Context Free Grammar – Quick Review

- Grammar - quaduple
  - A set of tokens (terminals): T
  - A set of non-terminals: N
  - A set of productions { lhs -> rhs , ... }
    - lhs in N
    - rhs is a sequence of N U T
  - A Start symbol: S (in N)

- Shorthands
  - Provide only the productions
    - All lhs symbols comprise N
    - All other sysmbols comprise T
    - lhs of first production is S

Using Grammars to derive Strings

- Rewriting rules
  - Pick a non-terminal to replace. Which order?
    - left-to-right
    - right-to-left
- Derives relation: $\alpha A \gamma \Rightarrow \alpha \beta \chi$
  - When A -> β is a production
- Derivations (a list if productions used to derive a string from a grammar).
- A sentence of G: L(G)
  - Start with S
  - $S \Rightarrow^* w$ where w is only terminal symbols
  - all strings of terminals derivable from S in 1 or more steps

CF Grammar Terms

- Parse trees.
  - Graphical representations of derivations.
  - The leaves of a parse tree for a fully filled out tree is a sentence.

- Regular language v.s. Context Free Languages
  - how do CFL compare to regular expressions?
  - Nesting (matched ()’s) requires CFG,’s RE's are not powerful enough.

- Ambiguity
  - A string has two derivations
  - E -> E + E | E * E | id
    - x + x * y

- Left-recursion
  - E -> E + E | E * E | id
  - Makes certain top-down parsers loop

Parsing

- Act of constructing derivations (or parse trees) from an input string that is derivable from a grammar.

- Two general algorithms for parsing
  - Top down - Start with the start symbol and expand Non-terminals by looking at the input
    - Use a production on a left-to-right manner
  - Bottom up - replace sentential forms with a non-terminal
    - Use a production in a right-to-left manner

Top Down Parsing

- Begin with the start symbol and try and derive the parse tree from the root.
- Consider the grammar
  1. Exp -> Id | Exp + Exp | Exp * Exp | (Exp)
  2. Id -> x | y

Some strings derivable from the grammar

x

x+x

x+x+x,

x * y

x + y * z ...

Example Parse (top down)

– stack input

```
   Exp            x + y * z

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

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

Top Down Parse (cont)

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

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

Top Down Parse (cont.)

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

Problems with Top Down Parsing

- Backtracking may be necessary:
  - S ::= ee | bAc | bAe
  - A ::= d | cA

  try on string "bcde"

- Infinite loops possible from (indirect) left recursive grammars.
  - E ::= E + id | id

- Ambiguity is a problem when a unique parse is not possible.

- These often require extensive grammar restructuring (grammar debugging).

Grammar Transformations

- Backtracking and Factoring
- Removing ambiguity.
  - Simple grammars are often easy to write, but might be ambiguous.
- Removing Left Recursion

Backtracking and Factoring

- Backtracking may be necessary:

```
S -> ee | bAc | bAe
A -> d | cA
```

- try on string "bcde"

```
S -> bAc      (by S -> bAc)
  -> bcAc     (by A -> cA)
  -> bcdc     (by A -> d)
```

- But now we are stuck, we need to backtrack to
  - S -> bAc
  - And then apply the production (S -> bAe)

- Factoring a grammar
  - Factor common prefixes and make the different postfixes into a new non-terminal

```
S -> ee | bAQ
Q -> c  | e
A -> d  | cA
```

Removing ambiguity.

- Adding levels to a grammar

E -> E + E | E * E | id | (E)

Transform to an equivalent grammar

E -> E + T | T

T -> T * F | F

F -> id | (E)

Levels make formal the notion of precedence. Operators that bind “tightly” are on the lowest levels

The dangling else grammar.

- st -> if exp then st else st

  | if exp then st

  | 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

```
st ::=  if exp  then st  else st
     |  if exp then st
     |  id := exp
```

- Assume that every `st` between then and else must be matched, i.e. it must have both a then and an else.
- New Grammar with addtional levels

```
st         ->  match | unmatch
match      ->  if exp then match else match
           |   id := exp
unmatch    ->  if exp then st
           |   if exp then match else unmatch
```

Top Down Recursive Descent Parsers

- One function (procedure) per non-terminal
- Functions call each other in a mutually recursive way.
- Each function “consumes” the appropriate input.
- If the input has been completely consumed when the function corresponding to the start symbol is finished, the input is parsed.

Example Recursive Descent Parser

```
E -> T+E | T
T -> F*T | F
F -> x | ( E )

expr =
  do { term
     ; iff (match '+') expr }

term =
  do { factor
     ; iff (match '*') term }

factor =
   pCase
   [ 'x' :=> return ()
   , '(' :=> do { expr; match ')'; return ()}
   ]
```

Removing Left Recursion

- Top down recursive descent parsers require non-left recursive grammars
- Technique: Left Factoring

```
E -> E + E  |  E * E |  id

E   ->  id E′
E′  ->  +  E  E′
      | *  E  E′
      | Λ
```

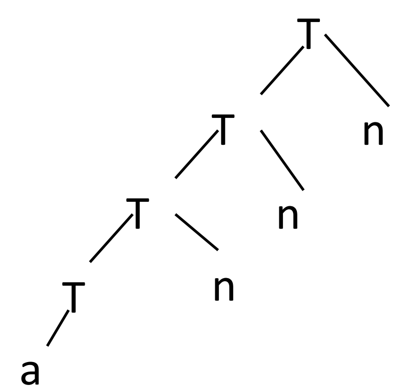
General Technique to remove direct left recursion

- Every Non terminal with productions

```
T  ->  T n  |  T m      (left recursive productions)
       | a     |  b     (non-left recursive productions)
```

“a” and “b” because they are the rhs of the non-left recurive productions.

(a | b) (n | m) *



- Make a new non-terminal T’
- Remove the old productions
- Add the following productions

```
T   ->   a T’  |  b T’
T’  ->   n  T’ |  m  T’ | Λ
```

Recursive Descent Parsing

- One procedure (function) for each non-terminal.
- Procedures are often (mutually) recursive.
- They can return a bool (the input matches that non-terminal) or more often they return a data-structure (the input builds this parse tree)
- Need to control the lexical analyzer (requiring it to “back-up” on occasion)

A grammar suitable for Recursive Descent parsers for Regular Expresions

- Build an instance of the datatype:

```
data RegExp a
   = Lambda                          -- the empty string ""
   | Empty                           -- the empty set
   | One a                           -- a singleton set {a}
   | Union (RegExp a) (RegExp a)     -- union of two RegExp
   | Cat (RegExp a) (RegExp a)       -- Concatenation
   | Star (RegExp a)                 -- Kleene closure
```

Ambiguous grammar

```
RE -> RE bar RE
RE -> RE RE
RE -> RE *
RE -> id
RE -> ^
RE -> ( RE )
```

- Transform grammar by layering
- Tightest binding operators (*) at the lowest layer
- Layers are Alt, then Concat, then Closure, then Simple.

```
Alt -> Alt bar Concat
Alt -> Concat
Concat -> Concat  Closure
Concat -> Closure
Closure -> simple star
Closure -> simple
simple -> id  | ( Alt ) | ^
```

Left Recursive Grammar

```
Alt -> Alt bar Concat
Alt -> Concat
Concat -> Concat  Closure
Concat -> Closure
Closure -> simple star
Closure -> simple
simple -> id  |  (Alt )  | ^
```

For every Non terminal with productions

```
   T ::= T n | T m     (left recursive prods)
         |  a  |  b    (non-left recursive prods)
```

Make a new non-terminal T’

Remove the old productions

Add the following productions

```
   T  ::=  a T’  |  b T’
   T’ ::=  n  T’ |  m  T’ | Λ
```

```
Alt          ->  Concat moreAlt
moreAlt      -> Bar Concat moreAlt
              | Λ
Concat       -> Closure moreConcat
moreConcat   -> Closure moreConcat
              | Λ
Closure      ->  Simple Star
              |  Simple
Simple       -> Id
              | ( Alt )
              | ^
```

Predictive Parsers

- Using a stack to avoid recursion. Encoding the diagrams in a table
- The Nullable, First, and Follow functions
  - Nullable: Can a symbol derive the empty string. False for every terminal symbol.
  - First: all the terminals that a non-terminal could possibly derive as its first symbol.
    - term or nonterm -> set(term)
    - sequence(term + nonterm) -> set(term)
  - Follow: all the terminals that could immediately follow the string derived from a non-terminal.
    - non-term -> set(term)

Example First and Follow Sets

```
E   ->  T E' $
E'  ->  + T E'
E’  ->  Λ
T   ->  F T'
T'  ->  * F T'
T’  ->  Λ
F   ->  ( E )
F   ->  id
```

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

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

Nullable

- if Λ is in First(symbol) then that symbol is nullable.
- Sometime rather than let Λ be a symbol we derive an additional function nullable.

```
E   ->  T E' $
E'  ->  + T E'
E'  ->  Λ
T   ->  F T'
T'  ->  * F T'
T'  ->  Λ
F   ->  ( E )
F   ->  id
```

- Nullable (E’) = true
- Nullable(T’) = true
- Nullable for all other symbols is false

Computing First

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

1) if X is a term. FIRST(X) = {X}

2) if X -> Λ is a production then add Λ to FIRST(X), (Or set nullable of X to true).

3) if X is a non-term and
   - X -> Y1 Y2 ... Yk
   - add a to FIRST(X)
     - if a in FIRST(Yi) and
     - for all j<i Λ in FIRST(Yj)

- E.g.. if Y1 can derive Λ then if a is in FIRST(Y2) it is surely in FIRST(X) as well.

Example First Computation

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

```
E  ->  T E' $
E' ->  + T E'
E' ->  Λ
T  ->  F T'
T' ->  * F T'
T' ->  Λ
F  ->  ( E )
F  ->  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.

2) If A -> *a* B *b*
   then everything in FIRST(*b*) except Λ is in FOLLOW(B)

3) If there is a production A -> *a* B
   or A -> *a* B *b* where FIRST(*b*)
   contains Λ (i.e. *b* can derive the empty string) then everything in FOLLOW(A) is in FOLLOW(B)

Ex. Follow Computation

- Rule 1, Start symbol
  - Add $ to Follow(E)
- Rule 2, Productions with embedded nonterms
  - Add First( )) = {) } to follow(E)
  - Add First($) = { $ } to Follow(E’)
  - Add First(E’) = {+, Λ } to Follow(T)
  - Add First(T’) = {*, Λ} to Follow(F)
- Rule 3, Nonterm in last 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’ --> Λ
  - Add follow(T’) to follow(F) since T’ --> Λ

```
E   ->  T E' $
E'  ->  + T E'
E’  ->  Λ
T   ->  F T'
T'  ->  * F T'
T’  ->  Λ
F   ->  ( E )
F   ->  id
```

Table from First and Follow

1. For each production A -> alpha do 2 & 3
2. For each a in First alpha do add A -> alpha to M[A,a]
3. if ε is in First alpha, add A -> alpha to M[A,b] for each terminal b in Follow A. If ε is in First alpha and $ is in Follow A add A -> alpha to M[A,$].

First E = {"(","id"} Follow E = {")","$"}

First F = {"(","id"} Follow F = {"+","*",”)”,"$"}

First T = {"(","id"} Follow T = {{"+",")","$"}

First E' = {"+",ε} Follow E' = {")","$"}

First T' = {"*",ε} Follow T' = {"+",")","$"}

```
1. E   ->  T E' $
2. E'  ->  + T E'
3. E’  ->  Λ
4. T   ->  F T'
5. T'  ->  * F T'
6. T’  ->  Λ
7. F   ->  ( E )
8. F   ->  id
```

M[A,t] terminals

nonterms

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

Predictive Parsing Table

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

Table Driven Algorithm

```
push start symbol
Repeat
   begin
    let X top of stack, A next input
        if terminal(X)
           then if X=A
                      then pop X; remove A
                      else error()
           else (* nonterminal(X) *)
  begin
    if M[X,A] = Y1 Y2 ... Yk
       then pop X;
              push Yk YK-1 ... Y1
       else error()
end
until stack is empty, input = $
```

Example Parse

	id	+	*	(	)	$
E	T E′			T E′		
E′		+ T E′			ε	ε
T	F T′			F T′		
T′		ε	* F T′		ε	ε
F	id			(E)		

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