1 Recursive Descent

Recursive descent parsing can be natural and flexible. Try writing a recursive descent parser for the following grammar. Assume the existence of `scan(C)`, `next()`, and `ERROR()` as defined in lecture. Once you’re done, list some pros/cons of using recursive descent. Bonus: what language is this?

```
var : ID ;
expr : var
     | '(' 'λ' var '.' expr ')' 
     | '(' expr expr ')' ;
```

```python
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)
```

Is it possible to write a regex to capture this language? Why or why not? What does the answer say about this grammar?

**Answers:** Recursive descent parsing is simple, but sometimes onerous (lookaheads must be handled explicitly). DFA-based parsers can be more efficient. There is no regex for this language because classical regular expressions can’t count, or balance parens. Conclusion: this language is not regular. Bonus: this is the λ-calculus, famous for its outsized descriptive power.
2 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 $\epsilon$-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.
- **Reverse rightmost derivation**: start at the bottom-left of the parse tree, then unapply rules left-to-right until the root is reached.

Use the grammar from Section 1 to construct leftmost and reverse rightmost derivations of the following strings.

- $(\lambda f.(f f))$
- $(\lambda f.(\lambda x.(f (f x))))$

**Leftmost derivations:**

- $\text{expr} \rightarrow \lambda \text{var.}(\lambda f.(f f)) \rightarrow 2 (\lambda f.((\lambda x.\text{expr})) \rightarrow 4 (\lambda f.(\lambda x.(\lambda x.(\text{expr})))) \rightarrow 2 (\lambda f.((\lambda x.\text{expr})))) \rightarrow 1 (\lambda f.((\lambda x.\text{expr}))))$
- $\text{expr} \rightarrow \lambda \text{var.}(\lambda x.(\text{expr} \lambda x.(\lambda x.(\text{expr})))) \rightarrow 2 (\lambda f.((\lambda x.\text{expr})))) \rightarrow 1 (\lambda f.((\lambda x.\text{expr}))))$

**Reverse rightmost derivations:**

- $(\lambda f.(f f)) \leftarrow 1 (\lambda \text{var.}(\lambda f.((\lambda x.\text{expr})))) \leftarrow 2 (\lambda \text{var.}(\lambda \text{var.}(\lambda f.((\lambda x.\text{expr})))))) \leftarrow 1 (\lambda \text{var.}(\lambda \text{var.}(\lambda f.((\lambda x.\text{expr}))))))$
- $(\lambda \text{var.}(\lambda \text{var.}(\lambda f.((\lambda x.\text{expr})))))) \leftarrow 2 (\lambda \text{var.}(\lambda \text{var.}(\lambda f.((\lambda x.\text{expr})))))) \leftarrow 1 (\lambda \text{var.}(\lambda \text{var.}(\lambda f.((\lambda x.\text{expr}))))))$

3 Ambiguous Grammars

A grammar is ambiguous if it permits multiple distinct parse trees for some string. For example, without the order of operations, $2 + 3 \ast 4$ could parse as 20 or as 14. Make the following grammar unambiguous by giving precedence to $\ast$ over $\plus$.

```plaintext
e : INT
| e 'plus' e
| e 'multiply' e
```

**Answer:** Thanks to Chris Hall for catching a problem with the original solution.

```plaintext
e : INT
| e 'multiply' INT
| e 'plus' e
```
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 $$\texttt{\$\$}$$ refers to a location into which the semantic value of the current symbol can be stored. The variables $\texttt{\$1}$, ..., $\texttt{\$n}$ refer to the semantic values of the symbols used to match the current rule. Here's an example:

```plaintext
p : e ';'
  { printf("Result: \%d\n", $1); }

e : INT
  { $$ = $1; }
  | e '+' e
    { $$ = $1 + $3; }
  | e '*' e
    { $$ = $1 * $3; }
;
```

Write a syntax-directed translator for your solution grammar from Section 3. Next, do the same for the grammar from Section 1.

**Answers:**

```plaintext
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); }
;
```

---

1There is no Further Reading for this week. Use this as a chance to get caught up in the 164 Course Reader!