Lecture 6: Top-Down Parsing

Administrivia

- Need teams!!
- Project #1 will be posted (late) tomorrow (homework, too). Due 27 Feb.
- Test #1: March 10 (in class).
- Estimate Project #2 will be due 3 April, test #2 14 April, project #3 1 May.

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 semantic value that the lexer assigned to the previous token in 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.

Example: Lisp Expression Recognizer

Grammar

```
prog : sexp '⊣'
sexp : atom
     | '(' elist ')'
     | '\'' sexp
elist : ϵ
      | sexp elist
atom : SYMBOL
     | NUMERAL
     | STRING
```

```
def prog ():
   ______________________________

def sexp ():
   if ____________________:
      ______________________________
   elif ____________________:
      ______________________________
   elif ____________________:
      ______________________________

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

def elist ():
   if ____________________________:
      ______________________________
```

Expression Recognizer with Actions

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

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

def elist ():
   if next() not in (')', '⊣'):
       ______________________________
   else:
       return emptyList
```

Grammar Problems I

What goes wrong here?

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

Grammar Problems II

Well then: What goes wrong here?

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

Grammar Problems III

What actions?

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

What are FIRST and FOLLOW?

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 ___________________________:
        ______________________________
```