# 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 $A$ 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

```
prog : sexp '⊣'
sexp : atom
     | '(' elist ')'
     | '\'' sexp
elist : ε
      | sexp elist
atom : SYM
     | NUM
     | STRING
```

```
def prog ():
    ________________

def sexp ():
    if ______________________________:
        ______
    elif ______________:
        ______________________________
    else:
        ___________________

def atom ():
    if ______________________________:
        _____________
    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 ________________________:
        ______
    elif _____________:
        ________________________
    else:
        ________________

def atom ():
    if ________________________:
        ___________
    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 ________:
        ____________________
    else:
        ____________

def atom ():
    if ____________________:
        ________
    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:
        ____________________

def atom ():
    if ______________________________:
        _____________
    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 ______________________________:
        _____________
    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:
        ________

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 ______________________________:
        ____________
```

# 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 next() in [SYM, NUM, STRING, '(', "'"]:
        sexp(); elist();
```

# 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() 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

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

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

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 '⊣'
e : s t
s : ϵ | '+' | '-'
t : ID | '(' e ')'
```

  Since `e` $\Rightarrow$ `s t` $\Rightarrow$ `( e )` $\Rightarrow$ `...`, 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 ')' ⊣` $\Rightarrow$ `...`, 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?

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

## 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?**

# 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:

```
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:

```
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:

```
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:

```
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:

```
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
```