Polytechnic University
Prof. Boris Aronov
Home page: http://cis.poly.edu/~aronov
Office: LC236 (718) 260-3092
email: aronov@poly.edu

CS312A: Programming Languages — Spring 2002
Class page: http://cis.poly.edu/~aronov/cs312-spring2002
Mon 2–3 Wed 2–4 RH705
Office hours: just drop by, except Tue

Homework #1

Assigned February 19, 2002. Due March 4, 2002.

Note: The homework must be brought to class or dropped off at Prof. Aronov's office or in his mailbox at the department. No electronic submission of this homework is accepted. NO late homeworks are accepted.

Problem 1: Associativity, precedences, and ASTs Recall the grammar introduced in class to solve the ambiguity problem for simple arithmetic expressions:

E	→	T
	|	E + T
	|	E - T
T	→	F
	|	T * F
	|	T / F
F	→	⟨id⟩
	|	⟨num⟩
	|	(E)

As before, ⟨id⟩ stands for an identifier and ⟨num⟩ stands for a number. Draw *both* the *full* parse tree *and* the abstract syntax tree for the following expressions:

1. 1
2. $3 + q * 1$
3. $1 * (3 * j)$
4. $(3 - 4) * (5 + 6)$
5. ((((*i*))))

Please remember that parentheses are not necessary in ASTs, as grouping is already represented in the structure of the tree. After working through these examples, I hope you realize why parse trees are not used to store the syntax information, after parsing is done.

Problem 2: Parse trees Consider the following BNF grammar:

⟨statement⟩	→	"{" ⟨statement list⟩ "}"
	|	**loop** ⟨statement list⟩ **forever**
	|	**repeat** ⟨statement list⟩ **until** ⟨expression⟩ **end loop**
	|	**if** "(" ⟨expression⟩ ")" ⟨statement list⟩ **end if**
	|	**if** "(" ⟨expression⟩ ")" ⟨statement list⟩ **else** ⟨statement list⟩ **end if**
	|	⟨id⟩ "←" ⟨expression⟩
	|	⟨nothing⟩
⟨statement list⟩	→	⟨statement⟩
	|	⟨statement⟩ ";" ⟨statement list⟩
⟨expression⟩	→	⟨id⟩ | ⟨num⟩
	|	"(" ⟨expression⟩ ")"
	|	⟨expression⟩ ⟨binary operator⟩ ⟨expression⟩
⟨binary operator⟩	→	"+" | "−" | "/" | "×" | "<" | "=" | ">"

Terminals are **boldfaced** keywords, ⟨id⟩ (for an identifier), ⟨num⟩ (for a numeric constant: one or more digits), the arithmetic operations, and punctuation. All other grammar symbols are terminals. Use standard precedences to disambiguate arithmetic expressions (comparisons after add/subtract after divide/multiply). Symbol "=" stands for the Boolean equality test operator. Starting symbol of the grammar is ⟨statement⟩.

For each string given draw its *full* parse tree:

1. loop forever
2. $i \leftarrow 1$
3. $\{i \leftarrow j + 2\}$
4. $\{i \leftarrow 1$; repeat $i \leftarrow i \times 2$ until $i < n$ end loop $\}$
5. if $(i \times j - k = m/n + 3)$ repeat $j \leftarrow j - 3$; $k \leftarrow k + 2$ until $k < j$ end loop else if $(k > 2)$ else $i \leftarrow 2$ end if end if
   (Yes, this is a syntactically correct input and, yes, it is very messy! Do this one on a separate page, please!)

Problem 3: Extended BNF (EBNF) Write a grammar in EBNF that corresponds to the BNF grammar in Problem 2. Aim for a compact but readable grammar. Use quotation marks, if necessary, to distinguish special EBNF characters and punctuation in the language you are defining.

Problem 4: Regular expressions Write a regular expression matching each of the following classes of strings, and no other string.

1. All alphanumeric strings starting with an upper case letter and ending in a digit.
2. All legal Pascal comments: a comment starts with a "`{`", ends with "`}`", and does not contain a "`}`" inside.
3. Numbers with floating point, and *at least* one digit *either* before *or* after it, e.g. ".9" and "9." are legal but "." by itself is not.
4. All strings that are valid arithmetic expressions given by grammar in Problem 1, but without parentheses (more precisely, we use the same grammar, but the rule $F \rightarrow (E)$ is removed). *(Hint: look at the EBNF version of the grammar. Rewrite it as one big rule and then turn it into a regular expression.)*
5. All strings of lowercase letters *not* containing the word `nice`. *(Tricky.)*