Lecture 7: Top-Down Parsing

• HW #2 now available.

• Please fill out our background survey (see homework page).
Beating Grammars into Programs

• A BNF grammar looks like a recursive program. Sometimes it works to treat it that way.

• Assume the existence of
  - A function ‘next’ that returns the syntactic category of the next token (without side-effects);
  - A function ‘scan(C)’ that checks that next syntactic category is C and then reads another token into next(). Returns the previous value of next().
  - A function ERROR for reporting errors.

• Strategy: Translate each nonterminal, $A$, into a function that reads an $A$ according to one of its productions and returns the semantic value computed by the corresponding action.

• Result is a **recursive-descent** parser.
Example: Lisp Expression Recognizer

Grammar

```
def prog ():
    
    def sexp ():
        if ________________:
            
            elif ____________:
                
                else:
                    
                    def atom ():
                        if ________________:
                            
                            else:
                                
                                def elist ():
                                    if ________________:
                                        
                                        ```

prog : sexp ’-‘
sexp : atom
    | ’(‘ elist ’)’
    | ’\’’ sexp
elist : ε
    | sexp elist
atom : SYM
    | NUM
    | STRING

Last modified: Sun Feb 8 17:56:50 2015
Example: Lisp Expression Recognizer

Grammar

```
def prog ():
    sexp(); scan(-)

    def sexp ():
        if ________________:
            ________
        elif ____________:
            __________________
        else:
            __________________

def atom ():
    if ____________________:
        ________
    else:
        ________

def elist ():
    if ____________________:
        ________
```

```
prog: sexp ’-’
sexp: atom
    | ’(’ elist ’)’
    | ’\’’ sexp
elist: ε
    | sexp elist
atom: SYM
    | NUM
    | STRING
```

Last modified: Sun Feb 8 17:56:50 2015
Example: Lisp Expression Recognizer

Grammar

```python
def prog ():
    sexp(); scan(-)

def sexp () :
    if next() in [SYM, NUM, STRING]:
        atom()
    elif ________:
        ________________________
    else:
        ________________________

def atom () :
    if ________________________:
        _______
    else:
        _______

def elist () :
    if ________________________:
        __________
```

```
prog : sexp '→'
sexp : atom
    | ' (' elist ')'
    | ' \' ' sexp
elist : ε
    | sexp elist
atom : SYM
    | NUM
    | STRING
```
Example: Lisp Expression Recognizer

Grammar

```python
def prog ():
    sexp(); scan(¬)

def sexp ():
    if next() in [SYM, NUM, STRING]:
        atom()
    elif next() == '(':  
        scan('‚'); elist(); scan(')')
    else:
        __________________________

def atom ():
    if __________________________:
        __________
    else:
        ______

def elist ():
    if __________________________:
        __________
```

prog : sexp ¬|
sexp : atom
    | ('elist ')
    | '\' sexp
elist : ε
    | sexp elist
atom : SYM
    | NUM
    | STRING

Last modified: Sun Feb 8 17:56:50 2015 CS164: Lecture #7 3
Example: Lisp Expression Recognizer

Grammar

```
prog : sexp '⊣'
sexp : atom
    | '(' elist ')' 
    | '\'' sexp
elist : ε
    | sexp elist
atom : SYM
    | NUM
    | STRING
```

```
def prog ():
    sexp(); scan('⊣')

def sexp ():
    if next() in [SYM, NUM, STRING]:
        atom()
    elif next() == '(':
        scan('('); elist(); scan(')')
    else:
        scan('\''); sexp()

def atom ():
    if ____________________________:
        
        
    else:
        
        

def elist ():
    if ____________________________:
        
        
```
Example: Lisp Expression Recognizer

Grammar

```
def prog ():
    exp(); scan(‘-‘)

def exp ():
    if next() in [SYM, NUM, STRING]:
        atom()
    elif next() == ‘(‘:
        scan(‘(‘); elist(); scan(‘)’)
    else:
        scan(‘\’’); exp()

def atom ():
    if next() in [SYM, NUM, STRING]:
        scan(next())
    else:
        __________

def elist ():
    if ________________:
        __________
```

```
prog : exp ’-‘
exp : atom
    | ‘(‘ elist ‘)’
    | ‘\’’ exp
elist : ε
    | exp elist
atom : SYM
    | NUM
    | STRING
```
Example: Lisp Expression Recognizer

**Grammar**

```python
def prog ():
    sexp(); scan(-|

def sexp ():
    if next() in [SYM, NUM, STRING]:
        atom()
    elif next() == '(':  # add 'as' for correct grammar
        scan('('); elist(); scan(')')
    else:
        scan('\'); sexp()

def atom ():
    if next() in [SYM, NUM, STRING]:
        scan(next())
    else:
        ERROR()

def elist ():
    if ________________:
        ________________
```

```python
def prog ():
    sexp(); scan(-|

def sexp ():
    if next() in [SYM, NUM, STRING]:
        atom()
    elif next() == '(':  # add 'as' for correct grammar
        scan('('); elist(); scan(')')
    else:
        scan('\'); sexp()

def atom ():
    if next() in [SYM, NUM, STRING]:
        scan(next())
    else:
        ERROR()

def elist ():
    if ________________:
        ________________
```
Example: Lisp Expression Recognizer

Grammar

```
def prog ():
    sexp(); scan(−)

def sexp () :
    if next() in [SYM, NUM, STRING]:
        atom()
    elif next() == ’(‘:
        scan(’’); elist(); scan(’’)
    else:
        scan(’’); sexp()

def atom () :
    if next() in [SYM, NUM, STRING]:
        scan(next())
    else:
        ERROR()

def elist () :
    if next() in [SYM, NUM, STRING, ’(‘, ’)’]:
        sexp(); elist();
```

```
prog : sexp ’−|’
sexp : atom
    | ’(’ elist ’)’
    | ’\’’ sexp
elist : ε
    | sexp elist
atom : SYM
    | NUM
    | STRING
```
Expression Recognizer with Actions

- Can make the nonterminal functions return semantic values.
- Assume lexer somehow supplies semantic values for tokens, if needed

```python
elist : ε           { $$ = emptyList; }
     | sexp elist   { $$ = cons($1, $2); }
```

def elist ():
    if next() in [SYM, NUM, STRING, '(', '\']:
        ________________________________
    else:
        return emptyList
```
Expression Recognizer with Actions

- Can make the nonterminal functions return semantic values.
- Assume lexer somehow supplies semantic values for tokens, if needed

```python
eлист : ε                 { $$ = emptyList; }
      | секс п елист         { $$ = cons($1, $2); }

def elist ():
    if next() in [SYM, NUM, STRING, '(', '\']:
        v1 = sexp(); v2 = elist(); return cons(v1, v2)
    else:
        return emptyList
```

Last modified: Sun Feb 8 17:56:50 2015
Grammar Problems I

In a recursive-descent parser, what goes wrong here?

\[
\begin{align*}
p : & \ e \ '−' \\
e : & \ t \quad \{ \ $$ = \$1; \} \\
| \ & e \ '/' t \quad \{ \ $$ = \text{makeTree}(\text{DIV}, \$1, \$3); \} \\
| \ & e \ '*' t \quad \{ \ $$ = \text{makeTree}(\text{MULT}, \$1, \$3); \}
\end{align*}
\]
Grammar Problems I

In a recursive-descent parser, what goes wrong here?

```
p : e '->'
   e : t { $$ = $1; }
   | e '/' t { $$ = makeTree(DIV, $1, $3); }
   | e '*' t { $$ = makeTree(MULT, $1, $3); }
```

If we choose the second or third alternative for $e$, we'll get an infinite recursion. If we choose the first, we'll miss '/' and '*' cases.
Grammar Problems II

Well then: What goes wrong here?

\[ p : e \ '−' \]
\[ e : t \{
\text{ $$ = $1; } \}
\]
\[ t '/' e \{ $$ = \text{makeTree(DIV, $1, $3); } \}
\]
\[ t '*' e \{ $$ = \text{makeTree(MULT, $1, $3); } \}
\]
Grammar Problems II

Well then: What goes wrong here?

\[
\begin{align*}
p & : e \;'-'
\end{align*}
\]

\[
\begin{align*}
e & : t \quad \{ \quad $$ = $1; \quad } \\
| & \quad t \; '\/'\; e \quad \{ \quad $$ = \text{makeTree}(\text{DIV}, \; $1, \; $3); \quad } \\
| & \quad t \; '\*\'\; e \quad \{ \quad $$ = \text{makeTree}(\text{MULT}, \; $1, \; $3); \quad }
\end{align*}
\]

No infinite recursion, but we still don’t know which right-hand side to choose for e.
FIRST and FOLLOW

- If $\alpha$ is any string of terminals and nonterminals (like the right side of a production) then $\text{FIRST}(\alpha)$ is the set of terminal symbols that start some string that $\alpha$ produces, plus $\epsilon$ if $\alpha$ can produce the empty string. For example:

  \[
  \begin{align*}
  p & : e \rightarrow ' \\
  e & : s \ t \\
  s & : \epsilon \mid '+' \mid '-' \\
  t & : \text{ID} \mid '(' e ')'
  \end{align*}
  \]

  Since $e \Rightarrow s \ t \Rightarrow ( e ) \Rightarrow \ldots$, we know that `' $\in \text{FIRST}(e)$. Since $s \Rightarrow \epsilon$, we know that $\epsilon \in \text{FIRST}(s)$.

- If $X$ is a non-terminal symbol in some grammar, $G$, then $\text{FOLLOW}(X)$ is the set of terminal symbols that can come immediately after $X$ in some sentential form that $G$ can produce. For example, since $p \Rightarrow e \rightarrow \Rightarrow s \ t \rightarrow \Rightarrow s '\ ( e ') \rightarrow \rightarrow \ldots$, we know that `' $\in \text{FOLLOW}(s)$.
Using FIRST and FOLLOW

• In a recursive-descent compiler where we have a choice of right-hand sides to produce for non-terminal, $X$, look at the FIRST of each choice and take it if the next input symbol is in it...

• ...and if a right-hand side’s FIRST set contains $\epsilon$, take it if the next input symbol is in FOLLOW($X$).
Grammar Problems III

What actions?

\[
\begin{align*}
p & : e \ '->' \\
e & : t \ et \ { \ ?1 } \\
et & : \epsilon \ { \ ?2 } \\
| & \ '/' \ e \ { \ ?3 } \\
| & \ '*' \ e \ { \ ?4 } \\
t & : I \ { \ $$ = $1; } 
\end{align*}
\]

What are FIRST and FOLLOW?
Grammar Problems III

What actions?

\[
\begin{align*}
p & : e \rightarrow t \\
e & : t \text{ or } t e \rightarrow \text{?1} \\
et & : \epsilon \rightarrow \text{?2} \\
& \quad | \text{'/' } e \rightarrow \text{?3} \\
& \quad | \text{'*' } e \rightarrow \text{?4} \\
t & : I \rightarrow \text{?1} \\
\end{align*}
\]

Here, we don't have the previous problems, but how do we build a tree that associates properly (left to right), so that we don't interpret I/I/I as if it were I/(I/I)?

What are FIRST and FOLLOW?
Grammar Problems III

What actions?

\[
\begin{align*}
  p & : e \ '⊣' \\
  e & : t \ et \ \\  et & : \epsilon \ \\  & \ | \ ' / ' \ e \ \\  & \ | \ ' * ' \ e \\
  t & : I
\end{align*}
\]

Here, we don't have the previous problems, but how do we build a tree that associates properly (left to right), so that we don't interpret \( I/I/I \) as if it were \( I/(I/I) \)?

What are FIRST and FOLLOW?

\[
\begin{align*}
  \text{FIRST}(p) & = \text{FIRST}(e) = \text{FIRST}(t) = \{ I \} \\
  \text{FIRST}(et) & = \{ \epsilon, ' / ', ' * ' \} \\
  \text{FIRST}(' / ' \ e) & = \{ ' / ' \} \quad \text{(when to use ?3)} \\
  \text{FIRST}(' * ' \ e) & = \{ ' * ' \} \quad \text{(when to use ?4)} \\
  \text{FOLLOW}(e) & = \{ ' ⊣ ' \} \\
  \text{FOLLOW}(et) & = \text{FOLLOW}(e) \quad \text{(when to use ?2)} \\
  \text{FOLLOW}(t) & = \{ ' ⊣ ', ' / ', ' * ' \}
\end{align*}
\]
Using Loops to Roll Up Recursion

• There are ways to deal with problem in last slide within the pure framework, but why bother?

• Implement e procedure with a loop, instead:

  def e():
      ____________
      while ______________:
          if ____________:
              ______________
               ______________
          else:
              ______________
               ______________
      return ____________
Using Loops to Roll Up Recursion

- There are ways to deal with problem in last slide within the pure framework, but why bother?
- Implement e procedure with a loop, instead:

```python
def e():
    r = t()
    while ________________:
        if ____________:
            __________________
            __________________
            __________________
        else:
            __________________
            __________________
            __________________
    return ________________
```
Using Loops to Roll Up Recursion

- There are ways to deal with problem in last slide within the pure framework, but why bother?

- Implement `e` procedure with a loop, instead:

```python
def e():
    r = t()
    while next() in ['/','*']:
        if ____________:
            ______________
            ______________
            ______________
        else:
            ______________
            ______________
        return _
```
Using Loops to Roll Up Recursion

- There are ways to deal with problem in last slide within the pure framework, but why bother?

- Implement e procedure with a loop, instead:

```python
def e():
    r = t()
    while next() in [’/’, ’*’]:
        if next() == ’/’:
            scan(’/’); t1 = t()
            r = makeTree(DIV, r, t1)
        else:
            #
            #
            #
    return _
```

Last modified: Sun Feb 8 17:56:50 2015
Using Loops to Roll Up Recursion

• There are ways to deal with problem in last slide within the pure framework, but why bother?

• Implement e procedure with a loop, instead:

```python
def e():
    r = t()
    while next() in [’/’, ’*’]:
        if next() == ’/’:
            scan(’/’); t1 = t()
            r = makeTree (DIV, r, t1)
        else:
            scan(’*’); t1 = t()
            r = makeTree (MULT, r, t1)
    return _
```

Last modified: Sun Feb 8 17:56:50 2015 CS164: Lecture #7 10
Using Loops to Roll Up Recursion

• There are ways to deal with problem in last slide within the pure framework, but why bother?

• Implement e procedure with a loop, instead:

```python
def e():
    r = t()
    while next() in [’/’, ’*’]:
        if next() == ’/’:
            scan ’/’; t1 = t()
            r = makeTree (DIV, r, t1)
        else:
            scan ’*’; t1 = t()
            r = makeTree (MULT, r, t1)
    return r
```