Grammars: Recursive-Descent Parsing

CS236 - Discrete Structures
Instructor: Brett Decker
Fall 2020

Recursive Algorithms

We have discussed how using a table and a pushdown automaton allows us to systematically parse LL(k) grammars. Now we will learn how to apply this theory to develop code that will perform the parsing. A key element is that the pushdown automaton makes use of a stack. When you hear "stack" you should think "recursion." Stacks allow us to create recursive code. Let's consider an example. The Fibonacci sequence defines a sequence of numbers, each of which we will denote as F_i. The Fibonacci sequence is defined as $F_0 = 0, F_1 = 1$ and $F_n = F_{n-1} + F_{n-2}$. Note the recursive nature of this definition. The next Fibonacci number is based on the previous two, until we reach the base cases of F_0 and F_1. Consider the following recursive implementation (note this is a naive solution, but shown because we're trying to illustrate recursion):

```
int fib(int n) {
    if (n > 2) {
        return fib(n-1) + fib(n-2);
    }
    else if (n == 1) {
        return 1;
    }
    else { // n == 0
        return 0;
    }
}
```

Here is a visualization of how the run-time stack grows when you execute the above code (each time you call a function a new frame is pushed onto the stack; when then function returns, the frame is popped from the stack):

Example: We begin with a call to `fib(3)`. Note the stack grows up. Execution moves from left to right.

```
                fib(1)          fib(0)
        fib(2)  fib(2)  fib(2)  fib(2)  fib(2)          fib(1)
fib(3)  fib(3)  fib(3)  fib(3)  fib(3)  fib(3)  fib(3)  fib(3)  fib(3)
```

All that just to compute `fib(3)` (now you see how recursion can cause re-computation).

Recursive-Descent Parsing

We can use recursion in a similar manner to create a parser for our LL(k) grammars. We will only deal with LL(1) grammars, so our discussion below will be specific to them (though the concepts can be applied to all LL(k) grammars with slight modifications). There is a three step process for creating recursive-descent parsing code, hard-coded to a specific LL(1) grammar (generic recursive-descent parsers that parse any LL(1) grammar can be created using inheritance).

Steps:

1. Create a function for each nonterminal, A
2. Within the function, check input (current token) against the terminals generated by A for a match (this defines which production to use next)
3. Within the function, call corresponding function for nonterminals generated by A for the production matched (this will push the next function on the stack before returning from the current function – this is the recursive part)

Recursive-Descent Parsing Example:

Recall the prefix notation grammar we've seen:

$N = \{E, O, D\}$
$T = \{0, 1, 2, 3, +, *\}$
$S = E$
$P = \{$
$E \rightarrow D \mid OEE$
$O \rightarrow + \mid *$
$D \rightarrow 0 \mid 1 \mid 2 \mid 3$
$\}$

Let's use the above steps to create pseudo-code for this grammar. Assume that `tokens` is a collection of tokens where each token is of a type corresponding to our vocabulary: $E, O, D, 0, 1, 2, 3, +, *$. The method `tokens.Current()` will give us the first token in the collection (this is the current input token). The method `tokens.Next()` will consume the first token in the collection (thus advancing to the next token). Assume a function `Match` that returns two when its two arguments are equal. We will also assume a function `FIRST(...)` that takes a nonterminal as its argument and returns a collection of all terminals produced by it. Here is the pseudo-code:

```
func ParseE(tokens) begin
   if FIRST(Token.D).contains(tokens.Current()) begin
      // Production: E -> D
      ParseD(tokens)
   else if FIRST(Token.O).contains(tokens.Current()) begin
      // Production: E -> OEE
```

```
      ParseO(tokens)
      ParseE(tokens)
      ParseE(tokens)
   else
      // the first terminal does not match any productions
      throw exception
   end
end

func ParseD(tokens) begin
   if Match(tokens.Current(), Token.Zero) or
      Match(tokens.Current(), Token.One) or ... begin
      tokens.Next()
   else
      // the first terminal does not match any producible terminals
      throw exception
   end
end

func ParseO(tokens) begin
   if Match(tokens.Current(), Token.Plus) or
      Match(tokens.Current(), Token.Multiply) begin
      tokens.Advance()
   else
      // the first terminal does not match any producible terminals
      throw exception
   end
end
```

Note the similarity of the functions `ParseO` and `ParseD`. Both nonterminals O and D only produce terminals. Thus their corresponding functions only need to check that the first token matches one of the types of one of the terminals they produce. Otherwise, when the string we're trying to parse is not produced by our grammar – that is when we throw an exception. It is helpful for the exception to contain the information of what is wrong. For example, if we're trying to parse O and have a '1' as the current token, the exception could be: "ERROR when parsing O; Expected + or *; Received 1". When a nonterminal produces other nonterminals more functions will be called and pushed onto the stack. For each possible production there will be a separate branch (there are two branches for `ParseE`). These patterns of how to parse terminals and nonterminals allow for generic parsing via inheritance, though we will not cover that.

Parse Trace Example:

Let's do a trace for parsing the terminal string '+*321' just keeping track of which function is called with the input:

```
ParseE     +*321
ParseO     +*321 -> *321
ParseE      *321
ParseO      *321 -> 321
ParseE       321
ParseD       321 -> 21
ParseE        21
ParseD        21 -> 1
ParseE         1
ParseD         1
Done!
```

Let's do another trace for an invalid terminal string '+1':

```
ParseE     +1
ParseO     +1 -> 1
ParseE      1
ParseE      (no more input)
throw an exception: "ERROR need more tokens"
```

Conclusion

In the second project of this course you will implement a recursive-descent LL(1) parser for a Datalog program to parse the tokens generated from your lexical analysis in the first project. Thanks to Dr. Michael Goodrich for the prefix notation grammar and examples.