Week 4: Book

Wednesday, February 9, 2022 4:20 PM

ε (Epsilon) : lexical component, anything that you can write in a language.
ε stands for "empty string of symbols"

Parsing : is the problem of taking a string of terminals to figure out how to derive it from the start symbol of the grammar or report syntax errors.

↳ We begin with a simple source program:

9 − 5 + 2 } each character is a TERMINAL

In reality, a program has multichar lexemes that the lexical converts to tokens

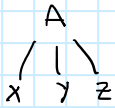
the token's 1st component (token name) are terminals processed by the PARSER.

(2.2.1 Derivation)
↑
2.2.3 Parse Trees (page 45)

Parse Tree: Pictorically shows how the START SYMBOL of a grammar derives a string in the language.

→ ex. If NONTERMINAL "A" has a production:
A → XYZ

Then a parse tree has an interior NODE A, with 3 children labeled X, Y and Z from left to right:



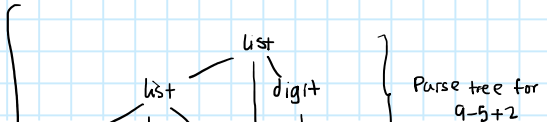
Given a context-free grammar, a parse tree according to the grammar has the properties:

1. The root is labeled by the START SYMBOL.
2. Each leaf is labeled by a TERMINAL or by ε.
3. Each interior node is labeled by a NONTERMINAL.
4. If A is the NONTERMINAL labeling of an interior node and $X_1, X_2, \ldots, X_n$ are the labels of the children of that node from L→R, then there must be a production $A \to X_1 X_2 \ldots X_n$ (X_i is a symbol either T or NONT)

→ if A → ε is a production, then a node labeled A has a single child ε.

Example: (2.4)
- The derivation of 9 − 5 + 2 is the tree below
- Each node in the three is labeled by a grammar symbol.
- A interior node and its children correspond to a production.
  - HEAD of the production
  - ↳ BODY of the production
- If we got list → list + digit
  - ✓ the root is labeled list, which is the START SYMBOL of the grammar in Example 2.1
  - ✓ the children of the root are labeled, L→R: list, +, digit.

From L to R the leaves of a parse tree form the YIELD of the



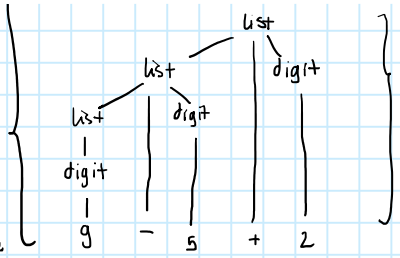
Parse tree for 9−5+2

From L to R the leaves of a parse tree form the YIELD of the tree.

↓

the string generated or derived from the NONT at the root of the tree

↳ "9-5+2"



Parse tree for 9-5+2 according to Ex 2.1 grammar (productions)

this way. Any tree imparts a natural left-to-right order to its leaves, based on the idea that if X and Y are two children with the same parent, and X is to the left of Y, then all descendants of X are to the left of descendants of Y.

→ Another definition for "the language generated by the grammar" is "the set of strings that can be generated by a parse tree"

Parsing a string: the process of finding a parse tree for a given string of terminals.

2.2.4 Ambiguity

↳ the structure of a string according to a grammar

↳ a grammar can have more than 1 parse tree that generates (YIELD) a string of terminals. } This is an ambiguous grammar

↳ to show that a grammar is ambiguous:

- find a terminal string (YIELD) that is the YIELD of more than 1 parse tree = if a string can be yielded by 2+ trees.

↳ Since a string with more than 1 parse tree can have ≠ meanings, it is recommended not to design ambiguous grammars.

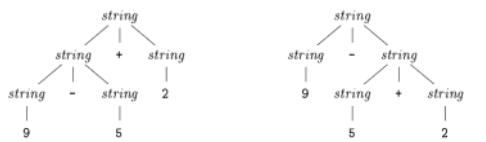
Example 2.5: Suppose we used a single nonterminal *string* and did not distinguish between digits and lists, as in Example 2.1. We could have written the grammar

$$string \rightarrow string + string \mid string - string \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

with this grammar (productions), 9+5-2 has more than 1 tree.

The 2 trees correspond to each:

6 ← (9-5)+2 9-(5+2) → 2



2.2.5 Associativity of Operators

9+5+2 = (9+5)+2 9-5-2 = (9-5)-2

When an operand like 5 has operators to its left and right, CONVENTIONS are needed for deciding which operator applies to that operand (5).

↳ We say that + operator ASSOCIATES to the left

↳ an operand(#) w/ plus on both sides of it belongs to the operator (+) to its left.

(9+5)+2

arithmetic operators +, -, ×, ÷ : left associative

assignment operator

a = (b = c)

arithmetic operators
+, −, ×, ÷ : left associative

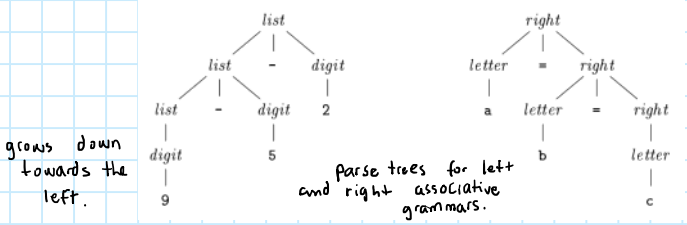
assignment operator
= : right associative, that is, a=b=c is treated as a=(b=c)

a = (b = c)

→ Strings like a=b=c with a right associative operator are generated by the grammar

right → letter = right | letter
letter → a | b | ... | z

→ The contrast of the parse tree for a left associative operator (−) and a right associative operator (=) is below.



2.2.6 Precedence of Operators

Consider the expression 9+5*2

↳ Two possible interpretations: (9+5)*2 or 9+(5*2)
↳ THE ASSOCIATIVITY RULES FOR + AND * (left association) APPLY TO OCCURRENCES OF THE SAME OPERATOR.

⇩

Rules defining the relative precedence of operators are needed when more than one operator.

→ We say that * has HIGHER PRECEDENCE than +
- IF * takes its operands (#) before +
- Thus, 5 is taken by * in both 9+5*2 and 9*5+2
  ↳ which means 9+(5*2) and (9*5)+2

ex: (2.6) • a grammar for arithmetic expressions can be constructed from a table showing the associativity and precedence of operators

- We start with the four common arithmetic operators and a precedence table, showing the operators in order of increasing precedence.
- Operators on the same line have the same associativity and precedence

left associative: + − → expr
left associative: * / → term

We create two nonterminals *expr* and *term* for the two levels of precedence, and an extra nonterminal *factor* for generating basic units in expressions. The basic units in expressions are presently digits and parenthesized expressions.

$$factor \rightarrow \textbf{digit} \mid (\ expr\ )$$

Now consider the binary operators, * and /, that have the highest precedence. Since these operators associate to the left, the productions are similar to those for lists that associate to the left.

$$\begin{array}{rcl} term & \rightarrow & term * factor \\ & \mid & term / factor \\ & \mid & factor \end{array}$$

highest precedence = grouped first

Similarly, *expr* generates lists of terms separated by the additive operators.

$$\begin{array}{rcl} expr & \rightarrow & expr + term \\ & \mid & expr - term \\ & \mid & term \end{array}$$

lowest precedence = grouped last

The resulting grammar is therefore

$$\begin{array}{rcl} expr & \rightarrow & expr + term \mid expr - term \mid term \\ term & \rightarrow & term * factor \mid term / factor \mid factor \\ factor & \rightarrow & \textbf{digit} \mid (\ expr\ ) \end{array}$$

higher level

higher level

p60

expr: list of terms separated by */

any parenthesized expr is factor

=

with parentheses we can develop expressions that have deep nesting (deep trees) arbitrarily.

() protect

A factor is something that cannot be torn apart : digit | (expr)

A term (that is not also a factor) is an expression that can be torn apart by operators of the highest precedence: * and / but not by the lower-precedence operators. An expression (that is not a term or factor) can be torn apart by any operator.

We can generalize this idea to any number n of precedence levels. We need $n+1$ nonterminals. The first, like *factor* in Example 2.6, can never be torn apart. Typically, the production bodies for this nonterminal are only single operands and parenthesized expressions. Then, for each precedence level, there is one nonterminal representing expressions that can be torn apart only by operators at that level or higher. Typically, the productions for this nonterminal have bodies representing uses of the operators at that level, plus one body that is just the nonterminal for the next higher level.

n=2
3 NONT: expr, term, factor.