# 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 $A \text{ -> } \beta$ 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).

# Bottom up Parsing

- Bottom up parsing tries to transform the input string into the start symbol.
- Moves through a sequence of sentential forms (sequence of Non-terminal or terminals). Tries to identify some *substring* of the sentential form that is the rhs of some production.
- E -> E + E | E * E | x
  - *x* + x * x
  - E + *x* * x
  - *E + E* * x
  - E * *x*
  - *E * E*
  - E

The substring (shown in color and italics) for each step) may contain both terminal and non-terminal symbols. This string is the rhs of some production, and is often called a handle.

# Bottom Up Parsing

Implemented by Shift-Reduce parsing

- data structures: input-string and stack.
- look at symbols on top of stack, and the input-string and decide:
  - shift (move first input to stack)
  - reduce (replace top n symbols on stack by a non-terminal)
  - accept (declare victory)
  - error (be gracious in defeat)

# Example Bottom up Parse

Consider the grammar: (note: left recursion is NOT a problem, but the grammar is still layered to prevent ambiguity)

```
1. E ::= E + T
2. E ::= T
3. T ::= T * F
4. T ::= F
5. F ::= ( E )
6. F ::= id
```

| *stack* | *Input* | *Action* |
|---|---|---|
| | x + y | shift |
| x | + y | reduce 6 |
| F | + y | reduce 4 |
| T | + y | reduce 2 |
| E | + y | shift |
| E + | y | shift |
| E + y | | reduce 6 |
| E + F | | reduce 4 |
| E + T | | reduce 1 |
| E | | accept |

The concatenation of the stack and the input is a sentential form. The input is all terminal symbols, the stack is a combination of terminal and non-terminal symbols

# LR(k)

- Grammars which can decide whether to shift or reduce by looking at only k symbols of the input are called LR(k).
  - Note the symbols on the stack don't count when calculating k

- L is for a Left-to-Right scan of the input

- R is for the Reverse of a Rightmost derivation

# Problems (ambiguous grammars)

1) shift reduce conflicts:

| *stack* | *Input* | *Action* |
|---|---|---|
| x + y | + z | ? |

| *stack* | *Input* | *Action* |
|---|---|---|
| if x t if y t s2 | e s3 | ? |

2) reduce reduce conflicts:

suppose both procedure call and array reference have similar syntax:

- x(2) := 6
- f(x)

| *stack* | *Input* | *Action* |
|---|---|---|
| id ( id | ) id | ? |

Should id reduce to a parameter or an expression. Depends on whether the bottom most id is an array or a procedure.

# Parsing Algorithms

- Top Down
  - Recursive descent parsers
  - LL(1) or predictive parsers
- Bottom up
  - Precedence Parsers
  - LR(k) parsers

# 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.
- 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)

# 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 ()}
   ]
```

# Predictive Parsers

- Use a stack to avoid recursion. Encoding parsing rules in a 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′ | $ |
| | $ |

# Bottom up table driven parsers

- Operator precedence parsers
- LR parsers

# Example operator precedence parser

| | + | * | ( | ) | id | $ |
|---|---|---|---|---|---|---|
| + | :> | <: | <: | :> | <: | :> |
| * | :> | :> | <: | <: | <: | :> |
| ( | <: | <: | <: | = | <: | |
| ) | :> | :> | | :> | | :> |
| id | :> | :> | | :> | | :> |
| $ | <: | <: | <: | | accept <: | |

input : x * x + y

| *stack* | *Input* | *Action* |
|---|---|---|
| $ E * E | + y $ | reduce! |

↑ topmost terminal (points to * in the stack)

↑ next input (points to + in the input)

# Precedence parsers

- Precedence parsers have limitations
- No production can have two consecutive non-terminals
- Parse only a small subset of the Context Free Grammars
- Need a more robust version of shift- reduce parsing.
- LR - parsers
  - State based - finite state automatons (w / stack)
  - Accept the widest range of grammars
  - Easily constructed (by a machine)
  - Can be modified to accept ambiguous grammars by using precedence and associativity information.

# LR Parsers

- Table Driven Parsers
- Table is indexed by *state* and *symbols* (both term and non-term)
- Table has two components.
  - ACTION part
  - GOTO part

| | terminals | | | | | | non-terminals | | |
|---|---|---|---|---|---|---|---|---|---|
| state | id | + | * | ( | ) | $ | E | T | F |
| 0 | shift (state = 5) | | | | | | | | |
| 1 | | | | | | | | | |
| 2 | | | | | | | goto(state = 2) | | |
| | reduce(prod = 12) | | | | | | | | |
| | *ACTION* | | | | | | *GOTO* | | |

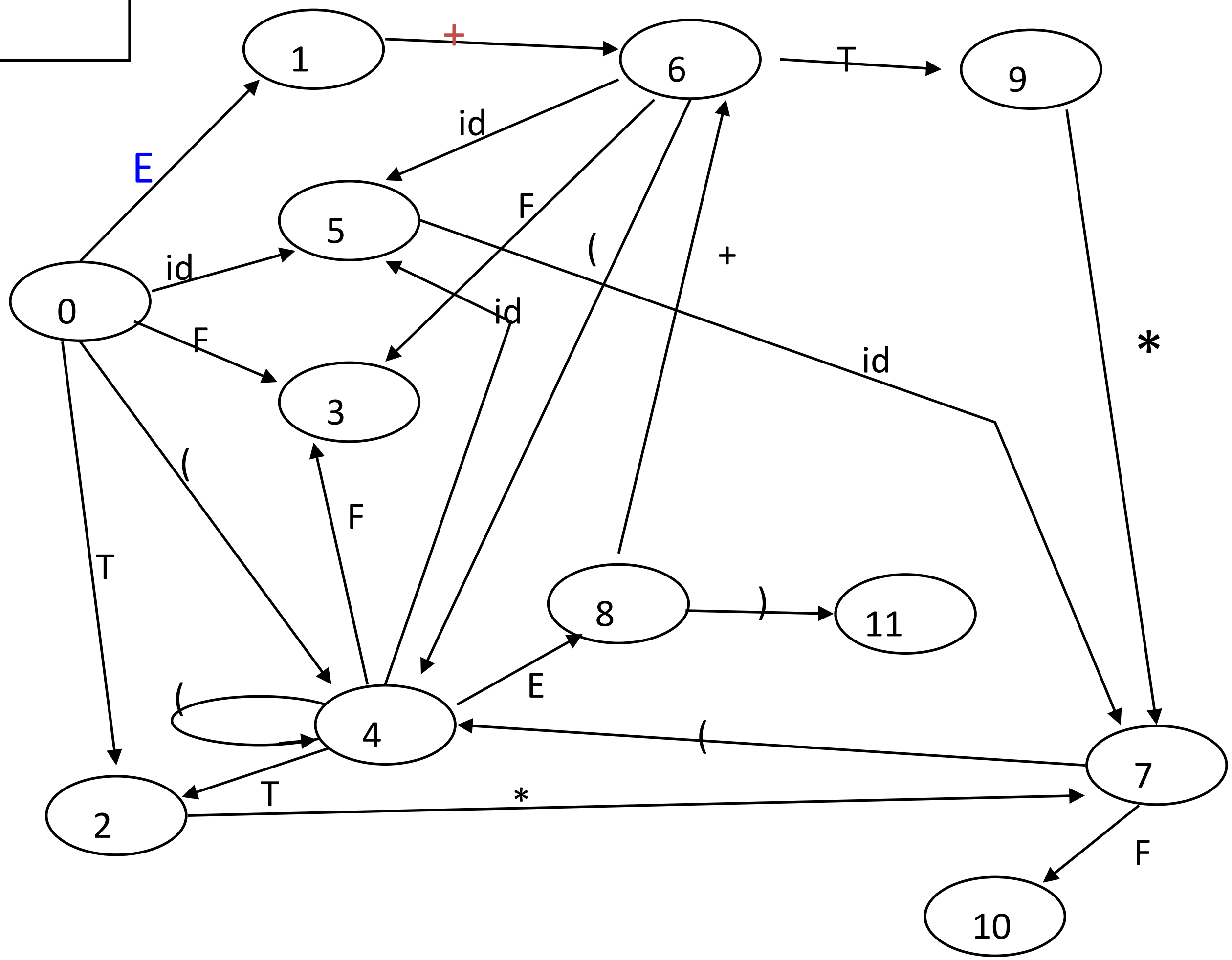
# LR Table encodes FSA

E -> E + T | T

T -> T * F | F

F -> ( E ) | id

transition on terminal is a shift in action table, on nonterminal is a goto entry



# Table vs FSA

- The Table encodes the FSA
- The action part encodes
  - Transitions on terminal symbols (shift)
  - Finding the end of a production (reduce)
- The goto part encodes
  - Tracing backwards the symbols on the RHS
  - Transition on non-terminal, the LHS
- Tables can be quite compact

# LR Table

| state | terminals | | | | | | non-terminals | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | s5 | | | s4 | | | 1 | 2 | 3 |
| 1 | | s6 | | | | acc | | | |
| 2 | | r2 | s7 | | r2 | r2 | | | |
| 3 | | r4 | r4 | | r4 | r4 | | | |
| 4 | s5 | | | s4 | | | 8 | 2 | 3 |
| 5 | | r6 | r6 | | r6 | r6 | | | |
| 6 | s5 | | | s4 | | | | 9 | 3 |
| 7 | s5 | | | s4 | | | | | 10 |
| 8 | | s6 | | | s11 | | | | |
| 9 | | r1 | s7 | | r1 | r1 | | | |
| 10 | | r3 | r3 | | r3 | r3 | | | |
| 11 | | r5 | r5 | | r5 | r5 | | | |

# Reduce Action

- If the top of the stack is the rhs for some production n
- And the current action is “reduce n”
- We pop the rhs, then look at the state on the top of the stack, and index the goto-table with this state and the LHS non-terminal.
- Then push the lhs onto the stack in the new s found in the goto-table.

(?,0)(id,5)     * id + id $

Where:     Action(5,*) = reduce 6

Production 6 is:     F ::= id

And:     GOTO(0,F) = 3

(?,0)(F,3)     * id + id $

# Example Parse

1) E -> E + T
2) E -> T
3) T -> T * F
4) T -> F
5) F -> ( E )
6) F -> id

| *Stack* | *Input* |
|---|---|
| (?,0) | id * id + id $ |
| (?,0)(id,5) | * id + id $ |
| (?,0)(F,3) | * id + id $ |
| (?,0)(T,2) | * id + id $ |
| (?,0)(T,2)(*,7) | id + id $ |
| (?,0)(T,2)(*,7)(id,5) | + id $ |
| (?,0)(T,2)(*,7)(F,10) | + id $ |
| (?,0)(T,2) | + id $ |
| (?,0)(E,1) | + id $ |
| (?,0)(E,1)(+,6) | id $ |
| (?,0)(E,1)(+,6)(id,5) | $ |
| (?,0)(E,1)(+,6)(F,3) | $ |
| (?,0)(E,1)(+,6)(T,9) | $ |
| (?,0)(E,1) | $ |

# Review

- Bottom up parsing transforms the input into the start symbol.
- Bottom up parsing looks for the rhs of some production in the partially transformed intermediate result
- Bottom up parsing is OK with left recursive grammars
- Ambiguity can be used to your advantage in bottom up partsing.
- The LR(k) languages = LR(1) languages = CFL

# More detail

- The slides that follow give more detail on several of the parsing algorithms
- These slides are for your own edification.

# Using ambiguity to your advantage

- Shift-Reduce and Reduce-Reduce errors are caused by ambiguous grammars.

- We can use resolution mechanisms to our advantage. Use an ambiguous grammar (smaller more concise, more natural parse trees) but resolve ambiguity using rules.

- Operator Precedence
  - Every operator is given a precedence
  - Precedence of the operator closest to the top of the stack and the precedence of operator next on the input decide shift or reduce.
  - Sometimes the precedence is the same. Need more information: Associativity information.

# Operations on Grammars

- 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 $\Lambda$ is in First(symbol) then that symbol is nullable.
- Sometime rather than let $\Lambda$ 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 -> $\Lambda$ is a production then add $\Lambda$ 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 $\Lambda$ in FIRST(Yj)

- E.g.. if Y1 can derive $\Lambda$ 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 $\Lambda$ to First(E'), add $\Lambda$ to First(T')
- Other NonTerminals
  - Computing from the lowest layer (F) up
    - First(F) = {id , ( }
    - First(T') = { $\Lambda$, * }
    - First(T) = First(F) = {id, ( }
    - First(E') = { $\Lambda$, + }
    - 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 $\Lambda$ is in FOLLOW(B)

3) If there is a production A -> *a* B
or A -> *a* B *b* where FIRST(*b*)
contains $\Lambda$ (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 | |