Advanced Programming
Andrew Black and Tim Sheard

Lecture 10
Grammars
NFAs
Grammars 1

• Grammar
  – A set of tokens (terminals): T
  – A set of non-terminals: N
  – A set of productions \{ \text{lhs} \rightarrow \text{rhs} , \ldots \}
    • \text{lhs} \in N
    • \text{rhs} is a sequence of N U T
  – A Start symbol: S (in N)
In Haskell

data Production symbol =
    Prod { lhs:: symbol
          , rhs:: [symbol] }

data Grammar symbol =
    Gr { -- the set of terminal symbols
          term:: [symbol]
          -- the set of non-terminals
          , nonterm:: [symbol]
          -- the set of productions
          , prod:: [Production symbol]
          -- the start symbol
          , start:: symbol }

Record syntax

term :: Grammar a -> [a]
Main> :t nonterm
nonterm :: Grammar a -> [a]
Main> :t prod
prod :: Grammar a -> [Production a]
Main> :t start
start :: Grammar a -> a

Main> :t lhs
lhs :: Production a -> a
Main> :t rhs
rhs :: Production a -> [a]
Example Grammar

g1 = Gr ["1","2","","[","",""]"
    ["list1","elem","list2","list3"]
    [Prod "list1" ["[", "list2", "]"]]
    ,Prod "list2" []
    ,Prod "list2" ["elem", "list3"]
    ,Prod "list3" [ ",", "elem", "list3" ]
    ,Prod "list3" []
    ,Prod "elem" ["1"]
    ,Prod "elem" ["2"]
    "list1"
Terminals = {"","", "1", "2", "[", "]"} 
NonTerminals = {elem, list1, list2, list3} 
Start = list1 
  list1 -> [ list2 ] 
  list2 -> 
  list2 -> elem list3 
  list3 -> , elem list3 
  list3 -> 
  elem -> 1 
  elem -> 2
Shortcut construction

- We can build a grammar form a list of its productions
  - Provide only the productions
    - All lhs symbols comprise N
    - All other symbols comprise T
    - Lhs of first production is S

```
shortcut ps = Gr term nonterm ps start
  where (Prod start _ : _) = ps
    nonterm = norm(map (\ (Prod lhs rhs) -> lhs) ps)
    all = concat(map (\ (Prod lhs rhs) -> lhs:rhs) ps)
    term = norm all \ nonterm
```
Example 2

prods2 =
[Prod "Sent" ["nounPhrase","verbPhrase"]
,Prod "nounPhrase" [ "properNoun" ]
,Prod "nounPhrase" [ "article", "noun" ]
,Prod "article" [ "a " ]
,Prod "article" ["the "]
,Prod "properNoun" ["Tom "]
,Prod "noun" ["cat "]
,Prod "noun" ["man "]
,Prod "verbPhrase" [ "verb", "object"]
,Prod "verb" [ "ate "]
,Prod "verb" [ "stole "]
,Prod "object" [ "article", "adjective", "noun" ]
,Prod "adjective" [ "pretty "]
,Prod "adjective" [ "red "]
]
Main> shortcut prods2

Terminals = {"Tom ", "a ", "ate ", "cat ", "man ", "pretty ", "red ", "stole", "the "}
NonTerminals = {Sent, adjective, article, noun, nounPhrase, object, properNoun, verb, verbPhrase}
Start = Sent
   Sent -> nounPhrase verbPhrase
   nounPhrase -> properNoun
   nounPhrase -> article noun
   article -> a
   article -> the
   properNoun -> Tom
   noun -> cat
   noun -> man
   verbPhrase -> verb object
   verb -> ate
   verb -> stole
   object -> article adjective noun
   adjective -> pretty
   adjective -> red
Example 3

prods3 =
[Prod "E" ["T","E'", ",$"]
,Prod "E'" [ "+", "T", "E'" ]
,Prod "E'" []
,Prod "T" [ "F", "T'" ]
,Prod "T'" [ "+", "F", "T'" ]
,Prod "T'" []
,Prod "F" ["(" , "E", ")"]
,Prod "F" ["Id"]
,Prod "Id" [ "x" ]
]

g3 = shortcut prods3
Pretty Printed

Terminals = {"$", "(" , ")", "*", "+", "x"}
NonTerminals = {E, E', F, Id, T, T'}
Start = E
  E -> T E' $
  E' -> + T E'
  E' ->
  T -> F T'
  T' -> * F T'
  T' ->
  F -> ( E )
  F -> Id
  Id -> x
Meaning of a grammar

• A grammar can be given several meanings
  – As a generator
  – As a recognizer

• Any recursive grammar generates an infinite set of strings
Generating Grammars

• Start with a non-terminal
• Rewriting rules
  – Pick a non-terminal to replace. Replace it with the rhs of one of its production.
• Repeat until no non-terminals remain

• A sentence of G: \( L(G) \)
  – Start with S
  – only terminal symbols
  – all strings derivable from G in 1 or more steps
Technique

- Start with a nonterm
  - choose `nounPhrase`
- Pick a production
  - choose:
    - `nounPhrase -> article noun`
- Generate all strings from each symbol in the rhs
  - `article = ["a ","the "]`
  - `noun = ["cat ","man "]`
- Make all possible combinations
  - “a cat”
  - “a man”
  - “the cat”
  - “the man”
- If there is more than 1 production for a symbol, compute sentences for each one, and then union them together.
All Possible combinations

oneEach :: [[String]] -> [String]
oneEach [] = ["""]
oneEach (x:xs) = [ a++b | a <- x
                  , b <- oneEach xs ]

Main> oneEach [["x","y"],["1","2"],["A","B","C"]]
["x1A","x1B","x1C","x2A","x2B","x2C" ,"y1A","y1B","y1C","y2A","y2B","y2C"]
First Try

genA :: Grammar String -> String -> [String]

genA (Gr term nonterm ps start) symbol
  | elem symbol term = [symbol]

genA (gram@(Gr term nonterm ps start)) symbol
  = concat many
    where startsWith sym (Prod lhs rhs) = lhs==sym
        prods = filter (startsWith symbol) ps
        oneRhs (Prod lhs rhs) =
          oneEach(map (genA gram) rhs)
        many = map oneRhs prods
Test it

test1 = mapM putStrLn (genA g2 "Sent")

Main> test1
Tom ate a pretty cat
Tom ate a pretty man
Tom ate a red cat
Tom ate a red man
Tom ate the pretty cat
Tom ate the pretty man
Tom ate the red cat
Tom ate the red man
Tom stole a pretty cat
Tom stole a pretty man
Tom stole a red cat
Tom stole a red man
Tom stole the pretty cat
Tom stole the pretty man
Tom stole the red cat
Tom stole the red man
a cat ate a pretty cat
a cat ate a pretty man
...

Let's try it on grammar g1

- What happens?
- Why?

Terminals = {"", "1", "2", "[", "]"}
NonTerminals = {elem, list1, list2, list3}
Start = list1
  list1 -> [ list2 ]
  list2 ->
  list2 -> elem list3
  list3 -> , elem list3
  list3 ->
  elem -> 1
  elem -> 2
Cut of generation after using a non-term n times

• Create a table of cutoff depths for each non-term.
  – type Tab = [(String, Int)]

• Operations on tables
  – Decrementing the cutoff for a given symbol
    decrement :: String -> Tab -> Tab
decrement s [] = []
decrement s ((x, n):xs)
  | s==x = (x, n-1):xs
  | True = (x, n):decrement s xs

  – Finding the cutoff
    find :: Eq a => a -> [(a, b)] -> b
find s ((x, n):xs) | s==x = n
  | True = find s xs
Second try

\[
\text{genB} :: \text{Grammar String} \rightarrow \text{Tab} \rightarrow \text{String} \rightarrow [\text{String}]
\]

\[
\text{genB} \ (\text{Gr term nonterm ps start}) \ \text{table symbol} \\
\quad | \ \text{elem symbol term} = [\text{symbol}]
\]

\[
\text{genB} \ (\text{gram@(Gr term nonterm ps start)}) \ \text{table symbol} \\
\quad | \ \text{find symbol table} \leq 0 = [] \\
\quad | \ \text{True} = \text{concat many}
\]

where \text{startsWith sym} \ (\text{Prod lhs rhs}) = \text{lhs==sym}

\text{prods} = \text{filter} \ (\text{startsWith symbol}) \ \text{ps}

\text{new sym} = \text{decrement sym table}

\text{oneRhs} \ (\text{Prod lhs rhs}) =

\quad \text{oneEach(map (genB gram (new lhs)) rhs)}

\text{many} = \text{map oneRhs prods}
Putting it all together

gen n (gram@(Gr term nonterm ps start))
    = genB gram table start
where table = map f nonterm
    f sym = (sym,n)
Test it

Main> gen 3 g1
["[]","[1,1,1]","[1,1,2]","[1,1]","[1,2,1]","[1,2,2]","[1,2]","[1]","[2,1,1]","[2,1,2]","[2,1]","[2,2,1]","[2,2,2]","[2,2]","[2]"}
Top Down Parsing

- Begin with the start symbol and try to derive the parse tree from the root which matches the given string.
- Consider the grammar:
  \[ \text{Exp} \rightarrow \text{id} \]
  \[ \quad | \text{Exp} + \text{Exp} \]
  \[ \quad | \text{Exp} \times \text{Exp} \]
  \[ \quad | (\text{Exp}) \]

  Derives \( x, x+x, x+x+x, \)
  \( x \times y \)
  \( x + y \times z \ldots \)
Example Parse (top down)

- stack          input
  Exp             x + y * z
     Exp         x + y * z
       / | \       Exp + Exp
      / | \       Exp + Exp
     / | \       Exp + Exp
    / | \       Exp + Exp
   / | \       Exp + Exp
  / | \       Exp + Exp
 id(x)
Top Down Parse (cont)

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

Top Down Parse (cont.)

```
Exp
  /   \
Exp + Exp
  |    |
|    |   |
id(x) Exp * Exp
  |    |
|    |
id(y) id(z)
```
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'  ->  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  = {"+", "(" , "$"}  
First E' = { "+", ε}  Follow E' = {")", "$"}  
First T' = { "*", ε}  Follow T' = {"+","(" , "$"}  

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

- If \( E \rightarrow \varepsilon \) then \( E \) is nullable
- If \( E \rightarrow A \ B \ C \), and all of \( A, B, C \) are nullable then \( E \) is nullable.

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

- This is a fixpoint computation
In Haskell

- We’ll represent nullable, first and follow sets as tables.

```haskell
data Table elem = Tab [(String,elem)] deriving Show

instance Eq elem => Eq (Table elem) where
    (Tab xs) == (Tab ys) = sameRhs xs ys
    where sameRhs [] [] = True
    sameRhs ((x,rhs1):xs) ((y,rhs2):ys)
        = rhs1==rhs2 && sameRhs xs ys
    sameRhs _ _ = False
```
Operations on Tables

get :: String -> Table elem -> elem
get s (Tab xs) = find s xs

set :: Table elem -> String -> elem -> Table elem
set (Tab xs) s y = Tab(update xs)
  where update [] = []
       update ((t,_) : ys) | t==s = (t,y) : ys
       update (y:ys) = y : (update ys)

add :: String -> String -> Table [String] -> Table [String]
add symbol element (Tab xs) = Tab(insert xs)
  where insert [] = []
       insert ((s,xs) : ys)
            | s==symbol = (s,norm(element:xs)) : ys
       insert (y:ys) = y : (insert ys)
nullable :: Table Bool -> String -> Bool
nullable tab s = get s tab

nullStep (gram@(Gr term nonterm ps start)) table = newtable
  where nulls (Prod lhs rhs) = all (nullable table) rhs
    nullps = filter nulls ps
    newtable = foldl acc table nullps
    acc tab (Prod lhs rhs) = set tab lhs True

null (gram@(Gr term nonterm ps start)) =
  fixpoint (nullStep gram) simple
  where simple = Tab (map f (term++nonterm))
    f x = (x,False)
Main> null g1
Tab [("","False),("1",False),("2",False),("[",False),("]",False)
    ,("elem",False), ("list1",False),("list2",True)
    ,("list3",True)]

Main> null g2
Tab [("Tom ",False),("a ",False),("ate ",False)
    ,("cat ",False),("man",False),("pretty",False)
    ,("red",False),("stole ",False),("the",False),("Sent",False)
    ,("adjective",False),("article",False),("noun",False)
    ,("nounPhrase",False),("object",False),("properNoun",False)
    ,("verb",False),("verbPhrase",False)]

Main> null g3
Tab [("$",False),("("False),(""",False),("*",False),("+",False)
    ,("x",False),("E",False),("E'",True),("F",False)
    ,("Id",False),("T",False),("T'",True)]
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 \rightarrow ε is a production then add ε to FIRST(X), (Or set nullable of X to true).
  3) if X is a non-term and
     - X \rightarrow Y_1 Y_2 ... Y_k
     - add a to FIRST(X)
       - if a in FIRST(Y_i) and
       - for all j<i ε in FIRST(Y_j)

- E.g.. if Y_1 can derive ε then if a is in FIRST(Y_2) it is surely in FIRST(X) as well.
Example First Computation

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

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

- **Other NonTerminals**
  - Computing from the lowest layer (F) up
    - First(F) = {id, ( }
    - First(T') = { $\epsilon$, * }
    - First(T) = First(F) = {id, ( }
    - First(E') = { $\epsilon$, + }
    - 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.

2) If $ A \rightarrow aBb$ then everything in FIRST(b) except $\varepsilon$ is in FOLLOW(B)

3) If there is a production $ A \rightarrow aB$ or $A \rightarrow aBb$ where FIRST(b) contains $\varepsilon$ (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') = {+,ε} to Follow(T)
  - Add First(T') = {* ,ε} 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' --> 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

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,ε] for each terminal b in Follow A. If ε is in First alpha and $ is in Follow A add A -> alpha to M[A,$].

<table>
<thead>
<tr>
<th>First</th>
<th>Follow</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>{&quot;(&quot;&quot;id&quot;)}</td>
</tr>
<tr>
<td>F</td>
<td>{&quot;(&quot;,&quot;id&quot;)}</td>
</tr>
<tr>
<td>T</td>
<td>{&quot;(&quot;&quot;id&quot;)}</td>
</tr>
<tr>
<td>E'</td>
<td>{&quot;+&quot;,ε}</td>
</tr>
<tr>
<td>T'</td>
<td>{&quot;**&quot;,ε}</td>
</tr>
</tbody>
</table>

M[A,t] terminals

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>id</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>(</td>
<td>E</td>
<td>)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>id</td>
<td></td>
</tr>
</tbody>
</table>
# 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><strong>E</strong></td>
<td>T E'</td>
<td></td>
<td></td>
<td>T E'</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>E'</strong></td>
<td></td>
<td>+ T E'</td>
<td></td>
<td></td>
<td>ε</td>
<td>ε</td>
</tr>
<tr>
<td><strong>T</strong></td>
<td>F T'</td>
<td></td>
<td></td>
<td>F T'</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>T'</strong></td>
<td></td>
<td></td>
<td>* F T'</td>
<td></td>
<td>ε</td>
<td>ε</td>
</tr>
<tr>
<td><strong>F</strong></td>
<td>id</td>
<td></td>
<td></td>
<td>( E )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
  end
until stack is empty, input = $