# Programming Languages

Qualifiers Examination
Fall 2000

July 8, 2003

## Introduction

In this question we would like to discuss the issue of *non-determinism* in programming languages. The idea is simple: we would like to provide programming languages with the ability of *"making choices"* and later verify whether the choices made are correct or not (and take appropriate actions).

Let us consider a simple imperative programming language, that we call *C–*. The syntax of C– is the following:

```
Program ::= Command .
Command ::= Command ; Command
    | if Boolean then Command else Command
    | Identifier = Expression
Expression ::= Expression + Expression
    | Numeral
    | Identifier
Boolean ::= Expression == Expression
    | ! Expression
```

For simplicity we assume that the variables in the langauge are all of type integer.

## Question 1 [10 pts]

Let us consider the following extensions to the C– language:

- we add one command called **choose** with the following syntax:

```
Command ::= ... | choose Identifier ListValues
ListValues ::= Numeral
    | Numeral , ListValues
```

This command assigns to the identifier a value drawn from the list of values; the choice of the value is arbitrary. E.g.,

```
choose x 4, 6, 1, 0
```

will select one value from the list $4, 6, 1, 0$ and assign it to the variable `x`.

- we add one command called **fail** which is used to cause a failure in the current computation. The effect of the failure is to backtrack the computation back to the last **choose** operation and select a different value from the list of values. If all the values in the list have already been tried, then the computation continues to backtrack to the previous **choose** operation. If no alternatives are available, then the whole execution terminates with a failure message.

  For example,

  ```
  y = 256;
  choose x  8, 16, 32;
  if ! (x * x == y) then fail else output = x;
  ```

  will succeed with `x` equal to 16. If the initial value of `y` is instead 7 then the computation will fail.

Modify the the traditional denotational semantic definition of the imperative language (e.g., use the one in page 76 of the Schmidt's book) to properly deal with the new commands introduced. Explain the changes to the existing components of the denotational semantics (e.g., changes to the store or addition of new arguments) and laid out clearly the semantic algebras and valuation functions for the new commands.

# Question 2 [10 pts]

Describe how the implementation of a traditional imperative programming language needs to be modified to accommodate for the new commands.
*[Note:] make sure that your implementation mechanism correctly handles programs such as*

```
choose x 45, 39, 27
y = x + 2;
z = y + x;
if ! (y+z == 121) then fail;
```

*In particular you need to ensure that the final value of the variables* `y` *and* `z` *is correct (i.e., it reflects the current choice of value for* `x`*)—you can see this by tracing the execution of the program assuming that* `choose` *first assigns* `x = 45` *and then (after failing) it assigns* `x = 39`.