Top-Down Parsing; Recursive Descent Parsing pt 1

CS 440: Programming Languages and Translators, Fall 2019

A. Parsing

- Parsing is the action of figuring out if an input is in the language of a particular grammar.
- One way to think about this is to ask "can we find a derivation for this input?"
- The problem differs depending on whether we look for leftmost or rightmost derivations.
  - (This is assuming we're reading input left-to-right.)
- **Top-down parsing**: Look for a leftmost derivation of some input by trying to build a parse tree from the top down. We begin with the start symbol and repeatedly try to apply rules to expand the current sentential form until we get a complete derivation of the input
- **Bottom-up parsing**: Look for a rightmost derivation by finding parse trees for shorter pieces of input and combine them to build parse trees for more input. We begin with the rules that lead directly to terminal symbols and go to rules that lead to those and so on.

B. Top-Down Parsing

- Begin with the starting nonterminal
- Repeatedly choose a nonterminal that is a leaf of the tree so far
- Replace it by a subtree that represents a rule

*Basic Example: Expressions with identifiers, +, and **

- Parsing `x + x * x`, using grammar $E \rightarrow E + E \mid E \rightarrow E * E \mid$ `x`
- Begin with starting symbol

  E

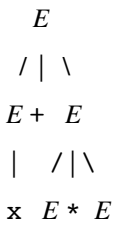
- Apply $E \rightarrow E + E$ rule

  E
  / | \
  E + E

- Apply $E \rightarrow$ `x` to left E

  E
  / | \
  E + E
  |
  `x`

- Apply $E \rightarrow E$ `*` E to remaining E



- Then apply $E \rightarrow$ `x` to the left remaining E and then the final remaining E (not shown).
- Use leftmost derivations:
  - Using a context-free grammar implies that expansions of different nonterminals are independent of each other.
  - But since we're reading the input left-to-right, the first terminal symbol is part of the yield of the leftmost path of the parse tree ($E \rightarrow E + E \rightarrow$ `x` $+ E$ above). Finding that path requires a leftmost derivation.

Left Recursion Problem

- But in the example, how do we know to expand the start symbol using $E \rightarrow E + E$ and not $E \rightarrow$ `x` ?
- If we look at the + after `x`, we know we need $E \rightarrow E + E$.
- But how do we expand the left E ?
  - We could use $E \rightarrow$ `x` and get $E \rightarrow E + E \rightarrow$ `x` $+ E$ as in the example.
  - But we could also use $E \rightarrow E + E$ and get $E \rightarrow E + E \rightarrow E + E + E$. And now the + following the `x` doesn't help us decide.
- The problem is that we have a nonterminal with a **left recursive** rule: The lhs nonterminal is also the first r.h.s. symbol: $B \rightarrow B$ etc.
- Left-recursive rules make top-down parsing nontrivial.
  - If you allow backtracking to take care of guessing "How many times should I recurse?", you can handle left recursion but with an enormous potential slowdown (possibly exponentially slower).
  - There are sophisticated techniques that do handle left recursion in leftmost derivations, but basic top-down parsing can't handle left recursion.

Removing Left Recursion (Without Breaking Other Properties?)

- To get rid of left recursion, we have to alter the grammar and introduce new nonterminals.
- The key in the $E \rightarrow E + E \mid E * E$ example is that if `x` is at the left end, then the next symbol has to be + or `*`. Applying precedences or associativities tells us that the plus in `x` + … has lower precedence than the * in `x` `*` ….
- If we think of evaluating an expression parse tree, we want to walk the tree in postorder (even if the operator is encountered before the end of the expression, we still have to look through the subexpressions to figure out on what to apply the operator).

- If we want to evaluate `x + x *` …, then the + operator has to be higher in the tree than the `*` operator. As we go through the tree in postorder, we need to complete everything associated with the `*` before we can work on the +.
- For the pluses in `x + x +` … to appear higher in the parse tree than the stars in `x * x *` …, we need to apply the rules for + before the rules for `*`.
- The traditional name for things we add together is a "term"; things we multiply together are "factors".
  - So an expression *E* will have the basic form *T* + *T* + …, where *T* is a term.
  - A term will have the basic form *F* `*` *F* `*` …, where *F* is a factor.
  - Factors don't get broken down directly into subexpressions involving + or `*` because those subexpressions need to be parts of terms or other factors. So `x` is a factor; if it's also a term, that's because we have a term that's a factor that's `x`.
  - We can also have parenthesized expressions as factors; the pluses and stars inside the parentheses have no effect on the pluses and stars outside the parentheses, so we don't need to introduce even deeper cousins of terms and factors that appear inside parenthesized expressions. We can simply restart the process of looking for expressions by using `(` *E* `)` as a factor. (We don't want it directly as a term because (*E*) can also be multiplied by other things, and things that get multiplied should be factors.)
- Note that `x` can be an expression or a factor or a term but we don't want all three of $E \rightarrow$ `x` and $T \rightarrow$ `x` and $F \rightarrow$ `x` because we also need $E \rightarrow^* T + T + \ldots$ and $T \rightarrow^* F * F * \ldots$ and we want to avoid ambiguous situations like $E \rightarrow$ `x` vs $E \rightarrow T \rightarrow$ `x`. In other words, we'll keep the precedences straight by avoiding shortcuts; *E* can yield *T* which can yield *F* which can yield `x`, but we can't derive x directly from *E* or *T*.
- The basic grammar for expressions involving id's, +, *, and parentheses is
  - $E \rightarrow T\ \ Ttail$
  - $Ttail \rightarrow +\ T\ TTail \mid \varepsilon$
  - $T \rightarrow F\ FTail$
  - $FTail \rightarrow *\ F\ FTail \mid \varepsilon$
  - $F \rightarrow$ `x` | `(` *E* `)`
- Using regular expression syntax along with context-free syntax, we can also write this as
  - $E \rightarrow T\ (+\ T)^*$
  - $T \rightarrow F\ (*\ F)^*$
  - $F \rightarrow$ `x` | `(` *E* `)`
- Note that the asymmetry between the *T* and *F* rules means that the derivations for `x + x * x` and `x * x + x` are asymmetric:
  - $E \rightarrow T + E \rightarrow$ `x` $+ E \rightarrow$ `x` $+ T \rightarrow$ `x` $+ F * T \rightarrow$ `x` $+$ `x` $* T \rightarrow$ `x` $+$ `x` $*$ `x`
  - $E \rightarrow T + E \rightarrow F * T + E \rightarrow$ `x` $* T + E \rightarrow$ `x` $*$ `x` $+ E \rightarrow$ `x` $*$ `x` $+$ `x`
  - But the asymmetry is a good thing: `x + x * x` is not a term `x + x` times a factor `x`.

C. Recursive Descent parsing

- **Recursive descent parsing** is a top-down technique where we write code that follows the recursive structure of the grammar rules.
- We use mutually recursive routines, one for each grammar nonterminal in the language. Each nonterminal's parser tries to parse an instance of the nonterminal (and remove it from the prefix of the input symbols).
- Recursive descent parsing works on **LL(1)** languages, where the parse is deterministic (no backtracking) and reads the input left-to-right (first L) producing a leftmost derivation (second L) using **top-down parsing** (from the start symbol toward the final terminal string yielded).
- The 1 in LL(1) means we get one symbol of **lookahead** as we parse, the head of the list of symbols that make up the current input to parse. This makes the parser **predictive**.
- Traditional recursive descent parsing has problems with grammars that have **left-recursive** rules (rules where *Nonterminal* → same *Nonterminal* plus other stuff).

*Expressions with + and **

- Going back to the grammar we started with
  - $E \rightarrow T \mid T + E$
  - $T \rightarrow F \mid F * T$
  - $F \rightarrow \texttt{x} \mid (\ E\ )$
- When a rule like $E \rightarrow T \mid T + E$ has alternatives that begin with the same nonterminal, we can't predict which of the alternatives to use. If we rewrite $E \rightarrow T \mid T + E$ as $E \rightarrow T\ \ (+\ E \mid \varepsilon)$ and similarly with terms and factors, we get
  - $E \rightarrow T\ \ (+\ E \mid \varepsilon)$
  - $T \rightarrow F\ (*\ T \mid \varepsilon)$
  - $F \rightarrow \texttt{x} \mid (\ E\ )$
- In effect, $E \rightarrow T\ \ (+\ E \mid \varepsilon)$ has us looking for a sequence of T separated by +, so we can look for terms directly using
  - $E \rightarrow T\ Ttail$
  - $Ttail \rightarrow +\ T\ Ttail \mid \varepsilon$
  - $T \rightarrow F\ Ftail$
  - $Ftail \rightarrow *\ F\ Ftail \mid \varepsilon$
  - $F \rightarrow \texttt{x} \mid (\ E\ )$
- You can think of *Ttail* as $(+\ T)^*$, so $E \rightarrow T\ (+\ T)^*$. Similarly, $T \rightarrow F\ (*\ F)^*$.

Example: Recognizing Expressions

- If we just want to recognize input (not build parse trees) the parsers are simpler:

- *E* → *T Ttail*

  parse_E (on current input)

  if *parse_T* on current input fails then we fail

  else call *parse_Ttail* on the input leftover from *parse_T* (and return what it returns)

  -- since *Ttail* → ε is a rule, *parse_Ttail* will always succeed

- *Ttail* → + *T Ttail* | ε

  parse_Ttail (on current input)

  if the next symbol is not a +

  then return *parse_empty* on the current input

  else

  remove the +

  if *parse_T* on current input fails then we fail

  else (call and return) *parse_Ttail* on the input leftover from *parse_T*

- *T* → *F Ftail*

  -- parsing a factor and factor tail are similar to expression and expression tail

  parse_T (on current input)

  if *parse_F* on current input fails then we fail

  else call (and return) *parse_Ftail* on the input leftover from *parse_F*

- *Ftail* → * *F Ftail* | ε

  parse_Ftail (on current input)

  If the next symbol is not a *

  then return *parse_empty* on the current input

  else

  remove the *

  if *parse_F* on current input fails then we fail

  else (call and return) *parse_Ftail* on the input leftover from *parse_F*

- *F* → x | (*E*)

  -- parsing a factor means parsing an identifier or parenthesized expression

  parse_F (on current input)

  if *parse_identifier* on current input succeeds

  then return what *parse_identifier* returned

  else


```
if next symbol is not a left parenthesis then fail
else
    remove the left paren
    call parse_E on the leftover of removing the '('
    if parse_E failed then we fail
    else if next symbol is not a right parenthesis' then fail
    else
        remove the right parenthesis from the input leftover from parse_E
        return the leftover
```

The recognizer code should be zipped with this pdf as `Lec_10_exp_recognizer.hs`

Activity Questions for Lecture 10

(Scanner questions, but for the recognizer)

1. What would we have to do to make tokens be strings instead of individual characters?
2. What would we have to do to have a more structured / keyword-oriented kind of symbol?
   I.e.,, `Id String`, `If_keyword`, `Then_keyword` and so on?
3. Why would it not work to have a symbol that combines the kind of symbol, text, and operation (something like `Operator("*",(*))`?

Changes to `Lec_10_recognizer_expr.hs`.

For these problems, continue using the same case-expression form as the existing recognizers

4. Currently *F* → x | (*E*). Change it to *F* → `x` | *Paren_E* where *Paren_E* → (*E*). Build a new parer *parse_paren_Expr* and move the code for (*E*) to this new parser.
5. Add a parser for `$` (*parse_end* input = `Just` [] if the input is [] and `Nothing` otherwise).
6. Write a recognizer for `x+` (i.e., *xplus* → `x` *xplus*? (i.e., `x` (`x+` | *epsilon*)).
7. Write a recognizer for `x+ (y?)` (i.e., `x+` (`y` | *epsilon*)).

Solutions to Selected Activity Questions

1. Just declare `type Token` = String instead of `type Token = Char`.
2. Declare `Token` to be a datatype: `data Token = Id String | If_keyword |` and so on.
3. The `next_symbol` routine requires symbols to support ==, but the embedded function makes `Operator("*", (*))` not support ==. (We'd need to write our own equality test for symbols.)

Changes to `Lec_10_recognizer_expr.hs`.

4. (Add *parse_paren_Expr* → (*parse_E*), make *paren_F* → `x` | *parse_paren_E*)

```
parse_F [] = Nothing
parse_F input =
    case parse_identifier input of
        Just left -> Just left
        Nothing -> parse_paren_E input

parse_paren_E input =
    case next_symbol '(' input of
        Nothing -> Nothing
        Just left2 ->paren
            case parse_E left2 of
                Nothing -> Nothing
                Just left3 ->
                    next_symbol ')' left3
```

5. (Parser for $ - end of input)

```
parse_end [] = Just []
parse_end _  = Nothing
```

6. (Parser for `x+` → `x` (`x+` | ε) `$`

```
xplus input =
    case next_symbol 'x' input of
        Nothing -> Nothing
        Just left1 ->
            case xplus left1 of
                Nothing -> Just left1
                Just left2 -> Just left2
```


7. (Parser for `x+ (y?)`)

```
-- parse x+ y? $
--
xplus_y_opt :: Recognizer
xplus_y_opt input =
    case xplus input of
        Nothing -> Nothing
        Just left1 ->                                   -- seen x+
            case next_symbol 'y' left1 of
                Nothing -> parse_end left1              -- x+ $
                Just left2 -> parse_end left2           -- x+ y $ ?
```