Warm-Up Problem

- Write a CFG that recognizes $L = \{a^i b^j c^i : i, j \in \mathbb{N}\}$.

CS 241 Lecture 10

Context Free Grammars, Ambiguity, Top-Down Parsing

With thanks to Brad Lushman, Troy Vasiga, Kevin Lanctot, and Carmen Bruni

Solutions

- For L_1: Arithmetic expressions from Σ without parentheses
  - $S \rightarrow a \mid b \mid c \mid SRS$
    $R \rightarrow + \mid - \mid * \mid /$
  - $S \Rightarrow SRS \Rightarrow aRS \Rightarrow a - S \Rightarrow a - b$
- Almost for L_2: Arithmetic expressions from Σ with balanced parentheses (incomplete, insufficient!)
  - $S \rightarrow a \mid b \mid c \mid (SRS)$
    $R \rightarrow + \mid - \mid * \mid /$
  - $S \Rightarrow (SRS) \Rightarrow (SRb) \Rightarrow (S - b) \Rightarrow (a - b)$

Solutions

- For L_2: Arithmetic expressions from Σ with balanced parentheses
  - $S \rightarrow a \mid b \mid c \mid SRS \mid (S)$
    $R \rightarrow + \mid - \mid * \mid /$
  - $S \Rightarrow (S) \Rightarrow (SRS) \Rightarrow ((S)RS) \Rightarrow ((a)RS) \Rightarrow ((a)\text{-}S) \Rightarrow ((a)\text{-}b)$

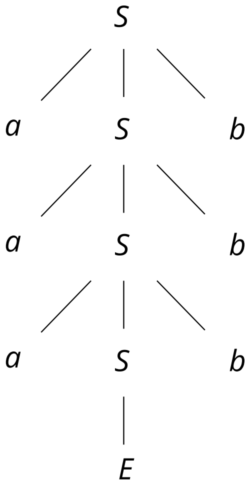
Note

Notice in these two derivations that we had a choice at each step which element of N to replace

- $S \Rightarrow SRS \Rightarrow aRS \Rightarrow a - S \Rightarrow a - b$
- $S \Rightarrow (S) \Rightarrow (SRS) \Rightarrow (SRb) \Rightarrow (S - b) \Rightarrow (a - b)$
- In the first derivation, we chose to do a left derivation, that is, one that always expands from the left first.
- In the second derivation, we chose to do a right derivation, that is, one that always expands from the right first.

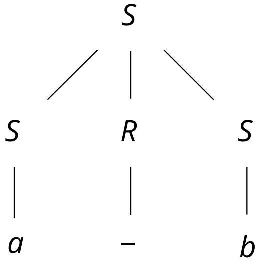
Parse Trees

$aaabbb$ in $S \rightarrow \varepsilon \mid aSb$



$a - b$ in

$$\begin{aligned} S &\rightarrow a \mid b \mid c \mid SRS \\ R &\rightarrow + \mid - \mid * \mid / \end{aligned}$$



Note: To every left- (right-) most derivation there exists a unique parse tree (and vice versa).

Question

- Is it possible for multiple leftmost derivations (or multiple rightmost derivations) to describe the same string?
- Yes! Consider the following two leftmost derivations (on the next slide) for $a - b * c$.

Left-Most Derivations

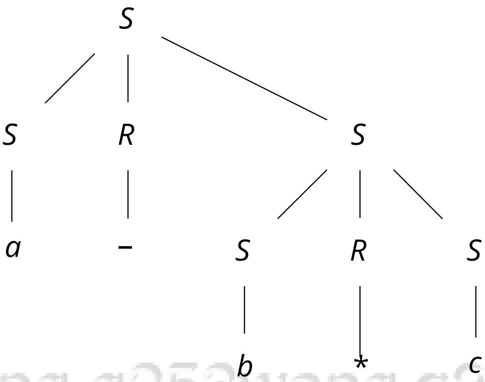
$$
\begin{aligned}
S &\rightarrow a \mid b \mid c \mid SRS \\
R &\rightarrow + \mid - \mid * \mid /
\end{aligned}
$$

$$
\begin{aligned}
S &\Rightarrow SRS \Rightarrow aRS \Rightarrow a-S \Rightarrow a-SRS \\
&\Rightarrow a-bRS \Rightarrow a-b*S \Rightarrow a-b*c \\
S &\Rightarrow SRS \Rightarrow SRSRS \Rightarrow aRSRS \Rightarrow a-SRS \\
&\Rightarrow a-bRS \Rightarrow a-b*S \Rightarrow a-b*c
\end{aligned}
$$

these correspond to different parse trees! Let's draw them.

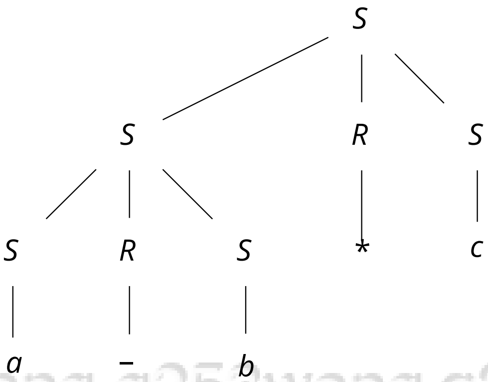
First Parse Tree

$$S \Rightarrow SRS \Rightarrow aRS \Rightarrow a-S \Rightarrow a-SRS$$
$$\Rightarrow a-bRS \Rightarrow a-b*S \Rightarrow a-b*c$$



Second Parse Tree

$$S \Rightarrow SRS \Rightarrow SRSRS \Rightarrow aRSRS \Rightarrow a-SRS$$
$$\Rightarrow a-bRS \Rightarrow a-b*S \Rightarrow a-b*c$$



Ambiguous Grammars

Definition

A grammar for which some word has more than one distinct leftmost derivation/rightmost derivation/parse tree is called **ambiguous**.

This:

$$\begin{aligned} S &\rightarrow a \mid b \mid c \mid SRS \\ R &\rightarrow + \mid - \mid * \mid / \end{aligned}$$

was an example of an ambiguous grammar

Sure But…

- Why do we care about this? Isn't our goal to determine whether or not $w \in L(G)$?
- As compiler writers, we care about where the derivation came from: parse trees give meaning to the string with respect to the grammar.
- Let's go back to the parse trees: they don't mean the same thing!
- How can we fix this?

Solutions

- Use some sort of precedence heuristics to guide the derivation process (very dependent on grammar, very ad hoc).
- Make the grammar unambiguous! This is what we did with our first (incomplete) L_2

$S \to a \mid b \mid c \mid (SRS)$
$R \to + \mid - \mid * \mid /$

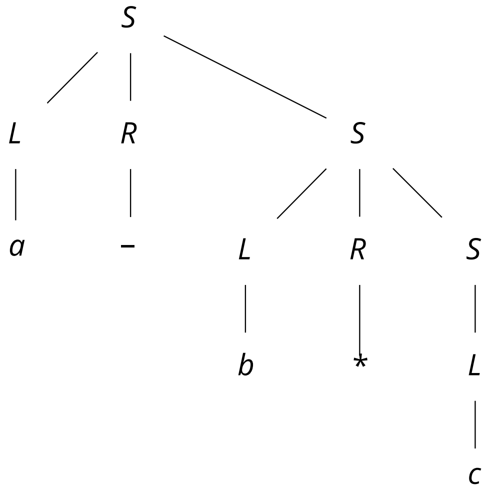
Ambiguity

- There’s a better way to eliminate ambiguity.
- In a parse tree, you evaluate the expression in a depth-first, post-order traversal (Left, Right, Root).
- We also have the other issue that we want to interpret [a − b + c] as [(a − b) + c] and NOT as [a − (b + c)] (that is, we want left associativity).
- We can make a grammar left/right associative by crafting the recursion in the grammar!

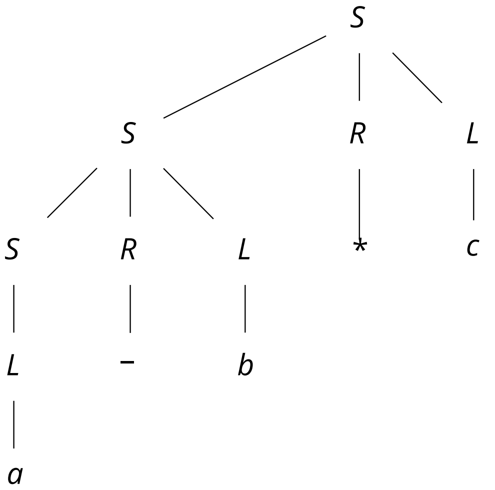
Forcing Right Associative

$$
\begin{aligned}
S &\rightarrow LR\mathbf{S} \mid L \\
L &\rightarrow a \mid b \mid c \\
R &\rightarrow + \mid - \mid * \mid /
\end{aligned}
$$

This forces a right-associative grammar (e.g., parse tree for $a - b * c$ recurses to the right)



Forcing Left Associative



$$S \rightarrow \mathbf{S} R L \mid L$$
$$L \rightarrow a \mid b \mid c$$
$$R \rightarrow + \mid - \mid * \mid /$$

This forces a left-associative grammar (e.g., parse tree for $a - b * c$ recurses to the left)

With the above...

- We can use this to create a grammar that follows BEDMAS/PEMDAS rules more closely by making $*$, / appear further down the tree (hence evaluated first!):

$S \to SPT | T$
$T \to TRF | F$
$F \to a \mid b \mid c \mid (S)$
$P \to + | -$
$R \to * \mid /$

- Exercise: Find a derivation of $a - b * c$ and then give the parse tree, and work out evaluating it.

Some Questions

- If *L* is a context-free language, is there always an unambiguous grammar such that L(*G*) = *L*?
- No! First proven in 1961 by Rohit Parikh. Quoth Wikipedia:

An example of an inherently ambiguous language is the union of $\{a^n b^m c^m d^n : n, m > 0\}$ with $\{a^n b^n c^m d^m : n, m > 0\}$. This set is context-free, since the union of two context-free languages is always context-free. But Hopcroft and Ulman in 1979 give a proof that there is no way to unambiguously parse strings in the (non-context-free) common subset $\{a^n b^n c^n d^n : n > 0\}$

Decidability of Ambiguous Grammars

- Can we write a computer program to recognize whether a grammar is ambiguous?
- No! Most textbooks (see for example Hopcroft, John; Motwani, Rajeev; Ullman, Jeffrey (2001). Introduction to automata theory, languages, and computation Theorem 9.20, pp. 405-406) reduce to the undecidability of Post's Correspondence Problem.
- Original proofs are due to Cantor (1962), Floyd (1962), and Chomsky and Schtzenberger (1963).

Yikes!

- What about an easier problem: given two CFGs G_1 and G_2, determine whether $L(G_1) = L(G_2)$.
- Maybe even easier: what about determining whether $L(G_1) \cap L(G_2) = \emptyset$?
- Still undecidable!

What We Can Answer

- Regular languages corresponded to abstract machines, namely DFAs.
- Is there a machine correspondence for CFLs?
- Yes! The correspondence is with a **pushdown automaton** (PDA). These machines are DFAs/NFAs with stack.
- The breakdown is that CFLs correspond to NPDAs, and NPDAs cannot be converted to DPDAs like NFAs can to DFAs.
- While NPDAs are computable, doing so is awful (inefficient in both time and space).

What We Can Answer

- While NPDAs are computable, doing so is awful (inefficient in both time and space.
- Instead, use a category of algorithm called *parsers*. Any practical parser can only handle a subset of CFLs (and of CFGs) but can generally do so with reasonable efficiency.

Formally

Given: a CFG $G = (N, \Sigma, P, S)$ and a terminal string $w \in \Sigma^*$
Find: The derivation, that is, the steps such that $S \Rightarrow \ldots \Rightarrow w$ or prove that $w \notin L(G)$.

Pragmatically, we care about the parse tree, not the derivation steps per se

How Do We Find This Derivation?

Two broad ideas (which cover all practical parsing algorithms):

- Start with S and then try to get to *w*, i.e., from the top down: top-down parsing.
- Start with *w* and work our way backwards to S: bottom-up parsing.
- *Each option is terrible in its own special way* ☺

Top-Down Parsing

- Start with S (the start symbol), and look for a derivation that gets us closer to *w*. Then, repeat with remaining non-terminals until we're done.
- The main trick is "look for a derivation": we can look at the first few symbols of *w*, but that won't necessarily match the RHS of a derivation, since there might be more non-terminals.
- Thus, the core problem is to *predict* which derivation is right.

Top-Down Parsing Reality

- We present the *LL(1) algorithm*
- In practice, almost no real compilers use LL(1)

Top-Down Parsing Reality

- Most compilers use hand-written parsers (often called “recursive descent parsing” because each non-terminal is a recursive function), and their underlying behavior is basically LL(1)
- LL(1) has limitations which we discuss; hand-written parsers bypass these limitations on an ad hoc basis

Top-Down Parsing Reality

- We don’t discuss recursive-descent parsing because it’s not an algorithm; it’s just a bunch of ad hoc programs
- Regardless, when we talk about LL(1), think of how you would implement the main algorithm with functions per non-terminal, instead of a formal algorithm, because this is what you’ll find in practice

Top-Down Parsing Reality

- As said, most real compilers use hand-written parsers, and those hand-written parsers are top-down, but a not-insignificant minority do use formal algorithms
- Those are almost always bottom-up parsers, which is the next module

First Try (note: broken even with faerie magic!)

Algorithm Top-Down Parsing Algorithm (with faerie magic)

push S
for each ‘a’ in input **do**
  while top of stack is A $\in$ N **do**
    pop A
    ask a magic faerie to tell you which production A $\rightarrow\gamma$ to use
    push the symbols in γ (right to left)
  end while
  // TOS is a terminal
  if TOS is not ‘a’ **then**
    Reject
  else
    pop ‘a’
  end if
end for
Accept