LL Parsing (part 3) and Bottom-Up Parsing

CS 440: Programming Languages and Translators, Spring 2019
Lecture 4, Mon Mar 10

Notation

- $a, b, ... \in T$ ($a$ and $b$ stand for members of $T$; $a, b, ...$ are members of $T$) Was $\Sigma$ for regular expressions.
- $x, y, ... \in T^*$
- $A, B, ... \in V$
- $W, X, ... \in T \mid V$
- $\alpha, \beta, ... \in (T \mid V)^*$  Sentential form

I. (More LL) Revising a Grammar to Make it More LL

- Recall the example from the last class
  - $S \rightarrow E$
  - $E \rightarrow ET$
  - $T \rightarrow +ET$
  - $T \rightarrow *ET$
  - $T \rightarrow \epsilon$
  - $E \rightarrow id$

- One problem with the grammar was that it had no way to distinguish between the rules $E \rightarrow ET$ and $E \rightarrow id$ because $id \in \text{First}(E)$ and $\text{First}(ET)$. Having a left recursive rule prevents a grammar from being LL.
- A second problem was that there's no preference of * over *, so an input like $id + id * id$ has a parse tree that is structured like $(id + id) * id$, if we had parentheses.
- The solution to both problems involves rewriting the grammar, first to remove left recursion and second to ensure that a parse tree for … * … + … puts the parse tree for … * … below the +.
- Removing left recursion: We saw this once already. The grammar basically makes $E$ generate an $E$ followed by a string of + or * $T$; in the base case, the initial $E$ becomes $id$.
  - We can replace $E \rightarrow ET$, $E \rightarrow id$ with $E \rightarrow E'T$, $E' \rightarrow id$ to get us to the base case earlier.
  - This optimizes to just $E \rightarrow id T$.

Giving * Precedence over +

- To give * precedence over +, we want the parse tree node for generating + to be higher than the one for *.
- So the path from the start symbol to the rule generating + has to be shorter than the one for *.
  - So the nominal structure of an expression needs to be … + … + … + … etc. where the ellipsed parts don't directly contain + (i.e., don't contain a + in this same context but possibly contain + in other contexts).
- So an $E$ is basically a sequence $E_1 (+ E_1)*$ where $E_1$ doesn't directly generate a + directly.
• Similarly, an $E_1$ is a sequence $E_2 (*) E_2^*$ where $E_2$ doesn't directly generate + or *.
• In the grammar as is, we have $E_2 \rightarrow id$, but if we want a way to indirectly generate + or *, we need more syntax for this structure.
  • Let $E_3$ be a new nonterminal for this kind of structure.
  • Since we might have $E \rightarrow * E_1 \rightarrow * E_2 \rightarrow E_3$, we know $First(E_3) \subseteq First(E_2) \subseteq First(E_1) \subseteq First(E)$, so we want to be careful to know we have an $E_3$ when we see one.
  • The easiest way to announce "We have an $E_3$" is to give it its own keyword.
  • It's also useful to know where the $E_3$ ends because + and * can appear inside an $E_3$ but also after an $E_3$.
  • And this is basically why we have $E_3 \rightarrow (E)$ – the parentheses serve as keywords to delimit the beginning and end of the $E_3$.
  • Note that how many levels of $E_1, E_2, E_3, \ldots$ we need is going to depend on how many distinct levels of precedence there are.
  • A quick look at a C precedence table makes it looks like there are 15 levels: Prefix * is at level 2 and postfix ++ is at level 1 (which takes precedence over level 2), so ++ is stronger than prefix * so *p++ means *(*(p++) and in (*p)+++, the parentheses are required.
  • But in reality there are more than 30 levels because within a level, the operators might be ordered. (Scary: Imagine a grammar with nonterminals $E_1$ through $E_{30}$.)
  • **Short version**: You can have many many levels of grammar, which gets awkward.
• With grammars for statements, the analogous problem to "Where does an $E_3$ end?" is that without some sort of ending delimiter, it's hard to know where **if** Header **Statement** **else** **Statement** ends.
  • One solution is basically like parentheses around **Statements** – this is where { and } come from in C and (begin and end in similar languages).
  • In C, it's useful for the statement parentheses to be different from the expression parentheses because a statement can begin with a parenthesized expression.
    • (Easy examples are $(x) = 1$; and `printf("hi\n")`; They look weird because the parentheses are redundant so we almost always leave them out.)
    • A similar problem comes up with semicolons: In while **Header** $S_1$ ; $S_2$, does $S_2$ go with the while? Or is while **Header** $S_1$ the whole statement? Answer: In C, while has stronger precedence than ;, so while **Header** $S_1$ is the whole while statement.
  • Going back to **if-else**, strictly speaking, you don't need { and } around the true branch as long as you know the false branch is going to follow directly and begin with **else**.
    • But since the false branch is optional, we get the ending problem above but also a new problem, the "dangling else":
    • When parsing **if Header**$_1$ **if Header**$_2$ **Statement**$_1$ **else** **Statement**$_2$, we've dropped an **else** **Statement**$_3$, but it's not obvious whether the **else** **Statement**$_2$ goes with **if Header**$_1$ or **if Header**$_2$.
    • Here, to remove ambiguity, C and similar languages such as Pascal use the "closest if" solution and match **else** **Statement**$_2$ with **if Header**$_2$. 
Another basic solution to the problem "Where does if Header Statement else Statement end?" is to add a keyword that explicitly marks the end, in the same way that a right parenthesis matches a left parenthesis.

The original "end if" keyword is "fi" ("if" spelled backwards); this comes from Algol 68.

Notice having an end if eliminates the dangling else problem:
- In if $B_1$ then $S_1$; if $B_2$ then $S_2$ fi fi, the missing else goes with if $B_2$.
- In if $B_1$ then $S_1$; if $B_2$ then $S_3$ fi else $S_2$ fi, the missing else goes with if $B_1$.

Algol 68 also has an end while loop keyword, but it's not "elihw", it's "od": The syntax for a while loop is while Boolean do Statement_List od where the statements are separated by semicolons. This makes … do $S_1$; $S_2$ ; $S_3$ ; … od syntactically analogous to an n-tuple $(e_1, e_2, e_3, … )$.

Prefix operator languages

A prefix operator language (where $x + y$ is written $+ x y$) has the interesting property that you don't really need parentheses in it. Since the lowest-level parts of an expression are all single terminal symbols (like $x$ or 17), if we read + and * only as binary operators, there's no ambiguity between $+ * x y z$ (the + of $* x y$ and $z$) and $* + x y z$ (the * of $+ x y$ and $z$). More generally, if each operator takes a fixed number of operands, then writing expressions in prefix doesn't require parentheses.

Of course, you can include parentheses if you want and this is where LISP-like syntax $( + ( * x y ) z )$ and $( * ( + x y ) z )$ come from. And if an operator can have a non-fixed number of operands then either you need parentheses or rewrite using operators that take a fixed number of operands. In Haskell, we see this in

$\{ E_1, E_2, E_3 \}$ versus $E_1 : E_2 : E_3 : [ ]$. (In prefix, this last expression is : $E_1 : E_2 : E_3 : [ ]$.)

Summary

Top-down parsing builds a parse tree starting with the root start symbol and expanding downwards.

To do this without backtracking, in LL(1) parsing we allow looking at the next input symbol.

(In practice, the parser might look two characters ahead for special cases.)

For looking at the next input character to be helpful, we have to know what information it establishes.

- Character $c \in First(A)$ set lets $c$ tell us that we're beginning the parse of some terminal string generated by $A$.
  - We also include $\varepsilon \in First(A)$ to tell us that $A \rightarrow \varepsilon$ is a rule. (So $First(A) \subseteq T \cup \{ \varepsilon \}$ in general.)
  - If you're expecting to parse an $A$ but don't see anything in the $First(A)$ set, we've detected a parse error.

- Character $c \in Follow(A)$ set lets $c$ tell us that we've ended the parse of some terminal string generated by $A$.
  - For example, with $B \rightarrow A C$ …, any next character in $Follow(A)$ tells us we've ended the $A$ and want to parse a $C$. If the next character is $\in Follow(A)$ and $\notin First(C)$ but is in $Follow(C)$, then we've ended the $A$ and used $C \rightarrow \varepsilon$ and want to parse whatever comes after the $C$.

- For LL table-driven parsing, we create a prediction table indexed by nonterminal and next input character to tell us what grammar rule to use to expand the nonterminal.
  - When building the table, if two rules want to be in the same slot (for an example take nonterminal and terminal $x$ with rules, $A \rightarrow x B$ and $A \rightarrow x C$), this tells us the grammar is not LL(1).
  - The fix for this particular grammar is easy: Factor out the leading $x$. $A \rightarrow x D, D \rightarrow B | C$. 

v. Mon 3/11
• A sneaker fix is to keep the two rules $A \to x B$ and $A \to x C$ and look at the character after $x$ to see if it begins a $B$ or a $C$. (This would be a hack to use LL(2) parsing in some special case.)
• In a recursive descent parser, each nonterminal $A$ has its own parsing function, which tries to parse the initial part of the input string to form an $A$.
  • (In this case, the problem with $A \to x B$ and $A \to x C$ comes up when writing the parser for $A$: If we see $x$, which routine should we call: the one for $B$ or the one for $C$?)

II. Bottom-Up Parsing
• In bottom-up parsing, we build the smallest subtrees at the outskirts of the parse tree before combining them to build their parents, which we in turn combine to form those parents, and so on, up to the root of the parse tree.
  • In bottom-up parsing, if we’ve processed some input and don’t know if we’re in the right-hand side of a rule $A \to \alpha$ or $B \to \beta$ (which can happen if $\alpha$ and $\beta$ can generate terminal strings with the initial substring), we don’t have to decide whether we’re parsing $A \to \alpha$ or $B \to \beta$ until we reach the end of the input generated by $\alpha$ or $\beta$. The decision as to which rule to use is lazy and can use non-local information (what made up $\alpha$ or $\beta$?).
• Contrast with top-down parsing, where we either have to predict which rule to apply to the current nonterminal or we have to be willing to backtrack if our initial decision is wrong.
  • Since backtracking can make parsing much slower, in LL($k$) parsing, we avoided it by using grammars that let us predict which $A \to \alpha$ rule to use based on $A$ and the next $k$ characters of input. This keeps the decision-making process eager and local at a cost of not being able to parse as many grammars.
• If we look at the whole parse tree generated by a parse, a bottom-up parser (that reads input left-to-right) builds small parse trees from the bottom left of the eventual overall parse tree, combing them to form larger bottom-left parse trees.
• A top-down LL parse visits the parse tree nodes in preorder. A bottom-up parse visits the nodes in postorder and generates a reverse rightmost derivation.
Example of Bottom-Up Parsing and Rightmost Derivation

- Here's an example of how bottom-up parsing generates a reverse rightmost derivation.

- **Grammar:**
  - \( S \rightarrow T U T \)
  - \( T \rightarrow a \ b \)
  - \( U \rightarrow c \ c \)

- **Derivation:** \( S \rightarrow T U T \rightarrow T U a \ b \rightarrow T c c a b \rightarrow a b c c a b \)

  - We start at left end of the input \( a b c c a b \). As we move through the input, we'll build a collection of terminal and nonterminal symbols. The terminal symbols are directly from the input; the nonterminals represent the roots of various parse trees that we've managed to parse so far.

  - The character to the right of the _ is the next input character to look at. The things to the left of the _ are items that we haven't completely parsed yet.

  - When we see a terminal symbol, we add it to our collection of terminal / non-terminal symbols; this is "shifting".

  - If our collection ends with the rhs of a rule, we remove the rhs symbols and replace them with the lhs nonterminal. This is "reducing".

    - \_a b c c a b
    - a b c c a b
    - a b c c a b
      - "Reduce" using \( T \rightarrow a \ b \) (Recognize rhs of rule \( T \rightarrow a \ b \), replace it with lhs non-term.)
    - T c c a b
      - Shift c
    - T c c a b
      - Shift c
    - T c c a b
      - Reduce using \( U \rightarrow c \ c \)
    - T U . a b
      - Shift a
    - T U . a b
      - Shift b
    - T U . a b
      - Reduce using \( T \rightarrow a \ b \)
    - T U T .
      - Reduce using \( S \rightarrow T U T \)
    - S
      - Done!

  - Going upward, we can read the derivation as \( S \rightarrow T U T \rightarrow T U a b \rightarrow T c c a b \rightarrow a b c c a b \), which is the rightmost derivation.

  - If we rewrite this as \( a b c c a b \leftarrow a b c c T \leftarrow a b U T \leftarrow T U T \leftarrow S \), which is what our actual process looked at, we see this is indeed the reverse of the rightmost derivation.

**Shift-Reduce Parsing**

- Shift-reduce parsers are the kind of bottom-up parser normally seen when people talk about "bottom-up parsing".

- A **shift-reduce** parser uses the technique above with a stack to hold the symbols we've seen so far (a collection of terminals &/or parsed nonterminals) but haven't yet parsed completely. If a parsed nonterminal is represented as a parse tree, it's the full tree, not a partially-built tree.
• Shift-reduce parser reads the input left-to-right and maintains a stack of all the previously-seen partly-used parse trees and symbols.
  • E.g., with x + y, at the point where we’ve seen x and +, we have two items at the top of the stack
    • If we’ve parsed Thing → x, then we have Thing and +. If not, we have x and + on the stack
    • Items on the stack have been parsed as much as possible; if Thing → x, we don’t have x and +.
  • The shift-reduce name comes from the two basic operations of the parser.
    • **Shift** - the lookahead symbol (i.e., the next input symbol) is pushed onto the stack of unprocessed items.
    • **Reduce** - Remove the n most recent items on the stack, use it as the rhs of a rule (with n symbols in its rhs), and add the lhs nonterminal to the stack.
      • If that causes the top elements of the stack to look like the rhs of a rule, then reduce that too; repeat until no more reductions can be done.
      • This ensures that the stack only contains items that have been as completely parsed as possible.
      • This lets us look at only the roots of trees on the stack (if they're trees) to see that we have the rhs of some rule. (If we didn’t reduce as much as possible, we’d have to look at something on the stack and realize we could reduce it.)
  • Shift-reduce parsers differ according to how and when they decide to do a reduce step and how they handle possible conflicts between shifting / reduction decisions.