Closures

To correctly implement the lexical scoping discipline, we somehow have to correctly bind all the (apparently) free variables which appear in a function body to their values/answers in the prevailing environment. Note that when we invoke a function with an actual argument, the intended operation is to *substitute the argument for the formal parameter in the body of the function*. Recall that we had introduced the "table" data structure so that variables could be quickly and efficiently looked up (in quick time) ... instead of the expensive operation of substituting answers (values) for each variable.

So if there are "global" variables in the body of a function, i.e., not parameters, and not locally defined using a let-definition, we should be able to find the bindings for these global variables in the prevailing table. Now instead of performing expensive substitution operations, we can "pack in" the table with the function code, and whenever we encounter a global variable, look up its binding in the table. The data structure consisting of the function expression and the table together is called a "*closure*" (because it closes up all the apparently free variable references).

$$Clos = Exp \times Table$$

with function closures represented as $\langle\langle \lambda x\,.\,e,\ \gamma \rangle\rangle$. We add such closures which we designate by $VClos = \{\lambda x\,.\,e \mid x \in \mathcal{X}, e \in Exp\} \times Table$ into our set of canonical answers.

We are now in a position to write the correct Big-step rules for function abstractions and application:

(CalcAbs) $$\frac{}{\gamma \vdash \underline{\lambda x\,.\,e} \Longrightarrow \langle\langle \lambda x\,.\,e,\ \gamma \rangle\rangle}$$

That is, an abstraction calculates to a closure as a canonical answer, packing in the current table into the closure. Of course, in a good implementation, one would not copy a table and create an unwieldy and large data structure, but instead insert a reference to the table. Some program analysis may also help to only keep relevant bindings from the table, and cut away all the unnecessary flab.

(CalcApp) $$\frac{\gamma \vdash \underline{e_1} \Longrightarrow \langle\langle \lambda x\,.\,e',\ \gamma' \rangle\rangle \qquad \gamma \vdash \underline{e_2} \Longrightarrow \underline{a_2} \qquad \gamma'[\underline{x} \mapsto \underline{a_2}] \vdash \underline{e'} \Longrightarrow \underline{a}}{\gamma \vdash \underline{(e_1\ e_2)} \Longrightarrow \underline{a}}$$

The operational semantics rule (**CalcApp**) details how the result of a function call $(e_1\ e_2)$ is calculated: First we calculate the answer for the "operator" expression e_1, which being a function, should result in a closure. Note that this answer closure is of the form $\langle\langle \lambda x\,.\,e',\ \gamma' \rangle\rangle$ where the expression component is an abstraction of the form $\lambda x\,.\,e'$, where x is the formal parameter, and e' the body of the function. Note also that the table packed into the closure *may* be different from the table γ with respect to which we calculated the answer (for example, it may be the result for looking up the table for a named function, which was created in another environment), and hence we indicate this possibly different table as γ'. Next we evaluate the "argument" expression e_2, in the same call-time environment, namely table γ as the operator expression, and let us call the obtained answer a_2. Now (in this call-by-value discipline), we bind the argument answer a_2 to the formal parameter x, and then evaluate the body e' of the function. But this evaluation of e'

is not done with respect to the call-time environment γ, but rather with the environment γ' saved in the closure — from where we will get the correct bindings in a lexical/statically scoped discipline for the global variables that appear in body e' — which is augmented by the binding of the formal parameter to the actual argument, namely $x \mapsto a_2$. The answer a obtained from this evaluation of the function body is returned as the answer of the function call.

You may have a noticed that we included (some) closures — which contain tables as a component — into our set of answers. And tables mapped variables to answers. So there is some mutual recursion in these definitions:
$Clos = Exp \times Table$ and $Table = \mathcal{X} \rightharpoonup_{fin} Ans$, where $VClos \subset Ans$.

Exercise: Write a function *unpack* from closures to expressions, that recursively unpacks a closure into an expression, by substituting for a variable appearing in the expression component of a closure the unpacking of the closure to which it is bound in the table of the closure.

Once you have done so, you may wish to try your hand at the following:

Exercise: Prove the new induction cases in the Soundness Theorem.

Exercise: Prove the new induction cases in the Completeness Theorem.

The Type Preservation theorem also continues to hold!

Exercise: Prove the new induction cases in the Type Preservation Theorem.
Caution: One has to be quite careful in this proof, in stating and using the IH.

Alternative Formulation: Big-step as closure evaluation

Recall that in our formulation of big-step natural semantics with a table, we had written the table γ to the left of a turnstile $\vdash$ to emphasise that the table did not change. But here in the (**CalcApp**) rule, we switch to a possibly different table γ' which is what prevailed when the closure was created, augment γ' with the formal-parameter-to-actual-argument binding in order to calculate the body of the function, we revert to the call-time table γ once we have obtained the answer a, we revert to the call-time table.

There is an alternative formulation of these rules where all answers are closures. tables are mappings from variables to closures, and we present all calculation as happening on closures ... where closures in $Clos$ are transformed into a subset of "canonical closures" in $VClos$.

(**CalcVar'**) $\dfrac{}{\langle\!\langle \underline{x}, \gamma \rangle\!\rangle \Longrightarrow_{clos} \gamma(x)}$

where the variable x is looked up in table γ, and we obtain a closure as the result.

(**CalcAbs'**) $\dfrac{}{\langle\!\langle \lambda x\,.\,e,\ \gamma \rangle\!\rangle \Longrightarrow_{clos} \langle\!\langle \lambda x\,.\,e,\ \gamma \rangle\!\rangle}$

is a trivial rule, and application is rewritten in terms of closure simplification as:

(CalcApp')

$$\frac{\langle\langle \underline{e_1}, \gamma\rangle\rangle \Longrightarrow_{clos} \langle\langle \lambda x\,.\,e', \gamma'\rangle\rangle \qquad \langle\langle \underline{e_2}, \gamma\rangle\rangle \Longrightarrow_{clos} vcl_2 \qquad \langle\langle \underline{e'}, \gamma'\,[\underline{x} \mapsto vcl_2]\rangle\rangle \Longrightarrow_{clos} vcl}{\langle\langle \underline{(e_1\ e_2)}, \gamma\rangle\rangle \Longrightarrow_{clos} vcl}$$

Now you have a choice to make about how to represent numeric constants such as the integer 3: Is it to be an answer as it is, or should we represent it as a closure of the form $\langle\langle 3, \gamma\rangle\rangle$ for any table γ (in particular, $\langle\langle 3, \varnothing\rangle\rangle$)? The former seems more space-efficient, the latter provides a simpler, uniform definition of tables: $Clos = Exp \times Table$ and $Table = \mathcal{X} \rightharpoonup_{fin} VClos.$ In fact, for the slightly different semantics of lazy evaluation (call-by-name), we will have $Table = \mathcal{X} \rightharpoonup_{fin} Clos.$

Exercise: Write a function *unpack* from closures to expressions, that recursively unpacks a closure into an expression, by substituting for a variable appearing in the expression component of a closure the unpacking of the closure to which it is bound in the table of the closure.

Once you have done so, you may wish to try your hand at the following:

Exercise: Prove the new induction cases in the Soundness Theorem.

Exercise: Prove the new induction cases in the Completeness Theorem.

The Type Preservation theorem also continues to hold!

Exercise: Prove the new induction cases in the Type Preservation Theorem.
Caution: One has to be quite careful in this proof, in stating and using the IH.

Compilation and Stack Machine Execution

How should the Stack Machine be modified to deal with closure formation and function call?

Peter J. Landin in the period 1964-66 wrote two very influential papers:

- The mechanical evaluation of expressions

- The Next 700 Programming Languages

The first paper « is a contribution to the "theory" of the activity of using computers. It shows how some forms of expression used in current programming languages can be modelled in Church's λ-notation, and then describes a way of "interpreting" such expressions. This suggests a method, of analyzing the things computer users write, that applies to many different problem orientations and to different phases of the activity of using a computer. Also a technique is introduced by which the various composite information structures involved can be formally characterized in their essentials, without commitment to specific written or other representations.»

Ideas such as abstract syntax, syntactic sugar, reducing a bigger language to a core sub-language, interpreters, byte code, compilation as a homomorphic traversal of an abstract syntax tree, and execution using a stack machine ... and many other seminal ideas are discussed in this paper.

The SECD Machine is a machine for executing *op-codes* obtained from *compiling* an expression from a high level language.

Language. For simplicity, let us consider the λ-calculus, i.e., the core sublanguage

$$e \ ::= \ x \mid \lambda x \,.\, e_1 \mid (e_1 \ e_2)$$

We can represent this language in OCaml as a data type:

```
type exp = V of string  |  Abs of string * exp  | App of exp * exp;;
```

This core language can be extended to embrace numerals, boolean constants, the unit value, pairs, injections, projections from pairs, if-then-else conditional expressions, case statements, etc.

In the sequel, we will assume that the expressions that we compile are well-typed and (at the top level) have no free variables.

Opcodes
We introduce a notion of opcodes, which will be the elementary operations of a run-time abstract machine.

```
type opcode =  LOOKUP(x)  |  APP  | MKCLOS(x, c)  |  RET;;
```

If we extend the core language, we will have to (re-)introduce opcodes for performing the operations at the machine level corresponding to the evaluation of the expression.

A compiled program is again going to be a list of opcodes. Note that the single opcode `MKCLOS`(x, c) has nested in it a list c of op-codes. (In practical implementations, this will be represented using a reference to a list, so that the opcode can be represented as a fixed size opcode. Also variables x will eventually be converted into memory addresses or register references).

Compilation. Expressions from our language are compiled into opcode sequences. As usual, we specify the *compiler* as a recursive function (a homomorphism, which happens to be a post-order traversal of the abstract syntax tree) as follows:

compile(x) = [`LOOKUP`(x)]
compile($\lambda x \,.\, e_1$) = [`MKCLOS`(x, *compile*(e_1) `@ [RET]`)]
compile($(e_1 \ e_2)$) = *compile*(e_1) `@` *compile* (e_2) @ [`APP`]

Notice that the compilation of applications is very similar to how one would compile an arithmetic expression $e_1 + e_2$ for a stack evaluator, i.e., compiling first e_1 then e_2 and then appending at the end an op-code such as `PLUS`.

More importantly, note that the compilation of λ–abstractions involves a recursive call to compile the body of the function e_1 but this is nested within the `MKCLOS` opcode. Note also that we append at the very end of the compiled body the `RET` opcode, to mark that the result of the function call must be returned to the calling context.

Exercise: Code the compile function in OCaml.

The components of the machine. The SECD machine is named for its constituent 4 components:

S — "Stack" stands for a stack of *answers,* corresponding to the sub-expressions which have already been evaluated so far.

E —"Environment" (which we will here write as a *table* γ), a mapping from variables $\mathcal{X}$ to answers *Ans*.

C — "Code" is a list of opcodes, and corresponds to the rest of the program that remains to be executed.

D — "Dump" is a stack of *(S, E, C)* triples, which represent the context to which control has to return after completion of a function call. (The λ-calculus is after all a higher-order functional language). (Again, in practice, we will not stack up such triples of large data-structures, but instead will use references to them).

Closures
At the machine level, since a syntactic expression has been compiled into an opcode sequence, we present a "machine-oriented" version of closures: as triples $vcl \in VClosI ::= \langle\langle x, c, \gamma \rangle\rangle$, where x is a variable (the formal parameter), c is an opcode sequence, and γ is a table, where tables are finite-domain functions from variables to answers, with $VClosI \subseteq AnsI$

We will see below that the opcode `MKCLOS`(x, c) will make this form of a "value closure" by packing in the current table (environment).

Exercise: Define a suitable data type representation of closures. Note that closures and tables will be mutually recursively defined.

Configurations of the SECD Machine

The configurations of the SECD machine are quadruples of the form

$$(S, \gamma, c, D)$$

where
- *S* is a stack (representable as a list) of *Ans;*
- $\gamma \in \mathcal{X} \rightharpoonup Ans$; (tables may be representable as finite domain functions or as lists)
- *c* is an op-code list; and
- *D* is a stack of *(S, E, C)* triples (again, a dump can be represented as a list of triples)

Note that in an efficient implementations, we will not actually stack up (S, E, C) triples, but only references to them.

Transition Rules of the SECD Machine.

The operation of the machine is presented as a single step transitions, based on the opcode at the head of the *C* component, There is only one rule for each opcode, and whether a transition is possible may depend on the state of some of the other component.
If the configuration does not match the expected shape, then the machine gets *stuck*.

- Look up a variable in the table.

$(S, \gamma, \texttt{LOOKUP}(x) :: c', D) \;\; ==> \;\; (a::S, \gamma, c' D),$ *provided x in dom(γ) and* $a = \gamma(x).$

- Make a closure by packing in the current table, and place it on the stack.

$(S, \gamma, \texttt{MKCLOS}(x, c') :: c'', D) ==> (\langle\langle\langle x, c', \gamma \rangle\rangle\rangle :: S, \gamma, c'', D)$

- Bind the actual argument a (at the top of the stack) to the formal parameter x, and add it to the environment (table) γ' that was packed in the operator closure $\langle\langle x, c', \gamma' \rangle\rangle$ that should be just below the top of the stack. Then execute the code c' of the operator closure, starting with an empty stack. Save the calling context (S, γ, c'') by pushing it onto the dump.

$(a::\langle\langle\langle x, c', \gamma' \rangle\rangle\rangle::S, \gamma, \texttt{APP} :: c'', D) ==> ([\,], \gamma' [x \mapsto a], c', (S, \gamma, c'') :: D)$

- Return from the function call by restoring — and popping — the calling context (S, γ, c'') from the dump, placing the answer a at the top of the restored stack S, and resuming executing the code c'' with the restored table γ. Note that we discard the context components S', c' and γ'' of the called function since they are no longer needed.

$(a::S', \gamma'', \texttt{RET} :: c', (S, \gamma, c'') :: D) ==> (a::S, \gamma, c'', D)$

Programming Exercise. Implement the SECD machine in OCaml — taking the (reflexive) transitive closure of the one-step transitions.

Exercise (**). Formulate a theorem that states that if a (closed well-typed) expression e evaluates to an answer a, then by compiling e into opcodes and running the compiled code on the SECD machine will result in a state where something analogous to a is on the top of the stack.

Exercise (***) Prove the theorem that you formulate. On what will you perform induction? Why? What generalisations on stack, environment, code and dump parameters will you need to make for the induction hypotheses to be applicable?