Lecture 5: 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 _________________:
        ______
Example: Lisp ExpressionRecognizer

Grammar

```
def prog ():
    sexp(); scan(-1)

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

```
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
```
Example: Lisp Expression Recognizer

Grammar

```
def prog ():
    sexp(); scan(\n)

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

```
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 ________________:
       __________
```
**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 next() in [SYM, NUM, STRING]:
        scan(next())
    else:
        ERROR()

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

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

Last modified: Mon Sep 17 17:06:09 2012
Grammar Problems I

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

\[
p : e \ '−' \\
  e : t \\
  \mid e '−' t \\
  \mid e '∗' t \\
\]

\[
\begin{align*}
\text{\{ } & \text{ \$\$ = } \text{udas;} \text{ \}} \\
\text{\{ } & \text{ \$\$ = } \text{makeTree(DIV, } \text{UB, } \text{US); } \text{ \}} \\
\text{\{ } & \text{ \$\$ = } \text{makeTree(MULT, } \text{UB, } \text{US); } \text{ \}}
\end{align*}
\]
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 of 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?

\[ p : e \rightarrow l \]

\[ e : t \quad \{ \quad $$ = $1; \quad \} \]

\[ | t \ '/' e \quad \{ \quad $$ = \text{makeTree}(\text{DIV}, \$1, \$3); \quad \} \]

\[ | t \ '*' e \quad \{ \quad $$ = \text{makeTree}(\text{MULT}, \$1, \$3); \quad \} \]
Grammar Problems II

Well then: What goes wrong here?

\[
p : e \ ' - ' \\
e : t \ { $$ = $1; } \\
| t \ '/ ' e \ { $$ = \text{makeTree}(\text{DIV}, $1, $3); } \\
| t \ ' * ' e \ { $$ = \text{makeTree}(\text{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:

\[
\begin{align*}
  p &: e \rightarrow \cdot \\
  e &: s \ t \\
  s &: \epsilon \mid + \mid - \\
  t &: \text{ID} \mid (\ e \ ) \\
\end{align*}
\]

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 $p \Rightarrow e \ \downarrow \Rightarrow s \ t \ \downarrow \Rightarrow s \ '(' \ e \ ')' \ \downarrow \Rightarrow \ldots$, 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 \quad \{ ?1 \} \\
et &: \epsilon \quad \{ ?2 \} \\
| '/' e \quad \{ ?3 \} \\
| '*' e \quad \{ ?4 \} \\
t &: \text{I} \quad \{ $$ = $1; \}
\end{align*}
\]

What are FIRST and FOLLOW?
Grammar Problems III

What actions?

\[
\begin{align*}
p & : e \rightarrow \mid \\
e & : t \text{ et } \{ ?1 \} \\
et & : \varepsilon \{ ?2 \} \\
\mid '/ ' e & \{ ?3 \} \\
\mid '* ' e & \{ ?4 \} \\
t & : I \{ $$ = $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?

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?

FIRST(p) = FIRST(e) = FIRST(t) = { I }
FIRST(et) = { ε, ’/’, ’*’ }
FIRST(’/’ e) = { ’/’ } (when to use ?3)
FIRST(’*’ e) = { ’*’ } (when to use ?4)
FOLLOW(e) = { ’⊣’ }
FOLLOW(et) = FOLLOW(e) (when to use ?2)
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:

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
```