Grammars: Table-Driven Parsing

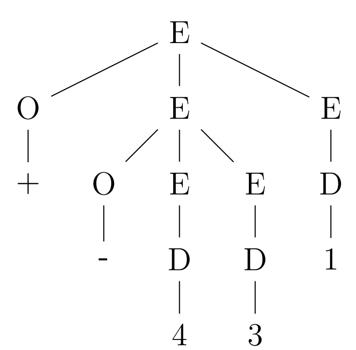
CS236 - Discrete Structures
Instructor: Brett Decker
Fall 2021

Systematic Parsing

We have not yet discussed a systematic way to create parse-trees. Having an algorithm that defines a systematic approach is vital in order for us to design code to create parse-trees. A naive approach would be to use a full search algorithm: create all possible parse-trees and see if any matches the terminal string we're trying to parse. This would be computationally expensive and inefficient. Let's develop an algorithm to automate parsing. To do this, let's consider the following grammar (it represents arithmetic expressions in prefix, or Polish, notation, see Section 11.3.4*):

$G = (V, T, S, P)$ where
  $N = \{E, O, D\}$
  $T = \{0, 1, 2, \ldots, 9, +, -, *, /\}$
  $S = E$
  $P = \{$
    $E \to D \mid OEE$
    $O \to + \mid - \mid * \mid /$
    $D \to 0 \mid 1 \mid 2 \mid \ldots \mid 9$
  $\}$

Let's create the parse tree for the terminal string '+ - 4 3 1':



We evaluate from bottom to top, so we evaluate '4 - 3' first, and then the result, 1 added with '1', thus '1 + 1 = 2.' Therefore, our prefix notation terminal string '+ - 4 3 1' evaluates to 2.

Is there a systematic way to create this parse-tree? How did we know to choose the production rule $E \to OEE$ instead of $E \to D$? We looked at the first terminal in our terminal string, which is '+.' Then we look to see which production rule would lead to the same terminal. That's the rule we choose. Thus we see the start of our algorithm. We check if we can match the current symbol of our terminal string with a producible terminal. If so, we choose that production. If the current nonterminal cannot produce the current symbol, then we must choose a production with a starting nonterminal. In order to choose the correct production, we have to look ahead at what each possible nonterminal produces. When we find a match with our current symbol, we've found the production. Let's make this concrete with our example above.

Our first symbol in our terminal string is '+.' Starting with our start symbol, E, we check if E has a production that starts with '+.' It does not. Now we must look at the productions from E that start with a nonterminal. Our choices are O and D. Now we check to see if either O or D has a production that starts with '+.' O does; D does not. Thus we choose the production OEE. We continue this process until we have parsed the entire terminal string.

LL Grammars

What happens if there are more than one possible production rule that produces the same terminal? We would have to guess at which production to take and backtrack if we later find out the production was the wrong one. Backtracking is expensive, so we'd like to avoid this. There is a subset of grammars, called *LL grammars* that do not require backtracking. An LL grammar is a special context-free grammar that can be parsed with an *LL parser*. An LL parser parses the input from left to right and produces a leftmost derivation (thus LL) without backtracking (note: a leftmost derivation is a strategy where the leftmost nonterminal is chosen as the next nonterminal to rewrite). An LL(k) parser is able to parse an LL grammar with just k look-ahead characters. That means it only needs to know the next k terminals, starting from the left in order to pick the correct next production rule. What is k for the prefix notation grammar, G?

$G = (V, T, S, P)$ where
$N = \{E, O, D\}$
$T = \{0, 1, 2, \ldots, 9, +, -, *, /\}$
$S = E$
$P = \{$
$E \rightarrow D \mid OEE$
$O \rightarrow + \mid - \mid * \mid /$
$D \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9$
$\}$

The prefix grammar above is an LL(1) grammar, thus $k = 1$, because we only have to look at the first character of the terminal string to choose the correct production rule. Let's look at another example. Starting at the start symbol, E, what production should be chosen if the terminal string is '6'? E does not produce '6' directly, so we must look at the first nonterminal of each production. D produces '6' (and thankfully O does not – if it did, we wouldn't have an LL(1) grammar). So we must choose $E \to D$. Note that for all digits, we choose this production, and that for all operators we choose $E \to OEE$.

FIRST Sets

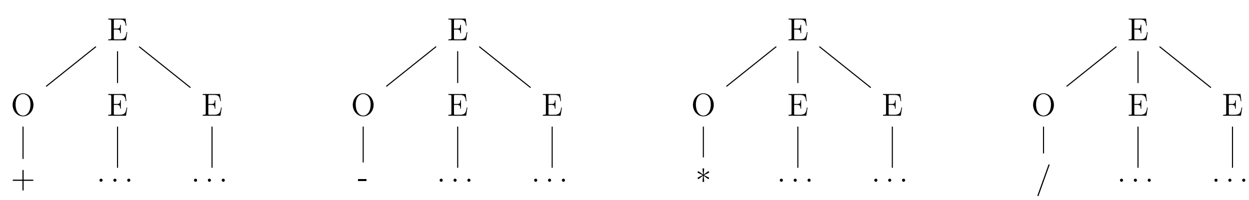
By definition, with LL(1) grammars we only have to look at the first character of the terminal string to choose the correct production, for all productions in all possible derivations. We use a concept known as $FIRST$ sets to help better understand LL(1) grammars.

Here is the intuition about $FIRST$ sets from Dr. Goodrich: "*Let A denote some nonterminal in the parse tree. The children of A in the parse tree are derived by applying the production $A \to RHS$ and putting the terminals and nonterminals from the right-hand side, RHS, into the tree as children of A. Any terminal that can be the leftmost descendent in the subtree under A belongs in the $FIRST$ set of A. FIRST sets capture what nonterminals can appear as a leftmost descendent of A when you apply $A \to RHS$.*"

We say that the $FIRST$ set of a production is the set of all first terminals it can *eventually* produce (since we are dealing with leftmost derivations, we mean the first, leftmost terminal produced by the current production). Consider again the LL(1) grammar we saw above:

$G = (V, T, S, P)$ where
$N = \{E, O, D\}$
$T = \{0, 1, 2, \ldots, 9, +, -, *, /\}$
$S = E$
$P = \{$
$E \to D \mid OEE$
$O \to + \mid - \mid * \mid /$
$D \to 0 \mid 1 \mid 2 \mid \ldots \mid 9$
$\}$

The $FIRST$ set of the production $E \to D$, denoted as $FIRST(E \to D)$ is $\{0, 1, 2, \ldots, 9\}$, since those are the first, leftmost terminals that can be produced from the nonterminal D (this is the nonterminal on the *right-hand* side of the production). $FIRST(E \to O) = \{+, -, *, /\ \}$. Let's visualize this. Here are the four possible parse trees with $E \to O$:



Note that the leftmost terminals for these parse trees are $+, -, *, /$. That's why $FIRST(E \to O) = \{+, -, *, /\ \}$. We say that a grammar is LL(1) if the $FIRST$ sets for all productions of all nonterminals are disjoint. For example, all the $FIRST$ sets with nonterminal E—$FIRST(E \to D)$ and $FIRST(E \to O)$—must not have any elements in common. If they have elements in common, we could not know which production to choose based off the current terminal we are trying to parse. We'd have to look ahead to the next terminal, thus the grammar would not be LL(1).

Note how trivial the $FIRST$ sets are that include the nonterminals O and D. Here are just a few: $FIRST(O \rightarrow +) = \{+\}$, $FIRST(O \rightarrow -) = \{-\}$, $FIRST(D \rightarrow 0) = \{0\}$, $FIRST(D \rightarrow 1) = \{1\}$, $FIRST(D \rightarrow 2) = \{2\}$, etc. $FIRST$ sets are interesting when we have productions with a nonterminal as the leftmost symbol on the *right-hand* side of a production.

Parse Tables

Because LL(1) grammars can be systematically parsed by an algorithm, they can make use of *parse tables* for efficient parsing. A parse table contains the cell for which production to use for every possible terminal character at every point in the parsing. We always use a stack in conjunction with our table. When using the table, we look up a cell by doing the following: (1) the nonterminal or terminal on the top of the stack determines the row in the table and (2) the current character of the input string we are parsing determines the column in the table. The cell contains all the new nonterminals and terminals to produce, which will get pushed onto the stack (after the current top symbol on the stack is popped). Recall our prefix notation grammar production rules shown below (we'll reduce the operators and digits for clarity):

$$
\begin{aligned}
P = \{ & \\
& E \rightarrow D \mid OEE \\
& O \rightarrow + \mid * \\
& D \rightarrow 0 \mid 1 \mid 2 \mid 3 \\
& \}
\end{aligned}
$$

We use the $FIRST$ sets to build our parse table:

	+	*****	**0**	**1**	**2**	**3**	**#**
E	OEE	OEE	D	D	D	D	
O	+	*					
D			0	1	2	3	
+	AdPop						
*****		AdPop					
0			AdPop				
1				AdPop			
2					AdPop		
3						AdPop	
#							Accept

To build the parse table we create a column for every terminal and the extra symbol # (we'll explain the pound–no, it's not a hashtag–shortly). We create a row for each symbol in the vocabulary (all nonterminals and terminals) and the extra symbol #. Keep the order of the terminal consistent for both rows and columns (just as done above).

How do we use the table? Note the 'AdPop' and 'Accept' entries in the table. These are present because we use the table with a pushdown automaton. Recall that a pushdown automaton (PDA) makes use of a stack. Before we start parsing a terminal string, we will push a pound symbol (#) to the stack. Each cell in the parse table tells the machine what to do next. The empty cells tell the PDA to reject the terminal string. The cells with 'AdPop' tell the PDA to pop the top element off the stack and advance the input. The PDA only accepts the terminal string if the cell with 'Accept' is reached. There is one more type of cell command in our table. Let's give a concrete example to explain how it works. Consider the cell for row E, column $+$, which is OEE. This cell tells the PDA that if we are parsing the nonterminal E and see a '+' as the current character (in the terminal string we are parsing), then we replace the E with OEE on the stack, with O at the very top. Notice that this corresponds to the production in our grammar: $E \to OEE$.

Parse Table Example:

Let's do an example. We will use the parse table above to parse the terminal string '*+123' (the input to our PDA). We call this a *trace* (the up arrow, ↑, will keep track of where we are in the input string). The output column is the production label used at each step – the pop action does not have a label, thus it does not affect the output.

Action	**Stack**	**Input**
Initialize, push '#' and start symbol	$E\#$	$\uparrow * + 123\#$
Action($E, *$)=Replace[E, OEE]	$OEE\#$	$\uparrow * + 123\#$
Action($O, *$)=Replace[$O, *$]	*$EE\#$	$\uparrow * + 123\#$
Action($*, *$)=AdPop	$EE\#$	*↑ +123#
Action($E, +$)=Replace[E, OEE]	$OEEE\#$	*↑ +123#
Action($O, +$)=Replace[$O, +$]	+$EEE\#$	*↑ +123#
Action($+, +$)=AdPop	$EEE\#$	*+↑ 123#
Action($E, 1$)=Replace[E, D]	$DEE\#$	*+↑ 123#
Action($D, 1$)=Replace[$D, 1$]	1$EE\#$	*+↑ 123#
Action($1, 1$)=AdPop	$EE\#$	*+1↑ 23#
Action($E, 2$)=Replace[E, D]	$DE\#$	*+1↑ 23#
Action($D, 2$)=Replace[$D, 2$]	2$E\#$	*+1↑ 23#
Action($2, 2$)=AdPop	$E\#$	*+12↑ 3#
Action($E, 3$)=Replace[E, D]	$D\#$	*+12↑ 3#
Action($D, 3$)=Replace[$D, 3$]	3#	*+12↑ 3#
Action($3, 3$)=AdPop	#	*+123↑ #
Action($\#, \#$)=Accept		*+123#↑
Accept!		

Let's run another trace for an invalid terminal string '1+2':
This terminal string will be rejected because the row pound, column '+' is empty – there is no possible production to use for this input.

Action	**Stack**	**Input**
Initialize, push '#' and start symbol	$E\#$	$\uparrow 1+2\#$
Action$(E,1)$=Replace$[E,D]$	$D\#$	$\uparrow 1+2\#$
Action$(D,1)$=Replace$[D,1]$	$1\#$	$\uparrow 1+2\#$
Action$(1,1)$=AdPop	$\#$	$1\uparrow +2\#$
Action$(\#,+)$=Reject	$\#$	$1\uparrow +2\#$
Reject!		

Conclusion

Using a table for parsing is inefficient in code. There is a better algorithm for parsing LL(1) grammars: recursive-descent parsing. We will study this next. Parsing grammars is important in the fields of programming languages, program analysis, and formal verification. Thanks to Dr. Michael Goodrich for the prefix notation grammar and examples.

**Discrete Mathematics and Its Applications*, by Kenneth H. Rosen.