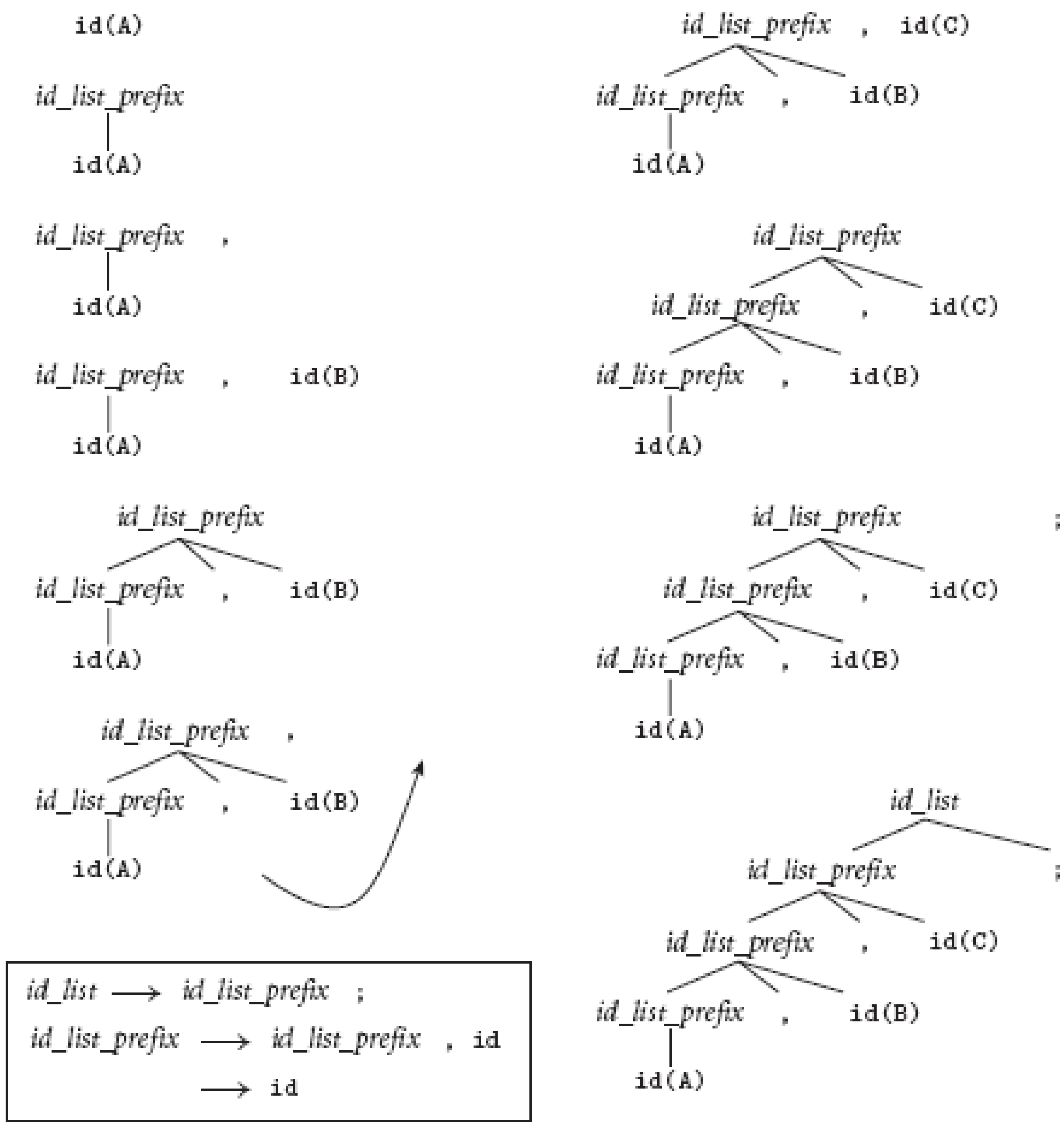
- The problem with previous grammar, for the purpose of bottom-up parsing, is that it forces the compiler to shift all the tokens of an id_list into its forest before it can reduce any of them.

  id_list → id_list_prefix;

  id_list_prefix → id_list_prefix, id | id

- This grammar cannot be parsed top-down, because when we see an id on the input and we're expecting an id_list_prefix, we have no way to tell which of the two possible productions we should predict.

Bottom-up Revision

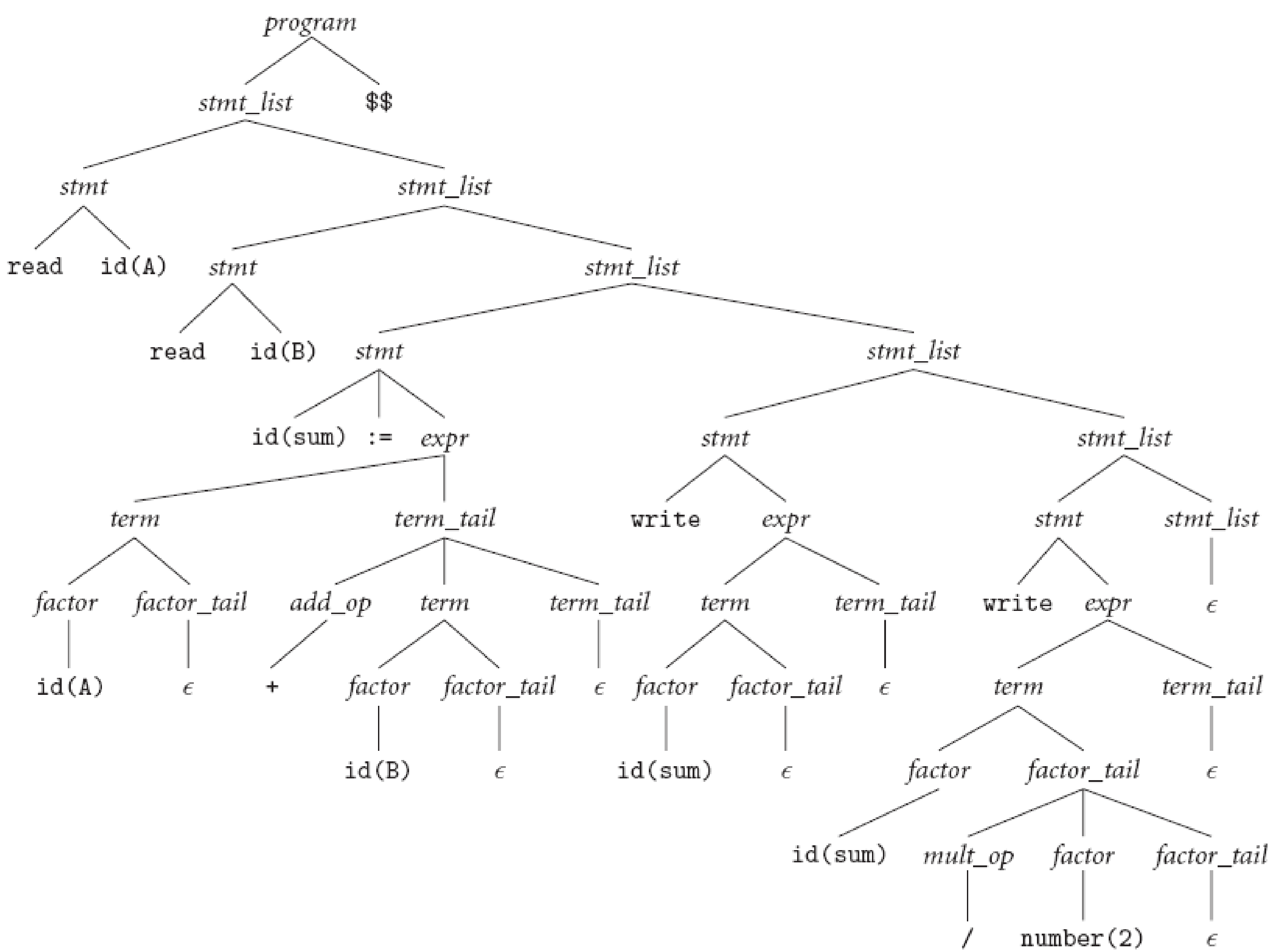


Recursive Descent

- An example of “calculate” language.

```
read A
read B
sum := A + B
write sum
write sum / 2
```

Sum-and-average program



FIRST Sets

- FIRST(α) is the set of all terminal symbols that can begin some sentential form that starts with α
- FIRST(α) = {a in V_t | α ➔* aβ } U { ε } if α ➔* ε
- Example:

 <stmt> → simple | begin <stmts> end

 FIRST(<stmt>) = {simple, begin}

Computing FIRST sets

Initially FIRST(A) is empty

1. For productions $A \rightarrow a\ \beta$, where a in V_t

   Add { a } to FIRST(A)

2. For productions $A \rightarrow \varepsilon$

   Add { ε } to FIRST(A)

3. For productions $A \rightarrow \alpha\ B\ \beta$, where $\alpha \Rightarrow^* \varepsilon$ and NOT ($B \Rightarrow \varepsilon$)

   Add FIRST(αB) to FIRST(A)

4. For productions $A \rightarrow \alpha$, where $\alpha \Rightarrow^* \varepsilon$

   Add FIRST(α) and { ε } to FIRST(A)

To compute FIRST across strings of terminals and non-terminals:

FIRST(ε) = { ε }

FIRST($A\alpha$) = A if A is a terminal

= FIRST(A) U FIRST(α)

if A ➔ ε

= FIRST(A) otherwise

Example 1

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

- FIRST(C) =
- FIRST(B) =
- FIRST(S) =

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 | b S c S T
- R → b | ε
- S → c | R n | ε
- T → R S q

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

Example 2

- $P \rightarrow i \mid c \mid n\ T\ S$
- $Q \rightarrow P \mid a\ S \mid b\ S\ c\ S\ T$
- $R \rightarrow b \mid \varepsilon$
- $S \rightarrow c \mid R\ n \mid \varepsilon$
- $T \rightarrow R\ S\ q$

- FIRST(P) = {i,c,n}
- FIRST(Q) = {i,c,n,a,b}
- FIRST(R) = {b, ε}
- FIRST(S) = {c,b,n, ε}
- FIRST(T) = {b,c,n,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) =
- 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 V_t | S $\Rightarrow^+$ …Aa…} U {$ (end of file)} if S $\Rightarrow^+$ …A
- For example, consider L $\Rightarrow^+$ (())(L)L --

  Both ‘)’ and end of file can follow L

Computing FOLLOW(A)

- If S is a start symbol, put \$ in FOLLOW(S)
- Productions of the form $B \rightarrow \alpha\ A\ a$, then add { a } to FOLLOW(A)
- 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}

$S \Rightarrow^{+} \alpha\ B\ \beta \Rightarrow \alpha\ A\ X\ \beta \Rightarrow^{+} \alpha\ A\ c\ \delta\ \beta$

- Productions of the form $B \rightarrow \alpha\ A$ or $B \rightarrow \alpha\ A\ \beta$ where $\beta \Rightarrow^* \varepsilon$

  Add FOLLOW(B) to FOLLOW(A)

  INTUITION:

  - Suppose $B \rightarrow Y\ A$

    $S \Rightarrow^+ \alpha\ B\ \beta \Rightarrow \alpha\ Y\ A\ \beta$
  - Suppose $B \rightarrow A\ X$ and $X \rightarrow \varepsilon$

    $S \Rightarrow^+ \alpha\ B\ \beta \Rightarrow \alpha\ A\ X\ \beta \Rightarrow \alpha\ A\ \beta$

NOTE: ε ***never*** in FOLLOW sets

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) =

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) = g,f

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

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

Example 5

- 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 5

- 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

- $E \rightarrow T\ E'$
- $E' \rightarrow +\ T\ E' \mid \varepsilon$
- $T \rightarrow F\ T'$
- $T' \rightarrow *\ F\ T' \mid \varepsilon$
- $F \rightarrow (\ E\ ) \mid id$

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

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

Example 6

- $E \rightarrow T\ E'$
- $E' \rightarrow +\ T\ E' \mid \varepsilon$
- $T \rightarrow F\ T'$
- $T' \rightarrow *\ F\ T' \mid \varepsilon$
- $F \rightarrow (\ E\ ) \mid id$

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

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

Example 7

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

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

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

Example 7

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

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

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

Writing an LL(1) Grammar

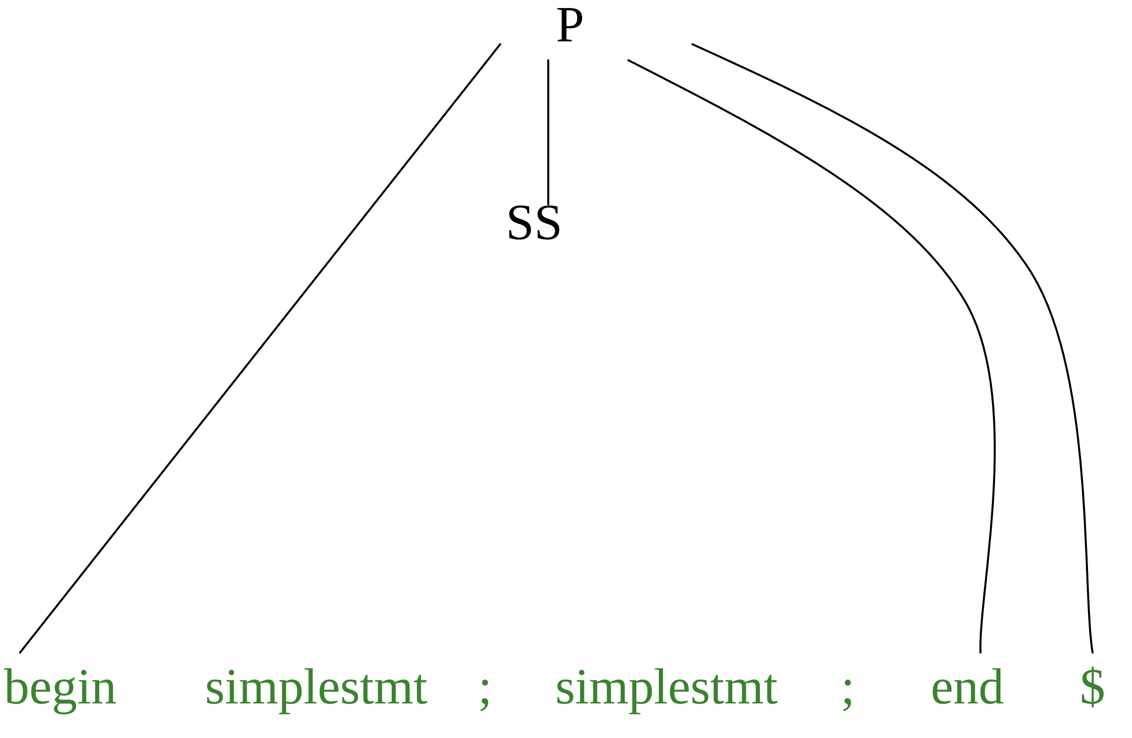
- The two most common obstacles to “LL(1)-ness” are
  - Left recursion
  - Common prefixes

Top Down (LL) Parsing

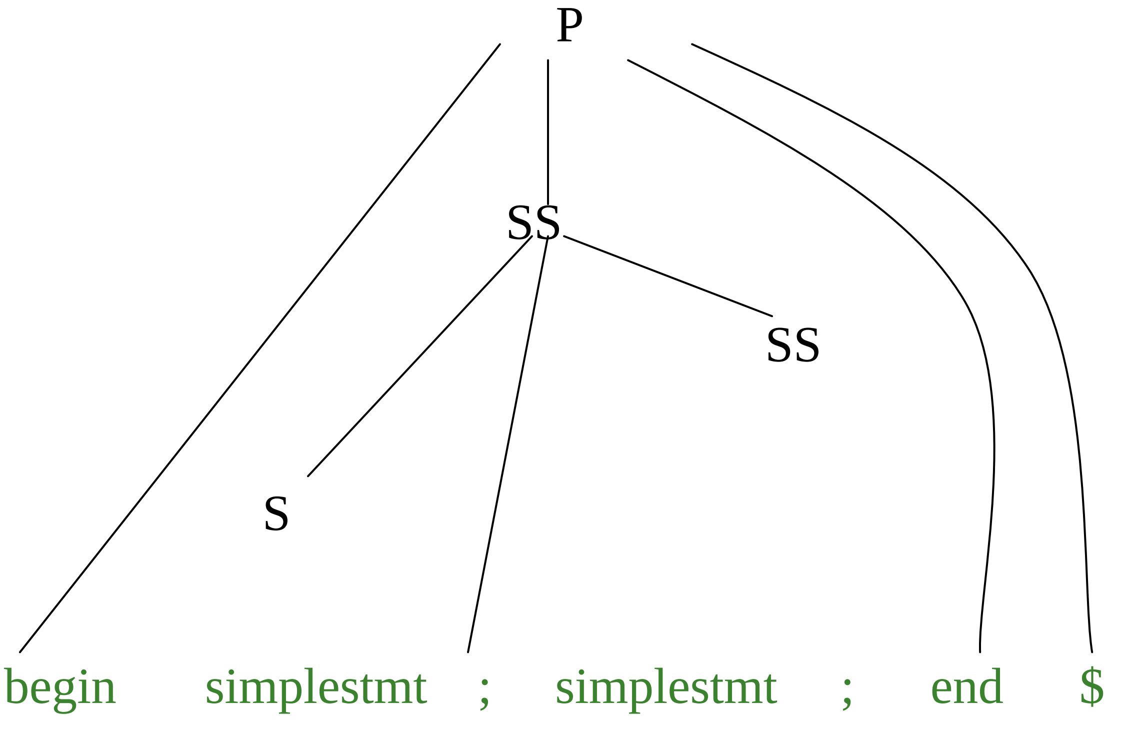
P

begin simplestmt ; simplestmt ; end $

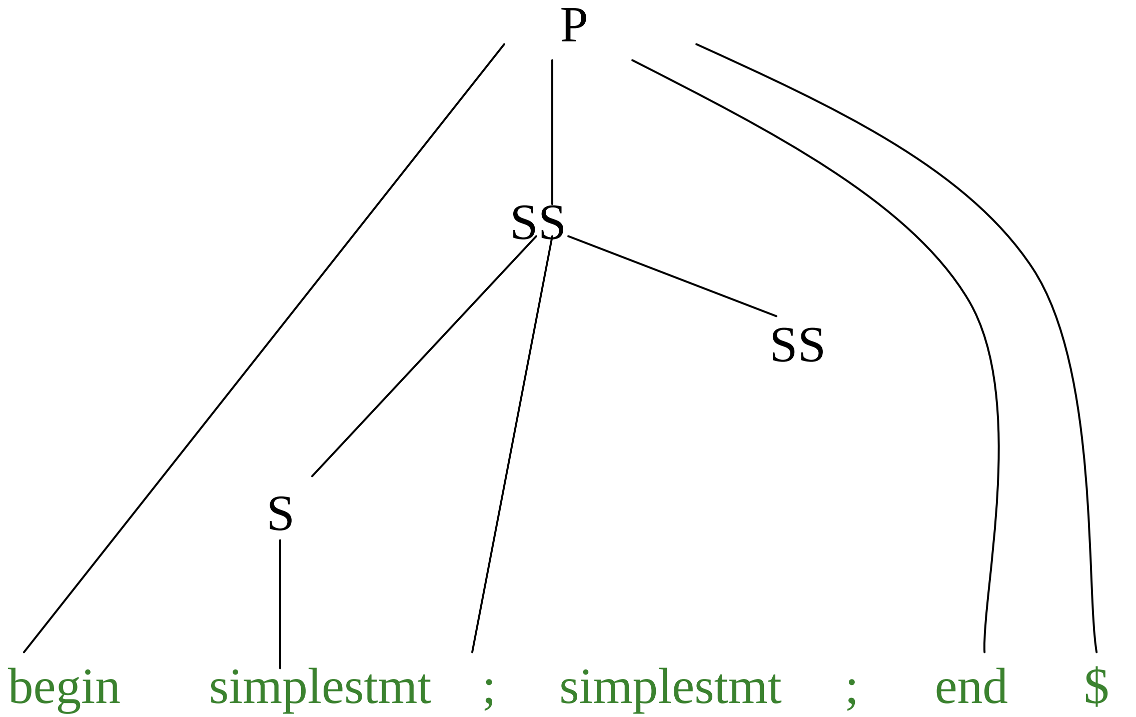
Top Down (LL) Parsing



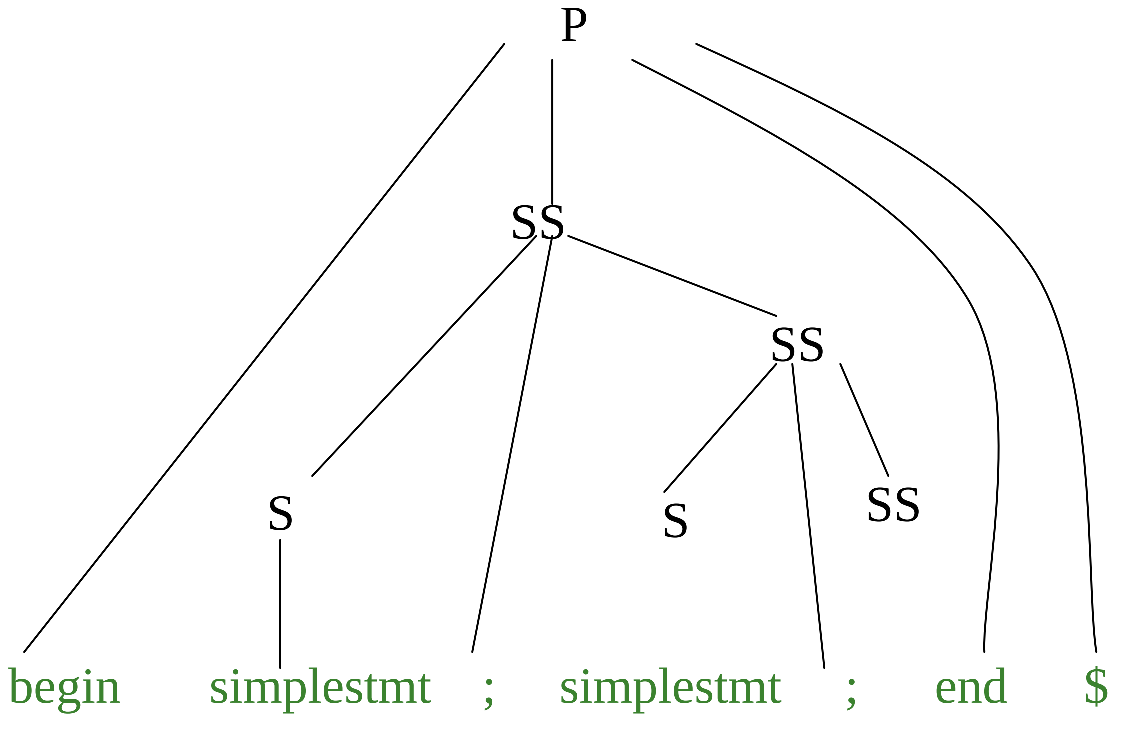
Top Down (LL) Parsing



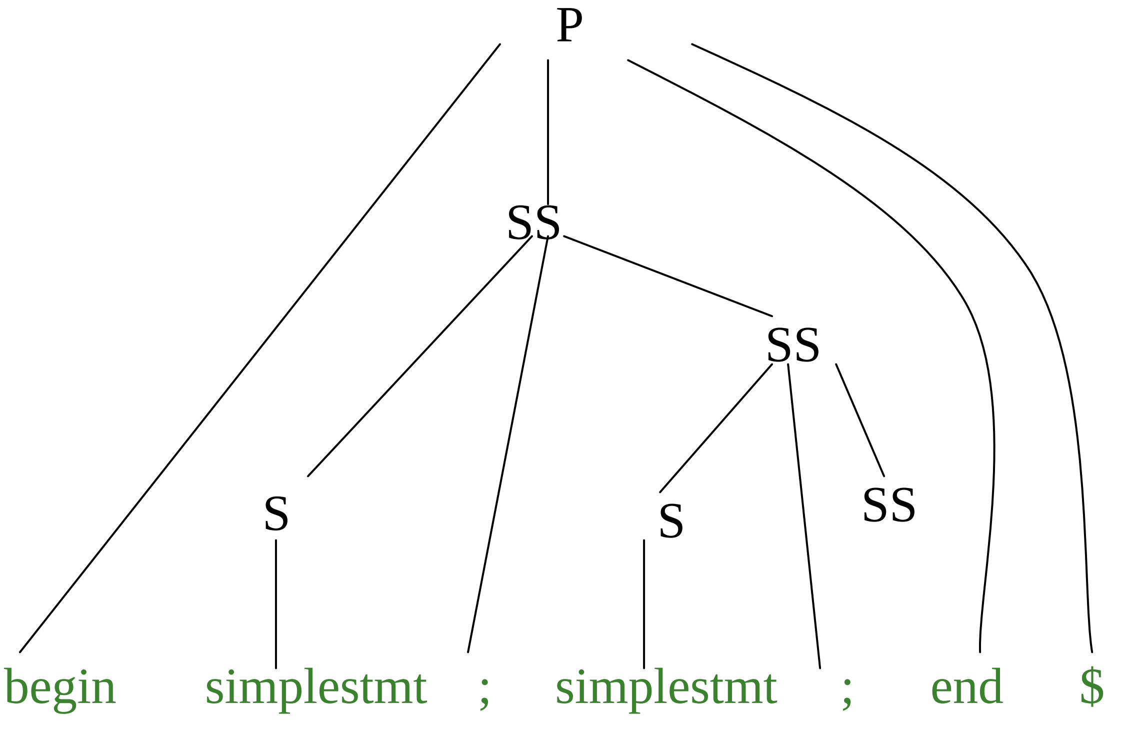
Top Down (LL) Parsing



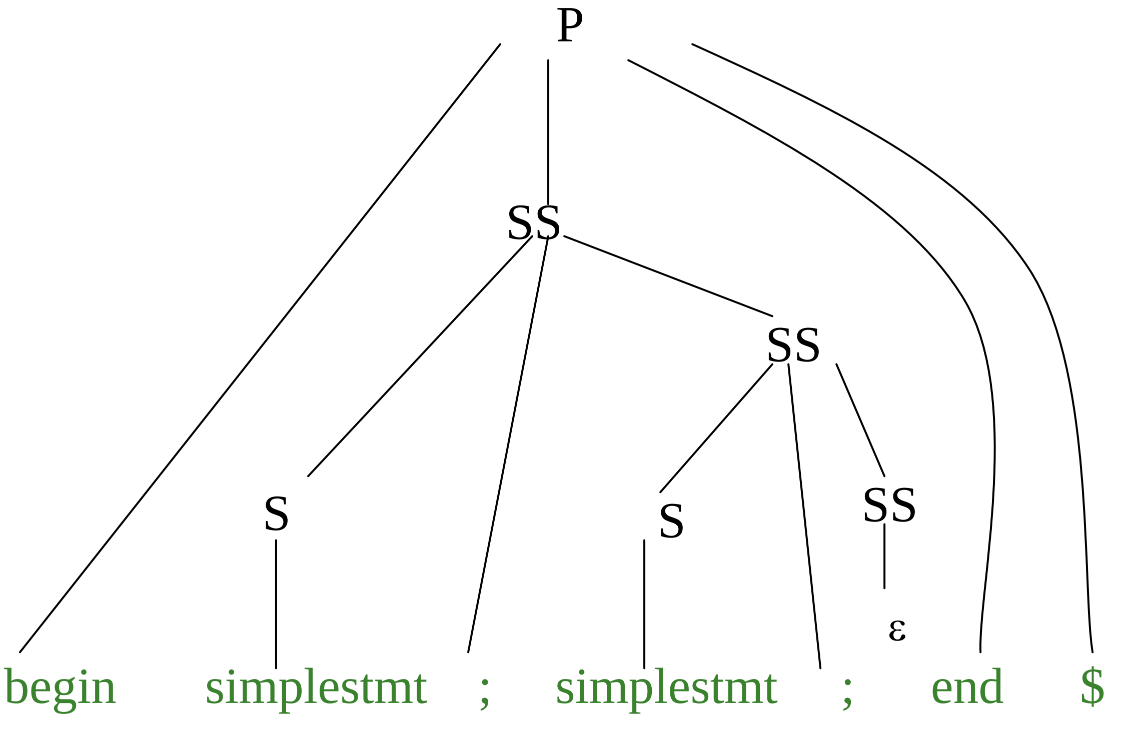
Top Down (LL) Parsing



Top Down (LL) Parsing



Top Down (LL) Parsing



Grammar

S → a B
  | b C
B → b b C
C → c c

Two strings in the language: abbcc and bcc

Can choose between them based on the first character of the input.

LL(*k*) parsing

- Process input *k* symbols at a time.
- Initially, current non-terminal is start symbol.
- Algorithm
  - Given next k input tokens and current non-terminal T, choose a rule R (T → …)
  - For each element X in rule R from left to right,

    if X is a non-terminal, call function for X

    else if symbol X is a terminal, see if next input symbol matches X; if so, update from the input
- Typically, we consider LL(1)

Two Approaches

- Recursive Descent parsing
  - Code tailored to the grammar
- Table Driven – predictive parsing
  - Table tailored to the grammar
  - General Algorithm

Writing a Recursive Descent Parser

- Procedure for each non-terminal.

  Use next token (lookahead) to choose which production to mimic.
  - for non-terminal X, call procedure X()
  - for terminals X, call ‘match(X)’
- ```
  match(symbol) {
    if (symbol = lookahead)
        lookahead = yylex()
    else error()  }
  ```
- Call `yylex()` before the first call to get first lookahead.
  ```

Back to grammar

```
S() {
    if (lookahead==a) { match(a);B(); }
   else if (lookahead == b) { match(b);
     C(); }
   else error("expecting a or b");
}
B() {match(b); match(b); C();}
C() { match(c) ; match(c) ;}

main() {
    lookahead==yylex();
   S();
}
```

S → a B

   | b C

B → b b C

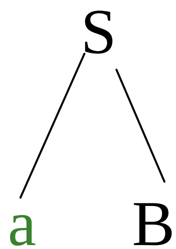
C → c c

Parsing abbcc

S

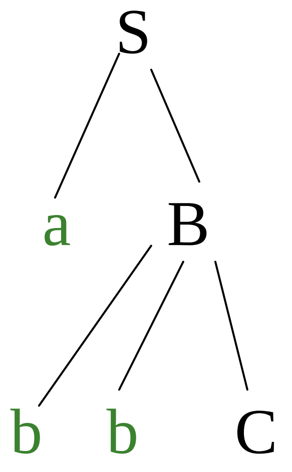
Remaining input: abbcc

Parsing abbcc



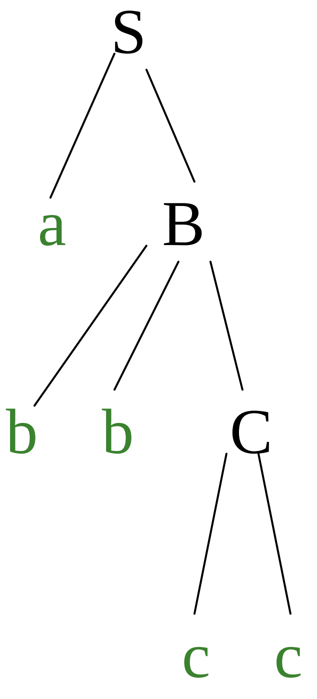
Remaining input: bbcc

Parsing abbcc



Remaining input: cc

Parsing abbcc



Remaining input:

How do we find the lookaheads?

- Can compute PREDICT sets from FIRST and FOLLOW
- PREDICT(A → α) =

  FIRST(α) – {ε} U FOLLOW(A) if ε in FIRST(α)

  FIRST(α) if ε not in FIRST(α)

NOTE: ε never in PREDICT sets

For LL(*k*) grammars, the PREDICT sets for a given non-terminal will be disjoint.

Example

Production	Predict
E → T E’	= FIRST(T) = {(,id}
E’ → + T E’	{+}
E’ → ε	= FOLLOW(E’) = {$,)}
T → F T’	= FIRST(F) = {(,id}
T’ → * F T’	{*}
T’ → ε	= FOLLOW(T’) = {+,$,)}
F → id	{id}
F → (E)	{(}

- FIRST(F) = {(,id}
- FIRST(T) = {(,id}
- FIRST(E) = {(,id}
- FIRST(T’) = {*,ε}
- FIRST(E’) = {+,ε}
- FOLLOW(E) = {$,)}
- FOLLOW(E’) = {$,)}
- FOLLOW(T) = {+$,)}
- FOLLOW(T’) = {+,$,)}
- FOLLOW(F) = {*,+,$,)}

```
E() {
  if (lookahead in {(,id}) T(); E_prime(); }                    E → T E'
  else error("(E) expecting ( or identifier");
}

E_prime() {
   if (lookahead in {+}) {match(+); T(); E_prime();}   E' → + T E'
   else if (lookahead in {),end_of_file}) return;          E' → ε
   else error("(E') expecting +, ) or end of file");
}

T() {
   if (lookahead in {(,id}) F(); T_prime(); }                 T → F T'
   else error("(T) expecting ( or identifier");
}
```

```
T_prime() {
   if (lookahead in {*}) {match(*); F(); T_prime();}   T' → * F T'
   else if (lookahead in {),end_of_file}) return;         T' → ε
   else error("(T') expecting *, ) or end of file"); }

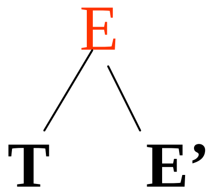
F() {
   if (lookahead in {id}) match(id);                      F → id
   else if (lookahead in {(}) match({); E(); match (}); }   F → ( E )
   else error("(F) expecting ( or identifier");}
```

Parsing a + b * c

E

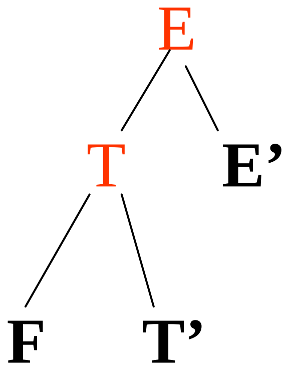
Remaining input: a+b*c

Parsing a + b * c



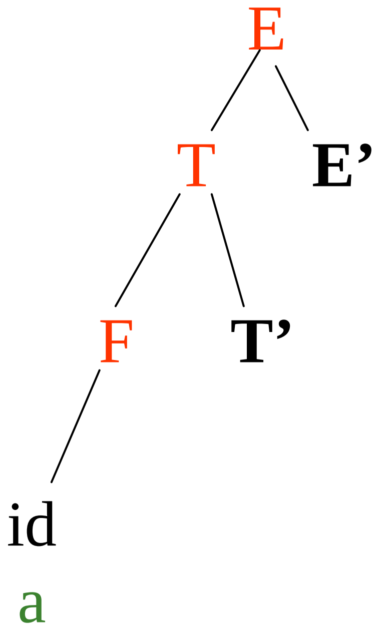
Remaining input: a+b*c

Parsing a + b * c



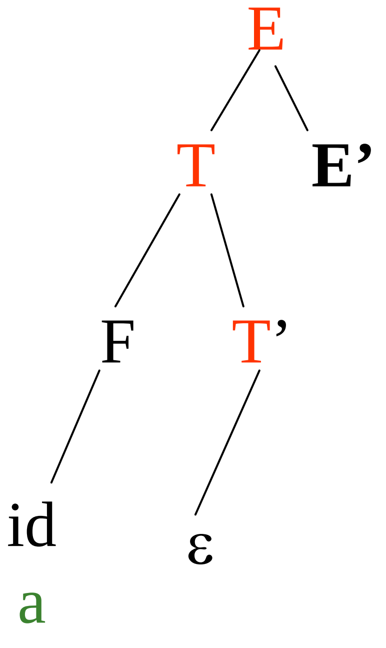
Remaining input: a+b*c

Parsing a + b * c



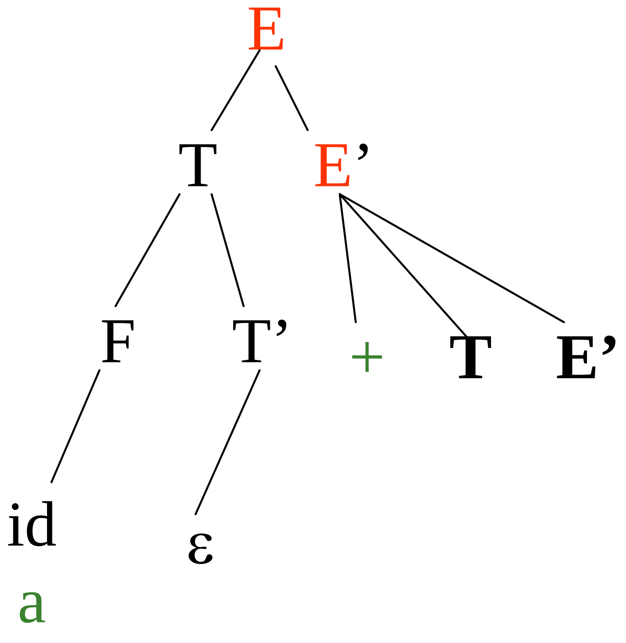
Remaining input: +b*c

Parsing a + b * c



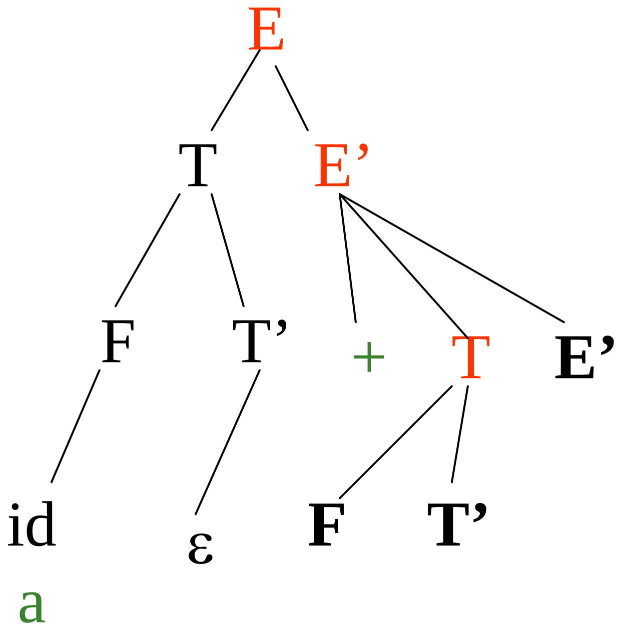
Remaining input: +b*c

Parsing a + b * c



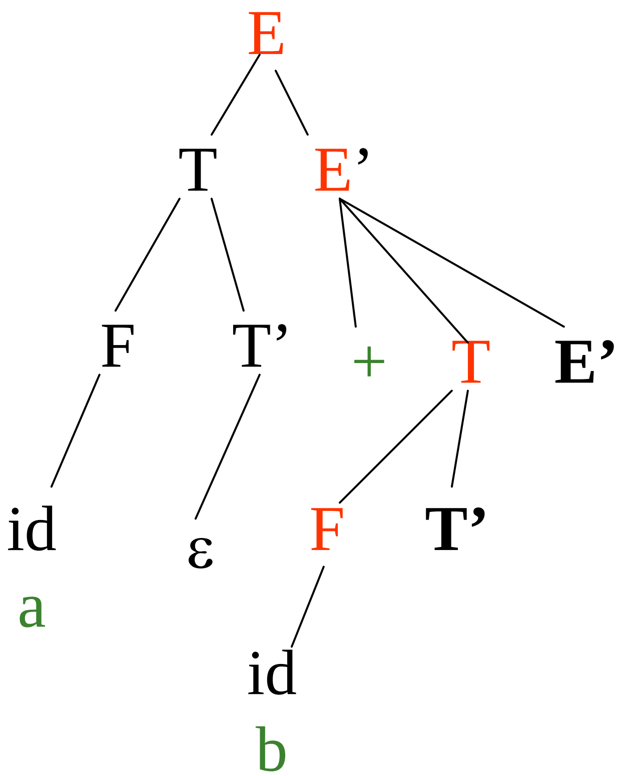
Remaining input: b*c

Parsing a + b * c



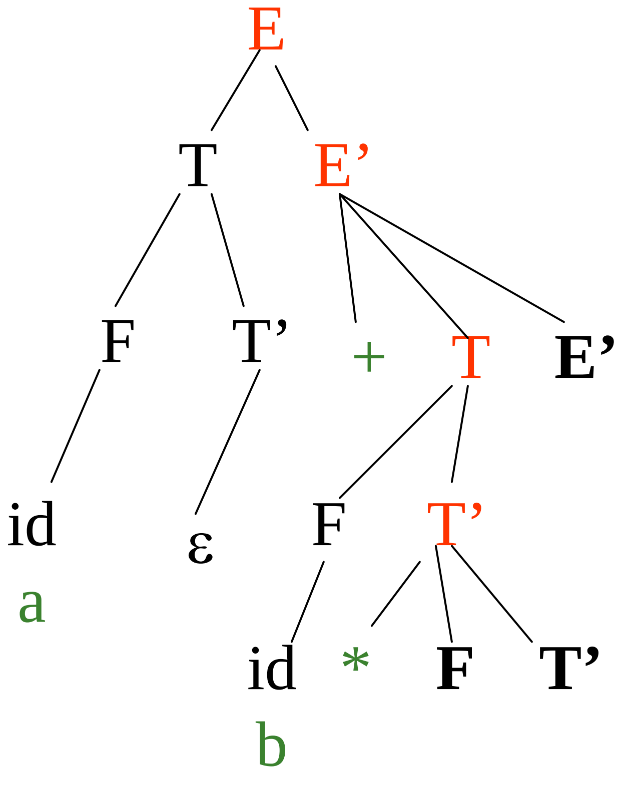
Remaining input: b*c

Parsing a + b * c



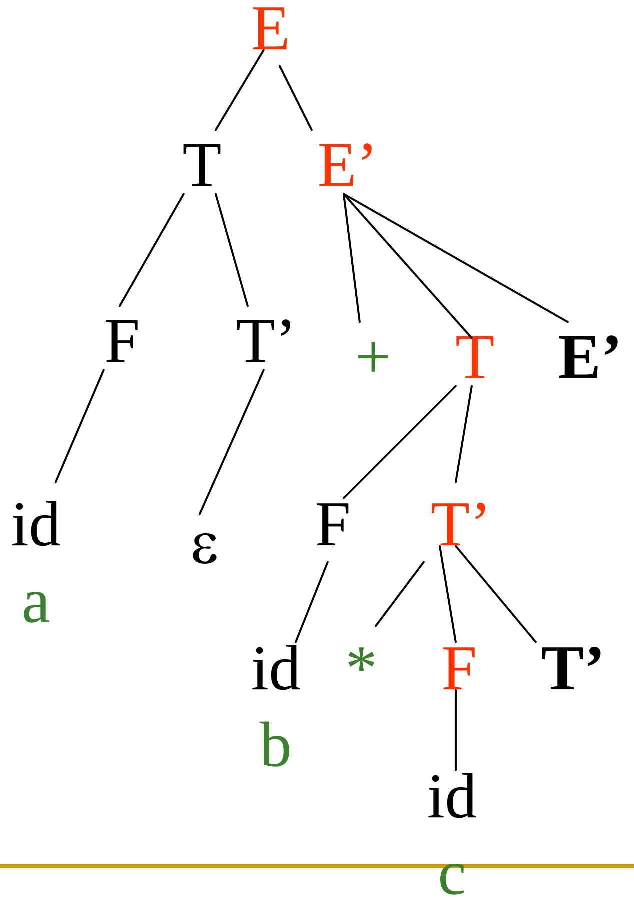
Remaining input: *c

Parsing a + b * c



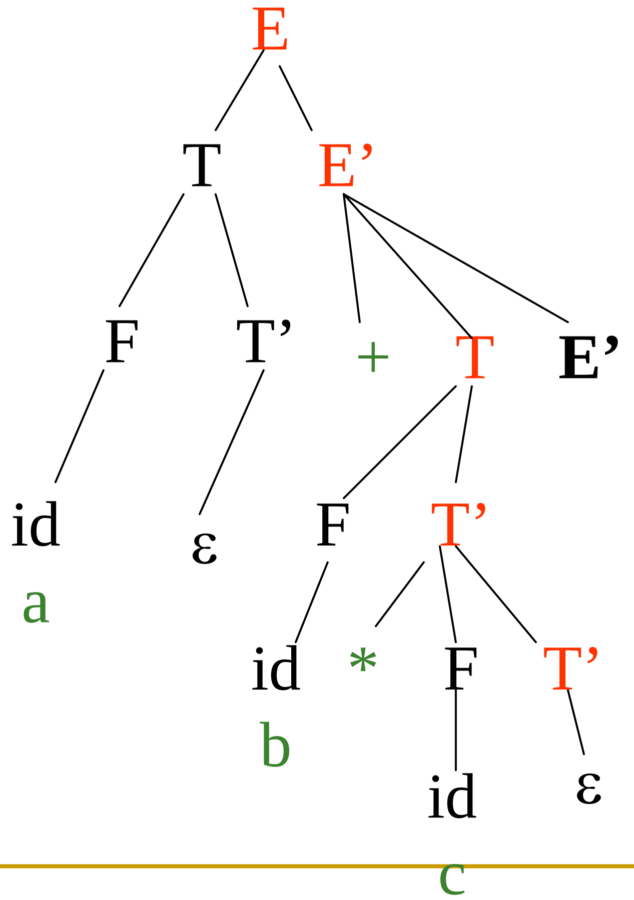
Remaining input: c

Parsing a + b * c



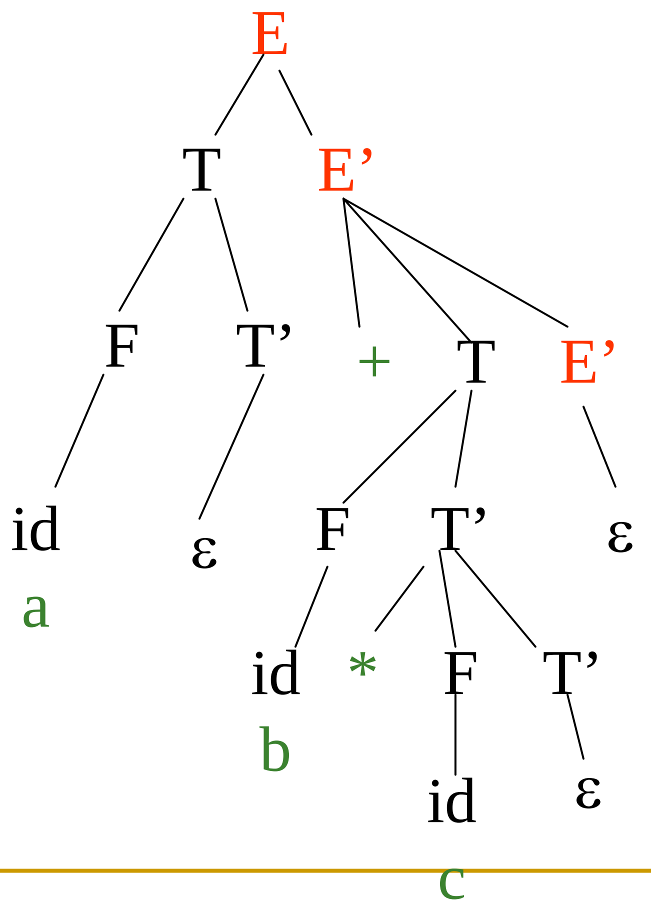
Remaining input:

Parsing a + b * c



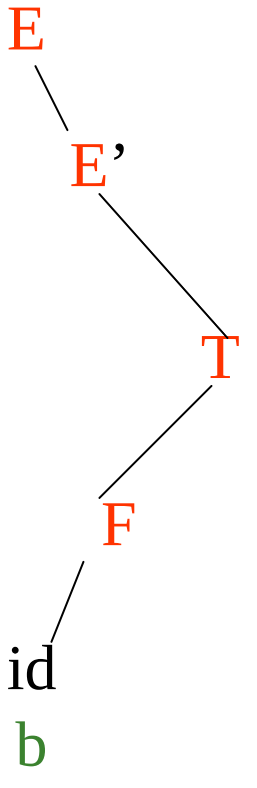
Remaining input:

Parsing a + b * c



Remaining input:

Stacks in Recursive Descent Parsing



- Runtime stack
- Procedure activations correspond to a path in parse tree from root to some interior node

LL(1) Predictive Parse Tables

An LL(1) Parse table is a mapping T: V_n x V_t → production P or error

1. For all productions A → α do
   - For each terminal a in Predict(A →α),
     T[A][a] = A → α
2. Every undefined table entry is an error.

Using LL(1) Parse Tables

ALGORITHM

INPUT: token sequence to be parsed, followed by '$' (end of file)

DATA STRUCTURES:

- Parse stack: Initialized by pushing '$' and then pushing the start symbol
- Parse table T

```
push($); push(start_symbol); lookahead = yylex()
repeat
  X = pop(stack)
  if X is a terminal symbol or $ then
    if X = lookahead then
        lookahead = yylex()
    else error()
  else  /* X is non-terminal */
    if T[X][lookahead] = X → Y1 Y2 …Ym
         push(Ym) … push (Y1)
    else error()
 until X = $ token
```

Expression Grammar

NT/T	+	*	(	)	ID	$
E			→ T E’		→ T E’	
E’	→ + T E’			→ ε		→ ε
T			→ F T’		→ F T’	
T’	→ ε	→ * F T’		→ ε		→ ε
F			→ (E)		→ ID	

Parsing a + b * c

Stack	Input	Action
$E	a+b*c$	E → T E’
$E’T	a+b*c$	T → F T’
$E’T’F	a+b*c$	F → id
$E’T’id	a+b*c$	match
$E’T’	+b*c$	T’ → ε
$E’	+b*c$	E’ → + T E’
$E’T+	+b*c$	match
$E’T	b*c$	T → F T’

Stack	Input	Action
$E’T’F	b*c$	F → id
$E’T’id	b*c$	match
$E’T’	*c$	T’ → * F T’
$E’T’F*	*c$	match
$E’T’F	c$	F → id
$E’T’id	c$	match
$E’T’	$	T’ → ε
$E’	$	E’ → ε
$	$	accept

Stack in Predictive Parsing

- Algorithm data structure
- Holds terminals and non-terminals from the grammar
  - terminals – still need to be matched from the input
  - non-terminals – still need to be expanded

Making a grammar LL(1)

- Not all context free languages have LL(1) grammars
- Can show a grammar is not LL(1) by looking at the predict sets
  - For LL(a) grammars, the PREDICT sets for a given non-terminal will be disjoint.

Example

Production	Predict
E → E + T	= FIRST(E) = {(,id}
E → T	= FIRST(T) = {(,id}
T → T * F	= FIRST(T) = {(,id}
T → F	= FIRST(F) = {(,id}
F → id	= {id}
F → (E)	= {(}

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

Two problems: E and T

Making a non-LL(1) grammar LL(1)

- Eliminate common prefixes

  Ex: A → B a C D | B a C E

- Transform left recursion to right recursion

  Ex: E → E + T | T

Eliminate Common Prefixes

- $A \rightarrow \alpha\ \beta \mid \alpha\ \delta$

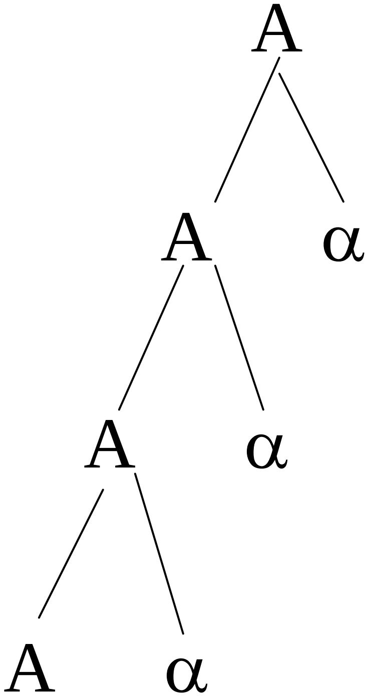
Can become:

$A \rightarrow \alpha\ A'$

$A' \rightarrow \beta \mid \delta$

Doesn’t always remove the problem. *Why?*

Why is left recursion a problem?



Remove Left Recursion

$A \rightarrow A\ \alpha_1 \mid A\ \alpha_2 \mid \ \ldots \mid \beta_1 \mid \beta_2 \mid \ldots$

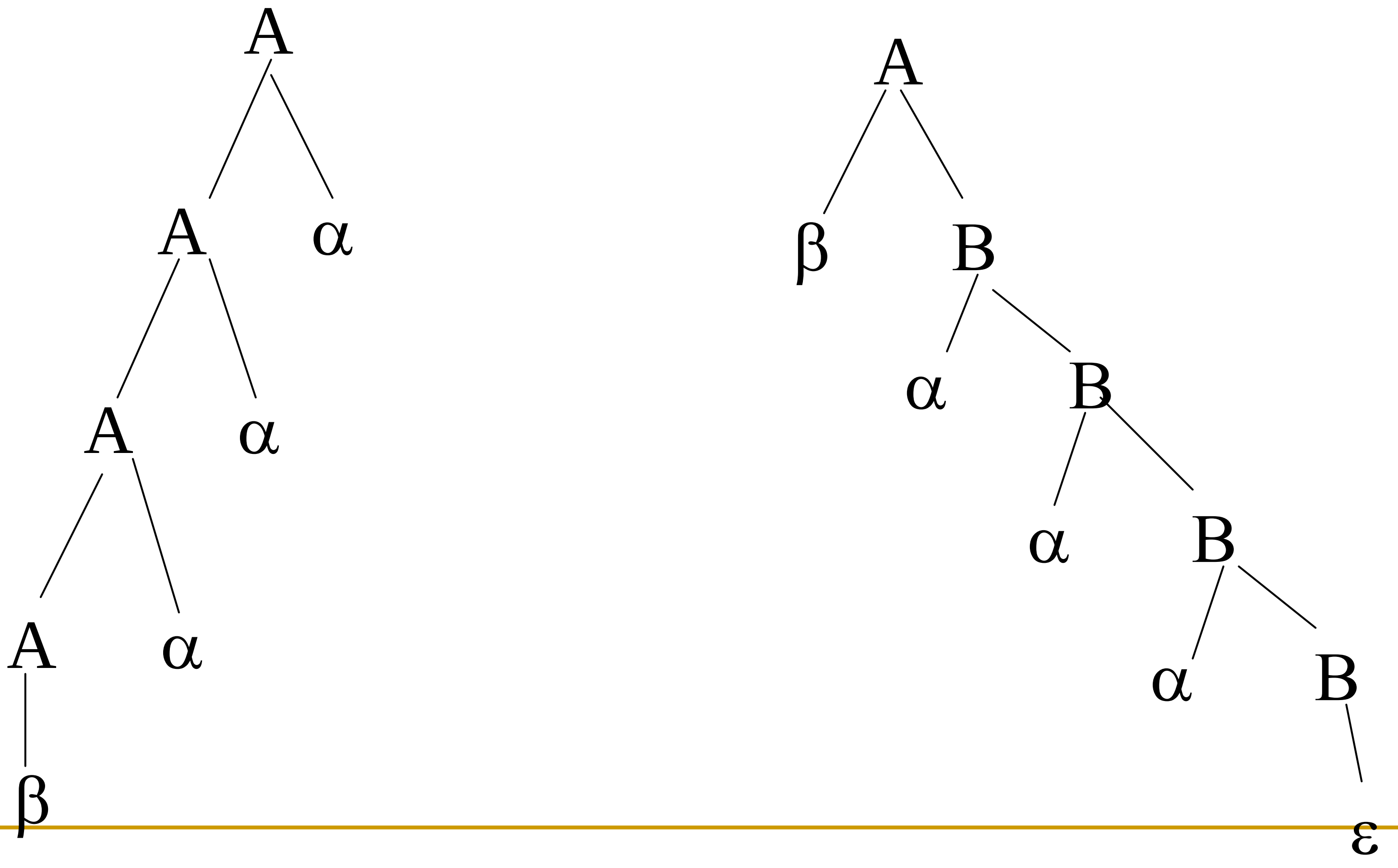
becomes

$A \rightarrow \beta_1\ A' \mid \beta_2\ A' \mid \ldots$

$A' \rightarrow \alpha_1\ A' \mid \alpha_2\ A' \mid \ldots \mid \varepsilon$

The left recursion becomes right recursion

$A \rightarrow A\ \alpha \mid \beta$ becomes $A \rightarrow \beta\ B$, $B \rightarrow \alpha\ B \mid \lambda$



Expression Grammar

- E → E + T | T

  T → T * F | F

  F → id | (E)   NOT LL(1)
- Eliminate left recursion:

  E → T E', E' → + T E' | ε

  T → F T', T' → * F T' | ε

  F → id | (E)

Non-Immediate Left Recursion

- Ex: $A_1 \rightarrow A_2\ a \mid b$

  $A_2 \rightarrow A_1\ c \mid A_2\ d$
- Convert to immediate left recursion
- Substitute A_1 in second set of productions by A_1's definition:

  $A_1 \rightarrow A_2\ a \mid b$

  $A_2 \rightarrow A_2\ a\ c \mid b\ c \mid A_2\ d$
- Eliminate recursion:

$A_1 \rightarrow A_2\ a \mid b$

$A_2 \rightarrow b\ c\ A_3$

$A_3 \rightarrow a\ c\ A_3 \mid d\ A_3 \mid \varepsilon$

Example

- A → B c | d
  B → C f | B f
  C → A e | g
- Rewrite: replace C in B
  B → A e f | g f | B f
- Rewrite: replace A in B
  B → B c e f | d e f | g f | B f

- Now grammar is:

  A → B c | d

  B → B c e f | d e f | g f | B f

  C → A e | g

- Get rid of left recursion (and C if A is start)

  A → B c | d

  B → d e f B' | g f B'

  B' → c e f B' | f B' | ε

Error Recovery in LL parsing

- Simple option: When see an error, print a message and halt
- “Real” error recovery
  - Insert “expected” token and continue – can have a problem with termination
  - Deleting tokens – for an error for non-terminal F, keep deleting tokens until see a token in follow(F).

For example:

```
E() {
  if (lookahead in {(,id}) T(); E_prime(); }
  else { printf("(E) expecting ( or identifier");
   while (lookahead != ( or $)  lookahead = yylex();
   }
}
```

E → T E'

Follow(E) = $)