Imperativeness and an imperative core *WHILE*

Principles of Programming Languages

Mark Armstrong

Fall 2020

1 Preamble

1.1 TODO Notable references

:TODO:

1.2 TODO Table of contents

2 Introduction

In this section we leave behind the “pure” languages which we have considered so far, and discuss the central features of imperative languages.

We then construct a simple imperative language, based on the `while` loop, giving several approaches to defining its semantics (and extend the language slightly.)

3 The “Von Neumann Architecture”

:TODO:

4 Imperative traits

:TODO:

5 The *WHILE* language

We now construct a simple imperative language, and give a stack-machine based semantics for it followed by a “small-step”, reduction based semantics for it.

After this, we will extend the language slightly and provide other approaches to its semantics.

To begin, the language consists of

- integer expressions, which may involve (integer-valued) variables,
- boolean tests, which can only appear in conditional/iteration constructs, and
- statements, which include assignment, composition, a conditional `if`, and the iterating `while` statement.

5.1 Syntax

```
⟨expr⟩ ::= constant_integer | variable
         | ⟨expr⟩ ('+' | '*' | '-' | '÷') ⟨expr⟩

⟨test⟩ ::= ⟨expr⟩ ('=' | '<' | '>') ⟨expr⟩
         | ⟨test⟩ ('and' | 'or') ⟨test⟩

⟨stmt⟩ ::= 'skip'
         | variable '≔' ⟨expr⟩
         | ⟨stmt⟩ ';' ⟨stmt⟩
         | 'if' ⟨test⟩ 'then' ⟨stmt⟩ 'else' ⟨stmt⟩
         | 'while' ⟨test⟩ 'do' ⟨stmt⟩
```

5.2 A stack-machine semantics for *WHILE*

Constants are simply moved from the control stack to the results stack.

$\langle \mathtt{n} \cdot \mathtt{c},\ \mathtt{r},\ \sigma\rangle \quad \langle \mathtt{c},\ \mathtt{n} \cdot \mathtt{r},\ \sigma\rangle$

For variables, we instead place the value of the variable at the current state onto the results stack.

$\langle \mathtt{v} \cdot \mathtt{c},\ \mathtt{r},\ \sigma\rangle \quad \langle \mathtt{c},\ \sigma(\mathtt{v}) \cdot \mathtt{r},\ \sigma\rangle$

$\langle \mathtt{'E_1\ op\ E_2'} \cdot \mathtt{c},\ \mathtt{r},\ \sigma\rangle \quad \langle \mathtt{E_2} \cdot \mathtt{E_1} \cdot \mathtt{op} \cdot \mathtt{c},\ \sigma(\mathtt{v}) \cdot \mathtt{r},\ \sigma\rangle$

$\langle \mathtt{op} \cdot \mathtt{c},\ \mathtt{n_1} \cdot \mathtt{n_2} \cdot \mathtt{r},\ \sigma\rangle \quad \langle \mathtt{c},\ \mathtt{n} \cdot \mathtt{r},\ \sigma\rangle \quad \mathtt{where\ n = n_1\ op\ n_2}$

5.3 A small-step semantics for ***WHILE***

6 The ***WHILE*** language with scoping

We now extend our *WHILE* language with the necessary syntax to express *variable scopes*, and then give scoping rules for programs.

Our approach introduces both *global* and *local* variables.

- Global variables may be considered to be the input/output to the program in this model.

6.1 Syntax

```
⟨expr⟩ ::= constant_integer | variable
        | ⟨expr⟩ ('+' | '*' | '-' | '÷') ⟨expr⟩

⟨test⟩ ::= ⟨expr⟩ ('=' | '<' | '>') ⟨expr⟩
        | ⟨test⟩ ('and' | 'or') ⟨test⟩

⟨stmt⟩ ::= 'skip'
        | 'local' variable 'in' ⟨stmt⟩
        | variable '≔' ⟨expr⟩
        | ⟨stmt⟩ ';' ⟨stmt⟩
        | 'if' ⟨test⟩ 'then' ⟨stmt⟩ 'else' ⟨stmt⟩
        | 'while' ⟨test⟩ 'do' ⟨stmt⟩

⟨prog⟩ ::= ⟨globals⟩ ⟨stmt⟩

⟨globals⟩ ::= 'global' { variable }
```

6.2 The scoping rules

:TODO:

6.3 A small-step semantics for ***WHILE*** with scoping

:TODO:

6.4 A ***big-step*** semantics for ***WHILE*** with scoping

:TODO: