flex

flex - fast lexical analyzer generator

- Flex is a tool for generating scanners/lexical analyzers
- Flex source is a table of regular expressions and corresponding program fragments.
- Generates `lex.yy.c` which defines a routine `yylex()`

Format of the Input File

- The flex input file consists of three sections, separated by a line with just %% in it:

```
definitions
%%
rules
%%
user code
```

Definitions Section

- The definitions section contains declarations of simple name definitions to simplify the scanner specification.
- Name definitions have the form:

```
name definition
```

- Example:

```
DIGIT     [0-9]
ID        [a-z][a-z0-9]*
```

Rules Section

- The rules section of the flex input contains a series of rules of the form:

```
pattern action
```

- Example:

```
{ID} printf( "An identifier: %s\n", yytext );
```

- The *yytext* and *yylength* variable.
- If action is empty, the matched token is discarded.

Action

- If the action contains a `'{'`, the action spans till the balancing `'}'` is found, as in C.
- An action consisting only of a vertical bar (`'|'`) means "same as the action for the next rule.“
- The *return* statement, as in C.
- In case no rule matches: simply copy the input to the standard output (A default rule).

A Simple Example

```
%{
   int num_lines = 0, num_chars = 0;
%}

%%
\n      ++num_lines; ++num_chars;
.       ++num_chars;

%%
main()  {
   yylex();
   printf( "# of lines = %d, # of chars = %d\n",
                num_lines, num_chars );
}
```

Programming Assignment 1

- Write a lexical analyzer using lex/flex to identify tokens of a typical C program. The program should be able to print series of token-ids for every lexical pattern that it recognizes. Please show the lex specification and the working of the lexical analyzer.

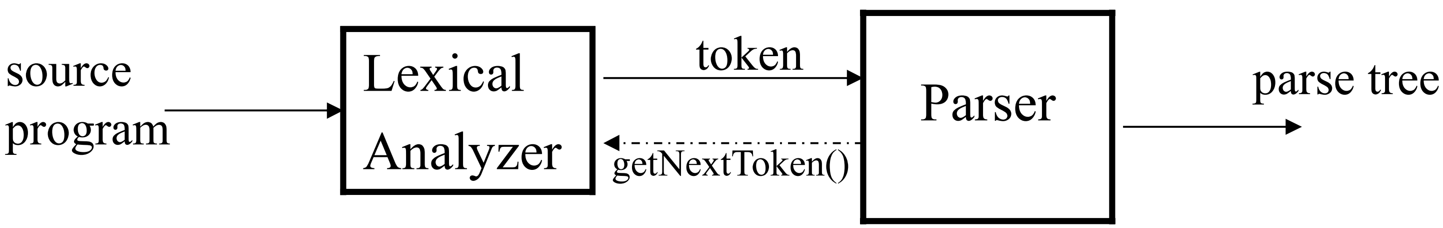
- Time Period : 2 weeks (deadline: 31st Jan 2021)

Syntax Analyzer

- *Syntax Analyzer* creates the syntactic structure of the given source program.
- This syntactic structure is mostly a *parse tree*.
- Syntax Analyzer is also known as *parser*.
- The syntax of a programming is described by a *context-free grammar (CFG)*. We will use BNF (Backus-Naur Form) notation in the description of CFGs.
- The syntax analyzer (parser) checks whether a given source program satisfies the rules implied by a context-free grammar or not.
  - If it satisfies, the parser creates the parse tree of that program.
  - Otherwise the parser gives the error messages.
- A context-free grammar
  - gives a precise syntactic specification of a programming language.
  - the design of the grammar is an initial phase of the design of a compiler.
  - a grammar can be directly converted into a parser by some tools (like yacc/bison)

Parser

- Parser works on a stream of tokens.
- The smallest item is a token.



Parsers (cont.)

- We categorize the parsers into two groups:

1. **Top-Down Parser**
   - the parse tree is created top to bottom, starting from the root.
2. **Bottom-Up Parser**
   - the parse is created bottom to top; starting from the leaves

- Both top-down and bottom-up parsers scan the input from left to right (one symbol at a time).
- Efficient top-down and bottom-up parsers can be implemented only for sub-classes of context-free grammars.
  - LL for top-down parsing
  - LR for bottom-up parsing

Context-Free Grammars

- Inherently recursive structures of a programming language are defined by a context-free grammar.
- In a context-free grammar, we have:
  - A finite set of terminals (in our case, this will be the set of tokens)
  - A finite set of non-terminals (syntactic-variables)
  - A finite set of productions rules in the following form
    - $A \rightarrow \alpha$ where A is a non-terminal and α is a string of terminals and non-terminals (including the empty string)
  - A start symbol (one of the non-terminal symbol)
- Example:

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid - E$$
$$E \rightarrow (E)$$
$$E \rightarrow id$$

Derivations

$E \Rightarrow E+E$

- E+E derives from E
  - we can replace E by E+E
  - to able to do this, we must have a production rule $E \rightarrow E+E$ in our grammar.

$E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+id$

- A sequence of replacements of non-terminal symbols is called a **derivation** of id+id from E.
- In general, a derivation step is

  $\alpha A \beta \Rightarrow \alpha\gamma\beta$ if there is a production rule $A \rightarrow \gamma$ in our grammar

  where α and β are arbitrary strings of terminal and non-terminal symbols

$\alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_n$ (α_n derives from α_1 or α_1 derives α_n)

$\Rightarrow$ derives in one step

$\overset{*}{\Rightarrow}$ derives in zero or more steps

$\overset{+}{\Rightarrow}$ derives in one or more steps

CFG - Terminology

- L(G) is *the language of G* (the language generated by G) which is a set of sentences.
- *A sentence of L(G)* is a string of terminal symbols of G.
- If S is the start symbol of G then

  ω is a sentence of L(G) iff $S \stackrel{+}{\Rightarrow} \omega$ where ω is a string of terminals of G.

- If G is a context-free grammar, L(G) is a *context-free language*.
- Two grammars are *equivalent* if they produce the same language.

- $S \stackrel{*}{\Rightarrow} \alpha$ - If α contains non-terminals, it is called as a *sentential* form of G.

  - If α does not contain non-terminals, it is called as a *sentence* of G

Note: α, β, γ and other initial Greek alphabets are used to denote string of terminals and non-terminals (sentential forms) while ω and other last Greek alphabets denote string of only terminals (sentence)

Derivation Example

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$

OR

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$

- At each derivation step, we can choose any of the non-terminal in the sentential form of G for the replacement.
- If we always choose the left-most non-terminal in each derivation step, this derivation is called as **left-most derivation**.
- If we always choose the right-most non-terminal in each derivation step, this derivation is called as **right-most derivation**.

Left-Most and Right-Most Derivations

Left-Most Derivation

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(id+E) \underset{lm}{\Rightarrow} -(id+id)$$

Right-Most Derivation

$$E \underset{rm}{\Rightarrow} -E \underset{rm}{\Rightarrow} -(E) \underset{rm}{\Rightarrow} -(E+E) \underset{rm}{\Rightarrow} -(E+id) \underset{rm}{\Rightarrow} -(id+id)$$

- We will see that the top-down parsers try to find the left-most derivation of the given source program.

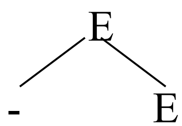
- We will see that the bottom-up parsers try to find the right-most derivation of the given source program in the reverse order.

Parse Tree

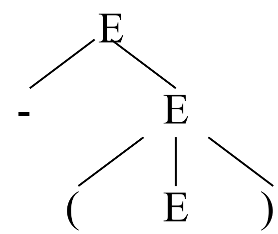
- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.

- A parse tree is a graphical representation of a derivation.

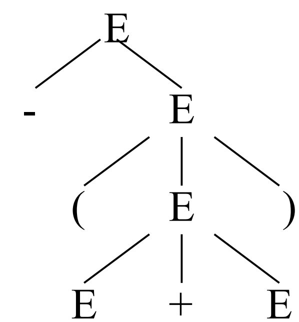
$E \Rightarrow -E$



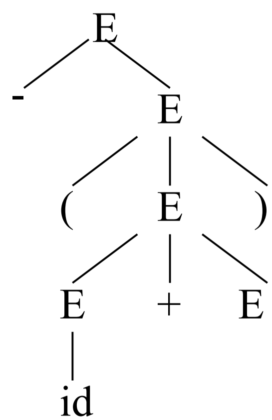
$\Rightarrow -(E)$



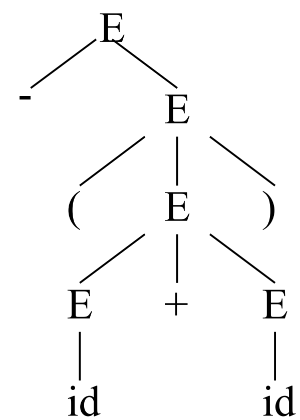
$\Rightarrow -(E+E)$



$\Rightarrow -(id+E)$



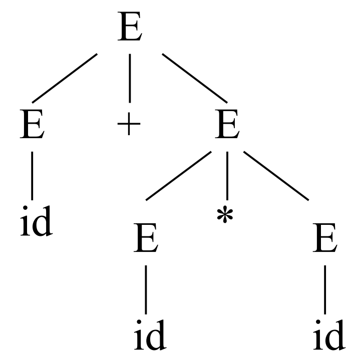
$\Rightarrow -(id+id)$



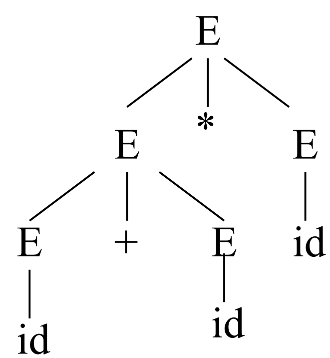
Ambiguity

- A grammar produces more than one parse tree for a sentence is called as an ***ambiguous*** grammar.

$$E \Rightarrow E{+}E \Rightarrow id{+}E \Rightarrow id{+}E{*}E$$
$$\Rightarrow id{+}id{*}E \Rightarrow id{+}id{*}id$$



$$E \Rightarrow E{*}E \Rightarrow E{+}E{*}E \Rightarrow id{+}E{*}E$$
$$\Rightarrow id{+}id{*}E \Rightarrow id{+}id{*}id$$



Ambiguity (cont.)

- For the most parsers, the grammar must be unambiguous.

- unambiguous grammar

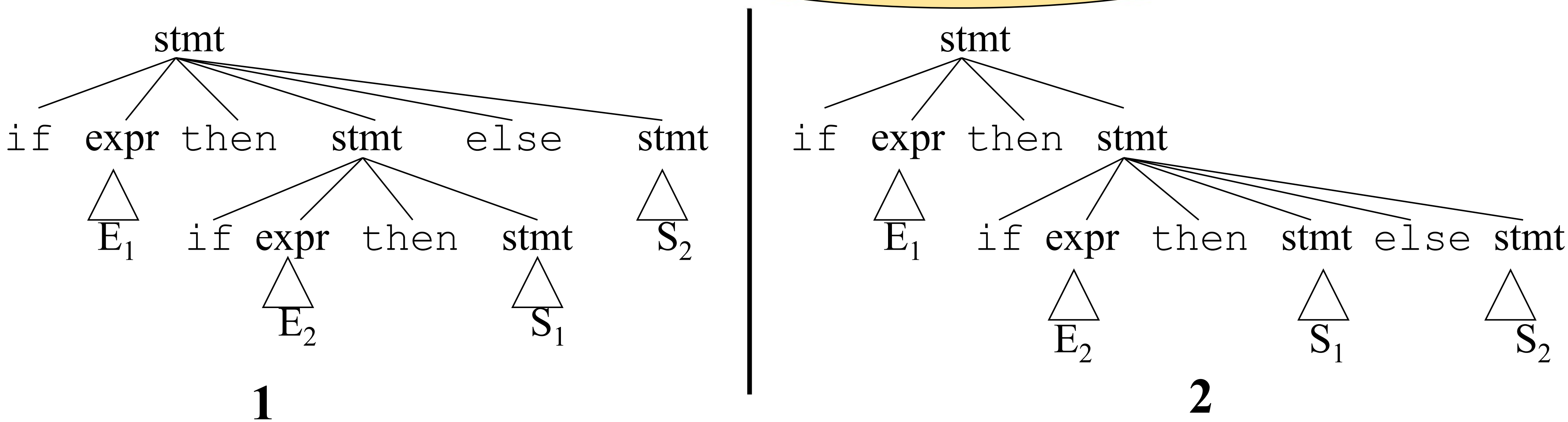
  ➔ unique selection of the parse tree for a sentence

- We should eliminate the ambiguity in the grammar during the design phase of the compiler.
- An unambiguous grammar should be written to eliminate the ambiguity
- We must prefer one of the parse trees of a sentence (generated by an ambiguous grammar) to disambiguate that grammar to restrict to this choice

Ambiguity (cont.)

stmt → `if` expr `then` stmt |
`if` expr `then` stmt `else` stmt | otherstmts

`if` E_1 `then` `if` E_2 `then` S_1 `else` S_2



Ambiguity (cont.)

- We prefer the second parse tree (else matches with closest if).
- So, we must disambiguate our grammar to reflect this choice.

- The unambiguous grammar will be:

stmt $\rightarrow$ matchedstmt | unmatchedstmt

matchedstmt $\rightarrow$ `if` expr `then` matchedstmt `else` matchedstmt | otherstmts

unmatchedstmt $\rightarrow$ `if` expr `then` stmt |
`if` expr `then` matchedstmt `else` unmatchedstmt

Ambiguity – Operator Precedence

- Ambiguous grammars (because of ambiguous operators) can be disambiguated according to the precedence and associativity rules.

$$E \rightarrow E+E \mid E*E \mid E\text{^}E \mid id \mid (E)$$

$\Downarrow$ disambiguate the grammar

precedence:

^ (right to left)

* (left to right)

\+ (left to right)

$$E \rightarrow E+T \mid T$$

$$T \rightarrow T*F \mid F$$

$$F \rightarrow G\text{^}F \mid G$$

$$G \rightarrow id \mid (E)$$

Left Recursion

- A grammar is ***left recursive*** if it has a non-terminal A such that there is a derivation:

  $A \stackrel{+}{\Rightarrow} A\alpha$ for some string α

- Top-down parsing techniques **cannot** handle left-recursive grammars
- So, we must convert left-recursive grammar into an equivalent grammar which is not left-recursive.
- The left-recursion may appear in a single step of the derivation (*immediate left-recursion*) or may appear in more than one step of the derivation.

Immediate Left-Recursion

$A \rightarrow A\,\alpha \mid \beta$ where β does not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta\, A'$

$A' \rightarrow \alpha\, A' \mid \varepsilon$ an equivalent grammar

In general,

$A \rightarrow A\,\alpha_1 \mid \ldots \mid A\,\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ where $\beta_1 \ldots \beta_n$ do not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta_1\, A' \mid \ldots \mid \beta_n\, A'$

$A' \rightarrow \alpha_1\, A' \mid \ldots \mid \alpha_m\, A' \mid \varepsilon$ an equivalent grammar

Immediate Left-Recursion -- Example

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id \mid (E)$

$\Downarrow$ eliminate immediate left recursion

$E \rightarrow T\, E'$

$E' \rightarrow +T\, E' \mid \varepsilon$

$T \rightarrow F\, T'$

$T' \rightarrow *F\, T' \mid \varepsilon$

$F \rightarrow id \mid (E)$

Left-Recursion -- Problem

- A grammar cannot be immediately left-recursive, but it still can be left-recursive.
- By just eliminating the immediate left-recursion, we may not get a grammar which is not left-recursive.

$S \rightarrow Aa \mid b$

$A \rightarrow Sc \mid d$ This grammar is not immediately left-recursive, but it is still left-recursive.

$\underline{S} \Rightarrow Aa \Rightarrow \underline{S}ca$ or

$\underline{A} \Rightarrow Sc \Rightarrow \underline{A}ac$ causes to a left-recursion

- So, we must eliminate all left-recursions from our grammar

Left-Factoring

- A predictive parser (a top-down parser without backtracking) insists that the grammar must be *left-factored*.

  grammar ➔ a new equivalent grammar suitable for predictive parsing

stmt $\rightarrow$ `if` expr `then` stmt `else` stmt |
`if` expr `then` stmt

- when we see `if`, we cannot know which production rule to choose to re-write *stmt* in the derivation.

Left-Factoring (cont.)

- In general,

  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ where α is non-empty and the first symbols of β_1 and β_2 (if they have one)are different.

- when processing α we cannot know whether expand

  A to $\alpha\beta_1$ or

  A to $\alpha\beta_2$

- But if we re-write the grammar as follows

  $A \rightarrow \alpha A'$

  $A' \rightarrow \beta_1 \mid \beta_2$ so, we can immediately expand A to $\alpha A'$

Left-Factoring -- Algorithm

- For each non-terminal A with two or more alternatives (production rules) with a common non-empty prefix, let say

$$A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m$$

convert it into

$$A \rightarrow \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_m$$
$$A' \rightarrow \beta_1 \mid \ldots \mid \beta_n$$

Left-Factoring – Example1

$A \rightarrow \underline{a}bB \mid \underline{a}B \mid cdg \mid cdeB \mid cdfB$

$\Downarrow$

$A \rightarrow aA' \mid \underline{cd}g \mid \underline{cd}eB \mid \underline{cd}fB$

$A' \rightarrow bB \mid B$

$\Downarrow$

$A \rightarrow aA' \mid cdA''$

$A' \rightarrow bB \mid B$

$A'' \rightarrow g \mid eB \mid fB$

Left-Factoring – Example2

$A \rightarrow ad \mid a \mid ab \mid abc \mid b$

$\Downarrow$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid b \mid bc$

$\Downarrow$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid bA''$

$A'' \rightarrow \varepsilon \mid c$

Non-Context Free Language Constructs

- There are some language constructions in the programming languages which are not context-free. This means that, we cannot write a context-free grammar for these constructions.

- L1 = { $\omega c \omega$ | ω is in (a|b)*} is not context-free

➔ declaring an identifier and checking whether it is declared or not later. We cannot do this with a context-free language. We need semantic analyzer (which is not context-free).

- L2 = {$a^n b^m c^n d^m$ | $n \geq 1$ and $m \geq 1$ } is not context-free

➔ declaring two functions (one with n parameters, the other one with m parameters), and then calling them with actual parameters.

Top-Down Parsing

- The parse tree is created top to bottom.
- Top-down parser
  - Recursive-Descent Parsing
    - Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
    - It is a general parsing technique, but not widely used.
    - Not efficient
    - Maybe bigger than LL(1) but may not terminate
  - Predictive Parsing
    - No backtracking
    - Efficient
    - Needs a special form of grammars (LL(1) grammars).
    - Recursive Predictive Parsing is a special form of Recursive Descent parsing without backtracking.
    - Non-Recursive (Table Driven) Predictive Parser is also known as LL(1) parser.

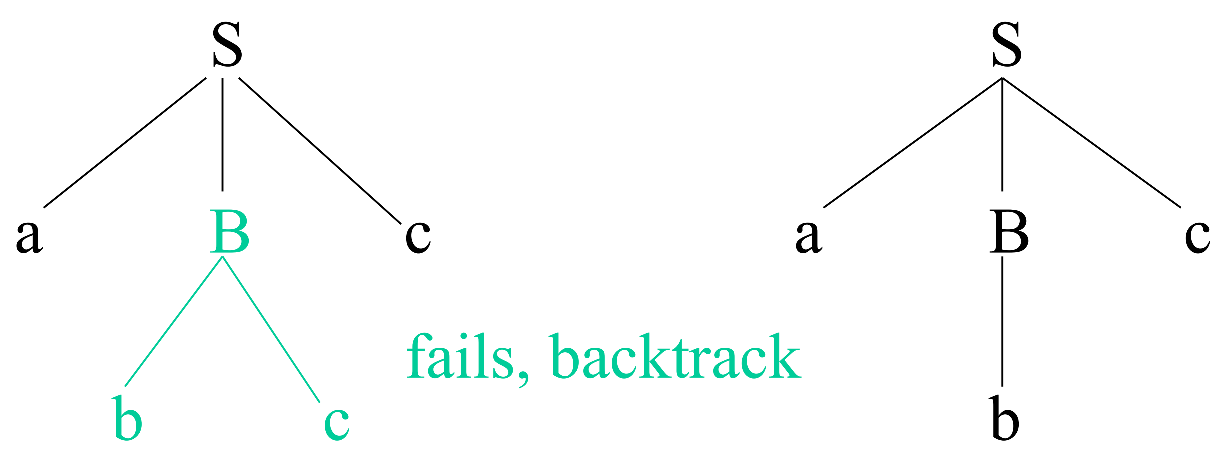
Recursive-Descent Parsing (uses Backtracking)

- Backtracking is needed.
- It tries to find the left-most derivation.

$S \rightarrow aBc$

$B \rightarrow bc \mid b$

input: abc



Predictive Parser

a grammar $\Rightarrow$ eliminate left recursion $\Rightarrow$ left factor $\Rightarrow$ a grammar suitable for predictive parsing (a LL(1) grammar)

- When re-writing a non-terminal in a derivation step, a predictive parser can uniquely choose a production rule by just looking the current symbol in the input string.

$A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$ input: ... a ....

current token

Recursive Predictive Parsing

- Each non-terminal corresponds to a procedure.

Ex: $A \rightarrow aBb$ (This is only the production rule for A)

```
proc A {
    - match the current token with a, and move to the next token;
    - call 'B';
    - match the current token with b, and move to the next token;
}
```

Recursive Predictive Parsing (cont.)

$A \rightarrow aBb \mid bAB$

```
proc A {
   case of the current token {
        'a':  - match the current token with a, and move to the next token;
              - call 'B';
              - match the current token with b, and move to the next token;
        'b':  - match the current token with b, and move to the next token;
              - call 'A';
              - call 'B';
   }
}
```

Recursive Predictive Parsing (cont.)

- When to apply ε-productions.

  $A \to aA \mid bB \mid \varepsilon$

- If all other productions fail, we should apply an ε-production. For example, if the current token is not a or b, we may apply the ε-production.
- Most correct choice: We should apply an ε-production for a non-terminal A when the current token is in the follow set of A (which terminals can follow A in the sentential forms).

Recursive Predictive Parsing (Example)

$A \rightarrow aBe \mid cBd \mid C$

$B \rightarrow bB \mid \varepsilon$

$C \rightarrow f$

```
proc C {   match the current token with f,
           and move to the next token; }

proc A {
    case of the current token {
        a:   - match the current token with a,
               and move to the next token;
             - call B;
             - match the current token with e,
               and move to the next token;
        c:   - match the current token with c,
               and move to the next token;
             - call B;
             - match the current token with d,
               and move to the next token;
        f:   - call C
    }
}

proc B {
    case of the current token {
        b:  - match the current token with b,
              and move to the next token;
            - call B
        e,d:  do nothing
    }
}
```

first set of C

follow set of B

Non-Recursive Predictive Parsing -- LL(1) Parser

- Non-Recursive predictive parsing is a table-driven parser.
- It is a top-down parser.
- It is also known as LL(1) Parser.

input buffer

↕

stack ⟷ Non-recursive Predictive Parser ⟶ output

↕

Parsing Table

LL(1) Parser

input buffer

– our string to be parsed. We will assume that its end is marked with a special symbol $.

output

– a production rule representing a step of the derivation sequence (left-most derivation) of the string in the input buffer.

stack

– contains the grammar symbols
– at the bottom of the stack, there is a special end marker symbol $.
– initially the stack contains only the symbol $ and the starting symbol S. $S ← initial stack
– when the stack is emptied (ie. only $ left in the stack), the parsing is completed.

parsing table

– a two-dimensional array M[A,a]
– each row is a non-terminal symbol
– each column is a terminal symbol or the special symbol $
– each entry holds a production rule.

LL(1) Parser – Parser Actions

- The symbol at the top of the stack (say X) and the current symbol in the input string (say a) determine the parser action.
- There are four possible parser actions.

1. If X and a are \$ ➔ parser halts (successful completion)
2. If X and a are the same terminal symbol (different from \$)

   ➔ parser pops X from the stack and moves the next symbol in the input buffer.
3. If X is a non-terminal

   ➔ parser looks at the parsing table entry M[X,a]. If M[X,a] holds a production rule $X \rightarrow Y_1 Y_2 \ldots Y_k$, it pops X from the stack and pushes $Y_k, Y_{k-1}, \ldots, Y_1$ into the stack. The parser also outputs the production rule $X \rightarrow Y_1 Y_2 \ldots Y_k$ to represent a step of the derivation.
4. none of the above ➔ error
   - all empty entries in the parsing table are errors.
   - If X is a terminal symbol different from a, this is also an error case.

LL(1) Parser – Example1

$S \rightarrow aBa$

$B \rightarrow bB \mid \varepsilon$

	a	**b**	**$**
S	$S \rightarrow aBa$		
B	$B \rightarrow \varepsilon$	$B \rightarrow bB$	

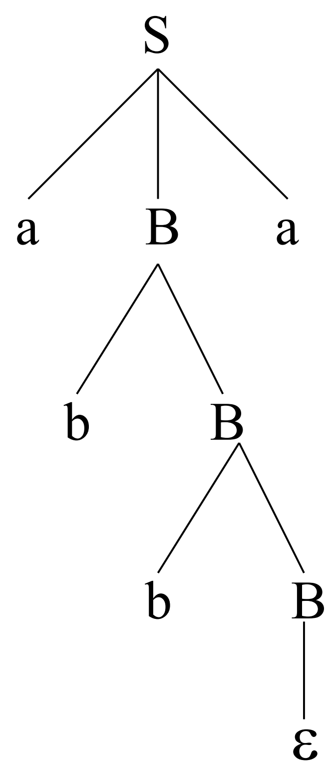
LL(1) Parsing Table

stack	**input**	**output**
$S	abba$	$S \rightarrow aBa$
$aBa	abba$	
$aB	bba$	$B \rightarrow bB$
$aBb	bba$	
$aB	ba$	$B \rightarrow bB$
$aBb	ba$	
$aB	a$	$B \rightarrow \varepsilon$
$a	a$	
$	$	accept, successful completion

LL(1) Parser – Example1 (cont.)

Outputs: $S \rightarrow aBa$ $B \rightarrow bB$ $B \rightarrow bB$ $B \rightarrow \varepsilon$

Derivation(left-most): $S \Rightarrow aBa \Rightarrow abBa \Rightarrow abbBa \Rightarrow abba$



parse tree

LL(1) Parser – Example2

$E \rightarrow TE'$
$E' \rightarrow +TE' \ | \ \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \ | \ \varepsilon$
$F \rightarrow (E) \ | \ id$

	id	**+**	*****	**(**	**)**	**$**
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

LL(1) Parser – Example2

stack	input	output
$\$E$	$id+id\$$	$E \rightarrow TE'$
$\$E'T$	$id+id\$$	$T \rightarrow FT'$
$\$E'T'F$	$id+id\$$	$F \rightarrow id$
$\$E'T'id$	$id+id\$$	
$\$E'T'$	$+id\$$	$T' \rightarrow \varepsilon$
$\$E'$	$+id\$$	$E' \rightarrow +TE'$
$\$E'T+$	$+id\$$	
$\$E'T$	$id\$$	$T \rightarrow FT'$
$\$E'T'F$	$id\$$	$F \rightarrow id$
$\$E'T'id$	$id\$$	
$\$E'T'$	$\$$	$T' \rightarrow \varepsilon$
$\$E'$	$\$$	$E' \rightarrow \varepsilon$
$\$$	$\$$	accept

Constructing LL(1) Parsing Tables

- Two functions are used in the construction of LL(1) parsing tables:
  - FIRST FOLLOW

- **FIRST(α)** is a set of the terminal symbols which occur as first symbols in strings derived from α where α is any string of grammar symbols.
- if α derives to ε, then ε is also in FIRST(α) .

- **FOLLOW(A)** is the set of the terminals which occur immediately after (follow) the *non-terminal A* in the strings derived from the starting symbol.
  - a terminal a is in FOLLOW(A) if $S \overset{*}{\Rightarrow} \alpha A a \beta$
  - \$ is in FOLLOW(A) if $S \overset{*}{\Rightarrow} \alpha A$

Compute FIRST for Any String X

- If X is a terminal symbol ➔ FIRST(X)={X}
- If X is a non-terminal symbol and $X \rightarrow \varepsilon$ is a production rule
  ➔ ε is in FIRST(X).
- If X is a non-terminal symbol and $X \rightarrow Y_1Y_2..Y_n$ is a production rule
  ➔ if a terminal **a** in FIRST(Y_i) and ε is in all FIRST(Y_j) for j=1,...,i-1 then **a** is in FIRST(X).
  ➔ if ε is in all FIRST(Y_j) for j=1,...,n then ε is in FIRST(X).
- If X is ε ➔ FIRST(X)={ε}
- If X is $Y_1Y_2..Y_n$
  ➔ if a terminal **a** in FIRST(Y_i) and ε is in all FIRST(Y_j) for j=1,...,i-1 then **a** is in FIRST(X).
  ➔ if ε is in all FIRST(Y_j) for j=1,...,n then ε is in FIRST(X).

FIRST Example

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

FIRST(F) = { (,id}
FIRST(T′) = {*, ε}
FIRST(T) = { (,id}
FIRST(E′) = {+, ε}
FIRST(E) = { (,id}

FIRST(TE′) = { (,id}
FIRST(+TE′) = {+}
FIRST(ε) = {ε}
FIRST(FT′) = { (,id}
FIRST(*FT′) = {*}
FIRST(ε) = {ε}
FIRST((E)) = { (}
FIRST(id) = {id}

Compute FOLLOW (for non-terminals)

- If S is the start symbol ➔ \$ is in FOLLOW(S)

- if $A \rightarrow \alpha B\beta$ is a production rule
  ➔ everything in FIRST(β) is FOLLOW(B) except ε

- If ($A \rightarrow \alpha B$ is a production rule) or
  ($A \rightarrow \alpha B\beta$ is a production rule and ε is in FIRST(β))
  ➔ everything in FOLLOW(A) is in FOLLOW(B).

We apply these rules until nothing more can be added to any follow set.

FOLLOW Example

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

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

Constructing LL(1) Parsing Table -- Algorithm

- for each production rule $A \rightarrow \alpha$ of a grammar G
  - for each terminal a in FIRST(α)
    ➔ add $A \rightarrow \alpha$ to M[A,a]
  - If ε in FIRST(α)
    ➔ for each terminal a in FOLLOW(A) add $A \rightarrow \alpha$ to M[A,a]
  - If ε in FIRST(α) and \$ in FOLLOW(A)
    ➔ add $A \rightarrow \alpha$ to M[A,\$]

- All other undefined entries of the parsing table are error entries.

Constructing LL(1) Parsing Table -- Example

$E \to TE'$	FIRST(TE')={(,id}	➔ $E \to TE'$ into M[E,(] and M[E,id]
$E' \to +TE'$	FIRST($+TE'$)={+}	➔ $E' \to +TE'$ into M[E',+]
$E' \to \varepsilon$	FIRST(ε)={ε}	➔ none
	but since ε in FIRST(ε)	
	and FOLLOW(E')={\$,)}	➔ $E' \to \varepsilon$ into M[E',\$] and M[E',)]
$T \to FT'$	FIRST(FT')={(,id}	➔ $T \to FT'$ into M[T,(] and M[T,id]
$T' \to *FT'$	FIRST($*FT'$)={*}	➔ $T' \to *FT'$ into M[T',*]
$T' \to \varepsilon$	FIRST(ε)={ε}	➔ none
	but since ε in FIRST(ε)	
	and FOLLOW(T')={\$,),+}	➔ $T' \to \varepsilon$ into M[T',\$], M[T',)] and M[T',+]
$F \to (E)$	FIRST((E))={(}	➔ $F \to (E)$ into M[F,(]
$F \to id$	FIRST(id)={id}	➔ $F \to id$ into M[F,id]

LL(1) Grammars

- A grammar whose parsing table has no multiply-defined entries is said to be LL(1) grammar.

one input symbol used as a look-head symbol do determine parser action

LL(1) left most derivation

input scanned from left to right

- The parsing table of a grammar may contain more than one production rule. In this case, we say that it is not a LL(1) grammar.

A Grammar which is not LL(1)

$S \rightarrow i\ C\ t\ S\ E \mid a$

$E \rightarrow e\ S \mid \varepsilon$

$C \rightarrow b$

FOLLOW(S) = { \$,e }

FOLLOW(E) = { \$,e }

FOLLOW(C) = { t }

FIRST(iCtSE) = {i}

FIRST(a) = {a}

FIRST(eS) = {e}

FIRST(ε) = {ε}

FIRST(b) = {b}

	a	**b**	**e**	**i**	**t**	**\$**
S	$S \rightarrow a$			$S \rightarrow iCtSE$		
E			$E \rightarrow e\ S$ $E \rightarrow \varepsilon$			$E \rightarrow \varepsilon$
C		$C \rightarrow b$				

two production rules for M[E,e]

Problem ➔ ambiguity

A Grammar which is not LL(1) (cont.)

- What do we have to do it if the resulting parsing table contains multiply defined entries?
  - If we didn't eliminate left recursion, eliminate the left recursion in the grammar.
  - If the grammar is not left factored, we have to left factor the grammar.
  - If its (new grammar's) parsing table still contains multiply defined entries, that grammar is ambiguous or it is inherently not a LL(1) grammar.
- A left recursive grammar cannot be a LL(1) grammar.
  - $A \rightarrow A\alpha \mid \beta$
    - ➔ any terminal that appears in FIRST(β) also appears FIRST($A\alpha$) because $A\alpha \Rightarrow \beta\alpha$.
    - ➔ If β is ε, any terminal that appears in FIRST(α) also appears in FIRST($A\alpha$) and FOLLOW(A).
- A grammar is not left factored, it cannot be a LL(1) grammar
  - $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$
    - ➔ any terminal that appears in FIRST($\alpha\beta_1$) also appears in FIRST($\alpha\beta_2$).
- An ambiguous grammar cannot be a LL(1) grammar.

Error Recovery in Predictive Parsing

- An error may occur in the predictive parsing (LL(1) parsing)
  - if the terminal symbol on the top of stack does not match with the current input symbol.
  - if the top of stack is a non-terminal A, the current input symbol is a, and the parsing table entry M[A,a] is empty.
- What should the parser do in an error case?
  - The parser should be able to give an error message (as much as possible meaningful error message).
  - It should be recover from that error case, and it should be able to continue the parsing with the rest of the input.

Error Recovery Techniques

- Panic-Mode Error Recovery
  - Skipping the input symbols until a synchronizing token is found.
- Phrase-Level Error Recovery
  - Each empty entry in the parsing table is filled with a pointer to a specific error routine to take care that error case.
- Error-Productions
  - If we have a good idea of the common errors that might be encountered, we can augment the grammar with productions that generate erroneous constructs.
  - When an error production is used by the parser, we can generate appropriate error diagnostics.
  - Since it is almost impossible to know all the errors that can be made by the programmers, this method is not practical.
- Global-Correction
  - Ideally, we we would like a compiler to make as few change as possible in processing incorrect inputs.
  - We have to globally analyze the input to find the error.
  - This is an expensive method, and it is not in practice.

Panic-Mode Error Recovery in LL(1) Parsing

- In panic-mode error recovery, we skip all the input symbols until a synchronizing token is found.
- What is the synchronizing token?
  - All the terminal-symbols in the follow set of a non-terminal can be used as a synchronizing token set for that non-terminal.
- So, a simple panic-mode error recovery for the LL(1) parsing:
  - All the empty entries are marked as ***synch*** to indicate that the parser will skip all the input symbols until a symbol in the follow set of the non-terminal A which on the top of the stack. Then the parser will pop that non-terminal A from the stack. The parsing continues from that state.
  - To handle unmatched terminal symbols, the parser pops that unmatched terminal symbol from the stack and it issues an error message saying that that unmatched terminal is inserted.

Panic-Mode Error Recovery - Example

$S \rightarrow AbS \mid e \mid \varepsilon$

$A \rightarrow a \mid cAd$

FOLLOW(S)={$}

FOLLOW(A)={b,d}

	a	**b**	**c**	**d**	**e**	**$**
S	$S \rightarrow AbS$	***sync***	$S \rightarrow AbS$	***sync***	$S \rightarrow e$	$S \rightarrow \varepsilon$
A	$A \rightarrow a$	***sync***	$A \rightarrow cAd$	***sync***	***sync***	***sync***

stack	**input**	**output**
$S	aab$	$S \rightarrow AbS$
$SbA	aab$	$A \rightarrow a$
$Sba	aab$	
$Sb	ab$	Error: missing b, inserted
$S	ab$	$S \rightarrow AbS$
$SbA	ab$	$A \rightarrow a$
$Sba	ab$	
$Sb	b$	
$S	$	$S \rightarrow \varepsilon$
$	$	accept

stack	**input**	**output**
$S	ceadb$	$S \rightarrow AbS$
$SbA	ceadb$	$A \rightarrow cAd$
$SbdAc	ceadb$	
$SbdA	eadb$	Error:unexpected e (illegal A)
(Remove all input tokens until first b or d, pop A)		
$Sbd	db$	
$Sb	b$	
$S	$	$S \rightarrow \varepsilon$
$	$	accept

Phrase-Level Error Recovery

- Each empty entry in the parsing table is filled with a pointer to a special error routine which will take care that error case.
- These error routines may:
  - change, insert, or delete input symbols.
  - issue appropriate error messages
  - pop items from the stack.
- We should be careful when we design these error routines, because we may put the parser into an infinite loop.