YASL Interpreter Semantics

Brian Howard

Fall 2015, Version 1.1

This is a slightly simplified version of the interpreter semantics described in class on October 29. The main change is that cells directly contain their corresponding primitive values, rather than a wrapped `IntValue` or `BoolValue`.

Interpret an entire program by creating an empty symbol table that maps identifiers to `Value`, then interpret the block inside a new scope whose name is the program name:

$$\mathcal{I}[\![\texttt{program ID ; Block .}]\!] = \begin{cases} \texttt{t = new SymbolTable[Value]} \\ \texttt{t.enter(ID)} \\ \mathcal{I}[\![\texttt{Block}]\!](t) \\ \texttt{t.exit()} \end{cases}$$

Interpret a block by interpreting in turn each of the declarations, followed by each of the statements of the block:

$$\mathcal{I}[\![\texttt{ConstDecls VarDecls ProcDecls begin Stmts end}]\!](t) = \begin{cases} \texttt{foreach cd} \in \texttt{ConstDecls:}\ \mathcal{I}[\![\texttt{cd}]\!](t) \\ \texttt{foreach vd} \in \texttt{VarDecls:}\ \mathcal{I}[\![\texttt{vd}]\!](t) \\ \texttt{foreach pd} \in \texttt{ProcDecls:}\ \mathcal{I}[\![\texttt{pd}]\!](t) \\ \texttt{foreach s} \in \texttt{Stmts:}\ \mathcal{I}[\![\texttt{s}]\!](t) \end{cases}$$

Interpret a constant declaration by binding the identifier to a new `IntValue` holding the number. It is an error if the identifier is already bound in the current scope:

$$\mathcal{I}[\![\texttt{const ID = NUM ;}]\!](t) = \texttt{t.bind(ID, new IntValue(NUM))}$$

Interpret a variable declaration by binding the identifier to a new cell containing the default value of the type (0 for integer, false for boolean). It is an error if the identifier is already bound in the current scope:

$$\mathcal{I}[\![\texttt{var ID : int ;}]\!](t) = \texttt{t.bind(ID, new IntCell(0))}$$
$$\mathcal{I}[\![\texttt{var ID : bool ;}]\!](t) = \texttt{t.bind(ID, new BoolCell(false))}$$

Interpret a procedure declaration by binding the identifier to a new `ProcValue` holding the parameters and block. It is an error if the identifier is already bound in the current scope:

$$\mathcal{I}[\![\texttt{proc ID Params ; Block ;}]\!](t) = \texttt{t.bind(ID, new ProcValue(Params, Block))}$$

Interpret an assignment statement by looking up the identifier on the left, interpreting the expression on the right, and then setting the new value of the cell on the left (it is an error if it is not a cell of the correct type) to the result of the expression:

$$\mathcal{I}[\![\texttt{ID = Expr ;}]\!](t) = \begin{cases} \texttt{lhs = t.lookup(ID)} \\ \texttt{rhs = } \mathcal{I}[\![\texttt{Expr}]\!](t) \\ \texttt{lhs.set(rhs)} \end{cases}$$

Interpret a procedure call by looking up the identifier to get a `ProcValue`, then interpreting the list of arguments to get a list of values, then creating a new local scope named for the procedure and binding each of the parameters to the corresponding argument. It is an error if the number of parameters does not match the number of arguments, if the type of a parameter does not match the type of its argument, or if a `var` parameter is not given a cell value. It is also an error if a parameter identifier is bound more than once in the new scope:

$$\mathcal{I}[\![\texttt{ID Args ;}]\!](t) = \begin{cases} \texttt{ProcValue(params, block) = t.lookup(ID)} \\ \texttt{args = foreach e} \in \texttt{Args: yield } \mathcal{I}[\![\texttt{e}]\!](t) \\ \texttt{t.enter(ID)} \\ \texttt{call(params, block, args, t) // see below} \\ \texttt{t.exit()} \end{cases}$$

The `call` function is defined by the following cases (note that `Nil` is an empty list, while `head :: tail` is a list with the given `head` element and rest of the list `tail`):

$$\begin{cases} \texttt{call(Nil, block, Nil, t) = } \mathcal{I}[\![\texttt{block}]\!](t) \\ \texttt{call((ID : int) :: params, block, arg :: args, t) =} \\ \quad \texttt{t.bind(ID, new IntCell(arg.intValue)); call(params, block, args, t)} \\ \texttt{call((ID : bool) :: params, block, arg :: args, t) =} \\ \quad \texttt{t.bind(ID, new BoolCell(arg.boolValue)); call(params, block, args, t)} \\ \texttt{call((var ID : int) :: params, block, arg :: args, t) =} \\ \quad \texttt{t.bind(ID, arg); call(params, block, args, t) // if arg is an IntCell} \\ \texttt{call((var ID : bool) :: params, block, arg :: args, t) =} \\ \quad \texttt{t.bind(ID, arg); call(params, block, args, t) // if arg is a BoolCell} \end{cases}$$

Interpret a compound statement by interpreting each of the contained statements in turn:

$$\mathcal{I}[\![\texttt{begin Stmts end ;}]\!](t) = \texttt{foreach s} \in \texttt{Stmts: } \mathcal{I}[\![\texttt{s}]\!](t)$$

Interpret an if-then statement by interpreting the test, which should give a boolean value. If the value is true, then interpret the statement:

$$\mathcal{I}[\![\texttt{if Expr then Stmt}]\!](t) = \begin{cases} \texttt{test = } \mathcal{I}[\![\texttt{Expr}]\!](t) \\ \texttt{if test.boolValue: } \mathcal{I}[\![\texttt{Stmt}]\!](t) \end{cases}$$

Interpret an if-then-else statement by interpreting the test, which should give a boolean value. If the value is true, then interpret the first statement; otherwise interpret the second statement:

$$\mathcal{I}[\![\texttt{if Expr then Stmt1 else Stmt2}]\!](t) = \begin{cases} \texttt{test = } \mathcal{I}[\![\texttt{Expr}]\!](t) \\ \texttt{if test.boolValue: } \mathcal{I}[\![\texttt{Stmt1}]\!](t) \\ \texttt{else: } \mathcal{I}[\![\texttt{Stmt2}]\!](t) \end{cases}$$

Interpret a while statement by interpreting the test, which should give a boolean value. If the value is true, then interpret the statement and reinterpret the test; repeat until the test is false:

$$\mathcal{I}[\![\texttt{while Expr do Stmt}]\!](t) = \begin{cases} \texttt{test = } \mathcal{I}[\![\texttt{Expr}]\!](t) \\ \texttt{while test.boolValue:} \\ \quad \mathcal{I}[\![\texttt{Stmt}]\!](t) \\ \quad \texttt{test = } \mathcal{I}[\![\texttt{Expr}]\!](t) \end{cases}$$

Interpret a prompt statement by printing the message string (without a trailing newline) and waiting for a line of input. If an identifier is specified along with the string, then attempt to convert the input to an integer and assign that value to the cell associated with that identifier. It is an error if the identifier isn't associated with an `IntCell`, or if the input string is not an integer:

$$\mathcal{I}[\![\texttt{prompt STRING ;}]\!](t) = \begin{cases} \texttt{print(STRING)} \\ \texttt{readLine()} \end{cases}$$

$$\mathcal{I}[\![\texttt{prompt STRING , ID ;}]\!](t) = \begin{cases} \texttt{lhs = t.lookup(ID)} \\ \texttt{print(STRING + " ")} \\ \texttt{input = readLine()} \\ \texttt{lhs.set(input.asInt)} \end{cases}$$

Interpret a print statement by printing each of its items (strings or integer expressions) on a single line:

$$\mathcal{I}[\![\texttt{print Items ;}]\!](t) = \begin{cases} \texttt{foreach it} \in \texttt{Items:} \\ \quad \texttt{if it is an Expr:} \\ \quad\quad \texttt{value =} \mathcal{I}[\![\texttt{it}]\!](t) \\ \quad\quad \texttt{print(value.intValue)} \\ \quad \texttt{else: // it is a STRING} \\ \quad\quad \texttt{print(it)} \\ \texttt{printLine()} \end{cases}$$

Interpret a binary operation expression by interpreting the left and right operands, then performing the indicated operation and returning the resulting value. It is an error if either operand is not of the desired type:

$$\mathcal{I}[\![\texttt{Expr1 op Expr2}]\!](t) = \begin{cases} \texttt{lhs =} \mathcal{I}[\![\texttt{Expr1}]\!](t) \\ \texttt{rhs =} \mathcal{I}[\![\texttt{Expr2}]\!](t) \\ \texttt{switch op:} \\ \quad \texttt{case AND, OR:} \\ \quad\quad \texttt{return new BoolValue(lhs.boolValue} \\ \quad\quad\quad \texttt{op rhs.boolValue)} \\ \quad \texttt{case EQ, NE, LE, LT, GE, GT:} \\ \quad\quad \texttt{return new BoolValue(lhs.intValue} \\ \quad\quad\quad \texttt{op rhs.intValue)} \\ \quad \texttt{case PLUS, MINUS, STAR, DIV, MOD:} \\ \quad\quad \texttt{return new IntValue(lhs.intValue} \\ \quad\quad\quad \texttt{op rhs.intValue)} \end{cases}$$

Interpret a unary operation expression by interpreting the operands, then performing the indicated operation and returning the resulting value. It is an error if the operand is not of the desired type:

$$\mathcal{I}[\![\texttt{op Expr}]\!](t) = \begin{cases} \texttt{value =} \mathcal{I}[\![\texttt{Expr}]\!](t) \\ \texttt{switch op:} \\ \quad \texttt{case MINUS:} \\ \quad\quad \texttt{return new IntValue(- value.intValue)} \\ \quad \texttt{case NOT:} \\ \quad\quad \texttt{return new BoolValue(not value.boolValue)} \end{cases}$$

Interpret an integer or boolean literal expression by returning the corresponding wrapped value:

$$\mathcal{I}[\![\texttt{NUM}]\!](t) = \texttt{return new IntValue(NUM)}$$
$$\mathcal{I}[\![\texttt{true}]\!](t) = \texttt{return new BoolValue(true)}$$
$$\mathcal{I}[\![\texttt{false}]\!](t) = \texttt{return new BoolValue(false)}$$

Interpret a variable expression by returning the value bound to the identifier in the symbol table. It is an error if the variable is undefined:

$$\mathcal{I}[\![\texttt{ID}]\!](t) = \texttt{return t.lookup(ID)}$$

There are five kinds of values that may be stored in the symbol table:

- `IntValue(n)`, for an integer `n`: the only operation on such a value `v` is `v.intValue`, which returns `n`;
- `BoolValue(b)`, for a boolean `b`: the only operation on such a value `v` is `v.boolValue`, which returns `b`;
- `IntCell(n)`, for an integer `n`: the operations on such a value `v` are `v.intValue`, which returns `n`, and `v.set(x)`, which modifies the integer stored in the cell from `n` to `x`;
- `BoolCell(b)`, for a boolean `b`: the operations on such a value `v` are `v.boolValue`, which returns `b`, and `v.set(x)`, which modifies the boolean stored in the cell from `b` to `x`;
- `ProcValue(params, block)`: described above when interpreting procedure declarations and calls.

It is an error to perform an operation on the wrong kind of value (for example, attempting to take the `intValue` of a `BoolValue`).