02/10/2010

Recitation #4

Administrative

- Any questions on lecture content?
- **Assignment 1b** was extended, now due on 15 February at 11:59 p.m.
- **Assignment 2** is due February 17th @ 11:59
- **Exam** is coming up in about 2 – 3 weeks, next week's recitation we will go through a practice exam provided by Dr. Hughes

Assignment 1a Grading

- Grading criteria and test/reference files on Dr. Hughes' website and WebCourses
- Compiler stuff → Point to Wiki page
- Point deductions when crash / not compile
- Weighting of 1a vs. 1b

Assignment 1b

- This builds on assignment 1a, so you should make sure that you get that done

Assignment 2

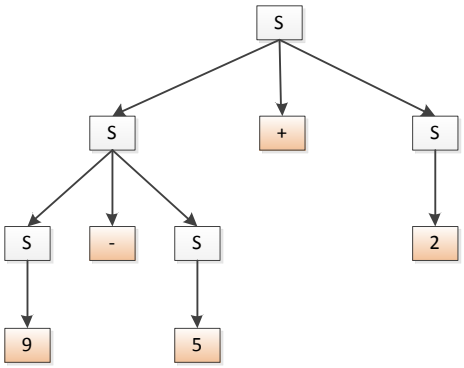
- Some explanation already done in Recitation #2
- Show little demo on using Flex, building an executable, and testing your program

Worksheet on grammars

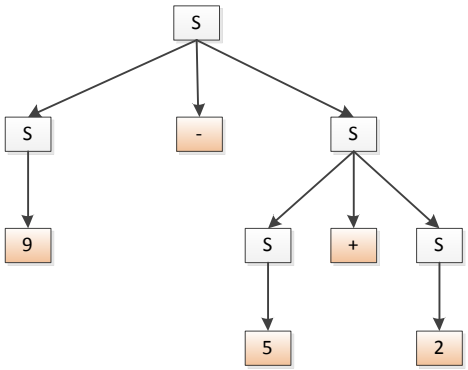
- Formal definition of **grammar** : $G = (V, \Sigma, P, S)$
  - V is a finite set, each element is called a non-terminal character (variable)
  - Σ is a set of terminals, disjoint from V, the alphabet
  - P are production rules with a single non-terminal on the left side and a single terminal or single terminal followed by a single nonterminal on the right side (right-linear grammar)
  - S is the start variable used to represent the whole program. $S \in V$.
  - See **worksheet question 1**.
- Grammar **ambiguity**
  - If there is more than one parse tree for a given string
  - This is bad, because there are multiple meanings for language elements.
  - The problem of deciding if a given grammar is ambiguous is undecidable!
  - Example: Parse string 9 – 5 + 2 with grammar

$$S \rightarrow S + S \mid S - S \mid D$$
$$D \rightarrow 0 \mid 1 \mid \ldots \mid 9$$

- Results in two distinct parse trees



This is (9 – 5) + 2 = 6

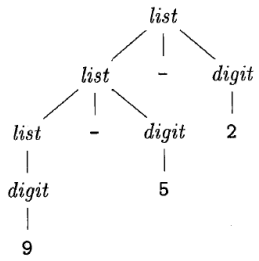


This is 9 – (5 + 2) = 2, which is unintended.

- **Operator associativity**
  - Associativity rules only apply to occurrences of the same operator
  - An operator is left-associative if an operand with the operator on both sides of it belongs to the operator to its left, e.g. +, -, *, /
    - Leads to implied left parentheses: a+b+c+d = (((a+b)+c)+d)
    - Left-associative grammar e.g.

      $\text{digit} \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9$

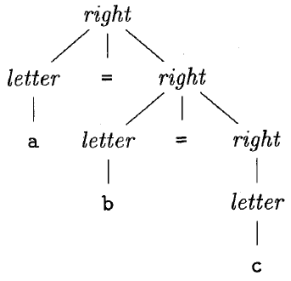
      $\text{list} \rightarrow \text{list - digit} \mid \text{digit}$
    - Parse trees "grow" to the left



  - Right-associative is the exact opposite, e.g. ^, =
    - Leads to implied right parentheses: a=b=c=d = (a=(b=(c=d)))
    - Right-associative grammar e.g.

      $\text{letter} \rightarrow a \mid b \mid c \mid \ldots \mid z$

      $\text{right} \rightarrow \text{letter} = \text{right} \mid \text{letter}$
    - Parse trees "grow" to the right



- **Operator precedence**
  - Associativity rules only apply to occurrences of the same operator
  - For expressions with mixed operators, e.g. 9 +5 * 2 we have to avoid ambiguity by considering operator precedence
  - To write a grammar whose parse trees expresses precedence correctly, use a different nonterminal for each precedence level. Start by writing a rule for the operator(s) with the lowest precedence, then write a rule for the operator(s) with the next higher precedence.

        - If binary operator "op" left-associative write $A_1 \rightarrow A_1 \text{ op } A_2 \mid A_2$
        - If binary operator "op" right-associative write $A_1 \rightarrow A_2 \text{ op } A_1 \mid A_2$
        - If unary operator "op" left-associative write $A_1 \rightarrow A_1 \text{ op } \mid A_2$
        - If unary operator "op" right-associative write $A_1 \rightarrow \text{op } A_1 \mid A_2$
    - This way operations with operators of higher precedence cannot be "torn apart" by operators of lower precedence
    - When you reach bottom level (e.g. A_5), take into account parentheses:

      $A_5 \rightarrow \text{digit} \mid (A_1)$
    - **Example**:
        - +, -, *, / (all are left-associative binary operators)

          $A_1 \rightarrow A_1 + A_2 \mid A_1 - A_2 \mid A_2$

          $A_2 \rightarrow A_2 * A_3 \mid A_2 / A_3 \mid A_3$

          $A_3 \rightarrow D \mid (A_1)$

          $D \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 9$
    - See **worksheet question 2**.

Parsing

- Is process of determining how a string of terminals can be generated by a grammar → construct parse tree
- For any context-free grammar there is a parser that takes at most $O(n^3)$ time to parse a string of n terminals → for practical programming languages parsing is linear
- Top-down (start at the root towards the leaves) vs. Bottom-up (start at bottom towards root)
- Top-down Parsing
    - Efficient parsers can be constructed more easily by hand
    - Sometimes, we have to backtrack if production turns out to be unsuitable → some element of trial-and-error involved → Recursive-descent parsers don't have that problem
- **Recursive-Descent Parsing**
    - Set of recursive procedures is used to process the input
    - One procedure is associated with each non-terminal in the grammar
    - FIRST(α) is the set of terminals that appear as the first symbols of one or more strings of terminals generated from α → must be considered if there are two productions $A \rightarrow \alpha$ and $A \rightarrow \beta$

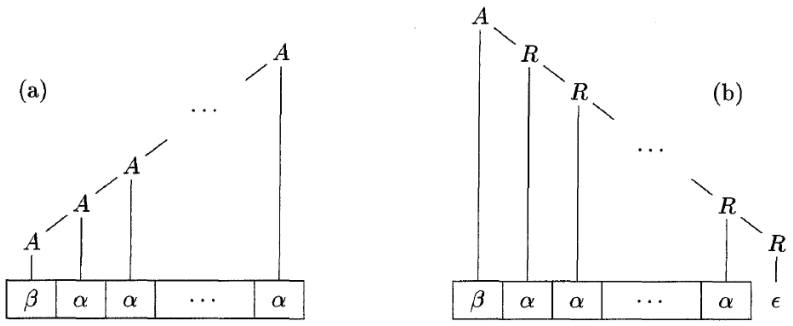
- **Left Recursion**
    - Recursive-descent parsers can run into infinite loops when encountering left-recursive production rules, e.g. $A \rightarrow A\alpha \mid \beta$ (immediate left recursion). Note that there could also be a derivation that leads to left-recursion, e.g. $A \rightarrow B$ and $B \rightarrow A\alpha \mid \beta$ leads to

$A \overset{*}{\Rightarrow} A\alpha$, which is left-recursive

- Eliminate by rewriting the offending rule as right recursive:

$A \to \beta R$

$R \to \alpha R \mid \varepsilon$



- **Left Factoring**
  - When the choice between two alternative productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice
  - Text book example:

$$\begin{array}{lcl} stmt & \to & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\ & \mid & \textbf{if } expr \textbf{ then } stmt \end{array}$$

  - In $A \to \alpha\beta_1 \mid \alpha\beta_2$ we do not know whether to expand A to $\alpha\beta_1$ or to $\alpha\beta_2$
  - Defer that decision by expanding A to $A \to \alpha A'$. Then after seeing input derived from α, we expand A' to β_1 or β_2. Hence the productions become:

$A \to \alpha A'$

$A' \to \beta_1 \mid \beta_2$

  - See **worksheet question 3**