# CSCI 3155: Principles of Programming Languages

Exercise sheet #3

6th June 2007

**Name #1:** _____

**Name #2:** _____

**Name #3:** _____

**Group name:** _____

## Syntax and Semantics

*Exercise* 1. (**Skill 2.7**) Explain, in your own words, what the concepts *meta-language* and *object language* mean. Support your explanation with an example.

*Exercise* 2. Consider the following grammar, with start symbol $\langle S \rangle$:

$$
\begin{array}{rcl}
\langle S \rangle & \to & \langle prog \rangle \\
\langle prog \rangle & \to & \langle stmt \rangle \ \texttt{;} \ \langle prog \rangle \\
& | & \varepsilon \\
\langle stmt \rangle & \to & \texttt{id} := \langle expr \rangle \\
& | & \texttt{id} \mathrel{+}= \langle expr \rangle \\
& | & \textbf{if} \ \langle expr \rangle \ \textbf{then} \ \langle stmt \rangle \ \textbf{end} \\
\langle expr \rangle & \to & \textbf{0} \\
& | & \textbf{1} \\
& | & \texttt{id}
\end{array}
$$

Here, id is a terminal corresponding to a token whose only two valid lexemes are "a" and "b".

(a) (**Skills 2.2, 2.3**) Generate three programs from the grammar. Each program must satisfy a certain condition:

i) This program must not contain a semicolon ("**;**") character.

ii) This program must contain precisely one semicolon but no "**if**".

iii) This program must contain precisely two "**if**"s, but no semicolon.

(b) (**Skills 2.2, 2.5**) Is the grammar ambiguous? If so, give an example to illustrate.

*Exercise* 3. In Handout A, our implementation of addition, subtraction etc. assumed that numbers can be of arbitrary size. In some languages (such as Java or C++), built-in numbers are normally bounded in size by the number of bits available for their representation. For example, in Java an `int` value can only range from $-2^{31}$ to $2^{31} - 1$, and in C (on modern machines) an `unsigned char` can only range from 0 to 255.

To reflect this restriction, we will update the denotational semantics of $\langle expr \rangle$`+`$\langle expr \rangle$ from the handout. The semantics should match those of a C `unsigned char`. You may use the first (easiest) definition of addition, from the beginning of Section 2.1 on the handout.

(a) Update the semantics to "wrap around" if an addition yields a number that does not fit into the interval from 0 to 255. For example, $[\![\texttt{200+57}]\!] = 1$. This matches the behaviour of C and is very efficient to implement.

(b) Update the semantics to signal an error if an addition yields a number that falls outside of the interval. This matches the behaviour of *Standard ML*.

(c) Both wrap-around and *overflow errors* are used in the implementations of existing programming languages, sometimes side by side. Which solution do you prefer? Argue, using our criteria.

*Exercise* 4. Handout A did not describe precisely how we can determine the semantics of integral numbers. Most language definitions leave out this part, since it is considered "obvious". However, a precise description of the semantics of integral numbers is not entirely trivial.

(a) Using digits and whatever other symbols you might need as terminals, write a BNF grammar that recognises all integers, of arbitrary size.

(b) (**Skill 2.4**) Write a denotational semantics for your grammar that translates the syntactic representation of an integer to an integer number.