McMaster University
Department of Computing and Software
Dr. W. Kahl

CAS 706
Fall 2010
Exercise Sheet 4

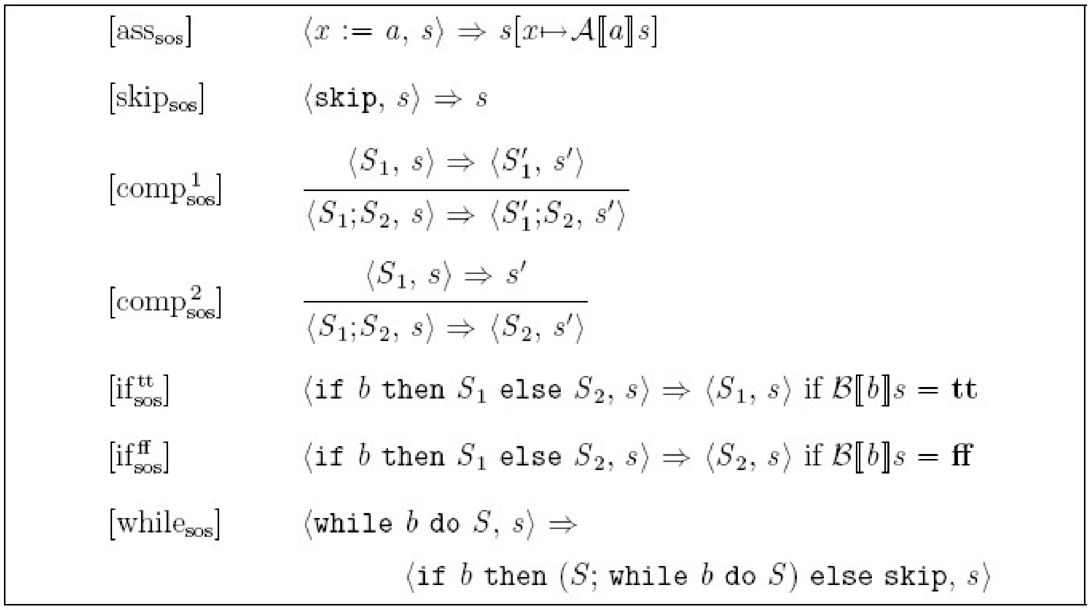
CAS 706 — Programming Languages

2 December 2010

1. Operational Semantics

(a) Produce the **Natural semantics rules** for the textbook programming language While.

(b) **Textbook Exercise 3.2**



$[\mathrm{ass}_{\mathrm{sos}}]$	$\langle x := a,\ s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$
$[\mathrm{skip}_{\mathrm{sos}}]$	$\langle \texttt{skip},\ s\rangle \Rightarrow s$
$[\mathrm{comp}^1_{\mathrm{sos}}]$	$\dfrac{\langle S_1,\ s\rangle \Rightarrow \langle S_1',\ s'\rangle}{\langle S_1;S_2,\ s\rangle \Rightarrow \langle S_1';S_2,\ s'\rangle}$
$[\mathrm{comp}^2_{\mathrm{sos}}]$	$\dfrac{\langle S_1,\ s\rangle \Rightarrow s'}{\langle S_1;S_2,\ s\rangle \Rightarrow \langle S_2,\ s'\rangle}$
$[\mathrm{if}^{\mathrm{tt}}_{\mathrm{sos}}]$	$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle S_1,\ s\rangle$ if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
$[\mathrm{if}^{\mathrm{ff}}_{\mathrm{sos}}]$	$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle S_2,\ s\rangle$ if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$
$[\mathrm{while}_{\mathrm{sos}}]$	$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \Rightarrow$ $\langle \texttt{if}\ b\ \texttt{then}\ (S;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else skip},\ s\rangle$

Table 2.2: Structural operational semantics for **While**

Extend While with the statement

`assert` b `before` S

The idea is that if b evaluates to true, then we execute S, and otherwise the execution of the complete program aborts. Extend the structural operational semantics of Table 2.2 to express this (without assuming that While contains the `abort`-statement). Show that `assert true before` S is semantically equivalent to S but that `assert false before` S is equivalent to neither `while true do skip` nor `skip`.

(c) **Textbook Exercise 3.4**

We shall now extend While with the statement `random(`x`)` and the idea is that its execution will change the value of x to be any positive natural number. Extend the natural semantics as well as the structural operational semantics to express this. Discuss whether `random(`x`)` is a superfluous construct in the case where While is also extended with the `or` construct.

(d) **Textbook Exercise 3.6**

The textbook defines evaluation of arithmetic expressions of While as follows:

$$
\begin{array}{lcl}
\mathcal{A}[\![n]\!]s & = & \mathcal{N}[\![n]\!] \\
\mathcal{A}[\![x]\!]s & = & s\ x \\
\mathcal{A}[\![a_1 + a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s + \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1 \star a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s \cdot \mathcal{A}[\![a_2]\!]s \\
\mathcal{A}[\![a_1 - a_2]\!]s & = & \mathcal{A}[\![a_1]\!]s - \mathcal{A}[\![a_2]\!]s
\end{array}
$$

Table 1.1: The semantics of arithmetic expressions

Specify a structural operational semantics for arithmetic expressions where the individual parts of an expression may be computed in parallel. Try to prove that you still obtain the result that was specified by the original semantics function.

(e) information hiding — how can the principle of information hiding be implemented in Java programs.

(f) encapsulation — what mechanism(s) is(are) used in Java to facilitate encapsulation.

2. Syntax and Semantics of Imperative Programs

We consider a simple imperative programming language with exceptions.

The **abstract syntax** of this programming language is the following:

Stmt	::=	**skip**
	|	*Id* **:=** *Expr*
	|	*Stmt* **;** *Stmt*
	|	**if** *Expr* **then** *Stmt* **else** *Stmt*
	|	**while** *Expr* **do** *Stmt*
	|	**throw** *Expr*
	|	**try** *Stmt* **catch(** *Id* **)** *Stmt*
Expr	::=	*Id*
	|	*Num*
	|	*Bool*
	|	*Expr Op Expr*
Op	::=	$+ \mid - \mid * \mid / \mid \leq \mid \geq \mid < \mid >$

(a) Draw **abstract syntax trees** for the following statements:

- (**if** $a < b$ **then** b **:=** a **else throw** 7) **; while** $a > 0$ **do** a **:=** $a - b$
- **try** (**if** $a < b$ **then** b **:=** a **else throw** 7) **;**
  while $a > 0$ **do** a **:=** $a - b$
  catch(e **) throw** $(1000 + e)$

We choose the following basic semantic domains:

Val	= *Bool* + *Num*	values
Store	= *Id* $\rightharpoonup$ *Val*	(simple) stores

($A \rightharpoonup B$ is the set of *partial functions* from the set A to the set B.)

We denote the elements of *Val* by True, False, 0, 1, 2, …

From an operational point of view, assuming that the expression e evaluates to the number k, the statement "**throw** e" raises exception k.

We allow **only numbers** as exceptions.

If a statement raising an exception is not enclosed by any "**try** _ **catch**" construct, then this exception immediately leads to program termination with an *uncaught exception.*

If there is an enclosing "**try** _ **catch**" construct, then this is of the shape "**try** _ **catch(** i **)** s_2" for some identifier i and a statement s_2. In that case, execution proceeds immediately to s_2 in an environment where the identifier i is bound to the numerical value of the caught exception.

(b) Write down the *Store* that the statement s_2 executes from when control arrives at s_2 in the following program:

$$k \mathbin{\mathbf{:=}} 100 \;\mathbf{;}\; \mathbf{try}\; q \mathbin{\mathbf{:=}} 42 \;\mathbf{;}\; \mathbf{throw}\; 14 \;\mathbf{;}\; s \mathbin{\mathbf{:=}} q + 1 \;\mathbf{;}\; \mathbf{catch(}\; n\; \mathbf{)}\; s_2$$

The statement semantics needs to accommodate the possibility of locally uncaught exceptions. Therefore, it produces a state transition function that returns either just a *Store* or a *Store* together with an exception number, i.e., statement semantics $\llbracket\_\rrbracket_S$ has the following type (the state transition function may be partial to accommodate non-termination):

$$\llbracket\_\rrbracket_S : \mathit{Stmt} \to (\mathit{Store} \nrightarrow (\mathit{Store} + (\mathit{Store} \times \mathit{Num})))$$

(c) Define $\llbracket\mathbf{try}\; s_1\; \mathbf{catch(}\; i\; \mathbf{)}\; s_2\rrbracket_S$ for arbitrary statements $s_1, s_2 : \mathit{Expr}$ and an arbitrary identifier $i : \mathit{Id}$.

Since we already have exceptions in our language, we want primitive operations to raise exceptions if they cannot execute properly, and **never** produce $\bot$.

In our concrete language, we want division by zero to raise exception 24, and all other operator applications have to terminate successfully.

(d) Propose **and explain** a type for expression semantics $\llbracket\_\rrbracket_E$.

(e) Define $\llbracket\mathbf{throw}\; e\rrbracket_S$ for an arbitrary expression $e : \mathit{Expr}$.

(f) Define $\llbracket v \mathbin{\mathbf{:=}} e\rrbracket_S$ for an arbitrary identifier $v : \mathit{Id}$ and an arbitrary expression $e : \mathit{Expr}$.

(g) Define $\llbracket\mathbf{if}\; b\; \mathbf{then}\; s_1\; \mathbf{else}\; s_2\rrbracket_S$ for an arbitrary expression $b : \mathit{Expr}$ and arbitrary statements $s_1, s_2 : \mathit{Stmt}$.

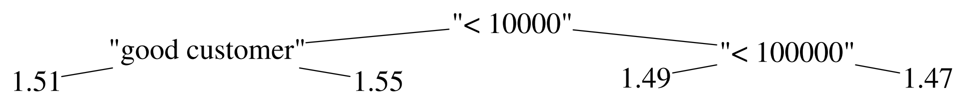
3. Decision Trees

Let the following definitions be given, defining a data type for decision trees parameterized by a *question* type `q` and an *answer* type `a`, and three example decision trees. (`DT` stands for `DecisionTree`; `R` stands for `Result` and `Br` for `Branch`.)

```
data DT q a = R a | Br q (DT q a) (DT q a)
```

```
dt1 = Br "tired?" (Br "evening?" (R 1) (R 2))
                  (Br "working?" (R 7) (R 5))
dt2 = Br (> 0) (R 1) (Br (< 0) (R (-1)) (R 0))
dt3 = Br 5 (Br 3 (R 'a') (R 'd'))
           (Br 12 (Br 6 (R 'e') (Br 9 (R 'g') (R 'j'))) (R 'p'))
```

(a) A bank might be using the following decision tree to arrive at a rate for a certain kind of currency transaction at a certain point in time:



Write an expression denoting this decision tree, either as an Haskell expression of type `DT String Double`.

(b) Draw the trees defined as `dt1`, `dt2` and `dt3`.

(c) What are the types of the trees defined as `dt1`, `dt2` and `dt3`?

Let the following **additional** definitions be given:

```
decide (R a) ds = a
decide (Br q left right) (d:ds)  =  if  q d  then  decide left ds
                                              else  decide right ds

repeat x = x : repeat x
```

(d) Simulate **lazy evaluation** to normal form for the following expression — write down the sequence of **all intermediate expressions**: `decide dt2 (repeat 0)`

(e) **Derive** the type of `decide`, and **explain** your method of derivation.

(f) Mark for **each** of the following statements whether it is true or false. All these statements are to be understood in the context of all the material in this question up to here.

Justify your answers.

1. True: ☐ False: ☐ The datatype "`DT`" has exactly three constructors.
2. True: ☐ False: ☐ "`Br`" is a constructor with three arguments.

(g) Would evaluating the expression "`decide dt2 (repeat 0)`" from (d) using the default evaluation method of ML produce the same sequence of intermediate expressions as the above lazy evaluation sequence? Would it arrive at the same result? Explain!

(h) **Define** a function

```
select :: DT q a -> [Bool] -> DT q a
```

such that `select dt bs` selects the subtree of `dt` addressed by the list `bs`, where True selects the left branch and False the right branch, and the first element of `bs` corresponds to the outermost branch in `dt`. For example, the following hold:

```
select dt1 [True]        = Br "evening?" (R 1) (R 2)
select dt1 [False, True] = R 7
```

The topic of the next few items are implementations of decision in C and Java.

(i) Provide C type definitions for a type `DTree` implementing decision trees of type `DT String Int`. **Document** your design choices (keywords).

(j) Produce a drawing representing storage allocation for the `DTree` value corresponding to the example tree `dt1` using the data structures you defined:

(k) Explain how a tree datatype allowing trees such as `dt2` would be implemented most naturally in C. Discuss the differences with respect to the type system.

(l) Define a C function `query()` that accepts a `DTree` as argument and then prompts the user on `stdout` with the questions in the decision tree and retrieves the user's answers (`True` or `False`) via `stdin`, following the path through the decision tree defined by those answers, until it encounters a result node, the contents of which serves as the function's return value. **Document** your code, and if you use library functions (existing or assumed) for input, describe their functionality.

(m) In **Java**, provide the following:

– **object-oriented** type definitions for a type `DTree` implementing decision trees of type `DT String Int`

- appropriate **constructors**
- an **appropriate** implementation of the function `query()`.

Document (keywords) your design choices (and library functions for input).

(n) Explain how a tree datatype allowing trees such as `dt2` would be implemented most naturally in Java. Discuss the differences with respect to the type system.

(o) Explain how a tree datatype allowing **all possible** decision trees that can be constructed using `DT` would be implemented most naturally in Java. Discuss the differences with respect to the type system!

4. `map` and `filter`

(a) Write the higer order functions `map` and `filter` in a functional language.

(b) Can you express `map` and `filter` in C or Pascal? If so, demonstrate how! If not, justify your answer!

(c) Can you express `map` and `filter` in Java or some other object-oriented language? If so, demonstrate how! If not, justify your answer!

(d) Discuss differences with respect to typing between the `map` and `filter` in functional programming languages on the one hand, and any solutions you may have proposed in imperative or object-oriented programming languages on the other hand.

5. Parameter Passing

For each of the following, give a pre-post-condition specification of a program and write the program in a While-like syntax with at least one procedure call that

(a) works properly with respect to the specification when parameters to procedures are passed by value, but does not work properly when parameters to procedures are passed by reference;

(b) works properly iwth respect to the specification when parameters to procedures are passed by reference, by does not work properly when parameters to procedures are passed by name;

6. Garbage Collection

Garbage collection of operating systems data structures usually is implemented by reference counting — why?

Garbage collection in Java implementations is **not** implemented by reference counting — why?

7. Modularization in Java

Discuss the following concepts in the context of Java programming:

(a) modularization — what kind of modules Java programs have, and what mechanism(s) may be used for modularization in Java programs.

(b) information hiding — how can the principle of information hiding be implemented in Java programs.

(c) encapsulation — what mechanism(s) is(are) used in Java to facilitate encapsulation.

8. Programming Language Features of Eiffel

For anwering this question, no previous knowledge of Eiffel is assumed, and no additional information about Eiffel is required.

Throughout this question, the pure object-oriented programming language Eiffel is described using Java terminology.

(a) In Eiffel, class interfaces do not distinguish between fields in the one hand, and methods with no arguments on the other hand.

What are the advantages and disadvantages of this scheme from a programmer's point of view?

What are the advantages and disadvantages of this scheme from a compiler writer's point of view?

In Eiffel, classes can be equipped with **invariants**, and methods with **preconditions** and **postconditions**. All these are called **assertions**. All assertions are expressions of Boolean type with no side-effects and may involve field references; preconditions and postconditions also may contain references to method arguments, and postconditions in addition may contain references to field values of the "**old**" state of the object, i.e., its state **before** the method call. Apart from that, assertions involve the usual relational operators `<`, `<=`, `=`, … and the usual propositional logic junctors `and`, `or`, `not`, … If an assertion is violated during program execution, an exception is raised.

(b) Mark for **each** of the following statements whether it is true or false.

For each question, provide a one-paragraph **justification.** (Answers without justification do not count.)

1. True: ☐ False: ☐	Assertions can be used to incorporate the full specification of every method into the Eiffel code.
2. True: ☐ False: ☐	Tested assertions are correctness proofs.
3. True: ☐ False: ☐	The runtime cost of checking assertions is likely negligible.

Eiffel supports multiple inheritance. It has been said, that

"there are two things that [Eiffel] got right that nobody else got right anywhere else: support for design by contract, and multiple inheritance."

Inheritance in Eiffel includes the following features:

- No class can have two members with the same name, no matter whether inherited or introduced in that class.
- Subclasses can **rename** selected superclass members.
- Subclasses can make `private` superclass members `public`.
- Subclasses can make `public` superclass members `private` or even exclude them altogether.
- Subclasses can override implementations of superclass methods.
- If a superclass method `m` accepted an argument of class `C`, subclasses can change `m` to accept arguments only of some subclass `D` of `C`. This is called the **covariance rule**.
- If `v` is a variable of class type `C` and `D` is a subclass of `C`, then objects of `D` can be assigned to `v`.
- A subclass inherits the invariants of all its superclasses.

(c) List at least three questions that you need to ask about apparent conflicts between the above-mentioned features of inheritance and member redefinition, and that are not answered above. Explain and discuss each question shortly.

The End