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 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(-)

def sexp ():
    if next() in [SYM, NUM, STRING]:
        atom()
    elif next() == '(':    # sexp elist
        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()
    else:
        return emptyList

def sexp ():
    if next() in [SYM, NUM, STRING]:
        cons($1, $2)
    else:
        emptyList
```

Example: Lisp Expression Recognizer with Actions

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

### Example

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

```
def sexp ():
    if next() in [SYM, NUM, STRING]:
        cons($1, $2)
    else:
        emptyList
```

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

```
def elist ():
    if next() in [SYM, NUM, STRING, '(', '"']:
        sexp(); elist()
    else:
        return emptyList
```
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); }  
```

```python
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?

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

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?

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

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:

\[
p : e '->'
\]
\[
e : s t
\]
\[
s : \epsilon | '+' | '-'
\]
\[
t : \text{ID} | '(' e ')'\]

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 \( p \Rightarrow e \Rightarrow s t \Rightarrow s '(' e ')' \Rightarrow \ldots \), 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?

\[ p : e \to^? \mathbin{-} \]
\[ e : t \phantom{e} \to \phantom{?} \]
\[ t : I \to \phantom{t} \{ \mathbin{\triangleleft} = $1; \} \]

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?

\[
\begin{align*}
\text{FIRST}(p) &= \text{FIRST}(e) = \text{FIRST}(t) = \{ I \} \\
\text{FIRST}(et) &= \{ \epsilon, \mathbin{-}, \mathbin{*} \} \\
\text{FIRST}(\mathbin{/} e) &= \{ \mathbin{/} \} \quad \text{(when to use ?3)} \\
\text{FIRST}(\mathbin{*} e) &= \{ \mathbin{*} \} \quad \text{(when to use ?4)} \\
\text{FOLLOW}(e) &= \{ \mathbin{-} \} \\
\text{FOLLOW}(et) &= \text{FOLLOW}(e) \quad \text{(when to use ?2)} \\
\text{FOLLOW}(t) &= \{ \mathbin{-}, \mathbin{/}, \mathbin{*} \}
\end{align*}
\]

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:

\[
\begin{align*}
def e(): \\
r &= t() \\
\text{while next() in [\mathbin{/}, \mathbin{*}]}: \\
\text{if next() == \mathbin{/}:} \\
\text{scan(\mathbin{/}); t1 = t()} \\
r &= \text{makeTree (DIV, r, t1)} \\
\text{else:} \\
\text{scan(\mathbin{*}); t1 = t()} \\
r &= \text{makeTree (MULT, r, t1)} \\
\text{return r}
\end{align*}
\]