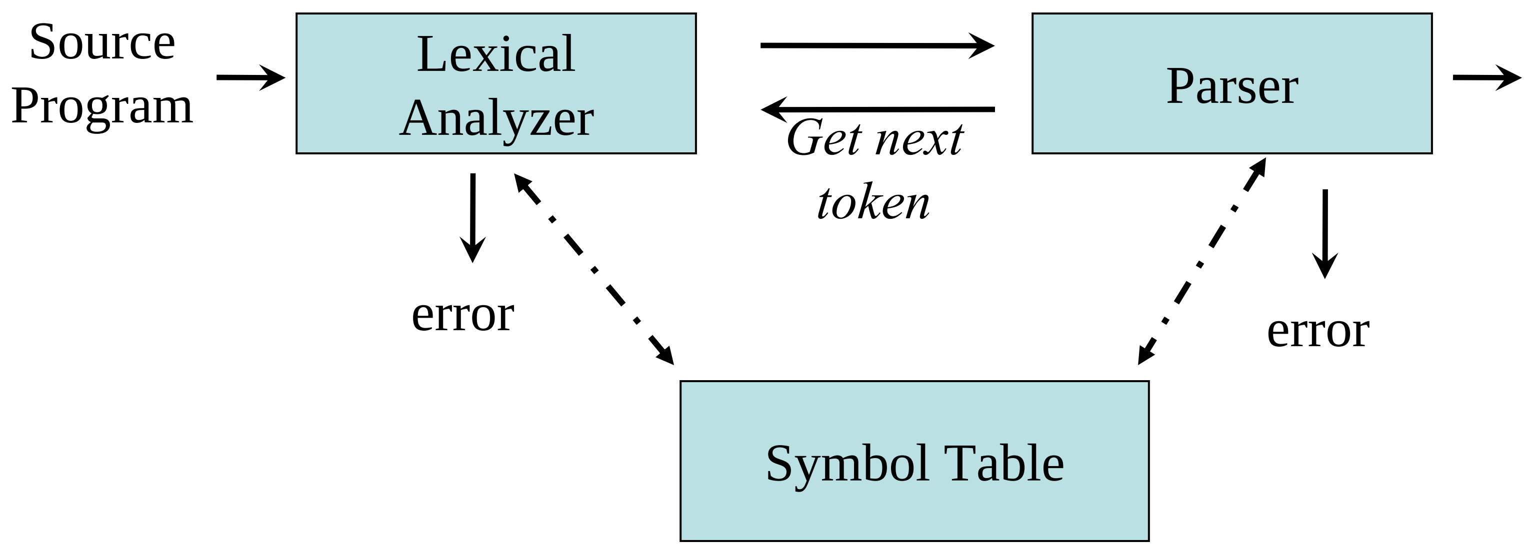
Syntax Analysis

Reading: Chapter 4

Deals with techniques for specifying and implementing parser

Parser



Syntax analyzer is also called the parser. Its job is to analyze the source program based on the definition of its syntax. It works in lock-step with the lexical analyzer and is responsible for creating a parse-tree of the source code.

Parser

A parser implements a Context-Free Grammar

The 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.

Parser

We categorize the parsers into two groups:

Top-Down Parser

the parse tree is created top to bottom, starting from the root.

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 (Recap)

Programming languages usually have recursive structures that can be defined by a context-free grammar (CFG).

CFGs are made of definitions of the form:

if S1 and S2 are statements and E is an expression, then

if E **then** S1 **else** S2 is a statement

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
  $\alpha \rightarrow \beta$
  where $\alpha \in (N \cup T)^* \, N \, (N \cup T)^*$ and $\beta \in (N \cup T)^*$
- $S \in N$ is a designated *start symbol*

CFG: Notational Conventions

Terminals are denoted by lower-case letters and symbols (single atoms) and **bold** strings (tokens)

$a,b,c,\ldots \in T$

specific terminals: **0**, **1**, **id**, **+**

Non-terminals are denoted by *lower-case italicized* letters or upper-case letters symbols

$A,B,C,\ldots \in N$

specific nonterminals: *expr*, *term*, *stmt*

Production rules are of the form $A \rightarrow \alpha$, that is read as “A can produce α”

Strings comprising of both terminals and non-terminals are denoted by greek letters (α, β etc.)

CFG: Derivations

$E \Rightarrow E+E$ means $E+E$ derives from E

- » we can replace E by $E+E$
- » to able to do this, we have to 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

$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ (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: Derivations

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.

CFG: Derivations

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

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**.

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

If we always choose the right-most non-terminal in each derivation step, this derivation is called as **right-most derivation**.

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

CFG: Derivations Example

Grammar $G = (\{E\}, \{\mathbf{+},\mathbf{*},\mathbf{(},\mathbf{)},\mathbf{-},\mathbf{id}\}, P, E)$ with productions $P =$

$$E \rightarrow E \mathbf{+} E$$
$$E \rightarrow E \mathbf{*} E$$
$$E \rightarrow \mathbf{(}\, E \,\mathbf{)}$$
$$E \rightarrow \mathbf{-}\, E$$
$$E \rightarrow \mathbf{id}$$

Example derivations:

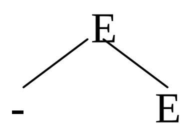
$$E \Rightarrow \mathbf{-}\, E \Rightarrow \mathbf{-}\,\mathbf{id}$$
$$E \Rightarrow_{rm} E \mathbf{+} E \Rightarrow_{rm} E \mathbf{+} \mathbf{id} \Rightarrow_{rm} \mathbf{id} \mathbf{+} \mathbf{id}$$
$$E \Rightarrow^{*} E$$
$$E \Rightarrow^{*} \mathbf{id} \mathbf{+} \mathbf{id}$$
$$E \Rightarrow^{+} \mathbf{id} \mathbf{*} \mathbf{id} \mathbf{+} \mathbf{id}$$

Parse Trees

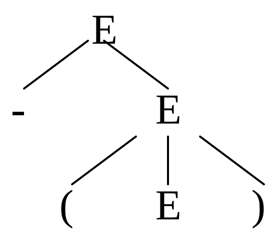
A Parse-tree is a graphical representation of a CFG derivation.

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

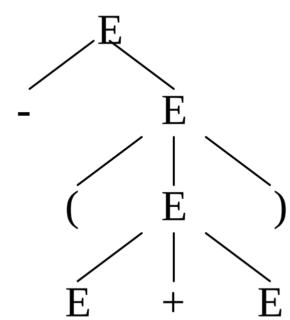
$E \Rightarrow -E$



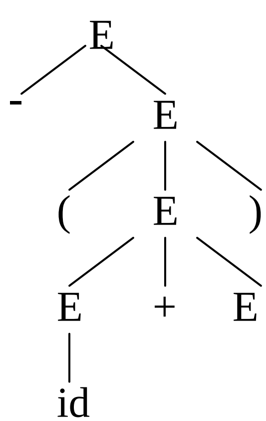
$\Rightarrow -(E)$



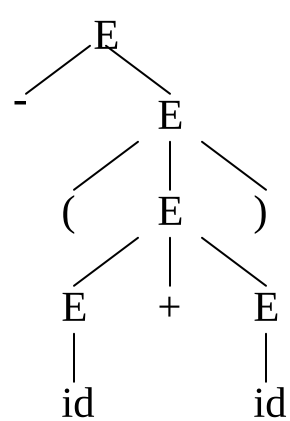
$\Rightarrow -(E+E)$



$\Rightarrow -(id+E)$



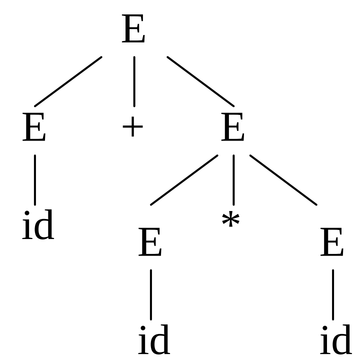
$\Rightarrow -(id+id)$



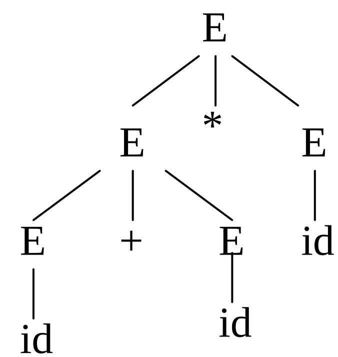
Ambiguity

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

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



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



Parsing

Given a stream of input tokens, *parsing* involves the process of "reducing" them to a non-terminal. The input string is said to represent the non-terminal it was reduced to.

Parsing can be either *top-down* or *bottom-up*.

Top-down parsing involves generating the string starting from the first non-terminal and repeatedly applying production rules.

Bottom-up parsing involves repeatedly rewriting the input string until it ends up in the first non-terminal of the grammar.

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
- Predictive Parsing
  - no backtracking
  - Efficient
  - Use LL (Left-to-right, Leftmost derivation) methods
  - 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

Backtracking is needed.

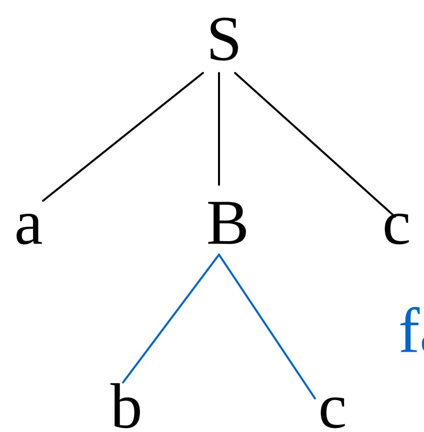
It tries to find the left-most derivation.

Method: let input w = abc, initially create the tree of single node S. The left most node a match the first symbol of w, so advance the pointer to b and consider the next leaf B. Then expand B using first choice bc. There is match for b and c, and advanced to the leaf symbol c of S, but there is no match in input, report failure and go back to B to find another alternative b that produce match.

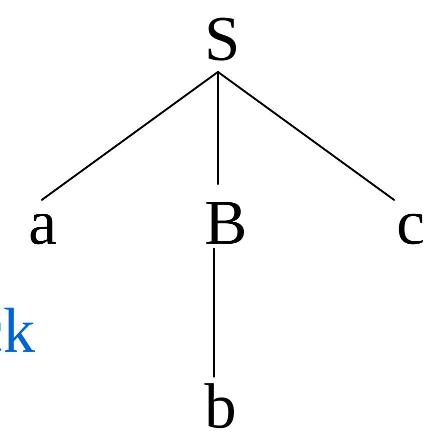
$S \rightarrow aBc$

$B \rightarrow bc|b$

input: abc



fails, backtrack



A left-recursive grammar can causes a recursive-decent parser to go into a infinite loop

Left Recursion

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

$A \Rightarrow A\alpha$ for some string α

Top-down parsing techniques **cannot** handle left-recursive grammars.

So, we have to convert our 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

$E \rightarrow T E'$

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

$T \rightarrow F T'$

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

$F \rightarrow id \mid (E)$

Non-Immediate Left-Recursion

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 have to eliminate all left-recursions from our grammar

Eliminate Left-Recursion - Algorithm

Input: Grammar G with no cycles or ε-productions

Output: An equivalent grammar with no left-recursion (but may have ε-productions)

Arrange non-terminals in some order: $A_1 \dots A_n$

for i **from** 1 **to** n **do** {

for j **from** 1 **to** i-1 **do** {

replace each production

$A_i \rightarrow A_j \gamma$

by

$A_i \rightarrow \alpha_1 \gamma \mid \dots \mid \alpha_k \gamma$

where $A_j \rightarrow \alpha_1 \mid \dots \mid \alpha_k$

}

eliminate immediate left-recursions among A_i productions

}

Eliminate Left-Recursion - Example

$S \to Aa \mid b$
$A \to Ac \mid Sd \mid f$

- Let the order of non-terminals: S, A

for S:

- do not enter the inner loop.
- there is no immediate left recursion in S.

for A:

- Replace $A \to Sd$ with $A \to Aad \mid bd$
  So, we will have $A \to Ac \mid Aad \mid bd \mid f$
- Eliminate the immediate left-recursion in A

$$A \to bdA' \mid fA'$$
$$A' \to cA' \mid adA' \mid \varepsilon$$

So, the resulting equivalent grammar which is not left-recursive is:

$$S \to Aa \mid b$$
$$A \to bdA' \mid fA'$$
$$A' \to cA' \mid adA' \mid \varepsilon$$

Left-Factoring

When a nonterminal has two or more productions whose right-hand sides start with the same grammar symbols, the grammar is not LL(1) and cannot be used for predictive parsing

Replace productions

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

with

$$A \to \alpha\, A' \mid \gamma$$
$$A' \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

Predictive Parsing

A *predictive parser* tries to predict which production produces the least chances of a backtracking and infinite looping.

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.

Example

```
stmt  →   if ......      |
          while ......   |
          begin ......   |
          for .....
```

When we are trying to write the non-terminal *stmt*, if the current token is `if` we have to choose first production rule.

Predictive Parsing

Two variants:

- Recursive (recursive-descent parsing)
- Non-recursive (table-driven parsing)

Recursive Predictive Parsing

Each non-terminal corresponds to a procedure.

Ex: $A \rightarrow aBb \mid bAB$

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

When to apply ε-productions.

$$A \rightarrow aA \mid bB \mid \varepsilon$$

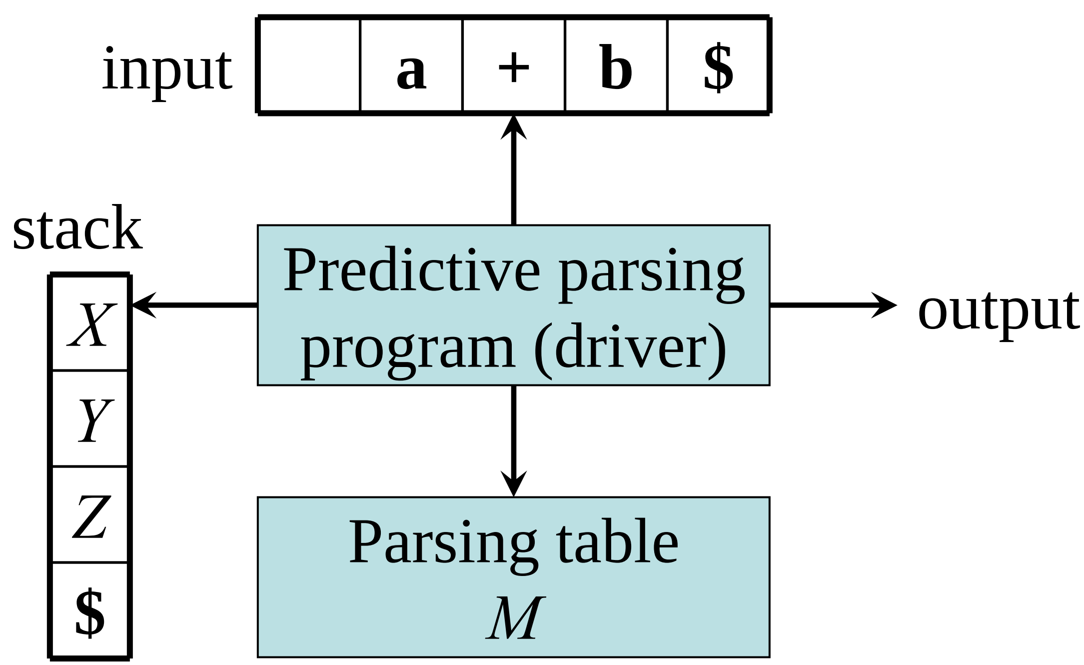
If all other productions fail, we should apply an ε-production. For example, if the current token is not a or b, we may apply the ε-production.

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).

Non-Recursive Predictive Parsing

Non-Recursive predictive parsing is a table-driven parser.

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*



Non-Recursive Predictive Parsing

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.
- 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.

Non-Recursive Predictive Parsing

Algorithm

Input : a string w.

Output: if w is in L(G), a leftmost derivation of w; otherwise error

1. Set *ip* to the first symbol of input stream
2. Set the stack to *$S* where *S* is the start symbol of the grammar
3. repeat

    Let *X* be the top stack symbol and *a* be the symbol pointed by *ip*

    If *X* is a terminal or *$* then

        if *X* = *a* then pop *X* from the stack and advance *ip*

        else error()

    else /* *X* is a non-terminal */

        if *M[X, a]* = *X* → *Y1, Y2, …, …, Yk* then

            pop *X* from stack

            push *Yk, Yk-1, …, …, Y1* onto stack (with *Y1* on top)

            output the production *X* → *Y1, Y2, …, Yk*

        else error()

4. until *X* = *$* /* stack is empty */

Non-Recursive Predictive Parsing

Example

$S \rightarrow aBa$
$B \rightarrow bB \mid \varepsilon$

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

LL(1) Parsing Table

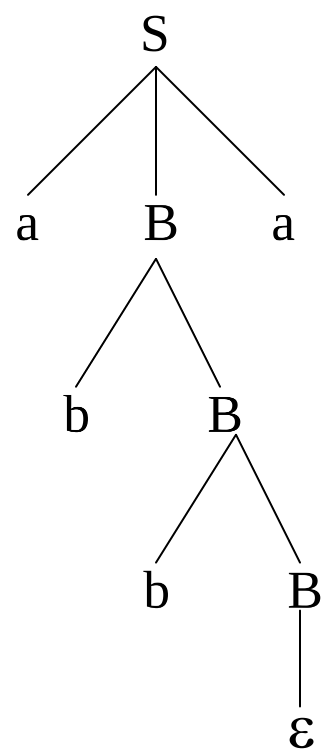
stack	**input**	**output**
$S	abba$	$S \rightarrow aBa$
$aBa	abba$	
$aB	bba$	$B \rightarrow bB$
$aBb	bba$	
$aB	ba$	$B \rightarrow bB$
$aBb	ba$	
$aB	a$	$B \rightarrow \varepsilon$
$a	a$	
$	$	accept, successful completion

Non-Recursive Predictive Parsing

Example

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

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



parse tree

Constructing LL(1) Parsing Tables

Eliminate left recursion from grammar

Left factor the grammar

a grammar $\xrightarrow{\text{eliminate left recursion}}$ $\xrightarrow{\text{left factor}}$ a grammar suitable for predictive parsing (a LL(1) grammar)

Compute FIRST and FOLLOW functions

Constructing LL(1) Parsing Tables
FIRST and 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 $S \stackrel{*}{\Rightarrow} \alpha A a \beta$
- \$ is in FOLLOW(A) if $S \stackrel{*}{\Rightarrow} \alpha A$

Constructing LL(1) Parsing Tables

Compute FIRST

1. If X is a terminal symbol then FIRST(X) = {X}
2. If X is a non-terminal symbol and $X \rightarrow \varepsilon$ is a production rule then
   FIRST(X) = FIRST(X) $\cup$ ε.
3. If X is a non-terminal symbol and $X \rightarrow Y_1 Y_2 .. Y_n$ is a production rule then
   a. if a terminal **a** in FIRST(Y_1) then FIRST(X) = FIRST(X) $\cup$ FIRST(Y_1)
   b. if a terminal **a** in FIRST(Y_i) and ε is in all FIRST(Y_j) for j=1,...,i-1 then
      FIRST(X) = FIRST(X) $\cup$ a.
   c. if ε is in all FIRST(Y_j) for j=1,...,n then FIRST(X) = FIRST(X) $\cup$ ε.

- If X is ε then FIRST(X)={ε}
- If X is $Y_1 Y_2 .. Y_n$
  a. if a terminal **a** in FIRST(Y_i) and ε is in all FIRST(Y_j) for j=1,...,i-1 then
     FIRST(X) = FIRST(X) $\cup$ a
  b. if ε is in all FIRST(Y_j) for j=1,...,n then FIRST(X) = FIRST(X) $\cup$ ε.

Constructing LL(1) Parsing Tables

Compute 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}

Constructing LL(1) Parsing Tables

Compute FOLLOW

Apply the following rules until nothing can be added to any FOLLOW set:

1. If S is the start symbol then \$ is in FOLLOW(S)
2. if $A \rightarrow \alpha B\beta$ is a production rule then everything in FIRST(β) is placed in FOLLOW(B) except ε
3. 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)) then everything in FOLLOW(A) is in FOLLOW(B).

Constructing LL(1) Parsing Tables

Compute 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 Tables
Algorithm

Input: LL(1) Grammar G

Output: Parsing Table M

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) then
            for each terminal a in FOLLOW(A)
                add $A \rightarrow \alpha$ to M[A,a]
        If ε in FIRST(α) and \$ in FOLLOW(A) then
            add $A \rightarrow \alpha$ to M[A,\$]

make all other undefined entries of the parsing table M be error

Constructing LL(1) Parsing Tables

Example

$E \rightarrow TE'$	FIRST(TE')={(,id}	$E \rightarrow TE'$ into M[E,(] and M[E,id]
$E' \rightarrow +TE'$	FIRST(+TE')={+}	$E' \rightarrow +TE'$ into M[E',+]
$E' \rightarrow \varepsilon$	FIRST(ε)={ε} but since ε in FIRST(ε) and FOLLOW(E')={$,)}	$E' \rightarrow \varepsilon$ into M[E',$] and M[E',)]
$T \rightarrow FT'$	FIRST(FT')={(,id}	$T \rightarrow FT'$ into M[T,(] and M[T,id]
$T' \rightarrow *FT'$	FIRST(*FT')={*}	$T' \rightarrow *FT'$ into M[T',*]
$T' \rightarrow \varepsilon$	FIRST(ε)={ε} but since ε in FIRST(ε) and FOLLOW(T')={$,),+}	$T' \rightarrow \varepsilon$ into M[T',$], M[T',)] and M[T',+]
$F \rightarrow (E)$	FIRST((E))={(}	$F \rightarrow (E)$ into M[F,(]
$F \rightarrow id$	FIRST(id)={id}	$F \rightarrow id$ into M[F,id]

LL(1) Grammars

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

What happen when a parsing table contains multiply defined entries ?
– The problem is ambiguity

A left recursive, not left factored and ambiguous grammar cannot be a LL(1) grammar (i.e. left recursive, not left factored and ambiguous grammar may have multiply –defined entries in parsing table)

There are no general rules by which multiply-defined entries can be made single-valued without affecting the language recognized by a grammar – therefore there should be LL(1) grammar as an input to construct the parsing table

Properties of LL(1) Grammars

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

LL(1) left most derivation

input scanned from left to right

A grammar G is LL(1) if and only if the following conditions hold for two distinctive production rules $A \rightarrow \alpha$ and $A \rightarrow \beta$

1. Both α and β cannot derive strings starting with same terminals.
2. At most one of α and β can derive to ε.
3. If β can derive to ε, then α cannot derive to any string starting with a terminal in FOLLOW(A).

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.

Exercise

Q. No. 4.1, 4.2, 4.3, 4.4, 4.5, 4.6, 4.11, 4.12, 4.14, 4.16 and 4.17