

# Lecture 6: Top-Down Parsing

## Administrivia

- If you don't have a team, let me know today.
- If you decide to drop this course, or just to "sit in" for the rest of the semester, please be sure to remove your registration and to remove yourself from any team you've joined.
- Test #1 will be on Wed, 10 March during class.

# 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 xnext().
  - 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 :  $\epsilon$   
      | 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 :  $\epsilon$   
      | 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 :  $\epsilon$   
      | 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 :  $\epsilon$   
      | 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 :  $\epsilon$   
      | 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 :  $\epsilon$            { $$ = emptyList; }  
      | sexp elist      { $$ = 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 :  $\epsilon$            { $$ = emptyList; }  
      | sexp elist      { $$ = 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 or 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.



# Using 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.
- 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.
- 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  $\text{FOLLOW}(X)$ .

# 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?

$\text{FIRST}(p) = \text{FIRST}(e) = \text{FIRST}(t) = \{ I \}$   
 $\text{FIRST}(et) = \{ \epsilon, '/', '*' \}$   
 $\text{FIRST}('/', e) = \{ '/' \}$  (when to use ?3)  
 $\text{FIRST}('*', e) = \{ '*' \}$  (when to use ?4)  
 $\text{FOLLOW}(e) = \{ '⊣' \}$   
 $\text{FOLLOW}(et) = \text{FOLLOW}(e)$  (when to use ?2)  
 $\text{FOLLOW}(t) = \{ '⊣', '/', '*' \}$

# 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
```