Top-down parsing using Recursive Descent
Top Down Parsing

• Begin with the start symbol and try and derive the parse tree from the root.

• Consider the grammar

\[ \text{Exp} \rightarrow \text{id} \]
\[ \mid \text{Exp} + \text{Exp} \]
\[ \mid \text{Exp} * \text{Exp} \]
\[ \mid (\text{Exp}) \]

• derives \( x, \ x+x, \ x+x+x, \ x * y, \ x + y * z \ldots \)
Example Parse (top down)

- Stack

```
input
Exp
```

```
Exp
```

```
Exp + Exp
```

```
Exp \* Exp
```

```
( Exp )
```

```
Exp
```

```
x + y \* z
```

```
Exp
```

```
x + y \* z
```

```
Exp + Exp
```

```
/  l  \n```

```
Exp  +  Exp
```
Example Parse (continued)

```
Exp → id
  | Exp + Exp
  | Exp * Exp
  | ( Exp )

Exp + Exp
  | Exp
  | y * z
  | id(x)
  | id

Exp
  | Exp + Exp
  | Exp * Exp
  | ( Exp )
```

Portland State University
Example Parse (continued)

Exp → id
    | Exp + Exp
    | Exp * Exp
    | ( Exp )

Exp
   / \
Exp + Exp
   / \
id(x)
Exp * Exp
Example Parse (continued)

\[
\begin{array}{c}
\text{Exp} & \rightarrow & \text{id} \\
& | & \text{Exp} + \text{Exp} \\
& | & \text{Exp} \times \text{Exp} \\
& | & (\text{Exp}) \\
\text{Exp} & \rightarrow & \text{id} \\
& | & \text{id}(x) \\
& | & \text{Exp} \times \text{Exp} \\
& | & \text{id}(y)
\end{array}
\]
Example Parse (continued)

```
Exp → id
    | Exp + Exp
    | Exp * Exp
    | ( Exp )
```

```
Exp
   /  \
Exp + Exp
   |   |
   /  \
id(x) Exp * Exp
   |   |
   |   |
id(y) id(z)
```
Problems with Top Down Parsing

- Backtracking may be necessary:
  
  \[ S \to ee \mid bAc \mid bAe \]
  
  \[ A \to d \mid cA \]

- try on string "bcde"

- Infinite loops possible from (indirect) left recursive grammars.

  \[ E \to E + id \mid id \]
Grammar Transformations

- Removing ambiguity
- Removing Left Recursion
- Backtracking and Factoring
Removing ambiguity

- Add levels to a grammar

\[
E \rightarrow E + E \mid E \ast E \mid \text{id} \mid (E)
\]

\[
E \rightarrow E + T \mid T
\]

\[
T \rightarrow T \ast F \mid F
\]

\[
F \rightarrow \text{id} \mid (E)
\]
Removing ambiguity

- The dangling else grammar.

\[
\text{stmt} \rightarrow \text{if exp then stmt else stmt} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{if exp then stmt} \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{id := exp}
\]

- Note that the following has two possible parses:

\[
\text{if } x=2 \text{ then if } x=3 \text{ then } y:=2 \text{ else } y := 4
\]

\[
\text{if } x=2 \text{ then (if } x=3 \text{ then } y:=2 \text{ ) else } y := 4
\]

\[
\text{if } x=2 \text{ then (if } x=3 \text{ then } y:=2 \text{ else } y := 4)
\]
Adding levels (cont)

• Original grammar

\[
\text{stmt} \rightarrow \text{if exp then stmt else stmt} \\
   \quad \mid \text{if exp then stmt} \\
   \quad \mid \text{id := exp}
\]

• Assume that every stmt between \textit{then} and \texttt{else} must be matched, i.e., it must have both a \textit{then} and an \texttt{else}.

• New Grammar with additional levels:

\[
\text{stmt} \rightarrow \text{match | unmatch} \\
\text{match} \rightarrow \text{if exp then match else match} \\
   \quad \mid \text{id := exp} \\
\text{unmatch} \rightarrow \text{if exp then stmt} \\
   \quad \mid \text{if exp then match else unmatch}
\]
Removing Left Recursion

• Top down recursive descent parsers require non-left recursive grammars

• Technique: *left factoring*
  
  \[
  E \rightarrow E + E \mid E \ast E \mid \text{id} \\
  E' \rightarrow + E E' \mid * E E' \mid \varepsilon
  \]
General Technique to remove direct left recursion

- For every variable with productions

\[ T \rightarrow T \, n \mid T \, m \]  
  \[ \mid a \mid b \]  
  \[ \text{(left recursive productions)} \]
  \[ \text{(non left recursive productions)} \]

1. Make a new variable \( T' \)

2. Remove the old productions

3. Add the following productions

\[ T \rightarrow a \, T' \mid b \, T' \]

\[ T' \rightarrow n \, T' \mid m \, T' \mid \varepsilon \]
Backtracking and Factoring

• Backtracking may be necessary:

\[
S \rightarrow \text{ee} \mid \text{bAc} \mid \text{bAe} \\
A \rightarrow \text{d} \mid \text{cA}
\]

• try on string “bcde”

\[
S \Rightarrow \text{bAc} \quad (\text{by } S \rightarrow \text{bAc}) \\
\Rightarrow \text{bcAc} \quad (\text{by } A \rightarrow \text{cA}) \\
\Rightarrow \text{bcdc} \quad (\text{by } A \rightarrow \text{d})
\]

• But this doesn’t match the input!
How to factor a grammar

• Combine productions with common prefixes; represent the different postfixes with a new variable

• Old grammar:

  \[
  S \rightarrow \text{ee} \mid \text{bAc} \mid \text{bAe} \\
  A \rightarrow \text{d} \mid \text{cA}
  \]

• Factored grammar:

  \[
  S \rightarrow \text{ee} \mid \text{bAQ} \\
  Q \rightarrow \text{c} \mid \text{e} \\
  A \rightarrow \text{d} \mid \text{cA}
  \]
Recursive Descent Parsing

• One procedure (function, method) for each variable.

• Procedures are often (mutually) recursive.

• Procedure can return a bool (true ⇒ the input matches that variable) or, more often, can return a data-structure (the input builds this parse tree)

• Usually depend on a “lexical analyzer” that is used to read the terminal symbols and “back up”.
Recursive Descent parser for REs

• Parser builds a value of the datatype:

```plaintext
datatype RE =
    Epsilon
  | Empty
  | Simple of string
  | Union of RE * RE
  | Concat of RE * RE
  | Closure of RE ;
```

• The lexical analyzer datatype:

```plaintext
datatype token =
    Done
  | Plus
  | Star
  | Hash
  | Zero
  | LeftParen
  | RightParen
  | Single of string
  | BadInput;
```
Ambiguous grammar

1. $\text{RE} \rightarrow \text{RE} + \text{RE}$
2. $\text{RE} \rightarrow \text{RE} \text{ RE}$
3. $\text{RE} \rightarrow \text{RE} \ast$
4. $\text{RE} \rightarrow \text{id}$
5. $\text{RE} \rightarrow \#$
6. $\text{RE} \rightarrow 0$
7. $\text{RE} \rightarrow ( \text{RE} )$
Ambiguous grammar

1. RE → RE + RE
2. RE → RE RE
3. RE → RE *
4. RE → id
5. RE → #
6. RE → 0
7. RE → ( RE )
Ambiguous grammar

1. RE → RE + RE
2. RE → RE RE
3. RE → RE *
4. RE → id
5. RE → #
6. RE → 0
7. RE → ( RE )
Ambiguous grammar

1. RE → RE + RE
2. RE → RE RE
3. RE → RE *
4. RE → id
5. RE → #
6. RE → 0
7. RE → ( RE )

• Transform grammar by layering
  • Tightest binding operators (*) at the lowest layer
  • Layers are alt, then concat, then closure, then simple.
Ambiguous grammar

1. RE → RE + RE
2. RE → RE RE
3. RE → RE *
4. RE → id
5. RE → #
6. RE → 0
7. RE → ( RE )

- Transform grammar by layering
  - Tightest binding operators (*) at the lowest layer
  - Layers are alt, then concat, then closure, then simple.

alt → alt + concat
Ambiguous grammar

1. RE → RE + RE
2. RE → RE RE
3. RE → RE *
4. RE → id
5. RE → #
6. RE → 0
7. RE → ( RE )

• Transform grammar by layering
  • Tightest binding operators (*) at the lowest layer
  • Layers are alt, then concat, then closure, then simple.

alt → alt + concat
alt → concat
Ambiguous grammar

1. $RE \rightarrow RE + RE$
2. $RE \rightarrow RE \ RE$
3. $RE \rightarrow RE \ *$
4. $RE \rightarrow id$
5. $RE \rightarrow #$
6. $RE \rightarrow 0$
7. $RE \rightarrow ( \ RE \ )$

- Transform grammar by layering
  - Tightest binding operators (*) at the lowest layer
  - Layers are $alt$, then $concat$, then $closure$, then $simple$. 

```
alt → alt + concat
alt → concat
concat → concat closure
```
Ambiguous grammar

1. RE → RE + RE
2. RE → RE RE
3. RE → RE *
4. RE → id
5. RE → #
6. RE → 0
7. RE → ( RE )

- Transform grammar by layering
  - Tightest binding operators (*) at the lowest layer
  - Layers are alt, then concat, then closure, then simple.

alt → alt + concat
alt → concat
concat → concat closure
closure → closure
Ambiguous grammar

• Transform grammar by layering
  • Tightest binding operators (*) at the lowest layer
  • Layers are alt, then concat, then closure, then simple.

1. RE → RE + RE
2. RE → RE RE
3. RE → RE *
4. RE → id
5. RE → #
6. RE → 0
7. RE → ( RE )

alt → alt + concat
alt → concat
concat → concat closure
concat → closure
closure → simple *
Ambiguous grammar

1. \( RE \to RE + RE \)
2. \( RE \to RE \ RE \)
3. \( RE \to RE \ * \)
4. \( RE \to id \)
5. \( RE \to \# \)
6. \( RE \to 0 \)
7. \( RE \to ( \ RE \ ) \)

- Transform grammar by layering
  - Tightest binding operators (*) at the lowest layer
  - Layers are alt, then concat, then closure, then simple.

alt \to alt + concat
alt \to concat
concat \to concat closure
closure \to closure
closure \to simple *
closure \to simple
Ambiguous grammar

1. RE → RE + RE
2. RE → RE RE
3. RE → RE *
4. RE → id
5. RE → #
6. RE → 0
7. RE → ( RE )

• Transform grammar by layering
  • Tightest binding operators (*) at the lowest layer
  • Layers are alt, then concat, then closure, then simple.

alt → alt + concat
alt → concat
concat → concat closure
concat → closure
closure → simple *
closure → simple
simple → id | (alt) | # | 0
Left Recursive Grammar
Left Recursive Grammar

\[ \text{alt} \rightarrow \text{alt} + \text{concat} \]
Left Recursive Grammar

alt → alt + concat
alt → concat
Left Recursive Grammar

\[
\begin{align*}
\text{alt} & \rightarrow \text{alt} + \text{concat} \\
\text{alt} & \rightarrow \text{concat} \\
\text{concat} & \rightarrow \text{concat} \text{ closure}
\end{align*}
\]
Left Recursive Grammar

alt → alt + concat
alt → concat
concat → concat closure
concat → closure
Left Recursive Grammar

alt → alt + concat
alt → concat
concat → concat closure
concat → closure
closure → simple *
Left Recursive Grammar

alt → alt + concat
alt → concat
concat → concat closure
concat → closure
closure → simple *
closure → simple
Left Recursive Grammar

alt → alt + concat
alt → concat
concat → concat closure
concat → closure
closure → simple *
closure → simple
simple → id | (alt) | # | ∅
Left Recursive Grammar

For every Non terminal with productions

\[ T \rightarrow T\ n \mid T\ m \quad \text{(left recursive)} \]
\[ \mid a \mid b \quad \text{(non-left rec.)} \]

1. Make a new variable \( T' \)
2. Remove the old productions
3. Add the following productions

\[ T \rightarrow a\ T' \mid b\ T' \]
\[ T' \rightarrow n\ T' \mid m\ T' \mid \varepsilon \]
Left Recursive Grammar

For every Non terminal with productions

\[ T \rightarrow T \, n \mid T \, m \quad \text{(left recursive)} \]
\[ \mid a \mid b \quad \text{(non-left rec.)} \]

1. Make a new variable \( T' \)
2. Remove the old productions
3. Add the following productions

\[ T \rightarrow a \, T' \mid b \, T' \]
\[ T' \rightarrow n \, T' \mid m \, T' \mid \varepsilon \]
Left Recursive Grammar

For every Non terminal with productions

\[ T \rightarrow T \ n \mid T \ m \quad \text{(left recursive)} \]
\[ \mid a \mid b \quad \text{(non-left rec.)} \]

1. Make a new variable \( T' \)
2. Remove the old productions
3. Add the following productions

\[ T \rightarrow a \ T' \mid b \ T' \]
\[ T' \rightarrow n \ T' \mid m \ T' \mid \varepsilon \]
Left Recursive Grammar

alt → alt + concat
alt → concat
concat → concat closure
concat → closure
closure → simple *
closure → simple
simple → id | (alt) | # | 0

For every Non terminal with productions
T → T n | T m (left recursive)
   | a | b (non-left rec.)
1. Make a new variable T'
2. Remove the old productions
3. Add the following productions
   T → a T' | b T'
   T' → n T' | m T' | ε

alt → concat moreAlt
moreAlt → + concat moreAlt
         | ε
concat → closure moreConcat
moreConcat → closure moreConcat
           | ε
closure → simple *
           | simple
Left Recursive Grammar

alt \rightarrow \text{alt} + \text{concat}  \\
alt \rightarrow \text{concat}  \\
\text{concat} \rightarrow \text{concat} \text{ closure}  \\
\text{concat} \rightarrow \text{closure}  \\
\text{closure} \rightarrow \text{simple} *  \\
\text{closure} \rightarrow \text{simple}  \\
\text{simple} \rightarrow \text{id} \mid (\text{alt}) \mid \# \mid \emptyset  \\

For every Non terminal with productions

\[
T \rightarrow T \text{ } n \mid T \text{ } m \quad \text{(left recursive)} \\
\mid a \mid b \quad \text{(non-left rec.)}
\]

1. Make a new variable $T'$
2. Remove the old productions
3. Add the following productions

\[
\begin{align*}
T & \rightarrow a \text{ } T' \mid b \text{ } T' \\
T' & \rightarrow n \text{ } T' \mid m \text{ } T' \mid \varepsilon
\end{align*}
\]
Lookahead and the Lexer

val lookahead = ref Done;
val input = ref [Done];
val location = ref 0;

fun nextloc () =
  (location := (!location) + 1; !location);

fun init s = ( location := 0;
    input := lexan s;
    lookahead := hd(!input);
    input := tl(!input)
    );

• Lexes the whole input
• Stores it in the variable input
• Keeps track of next token (so that backup is possible)
The Lexical Analyzer (Lexer)

fun lexan "" : token list = [ ]
| lexan s = case (first s)
  of " " => (lexan (rest s)) (* ignore spaces *)
  | "#" => Hash :: (lexan (rest s))
  | "0" => Zero :: (lexan (rest s))
  | "+" => Plus :: (lexan (rest s))
  | "*" => Star :: (lexan (rest s))
  | "(" => LeftParen :: (lexan (rest s))
  | ")" => RightParen :: (lexan (rest s))
  | ch => if ch >= "a" andalso ch <= "z"
        then (Single ch) :: (lexan (rest s))
        else [BadInput];
Matching a single Terminal

fun match t =
  if (!lookahead) = t
  then if null(!input)
    then lookahead := Done
     else ( lookahead := hd(!input);
            input := tl(!input) )
  else raise error ("looking for: " ^ (tok2str t)
                   ^ " found: " ^ (tok2str (!lookahead)));

• Match one token
• Advance the input
• Handle the end of input correctly
• Report errors in a sensible way
• This function will be called a lot!
moreAlt and moreConcat

When we removed left recursion, we added variables that might recognize $\varepsilon$.

i.e., moreAlt and moreConcat

Observe the shape of parse trees using those productions.

- moreConcat → closure moreConcat
  | $\varepsilon$

They always end in $\varepsilon$ at the far right of the tree.
Write one function for each Variable

• Recursive descent is a simple way to write a parser
• Each variable in the grammar is represented by a function that returns a syntax item corresponding to the element parsed by productions with that variable on the LHS.
• If the function can’t parse that element, it raises an error.
• When a production might match the empty string we handle that by using the ML\texttt{alpha} option datatype.

• The types of the parsing functions are:

\begin{verbatim}
alt : unit \rightarrow \text{RE}
moreAlt : unit \rightarrow \text{RE option}
concat : unit \rightarrow \text{RE}
closure : unit \rightarrow \text{RE}
moreConcat : unit \rightarrow \text{RE option}
simple : unit \rightarrow \text{RE}
\end{verbatim}
fun alt () =
  let val x = concat ()
  val y = moreAlt ()
  in case y of
    NONE => x
    | SOME z => Union(x, z)
  end
moreAlt → + alt moreAlt | ε

and moreAlt () =
    case (!lookahead) of
        Plus => let val _ = match Plus
                    val x = alt()
                    val y = moreAlt ()
                    in case y of
                        NONE => SOME x
                        | (SOME z) => SOME(Union(x,z))
                    end
        _ => NONE
moreAlt → + alt moreAlt | ε

and moreAlt () =
case (!lookahead) of
    Plus => let val _ = match Plus
        val x = alt()
        val y = moreAlt ()
in case y of
    NONE => SOME x
    | (SOME z) => SOME(Union(x,z))
end
| _ => NONE

“and” separates mutually recursive functions
concat → closure moreConcat

and concat () =
let val x = closure ()
    val y = moreConcat ()
in case y of
    NONE => x
    | SOME z => (Concat(x,z))
end
moreConcat → closure moreConcat | ε

and moreConcat () =
  if (couldBeSimple (!lookahead))
  then
    let val x = closure()
    val y = moreConcat()
    in case y of
    NONE => SOME x
    | SOME z => SOME(Concat(x,z))
    end
  else NONE

and couldBeSimple LeftParen = true
| couldBeSimple Hash = true
| couldBeSimple Zero = true
| couldBeSimple (Single _) = true
| couldBeSimple _ = false
closure → simple Star
  | simple

and closure () =
  let val x = simple()
  in case !lookahead of
    Star => (match Star; Closure x)
          |_  =>  x
  end
simple → id | ( alt ) | # | 0

and simple () =
  case !lookahead of
    Single c =>
      let val _ = match (Single c)
      in Simple(c)
      end
    | LeftParen =>
      let val _ = match LeftParen
      val x = alt();
      val _ = match RightParen
      in x
      end
    | Hash =>
      let val _ = match Hash
      in Epsilon
      end
    | Zero =>
      let val _ = match Zero
      in Empty
      end
    | x => raise error ("In simple no match: " ^ (tok2str x));
fun parse s =
    let val _ = init s
    val ans = alt()
    val _ = match Done
    in ans end;

(* Tests *)
val p1 = parse "a(b* + c)#";

> val p1 =
    Concat(Simple "a", Concat(Union(Closure(Simple "b"),
    Simple "c"), Epsilon))
  : RE/39
Top-down parsing with a Parse Table
Recall the theory:

\[ \varepsilon, \varepsilon \rightarrow S \]

where \( S \) is the grammar's start symbol

\[ \varepsilon, A \rightarrow \omega \]

for each rule \( A \rightarrow \omega \), \( \omega \) a sequence of terminals and variables

\[ \varepsilon, a \rightarrow \varepsilon \]

for each terminal \( a \in A \)
Recall the theory:

\[ \varepsilon, \varepsilon \rightarrow S \] where \( S \) is the grammar's start symbol

\[ \varepsilon, A \rightarrow \omega \] for each rule \( A \rightarrow \omega \), \( \omega \) a sequence of terminals and variables

\[ a, a \rightarrow \varepsilon \] for each terminal \( a \in A \)

- At each step, PDA can either
Recall the theory:

- \( \varepsilon, \varepsilon \rightarrow S \) where \( S \) is the grammar's start symbol

- \( \varepsilon, A \rightarrow \omega \) for each rule \( A \rightarrow \omega \), \( \omega \) a sequence of terminals and variables

- \( a, a \rightarrow \varepsilon \) for each terminal \( a \in A \)

- At each step, PDA can either

  1. read input \( a \), iff \( a \) is on the stack (match), or
Recall the theory:

\[ \varepsilon, \varepsilon \rightarrow S \]  
\[ \varepsilon, A \rightarrow \omega \]  
\[ a, a \rightarrow \varepsilon \]

where S is the grammar's start symbol  
for each rule \( A \rightarrow \omega \), \( \omega \) a sequence of terminals and variables  
for each terminal \( a \in A \)

- At each step, PDA can either
  1. read input \( a \), iff \( a \) is on the stack (match), or
  2. replace \( A \) on the stack with \( \omega \), where \( A \rightarrow \omega \) is a rule of the grammar (derive).
Recall the theory:

\[ \varepsilon, \varepsilon \rightarrow S \]  
where \( S \) is the grammar's start symbol

\[ \varepsilon, A \rightarrow \omega \]  
for each rule \( A \rightarrow \omega \), \( \omega \) a sequence of terminals and variables

\[ a, a \rightarrow \varepsilon \]  
for each terminal \( a \in A \)

- At each step, PDA can either
  
  1. read input \( a \), iff \( a \) is on the stack (match), or
  2. replace \( A \) on the stack with \( \omega \), where \( A \rightarrow \omega \) is a rule of the grammar (derive).

- How to choose \textit{which} \( A \rightarrow \omega \)?
Recall the Practice

and moreConcat () =
  if (couldBeSimple (!lookahead))
    then
      let val x = closure()
      val y = moreConcat()
      in case y of
        NONE => SOME x
        | SOME z => SOME(Concat(x,z))
      end
    else NONE
  end

and couldBeSimple LeftParen = true
| couldBeSimple Hash = true
| couldBeSimple Zero = true
| couldBeSimple (Single _) = true
| couldBeSimple _ = false
Top-down predictive parsers

- Use an explicit stack instead of recursion.
- Represent the transitions of the PDA in a table, rather than as code.
- Choice of action of the parser (which production to use to expend the stack symbol) represented by three functions:
  - Nullable
  - First
  - Follow
• Nullable: Can a symbol derive the empty string? False for every terminal symbol.
  » Nullable: (terminal or variable) → bool

• First: all the terminals that a variable could possibly derive as its first symbol.
  » First: (terminal or variable) → set( terminal )
  » First: sequence(terminal + variable) → set( terminal )

• Follow: all the terminals that could immediately follow the string derived from a variable.
  » Follow: variable → set( terminal )
Example First and Follow Sets

E → T E' $  
E' → + T E'  
E' → ε  
T → F T'  
T' → * F T'  
T' → ε  
F → ( E )  
F → id

First E = \{ "(", "id"\}  
Follow E = \{ ", "$\}\}  
First F = \{ "(", "id"\}  
Follow F = \{ ", ", ", ", ", ", "$\}\}  
First T = \{ "(", "id"\}  
Follow T = \{ ", ", ", "$\}\}  
First E' = \{ ", "$\}\}  
Follow E' = \{ ", "$\}\}  
First T' = \{ ", "$\}\}  
Follow T' = \{ ", "$\}\}  

- First of a terminal is itself.  
- First can be extended to be defined on a sequence of symbols.
Nullable

- if $\epsilon$ is in First(symbol) then that symbol is *nullable*.

- Sometimes, we use an additional function Nullable.
  - Nullable ($E'$) = true
  - Nullable ($T'$) = true
  - Nullable for all other symbols is false

$$
egin{align*}
E &\rightarrow T\ E'\ \$ \\
E' &\rightarrow +\ T\ E' \\
E' &\rightarrow \epsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \\
T' &\rightarrow \epsilon \\
F &\rightarrow (\ E ) \\
F &\rightarrow \text{id}
\end{align*}
$$
Computing *First*

- Use the following rules until no more terminals can be added to any *First* set.

For all symbols X:

1) if X is a terminal, First(X) = {X}

2) if $X \rightarrow \epsilon$ is a production then add $\epsilon$ to First(X),
   (or: set Nullable(X) to true).

3) if X is a variable and $X \rightarrow Y_1 \ Y_2 \ldots \ Y_k$
   
   - if $a \in$ First($Y_i$) $\land$ ($\forall j < i$. Nullable ($Y_j$)) then add a to First(X)
   
   - e.g., if $Y_1$ can derive $\epsilon$ then,
     if $a$ is in First($Y_2$), $a$ is surely in First(X) as well.
Example First Computation

- **Terminals**
  - First($) = {$}  First(*) = {}  First(+) = {+}  ...

- **Empty Productions**
  - add $\varepsilon$ to First(E'), add $\varepsilon$ to First(T')

- **Other Variables**
  - Computing from the lowest layer (F) up:
    - First(F) = {id, ( }
    - First(T') = { $\varepsilon$, * }
    - First(T) = First(F) = {id, ( }
    - First(E') = { $\varepsilon$, + }
    - First(E) = First(T) = {id, ( }
Computing FOLLOW

Use the following rules until nothing can be added to any follow set:

1) Place $ (the end of input marker) in FOLLOW(S) where S is the start symbol.  
   \textit{(This is Hein’s rule 1)}

2) If $A \rightarrow a B b$, then put everything in First($b$) (except $\epsilon$) into FOLLOW($B$)  
   \textit{(This is Hein’s rule 3)}

3) If there is a production $A \rightarrow a B$, or $A \rightarrow a B b$ where First($b$) contains $\epsilon$ (i.e., nullable $b$), then put everything in FOLLOW($A$) into FOLLOW($B$)  
   \textit{(This is a combination of Hein’s rules 2 & 4)}
Example: Follow Computation

- **Rule 1**, Start symbol
  - Add $ to Follow(E)

- **Rule 2**, Productions with embedded variables
  - Add First( ) = { ) } to Follow(E)
  - Add First($) = { $ } to Follow(E')
  - Add First(E') = {+, ε } to Follow(T)
  - Add First(T') = {* , ε} to Follow(F)

- **Rule 3**, Variable in rightmost position
  - Add follow(E') to follow(E') (doesn’t do much)
  - Add follow(T) to follow(T')
  - Add follow(T) to follow(F) since T' ⇒ ε
  - Add follow(T') to follow(F) since T' ⇒ ε

```
E → T E' $  
E' → + T E'  
E' → ε  
T → F T'  
T' → * F T'  
T' → ε  
F → ( E )  
F → id
```
Table from First and Follow

For each production $A \rightarrow \omega$ do steps 1–3 below:

1. For each terminal $a$ in $\text{First } \omega$ add $A \rightarrow \omega$ to $M[A,a]$.
2. If $\varepsilon$ is in $\text{First } \omega$, add $A \rightarrow \omega$ to $M[A,b]$ for each terminal $b$ in $\text{Follow } A$.
3. If $\varepsilon$ is in $\text{First } \omega$ and $\varepsilon$ is in $\text{Follow } A$, add $A \rightarrow \omega$ to $M[A,\varepsilon]$.

First $E = \{ "(", "id"\}$
Follow $E = \{ "\), "$\}$
First $F = \{ "(", "id"\}$
Follow $F = \{ "\), ", +", ", \*, \), "$\}$
First $T = \{ "(", "id"\}$
Follow $T = \{ "\), ", +", "$\}$
First $E' = \{ ", +", \varepsilon\}$
Follow $E' = \{ "\), "$\}$
First $T' = \{ ", \*, \varepsilon\}$
Follow $T' = \{ "\), ", +", "$\}$

<table>
<thead>
<tr>
<th>$M[A,t]$</th>
<th>+</th>
<th>*</th>
<th>)</th>
<th>id</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>$E'$</td>
<td>2</td>
<td>3</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
<td>4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>$T'$</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>

1  $E \rightarrow T E' \$  
2  $E' \rightarrow + T E'$  
3  \varepsilon
4  $T \rightarrow F T'$  
5  $T' \rightarrow \* F T'$  
6  \varepsilon
7  $F \rightarrow ( E )$  
8  id
### Predictive Parsing Table

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E→TE’</td>
<td>E→TE’</td>
<td>E→TE’</td>
<td>E→TE’</td>
<td>E→TE’</td>
<td>E→TE’</td>
</tr>
<tr>
<td>E’</td>
<td>E’→+TE’</td>
<td>E’→ε</td>
<td>E’→ε</td>
<td>E’→ε</td>
<td>E’→ε</td>
<td>E’→ε</td>
</tr>
<tr>
<td>T</td>
<td>T→FT’</td>
<td>T→FT’</td>
<td>T→FT’</td>
<td>T→FT’</td>
<td>T→FT’</td>
<td>T→FT’</td>
</tr>
<tr>
<td>T’</td>
<td>T’→ε</td>
<td>T’→ε</td>
<td>T’→ε</td>
<td>T’→ε</td>
<td>T’→ε</td>
<td>T’→ε</td>
</tr>
<tr>
<td>F</td>
<td>F→id</td>
<td>F→(E)</td>
<td>F→(E)</td>
<td>F→(E)</td>
<td>F→(E)</td>
<td>F→(E)</td>
</tr>
</tbody>
</table>
Table-driven Algorithm

push start symbol of grammar onto stack
repeat
    let X = top of stack, c = next input
    if isTerminal(X)
        then if X=c
            then pop X; read c
            else error(…)
        fi
    else (* isVariable(X) *)
        if M[X,c] = X → Y₁ Y₂ ... Yₖ
            then pop X;
                push Yₖ Yₖ₋₁ ... Y₁ (* Y₁ on top *)
            else error(…)
        fi
    fi
until stack is empty and input = $
## Example Parse

### Stack vs Input

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td><code>x + y $</code></td>
</tr>
<tr>
<td>T E'</td>
<td><code>x + y $</code></td>
</tr>
<tr>
<td>F T' E'</td>
<td><code>x + y $</code></td>
</tr>
<tr>
<td>id T' E'</td>
<td><code>x + y $</code></td>
</tr>
<tr>
<td>T' E'</td>
<td><code>+ y $</code></td>
</tr>
<tr>
<td>E'</td>
<td><code>+ y $</code></td>
</tr>
<tr>
<td>T E'</td>
<td><code>+ y $</code></td>
</tr>
<tr>
<td>F T' E'</td>
<td><code>y $</code></td>
</tr>
<tr>
<td>id T' E'</td>
<td><code>y $</code></td>
</tr>
<tr>
<td>T' E'</td>
<td><code>y $</code></td>
</tr>
<tr>
<td>E'</td>
<td><code>y $</code></td>
</tr>
<tr>
<td></td>
<td><code>y $</code></td>
</tr>
<tr>
<td></td>
<td><code>y $</code></td>
</tr>
<tr>
<td></td>
<td><code>y $</code></td>
</tr>
</tbody>
</table>

### Parsing Rules

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td><code>E → TE'</code></td>
</tr>
<tr>
<td>E'</td>
<td><code>E' → +TE'</code></td>
</tr>
<tr>
<td>T</td>
<td><code>T → FT'</code></td>
</tr>
<tr>
<td>T'</td>
<td><code>T' → ε</code></td>
</tr>
<tr>
<td>F</td>
<td><code>F → id</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td><code>E → TE'</code></td>
</tr>
<tr>
<td>E'</td>
<td><code>E' → ε</code></td>
</tr>
<tr>
<td>T</td>
<td><code>T → FT'</code></td>
</tr>
<tr>
<td>T'</td>
<td><code>T' → ε</code></td>
</tr>
<tr>
<td>F</td>
<td><code>F → (E)</code></td>
</tr>
</tbody>
</table>