CS311—Computational Structures

Parsing Techniques

Andrew P. Black

Lecture 10

Portland State
UNIVERSITY

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

$$
\begin{aligned}
\text{Exp} \rightarrow\ & \text{id} \\
\mid\ & \text{Exp} + \text{Exp} \\
\mid\ & \text{Exp} * \text{Exp} \\
\mid\ & (\ \text{Exp}\ )
\end{aligned}
$$

- derives x, x+x, x+x+x, x * y, x + y * z ..

Example Parse (top down)

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

– stack   input

Exp   x + y * z

Exp   x + y * z
/ | \
Exp   + Exp

Portland State University

Example Parse (continued)

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

```
     Exp            y * z
    /  |  \
Exp  +  Exp
 |
id(x)
```

Example Parse (continued)

```
         Exp                  y * z
       /  |  \
   Exp  +    Exp
    |        / | \
  id(x)   Exp *  Exp
```

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

Example Parse (continued)

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

```
        Exp                      z
      /  |  \
  Exp    +     Exp
   |          /  |  \
 id(x)    Exp  *   Exp
            |
          id(y)
```

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:

$$S \rightarrow ee \mid bAc \mid bAe$$
$$A \rightarrow d \mid cA$$

  - try on string "bcde"

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

$$E \rightarrow E + id \mid id$$

Grammar Transformations

- Removing ambiguity
- Removing Left Recursion
- Backtracking and Factoring

Removing ambiguity

- Add levels to a grammar

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

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow id \mid (E) \end{aligned}$$

Removing ambiguity

- The dangling else grammar.

stmt → **if** exp **then** stmt **else** stmt
| **if** exp **then** stmt
| id := exp

  - Note that the following has two possible parses:

if x=2 **then if** x=3 **then** y:=2 **else** y := 4

if x=2 **then** (**if** x=3 **then** y:=2) **else** y := 4

if x=2 **then** (**if** x=3 **then** y:=2 **else** y := 4)

Adding levels (cont)

- Original grammar

stmt → **if** exp **then** stmt **else** stmt
    | **if** exp **then** stmt
    | id := exp

- Assume that every stmt between **then** and **else** must be matched, i.e., it must have both a **then** and an **else**.
- New Grammar with additional levels:

stmt → match | unmatch
match → **if** exp **then** match **else** match
    | id := exp
unmatch → **if** exp **then** stmt
    | **if** exp **then** match **else** unmatch

Removing Left Recursion

- Top down recursive descent parsers require non-left recursive grammars
- Technique: *left factoring*
    - $E \rightarrow E + E \mid E * E \mid \text{id}$
    - $E \rightarrow \text{id}\ E'$
      $E' \rightarrow +\ E\ E' \mid *\ E\ E' \mid \varepsilon$

General Technique to remove direct left recursion

- For every variable with productions

$T \rightarrow T\ n \mid T\ m$ (left recursive productions)

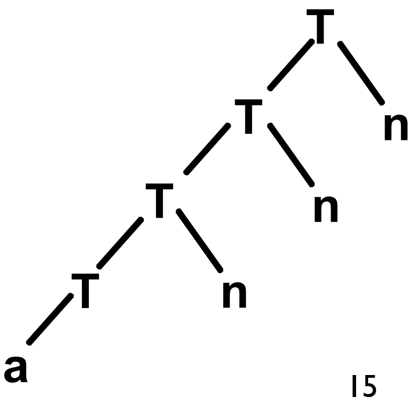
$\quad \mid a \mid b$ (non left recursive productions)

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

$T \rightarrow a\ T' \mid b\ T'$

$T' \rightarrow n\ T' \mid m\ T' \mid \varepsilon$

(a | b) (n | m) *



Backtracking and Factoring

- Backtracking may be necessary:

$$S \rightarrow ee \mid bAc \mid bAe$$
$$A \rightarrow d \mid cA$$

- try on string "bcde"

$$S \Rightarrow bAc \qquad \text{(by } S \rightarrow bAc\text{)}$$
$$\Rightarrow bcAc \qquad \text{(by } A \rightarrow cA\text{)}$$
$$\Rightarrow bcdc \qquad \text{(by } A \rightarrow d\text{)}$$

- 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:

    $S \rightarrow ee \mid bAc \mid bAe$
    $A \rightarrow d \mid cA$

  - Factored grammar:

    $S \rightarrow ee \mid bAQ$
    $Q \rightarrow c \mid e$
    $A \rightarrow d \mid cA$

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:

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

- The lexical analyzer datatype:

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

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 )
```

for ε

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`.

Portland State UNIVERSITY

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 → 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
```

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
```

Portland State University

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 *
```

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
```

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

```
alt → alt + concat
```

Left Recursive Grammar

```
alt → alt + concat
alt → concat
```

Left Recursive Grammar

```
alt → alt + concat
alt → concat
concat → concat  closure
```

Left Recursive Grammar

```
alt → alt + concat
alt → concat
concat → concat  closure
concat → closure
```

Portland State
UNIVERSITY

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) | # | 0
```

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
```

Portland State UNIVERSITY

Left Recursive Grammar

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

```
alt          → concat moreAlt
moreAlt      → + concat moreAlt
               | ε
```

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
```

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
```

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

```
alt          → concat moreAlt
moreAlt      → + concat moreAlt
               | ε
concat       → closure moreConcat
moreConcat   → closure moreConcat
               | ε
closure      → simple *
               | 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
```

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!

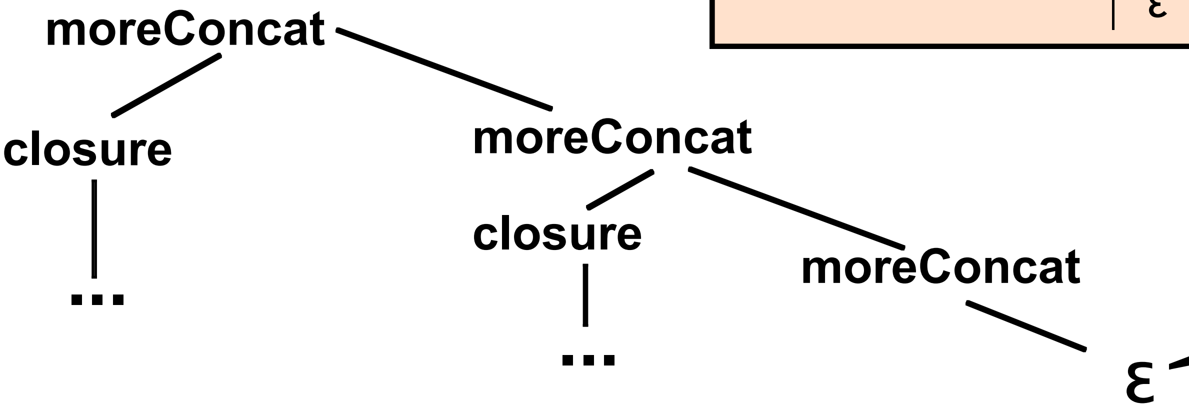
moreAlt and moreConcat

When we removed left recursion, we added variables that that might recognize ε.

i.e., `moreAlt` and `moreConcat`

Observe the shape of parse trees using those productions.

```
moreConcat → closure moreConcat
           | ε
```



moreConcat
closure
…
moreConcat
closure
…
moreConcat
ε

They always end in ε 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 `alpha option` datatype.
- The types of the parsing functions are:

```
alt : unit → RE
moreAlt : unit → RE option
concat : unit → RE
closure : unit → RE
moreConcat : unit → RE option
simple : unit → RE
```

```
alt          → concat moreAlt
moreAlt      → + Concat moreAlt
               | ε
concat       → closure moreConcat
moreConcat   → closure moreConcat
               | ε
closure      → simple *
               | simple
simple       → id
               | ( alt )
               | #
               | 0
```

Portland State
UNIVERSITY

alt → concat moreAlt

```
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));
```

Top Level Parser

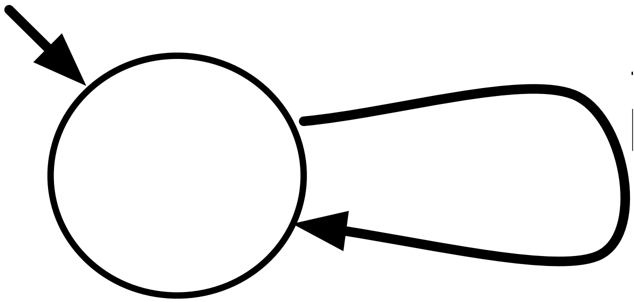
```
fun parse s =
    let val _ = init s
        val ans = alt()
        val _ = match Done
    in ans end;

(* 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:

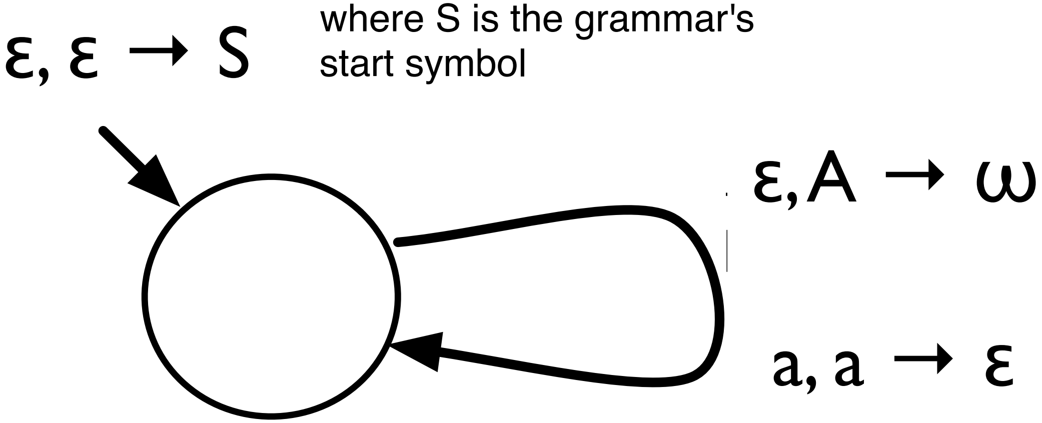


$\varepsilon, \varepsilon \to S$ where S is the grammar's start symbol

$\varepsilon, A \to \omega$ for each rule $A \to \omega$, ω a sequence of terminals and variables

$a, a \to \varepsilon$ for each terminal $a \in A$

Recall the theory:



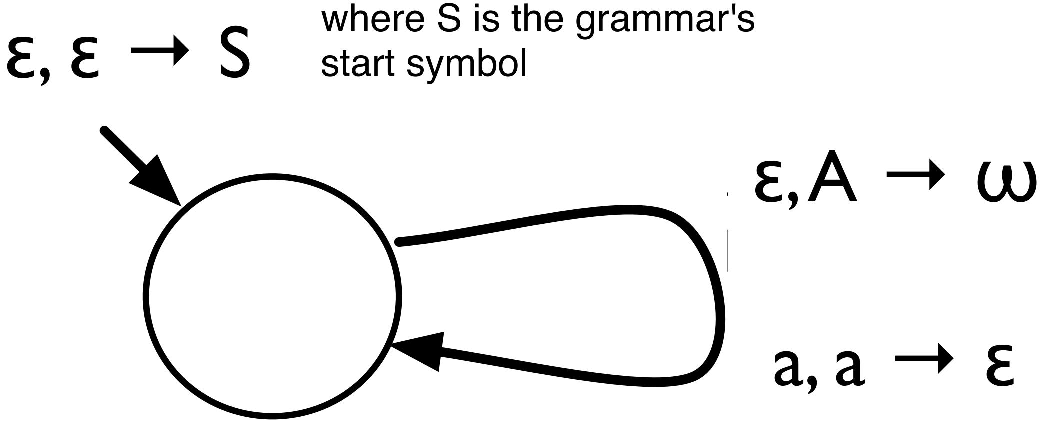
where S is the grammar's start symbol

for each rule A →ω , ω a sequence of terminals and variables

for each terminal a ∈ A

- At each step, PDA can either

Recall the theory:



where S is the grammar's start symbol

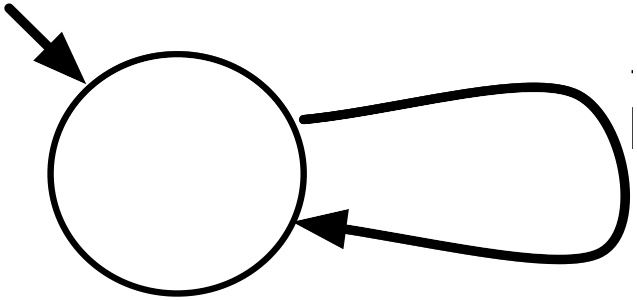
for each rule A →ω , ω a sequence of terminals and variables

for each terminal a ∈ A

- At each step, PDA can either

    1. read input *a* , iff *a* is on the stack (match), or

Recall the theory:



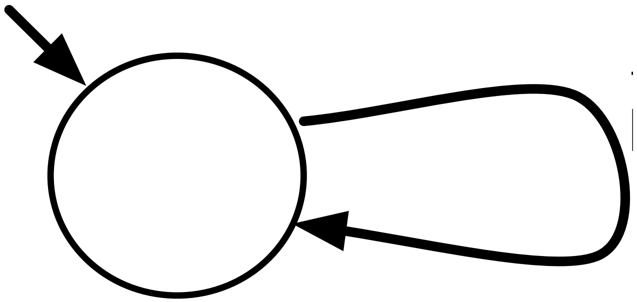
$\varepsilon, \varepsilon \rightarrow S$ where S is the grammar's start symbol

$\varepsilon, A \rightarrow \omega$ for each rule A →ω , ω a sequence of terminals and variables

$a, a \rightarrow \varepsilon$ for each terminal a ∈ 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 ω, where A →ω 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 →ω , ω a sequence of terminals and variables

$a, a \rightarrow \varepsilon$ for each terminal a ∈ 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 ω, where A →ω is a rule of the grammar (derive).
- How to choose *which* A →ω?

Recall the Practice

```
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
```

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

$E \rightarrow T\ E'\ \$$

$E' \rightarrow +\ T\ E'$

$E' \rightarrow \varepsilon$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T'$

$T' \rightarrow \varepsilon$

$F \rightarrow (\ E\ )$

$F \rightarrow id$

First E = { "(", "id"}

First F = { "(", "id"}

First T = { "(", "id"}

First E′ = { "+", **ε**}

First T' = { "*", **ε**}

Follow E = {")","$"}

Follow F = {"+","*",”)”,"$"}

Follow T = {{"+",")","$"}

Follow E' = {")","$"}

Follow T' = {"+",")","$"}

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

Nullable

- if ε 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

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

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 → ε is a production then add ε to First(X),
(or: set Nullable(X) to true).

3) if X is a variable and X → Y_1 Y_2 ... Y_k

» if $a \in \text{First}(Y_i) \wedge (\forall j < i .\ \text{Nullable}(Y_j))$ then add *a* to First(X)

– e.g., if Y_1 can derive ε then,
if *a* is in First(Y_2), *a* is surely in First(X) as well.

Portland State
UNIVERSITY

Example First Computation

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

- Terminals
  - First($) = {$} First(*) = {*} First(+) = {+} ...
- Empty Productions
  - add ε to First(E'), add ε to First(T')
- Other Variables
  - Computing from the lowest layer (F) up:
    - » First(F) = {id , (}
    - » First(T') = { ε, * }
    - » First(T) = First(F) = {id, (}
    - » First(E') = { ε, + }
    - » 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.

(This is Hein's rule 1)

2) If A → ***a*** B ***b***, then put everything in First(***b***) (except ε) into FOLLOW(B)

(This is Hein's rule 3)

3) If there is a production A → ***a*** B, or A → ***a*** B ***b***
where First(***b***) contains ε (i.e., nullable ***b***), then put everything in FOLLOW(A) into FOLLOW(B)

(This is a combination of Hein's rules 2 & 4)

Portland State
UNIVERSITY

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 → ω do steps 1–3 below:

1. For each terminal **a** in First ω add A → ω to M[A,**a**]
2. If **ε** is in First ω, add A → ω to M[A,b] for each terminal b in Follow A.
3. If **ε** is in First ω and $ is in Follow A, add A → ω to M[A,$].

First E = { "(", "id"}
First F = { "(", "id"}
First T = { "(", "id"}
First E′ = { "+", **ε**}
First T' = { "*", **ε**}

Follow E = {")","$"}
Follow F = {"+","*",”)”,"$"}
Follow T = {{"+",")","$"}
Follow E' = {")","$"}
Follow T' = {"+",")","$"}

M[A,t]

variables	+	*	)	(	id	$
	terminals					
E				1	1	
E’	2		3			3
T				4	4	
T’	6	5	6			6
F				7	8	

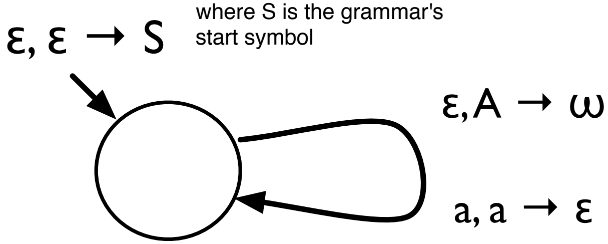
```
1 E  →  T E' $
2 E' →  + T E'
3       |  ε
4 T  →  F T'
5 T' →  * F T'
6       | ε
7 F  →  ( E )
8       | id
```

Predictive Parsing Table

	id	+	*	(	)	$
E	E→TE′			E→TE′		
E’		E′→+TE′			E′→ε	E′→ε
T	T→F T′			T→F T′		
T’		T′→ε	T′→*FT′		T′→ε	T′→ε
F	F→id			F→(E)		

Portland State UNIVERSITY

Table-driven Algorithm



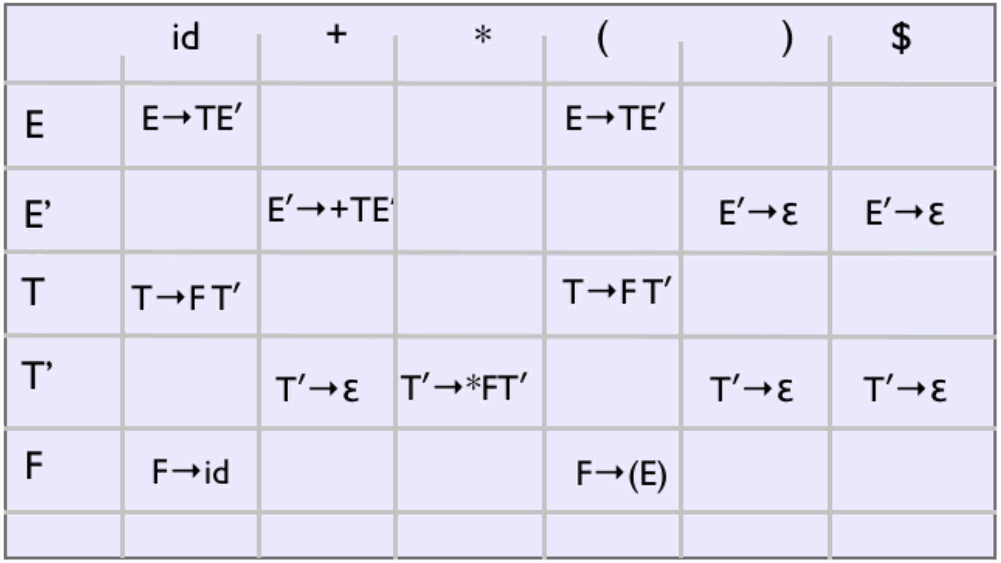
$\varepsilon, \varepsilon \to S$ where S is the grammar's start symbol

$\varepsilon, A \to \omega$ for each rule $A \to \omega$, ω a sequence of terminals and variables

$a, a \to \varepsilon$ for each terminal $a \in A$

```
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] = $X \to Y_1\ Y_2\ \dots\ Y_k$
          then   pop X;
                 push $Y_k\ Y_{K-1}\ \dots\ Y_1$     (* $Y_1$ on top *)
          else error(…)
          fi
    fi
until stack is empty and input = $
```

Example Parse



	id	+	*	(	)	$
E	E→TE′			E→TE′		
E’		E′→+TE′			E′→ε	E′→ε
T	T→FT′			T→FT′		
T’		T′→ε	T′→*FT′		T′→ε	T′→ε
F	F→id			F→(E)		

Top of stack

Stack	*Input*
E	x + y $
T E′	x + y $
F T′ E′	x + y $
id T′ E′	x + y $
T′ E′	+ y $
E′	+ y $
+ T E′	+ y $
T E′	y $
F T′ E′	y $
id T′ E′	y $
T′ E′	$
E′	$
	$