Recursive Descent Parsing

CS 440: Programming Languages and Translators, Spring 2019
Lecture 10, Wed 2/20

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
- Example: Parsing \( \text{id} + \text{id} \ast \text{id} \), using grammar \( E \rightarrow E + E \mid E \ast E \mid \text{id} \)
- Begin with starting symbol
  \[ E \]
- Apply \( E \rightarrow E + E \) rule
  \[ E \]
  \[ / | \ \]
  \[ E + E \]
- Apply \( E \rightarrow \text{id} \) to leftmost \( E \)
  \[ E \]
  \[ / | \ \]
  \[ E + E \]
  \[ | \]
  \[ \text{id} \]
- Apply \( E \rightarrow E \ast E \) to remaining \( E \)
  \[ E \]
  \[ / | \ \]
  \[ E + E \]
  \[ | / | \ \]
  \[ \text{id} E \ast E \]
- Apply \( E \rightarrow \text{id} \) to leftmost \( E \)
  \[ E \]
  \[ / | \ \]
  \[ E + E \]
  \[ | / | \ \]
  \[ \text{id} E \ast E \]
  \[ | \]
  \[ \text{id} \]
• Apply $E \to \mathtt{id}$ to remaining $E$

$$
E
/\|
E + E
| /\|
\mathtt{id} E * E
|   |
\mathtt{id} \mathtt{id}
$$

• We typically go left-to-right through the input and left-to-right through the children of a node.
• Always expand leftmost nonterminal

• Grammar derivation is leftmost
  
  • $E \to E + E \to \mathtt{id} + E \to \mathtt{id} + E * E \to \mathtt{id} + \mathtt{id} * E \to \mathtt{id} + \mathtt{id} * \mathtt{id}$

• In a recursive descent parser, each nonterminal corresponds to a recursive function that parses that kind of terminal string, and a grammar rule turns into combinations of recursive calls.
  
  • Prefer to do this without backtracking, to save time and space.
  • To avoid backtracking, we need restrictions on the grammar.
  • We need to know what routine to recursively call

• $E$ input = $E$ input followed by + on remaining input followed by $E$ on remaining input
  
  • With $\mathtt{id} + \mathtt{id} * \mathtt{id}$, first $E$ recognizes $\mathtt{id}$, second one recognizes $\mathtt{id} * \mathtt{id}$
  
  • $E = E$ followed by + followed by $E$ (implicitly, the input is being trimmed as we go)
  
  • $E = E$ followed by * followed by $E$ (the other recursive $E$ rule)
  
  • $E = \mathtt{id}$ (grab it from input)

  • Similar to what went on in homework 2: empty input is a base case, matching a character is a base case (matching a character) or does recursive calls.

  ```
  \text{match \_ \[\] = (False, \[\])}
  \text{match (P x) (y : ys)}
    = if x == y then (True, ys) else (False, y : ys)
  \text{match (POr pat1 pat2) xs}
    = case match pat1 xs of
      (True, leftover) -> (True, leftover)
        -- stop if pat1 succeeded
      (False, _) -> match pat2 xs
        -- else try pat2
  ```
• This is not recursive descent parsing because the calls don't involve a routine for a particular nonterminal.

\[
E \text{ str} = E \text{ str } \text{`And`} \text{ char } `+' \text{ remaining string } `\text{And`} \text{ E yet remaining string}
\]

• E returns \((True, \text{remaining input})\) to chain recursive calls, we want to pass that to \(E\) also

\[
E = E \text{ `followed by`} \text{ char } `+' \text{ `followed by`} E
\]

Define \(E\) by combining functions

• In a language with global variables and assignment, the remaining input is a global variable and each parser function just modifies it.
  • Then you can call \(E\) and call \(\text{char } `+'\) and call \(E\)
  • If these return trees, we have something like
    \[
    E = \text{let } t1 = E() \\
    t2 = \text{char } `+' \\
    t3 = E() \\
    \text{in build_tree_for_E t1 t2 t3} \\
    \text{end}
    \]
  • The result is that each nonterminal's function looks a lot like the rule we're applying.

• But this grammar had \(E \rightarrow E + E\) and \(E \rightarrow E * E\) rules; how do we distinguish between them?
  • This turns into the question of which function should we apply?
  • Rewrite grammar to \(E \rightarrow E (+ E | * E)\) -- find the left hand \(E\) in any case, then decide which of the + and * rules to apply. (Factor out the left \(E\).)
  • We're assuming we can look at the current input character. (So we can ask + or * ? and also Is char == c?)
    • One character of "lookahead"
  • We use another function to handle the "or" of \(+ E | * E\)
    • But this one gets to look at initial character, recognize + or * and choose that alternative without trying the other one.
• For this grammar, we do still have the problem of left recursion \((E \rightarrow E \ldots)\).
  • If \(E\) calls \(E\) without using any input, then each \(E\) call will just do another \(E\) call (infinite recursion).
  • There are techniques for avoiding this, but here we can rewrite the grammar to avoid it.
• This is where \(E \rightarrow T \mid T + E\) comes from.
  • Factor that to \(E \rightarrow T (\varepsilon | + E)\). Once we parse the T, we can use the one character of lookahead to see if we're at the end of the string (and match \(\varepsilon\)) or we have a +.

\[
E = \text{let } t1 = T \\
\text{in if next char == `+'} \\
\text{then parse `+';} \\
\text{let t2 = E} \\
\text{in build_tree_using t1 t2}
\]
else if at end of input, return t1
else error