Lecture 8: Top-Down Parsing
Beating Grammars into Programs

• A 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
```
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: Mon Feb 7 11:05:33 2011
Example: Lisp Expression Recognizer

Grammar

```
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
```

Last modified: Mon Feb 7 11:05:33 2011
Example: Lisp Expression Recognizer

**Grammar**

\[
\begin{align*}
\text{prog} & : \text{sexp} \ '('|')' \\
\text{sexp} & : \text{atom} \\
& \quad | '(' \text{elist} ')' \\
& \quad | '\\' \ '' \text{sexp} \\
\text{elist} & : \epsilon \\
& \quad | \text{sexp} \text{elist} \\
\text{atom} & : \text{SYM} \\
& \quad | \text{NUM} \\
& \quad | \text{STRING}
\end{align*}
\]

```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 
                
            ```
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 ______________:
        
        
    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:  # scan(')'); sexp()

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

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

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

Grammar

\[
\begin{align*}
\text{prog} & : \text{sexp } \texttt{'}-\texttt{'} \\
\text{sexp} & : \text{atom} \\
& \quad \mid ( \text{'} \text{elist} \texttt{')'} \\
& \quad \mid \backslash \texttt{'} \text{sexp} \\
\text{elist} & : \varepsilon \\
& \quad \mid \text{sexp } \text{elist} \\
\text{atom} & : \text{SYM} \\
& \quad \mid \text{NUM} \\
& \quad \mid \text{STRING} \\
\end{align*}
\]

```
def prog ():
    sexp(); scan(\texttt{'}-\texttt{'}

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

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

def elist ():
    if \text{el{\_}list()}:
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
eлист : ε                              { $$ = emptyList; }
     | сexp елист                   { $$ = cons($1, $2); }

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

Last modified: Mon Feb 7 11:05:33 2011
Expression Recognizer with Actions

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

\[
elist : \epsilon \quad \{ \$$ = \text{emptyList}; \} \\
| \text{sexp } \text{elist} \quad \{ \$$ = \text{cons}($1, $2); \}
\]

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

Last modified: Mon Feb 7 11:05:33 2011
Grammar Problems I

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

\[
p : e \,'-\,'
\]
\[
e : t \quad \{ \quad $$ = $1; \quad \}
\]
\[
| \quad e \,'/\,' t \quad \{ \quad $$ = \text{makeTree(DIV, }$1, $3); \quad \}
\]
\[
| \quad e \,'*' t \quad \{ \quad $$ = \text{makeTree(MULT, }$1, $3); \quad \}
\]
In a recursive-descent parser, what goes wrong here?

```plaintext
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 \; \rightarrow \\
 e : t \\
| t \; '/' \; e \quad \{ \; $$ = \text{makeTree}(\text{DIV}, \; \$1, \; \$3); \; \}
| t \; '*' \; e \quad \{ \; $$ = \text{makeTree}(\text{MULT}, \; \$1, \; \$3); \; \}
\]
Grammar Problems II

Well then: What goes wrong here?

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

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:

$$\begin{align*}
p : & \quad e \rightarrow \prime \\
e : & \quad s \ t \\
s : & \quad \epsilon \mid + \mid - \\
t : & \quad \text{ID} \mid (e) \\
\end{align*}$$

Since $e \Rightarrow s \ t \Rightarrow (e) \Rightarrow \ldots$, we know that '$\prime$' $\in$ $\text{FIRST}(e)$. 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 $p \Rightarrow e \rightarrow \Rightarrow e \Rightarrow s \ t \rightarrow \Rightarrow s \ (e) \rightarrow \ldots$, we know that '$\prime$' $\in$ $\text{FOLLOW}(s)$. 
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?

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

What are FIRST and FOLLOW?
Grammar Problems III

What actions?

\[ p : e \rightarrow|' \]
\[ e : t \ et \ \{ \ ?1 \ \} \]
\[ et: \varepsilon \ \{ \ ?2 \ \} \]
\[ \mid '/' \ e \ \{ \ ?3 \ \} \]
\[ \mid '*' \ e \ \{ \ ?4 \ \} \]
\[ t : I \ \{ \ $$ = $1; \} \]

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?

```
p : e '⊣'
e : t et { ?1 }
et: ε { ?2 }
   | '/' e { ?3 }
   | '*' e { ?4 }
t : I { $$ = $1; }
```

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?

- \( \text{FIRST}(p) = \text{FIRST}(e) = \text{FIRST}(t) = \{ I \} \)
- \( \text{FIRST}(et) = \{ ε, '/', '*' \} \)
- \( \text{FIRST}('/' e) = \{ '/' \} \) (when to use ?3)
- \( \text{FIRST}('*' e) = \{ '*' \} \) (when to use ?4)
- \( \text{FOLLOW}(e) = \{ '⊣' \} \)
- \( \text{FOLLOW}(et) = \text{FOLLOW}(e) \) (when to use ?2)
- \( \text{FOLLOW}(t) = \{ '⊣', '/', '*' \} \)
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 $e$():
    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 _
```
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 _
```
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
```