Discussion3 Answer

1. **Derivations**.

   If a string belongs to a particular language, we can find a *parse tree* for it. A single nonterminal roots the parse tree (e.g 'prog' or 'expr'). This top-level nonterminal branches off into constituent nonterminals, eventually breaking down into terminals and ϵ-symbols at the leaves. *Top-down parsing* is a method of matching strings via an abstract preorder traversal of a parse tree. We've already seen an example of this: recursive descent! As a recap from lecture:

   - *Leftmost derivation*: a sequence of rules following a preorder traversal of the parse tree.
   - *Rightmost derivation*: see previous definition, but with a right-to-left preorder traversal.

   The *reverse rightmost derivation* is the rightmost derivation in reverse. The reverse rightmost derivation is an instance of bottom-up parsing—the string's lowest level details are recognized first, then mid-level details, finally leading up to the start symbol.

   Consider the following grammar:

```
var  : ID ;
expr : var
     | ‘(’ ‘λ’ var ‘.’ expr ‘)’
     | ‘(’ expr expr ‘)’ ;
```

   Use this grammar to construct leftmost and reverse rightmost derivations of the following strings.

   - `(λf.(λx.(f (f x))))`

   Answer:

   Leftmost derivations: `expr` $\to^3$ `(λvar.expr)` $\to^1$ `(λf.expr)` $\to^3$ `(λf.(λvar.expr))` $\to^1$ `(λf.(λx.expr))` $\to^4$ `(λf.(λx.(expr expr)))` $\to^2$ `(λf.(λx.(var expr)))` $\to^1$

$(\lambda$f.$(\lambda$x.(f expr))) $\rightarrow^4$ $(\lambda$f.$(\lambda$x.(f (expr expr)))) $\rightarrow^2$ $(\lambda$f.$(\lambda$x.(f (var expr)))) $\rightarrow^1$ $(\lambda$f.$(\lambda$x.(f (f expr)))) $\rightarrow^2$ $(\lambda$f.$(\lambda$x.(f (f var)))) $\rightarrow^1$ $(\lambda$f.$(\lambda$x.(f (f x))))

Reverse rightmost derivations: $(\lambda$f.$(\lambda$x.(f (f x)))) $\leftarrow^1$ $(\lambda$var.$(\lambda$x.(f (f x)))) $\leftarrow^1$ $(\lambda$var.$(\lambda$var.(f (f x)))) $\leftarrow^1$ $(\lambda$var.$(\lambda$var.(var (f x)))) $\leftarrow^2$ $(\lambda$var.$(\lambda$var.(expr (f x)))) $\leftarrow^1$ $(\lambda$var.$(\lambda$var.(expr (var x)))) $\leftarrow^2$ $(\lambda$var.$(\lambda$var.(expr (expr x)))) $\leftarrow^1$ $(\lambda$var.$(\lambda$var.(expr (expr var)))) $\leftarrow^2$ $(\lambda$var.$(\lambda$var.(expr (expr expr)))) $\leftarrow^4$ $(\lambda$var.$(\lambda$var.(expr expr))) $\leftarrow^4$ $(\lambda$var.$(\lambda$var.expr)) $\leftarrow^3$ $(\lambda$var.expr) $\leftarrow^3$ expr

2. **Recursive Descent Parsers**.

   Try writing a recursive descent parser for the grammar in Problem 1. Assume the existence of `scan(C)`, `next()`, and `ERROR()` as defined in lecture.

   Answer:

```
def var():
    if next() == IDENT:
        return make_var(scan(next()))
    else:
        ERROR()

def expr():
    if next() != '(':
        return var()
    scan('(')
    if next() == 'λ':
        scan('λ')
        param = var()
        scan('.')
        body = expr()
        scan(')')
        return make_lambda(param, body)
    else:
        func = expr()
        args = expr()
        scan(')')
        return make_funcall(func, args)
```

   Recursive descent parsing is simple, but sometimes onerous (lookaheads must be handled explicitly).

3. **Ambiguous Grammars**.

A grammar is *ambiguous* if it permits multiple distinct parse trees for some string. For example, without the order of operations, $12-8/4$ could parse as 1 or as 10. Make the following grammar unambiguous, and also give precedence to '/' over '-'.

```
e : INT
  | e '-' e
  | e '/' e ;
```

Answer:

```
em : INT
   | em '/' INT ;
e  : em
   | e '-' em ;
```

As another example, consider the following grammar:

```
e : 0 | 1 ;

stmt : e ';'
     | if e then stmt
     | if e then stmt else stmt ;
```

Give an example of a string in the language that can produce multiple parse trees.

Consider the string:

```
if 0 then if 1 then 0; else 1;
```

The two parse trees corresponding to this statement could be:

(a) `if 0 then (if 1 then 0; else 1;)`

(b) Or

    `if 0 then (if 1 then 0;) else 1;`

In other words, the else could be grouped with the first or second if statement. Most programming languages solve this ambiguity greedily; the else statement would be matched with the closest if statement, so the first grouping is what parsers for most languages end up choosing.

4. **Syntax-directed Translation**.

Parser-generators usually support syntax-directed translation, which is a convenient way to execute an action every time a grammar rule is matched. While defining actions, the variable `$$` refers to a location into which the semantic value of the current symbol can be stored. The variables `$1`, ..., `$n` refer to the semantic values of the symbols used to match the current rule. Here's an example:

```
p : e ';'         { printf("Result: %d\n", $1); }
e : INT           { $$ = $1; }
  | e '-' e       { $$ = $1 - $3; }
  | e '/' e       { $$ = $1 / $3; } ;
```

Write a syntax-directed translator for the first grammar you wrote for Problem 3. Also, write one for Problem 1.

Answer:

```
em : INT          { $$ = $1; }
   | em '/' em    { $$ = $1 / $3; } ;
e  : em           { $$ = $1; }
   | e '-' e ;    { $$ = $1 - $3; } ;
```

```
var  : ID                         { $$ = make_var($1); } ;
expr : var                        { $$ = $1; }
     | '(' 'λ' var '.' expr ')'   { $$ = make_lambda($3, $5); }
     | '(' expr expr ')'          { $$ = make_funcall($2, $3); } ;
```