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 ():
    sexp(); scan(\t)

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

eelist : 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

```
```
def prog ():
de sexp ():
def elist ():
Expression Recognizer with Actions

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

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

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

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

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

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

\[
p : e '\rightarrow'
\]
\[
e : s t
| s : \epsilon | '+' | '-'
| t : ID | '(' e ')
\]

Since \( e \Rightarrow s t \Rightarrow ( e ) \Rightarrow \ldots \), we know that '(' \( \in \) FIRST(\( e \)). Since \( s \Rightarrow \epsilon \), we know that \( \epsilon \in \) FIRST(\( s \)).

• If \( X \) is a non-terminal symbol in some grammar, \( G \), then 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 \( p \Rightarrow e \Rightarrow s t \Rightarrow s '(' e ')' \Rightarrow \ldots \), we know that '(' \( \in \) 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?

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) = { $e$, '/', '*' } |
| 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():
    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
```