Implementing functional languages with abstract machines

Hayo Thielecke
University of Birmingham
http://www.cs.bham.ac.uk/~hxt

November 15, 2017

Contents

Structure of the module

Parsing ✓

- Progression from: Language + Logic, Models of Computation
- abstract machines, formal, "mathy"

Compiling C with Clang ✓

- Progression from: Computer Systems + Architecture, C/C++
- not so formal, by example, x86 machine code

Implementing functional languages

- Progression from: functional programming
- builds on abstract machines and C stack
- the most research-led section, goes beyond Dragon Book

Language design

- There are trade-offs in programming language design and implementation.
- If you know both C and functional languages, you understand much then just knowing one of them.
- They are at opposite ends of spectrum of language design.

First-class functions and language design trade-offs

Can a function be:

1. passed as a parameter to another function?
2. returned as the result of another function?
3. defined inside another function and thereby have access to its variables?

- In C, you can do 1 and 2, using function pointers, but not 3.
- In Pascal, you can do 1 and 3 but not 2. Needs a “static link” in the stack frame.
- With C++ lambda expressions, you can do all 3, but can get undefined behaviour from returning functions and by-reference capture
- In functional languages, you can do all 3 without restrictions. More powerful, less efficient, as it needs a garbage collector

Big ideas in functional language implementation

- Implementing languages like OCaml (or F♯, or Scheme, or Clojure, . . .) amounts to implementing call-by-value lambda calculus
- There are many abstract machines for functional languages
- They share some critical ingredients: call stack and closures
- There are similarities to the way C is implemented using stack and frame pointer

Lambda calculus and programming language (pre-)history

- 1879: Frege's Begriffsschrift clarified variables and functions
- 1941s: Church invents lambda calculus
- Church and Turing invent fixpoint combinators
- Haskell Curry compiles lambda calculus into S and K combinators
- Lisp implements lambda, but without closures (dynamic binding)
- Landin notices that Algol-like languages can be desugared into lambda calculus
- 1964: Landin's SECD machine includes closures
- 1980s–: modern functional languages, CEK machine, CPS transform, ...
- Now (2017): many different compilers for ML and Haskell; C++ has lambda expressions that build closures

Lambda calculus syntax

$$
\begin{array}{rcl}
M & ::= & x \\
 & \mid & M_1\, M_2 \\
 & \mid & \lambda x.M \\
 & \mid & n
\end{array}
$$

Here n ranges over constants such as integers.
Note: this is an ambiguous grammar, so no good for parsing.
Convention: left associative.

$$M_1\, M_2\, M_3$$

is read as

$$(M_1\, M_2)\, M_3$$

Lambda abstraction extends as far to the right as possible:
$\lambda x.x\, y$ is read as $\lambda x.(x\, y)$ and not $(\lambda x.x)\, y$

Lambda in programming languages

Most modern languages have lambda expressions
In Lisp/Scheme/Clojure: `lambda`
In C++14: `[=]`
In OCaml: λ is written as `function`
Example: $\lambda x.x$ in OCaml:

```
# function x -> x;;
- : 'a -> 'a = <fun>

# (function x -> x) 42;;
- : int = 42
```

Free variables of a lambda term

- In an abstraction
$$\lambda x.M$$
the scope of x is M.
- A variable is free in a lambda term if there is an occurrence of that variable that is not in scope of a lambda binding for that variable.
- A lambda term that contains no free variables is called closed
- Note: variables in lambda $\neq$ variables in most programming languages. The former cannot be updated.

Exercise on free variables

What are the free variables in:

1. $z\,(x\,y)$

Exercise on free variables

What are the free variables in:

1. $z\,(x\,y)$
2. $\lambda x.x$

Exercise on free variables

What are the free variables in:

1. $z\,(x\,y)$
2. $\lambda x.x$
3. $\lambda x.y\,x$

Exercise on free variables

What are the free variables in:

1. $z\,(x\,y)$
2. $\lambda x.x$
3. $\lambda x.y\,x$
4. $\lambda y.\lambda x.y\,x$

Exercise on free variables

What are the free variables in:

1. $z\,(x\,y)$
2. $\lambda x.x$
3. $\lambda x.y\,x$
4. $\lambda y.\lambda x.y\,x$
5. $(\lambda y.\lambda x.y\,x)\,x$

Exercise on free variables

What are the free variables in:

1. $z\,(x\,y)$
2. $\lambda x.x$
3. $\lambda x.y\,x$
4. $\lambda y.\lambda x.y\,x$
5. $(\lambda y.\lambda x.y\,x)\,x$
6. $(\lambda x.x)\,(\lambda y.x)\;y$

Exercise: for each occurrence of a variable, draw an arrow to the corresponding lambda binding, if there is one.

Substitution in lambda calculus

We write

$$M_1[x \mapsto M_2]$$

for the substitution in M_1 of x by M_2
Here we assume that the bound variables in M_1 do not occur in M_2, which can always be made to hold by renaming them
Example: $M_1 = \lambda y.xx$ and $M_2 = \lambda z.z$. Then

$$(\lambda y.xx)[x \mapsto \lambda z.z] = \lambda y.((\lambda z.z)\,(\lambda z.z))$$

Some authors write substitution differently, e.g.

$$M_1[M_2/x] \text{ or } M_1[x := M_2]$$

Exercise on substitution

What is

$$(x\,x)[x \mapsto y]$$

$$(x\,x)[y \mapsto x]$$

$$(z\,y)[y \mapsto \lambda a.a]$$

Beta reduction in lambda calculus

β-reduction:

$$((\lambda x.M_1)\,M_2) \to_\beta (M_1[x \mapsto M_2])$$

Example:

$$(\lambda f.f\,5)\,(\lambda x.x)$$

Beta reduction in lambda calculus

β-reduction:

$$((\lambda x.\, M_1)\, M_2) \to_\beta (M_1[x \mapsto M_2])$$

Example:

$$\begin{array}{ll} & (\lambda f.\, f\, 5)\, (\lambda x.x) \\ \to_\beta & (\lambda x.x)\, 5 \end{array}$$

Beta reduction in lambda calculus

β-reduction:

$$((\lambda x . M_1)\, M_2) \rightarrow_\beta (M_1[x \mapsto M_2])$$

Example:

$$\begin{array}{ll} & (\lambda f . f\, 5)\, (\lambda x . x) \\ \rightarrow_\beta & (\lambda x . x)\, 5 \\ \rightarrow_\beta & 5 \end{array}$$

Beta reduction in lambda calculus

β-reduction:

$$((\lambda x.M_1)\,M_2) \rightarrow_\beta (M_1[x \mapsto M_2])$$

Example:

$$\begin{aligned} & (\lambda f.f\,5)\,(\lambda x.x) \\ \rightarrow_\beta \quad & (\lambda x.x)\,5 \\ \rightarrow_\beta \quad & 5 \end{aligned}$$

Exercise: beta reduce this to 2:

$$(\lambda z.(\lambda p.p\,(p\,z))\,(\lambda y.y))\,2$$

Beta reduction and compiler terminology

β-reduction:

$$((\lambda x.\, M_1)\, M_2) \rightarrow_\beta (M_1[x \mapsto M_2])$$

This is a simple model of function calls:

- $(\lambda x.\, M_1)$ called function, callee
- M_1 function body
- x formal parameter
- M_2 actual parameter

Beta reduction says: compute function calls by replacing the formal parameter by the actual parameter in the body of the called function

Lambda calculus has first-class functions

Can a function be:

1. passed as a parameter to another function? Yes, e.g.: $(\lambda x . M_1)\,(\lambda y . M_2)$
2. returned as the result of another function? Yes, e.g.: $\lambda y . (\lambda x . M_1)$
3. defined inside another function and thereby have access to its variables? Yes, e.g.: $\lambda y . \lambda x . y$

Encoding let as lambda

A local let binding

$$\texttt{let}\ x = M_1\ \texttt{in}\ M_2$$

is encoded as

$$(\lambda x.M_2)\ M_1$$

Idea: evaluate M_1, bind the result to x and then evaluate M_2.

Programs idealized as lambda terms

Typical C-style program

```
f1(T1 x1) { return E1; }
...
fn(Tn xn) { return En; }

main() { return E;}
```

Idealized in lambda calculus

$\texttt{let}\ f_1 = \lambda x_1.E_1\ \texttt{in}$
$\cdots$
$\texttt{let}\ f_n = \lambda x_n.E_n\ \texttt{in}$
E

Exercise: explain how function linlining on the left corresponds to two beta reduction on the right.

Infinite loops

$$(\lambda y.42)((\lambda x.xx)(\lambda x.xx))$$

The same term may produce a constant or loop forever, depending where we choose to perform the beta reduction.

Encoding of pairs

Data structures such as pairs, lists and trees can be encoded in the lambda calculus. The idea is a kind of protocol between a server that holds data and clients that need the data.

$$\begin{aligned} \langle M_1, M_2 \rangle &= \lambda v . v\, M_1\, M_2 \\ \texttt{fst} &= \lambda p . p\,(\lambda x . \lambda y . x) \\ \texttt{snd} &= \lambda p . p\,(\lambda x . \lambda y . y) \end{aligned}$$

Exercise: Prove that

$$\begin{aligned} \texttt{fst}\,\langle M_1, M_2 \rangle &\to_\beta M_1 \\ \texttt{snd}\,\langle M_1, M_2 \rangle &\to_\beta M_2 \end{aligned}$$

Beta reduction is nondeterministic ☹

Beta reduction is nondeterministic, as there can be more than one redex in a term.

A programming language needs to fix an evaluation order; compare making parsing machines deterministic.

Consider the following term with multiple β-redexes R_1, R_2, and R_3:

$$\overbrace{(\lambda x_1 . (\underbrace{(\lambda x_2 . M_2)\, M_3}_{R_2}))\ (\underbrace{(\lambda x_4 . M_4)\, M_5}_{R_3})}^{R_1}$$

Under call-by-name, R_1 is the next redex to be reduced.

Under call-by-value, it is R_3.

R_2 is under a lambda. Reducing R_2 correspond to function inlining rather than function call.

Beta reduction is inefficient ☹

Beta reduction by replacing variables with values is fine for pencil and paper calculation, as originally intended in the lambda calculus.
As we have seen in C, the compiled code for a function should stay fixed.
Different calls have different values in stack frames and registers.
Consider a term

$$\lambda x . y\, x$$

Here x is our formal parameter. We can expect to be passed some value for it using some calling convention (stack or register, say).
But what about y? We need some way of knowing what its value is.
In C, we have seen how to find values in memory (and in particular using the frame pointer).
What should a machine for lambda calculus do?

Overview of lambda calculus so far

Conclusion so far: lambda calculus is a good idealization of functional programming languages
But it is not a practical language
Compare:
grammars $\to$ nondeterministic machine $\to$ deterministic parser (e.g. ANTLR, Menhir)
lambda calculus $\to$ abstract machine $\to$ ML compiler

Abstract machines for functional languages

- History: Peter Landin's 1964 paper "The mechanical evaluation of expressions"
- SECD machine [Lan64]
- Caml originally came from the Categorical Abstract Machine (CAM)
- Caml compiler based on ZINC machine, itself inspired by the Krivine Abstract Machine
- CEK machine is a cleaner version of the SECD machine by Friedman and Felleisen in the 1980s
- S and D are fused into K
- The CEK machine corresponds to CPS compiling
- CPS is related to the SSA form used in LLVM

CEK machine name

The name CEK is due to the fact that the machine has three components:

- C stands for control or code. Intuitively, it is the expression that the machine is currently trying to evaluate.
- E stand for environment. It gives bindings for the free variables in C.
- K stands for continuation (given that the letter C is already used up). It tells the machine what to do when it is finished the with the current C.

CEK machine is an interpreter

- The CEK machine is strictly speaking not a compiler but an interpreter
- The original program, represented as a lambda term, is manipulated at run tim.
- A real implementation would translate the lambda term into machine code which the simulates the CEK machine
- Nonetheless, the key ideas are stack and closures

CEK machine runs lambda terms as programes

The language that the basic CEK machine can interpret is λ calculus with constants n:

$$\begin{array}{rcl} M & ::= & x \\ & \mid & M_1\, M_2 \\ & \mid & \lambda x . M \\ & \mid & n \end{array}$$

More constructs could be added, e.g., exceptions or assignments.

How to compile lambdas

A closed lambda term can be implemented as a code pointer, as in C.
If a lambda term is not closed, the implementation "makes it closed", so to speak, by contructing a closure.

Environments

An environment E is a list of the form

$$x_1 \mapsto W_1, \ldots, x_n \mapsto W_n$$

that associates the value W_j to variable x_j.
We write $\emptyset$ for the special case when the list is empty.
An environent E can be updated with a new binding, giving x the value W, which we write as

$$E[x \mapsto W]$$

We write

$$\texttt{lookup}\,x\,\texttt{in}\,E$$

For W_i if $x = x_i$

Closures

A closure is of the form

$$\texttt{clos}(\lambda x.M, E)$$

Here x is called the parameter, M the body, and E the environment of the closure.
A value W can be a constant n or a closure:

$$\begin{array}{rcl} W & ::= & n \\ & \mid & \texttt{clos}(\lambda x.M, E) \end{array}$$

Objects and closures

In C, a function body has access to parameters (and also global variables).
With objects, a function has access to members of the surrounding object.
With closures, a function has access to variables in surrounding functions.
In both objects and closure, there are additional pointers to data.

CEK frames and stacks

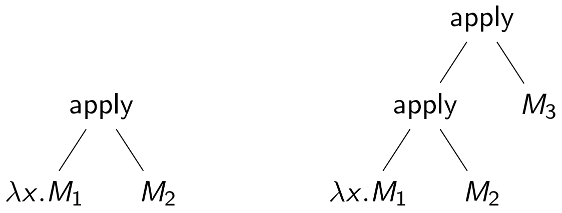
A frame is of the form

$$\begin{array}{rcl} F & ::= & (W \bigcirc) \\ & \mid & (\bigcirc\ M\ E) \end{array}$$

A continuation K is a stack of frames.
If F is a frame and K a continuation, we write $F\,,K$ for the continuation which we get by pushing F onto the top of K.
The empty stack is written as $\blacksquare$.

Evaluation and tree traversal

In a function application, the function and parameter may themselves be complex terms that need to be evaluated.



Compare: tree traversal using a stack

Function call (application) in the CEK machine

Evaluating an application

$$M_1\, M_2$$

is done in three stages:

1. evaluate M_1, while remembering M_2 and the current environment in frame $(\bigcirc\ M_2\ E)$
2. evaluate M_2, while remembering the value W_1 we got from M_1 in the frame $(W_1 \bigcirc)$
3. When M_2 has been evaluated to some W_2, pop the frame $(W_1 \bigcirc)$ and apply W_1 to W_2. This only works if W_1 is a closure, say

   $$W_1 = \mathtt{clos}(\lambda x.M, E)$$

   If so, x gets bound to W_2.

This may be a little subtle to understand due to the switching between M_1 and M_2.

LEVEL UP

CEK machine steps

$$\begin{array}{rcl}
\langle\, x \;\mid E \mid K\,\rangle & \longrightarrow & \langle\, \texttt{lookup}\, x\, \texttt{in}\, E \;\mid E \mid K\,\rangle \\
\langle\, M_1\, M_2 \;\mid E \mid K\,\rangle & \longrightarrow & \langle\, M_1 \;\mid E \mid (\bigcirc\; M_2\; E)\,,\, K\,\rangle \\
\langle\, \lambda x.M \;\mid E \mid K\,\rangle & \longrightarrow & \langle\, \texttt{clos}(\lambda x.M, E) \;\mid E \mid K\,\rangle \\
\\
\langle\, W \;\mid E_1 \mid (\bigcirc\; M\; E_2)\,,\, K\,\rangle & \longrightarrow & \langle\, M \;\mid E_2 \mid (W\, \bigcirc)\,,\, K\,\rangle \\
\langle\, W \;\mid E_1 \mid (\texttt{clos}(\lambda x.M, E_2)\, \bigcirc)\,,\, K\,\rangle & \longrightarrow & \langle\, M \;\mid E_2[x \mapsto W] \mid K\,\rangle
\end{array}$$

CEK machine steps with pattern matching highlighted

$$
\begin{array}{rcl}
\langle\, x \mid E \mid K \,\rangle & \longrightarrow & \langle\, \mathtt{lookup}\; x \;\mathtt{in}\; E \mid E \mid K \,\rangle \\
\langle\, M_1\, M_2 \mid E \mid K \,\rangle & \longrightarrow & \langle\, M_1 \mid E \mid (\bigcirc\; M_2\; E)\,,\, K \,\rangle \\
\langle\, \lambda x.M \mid E \mid K \,\rangle & \longrightarrow & \langle\, \mathtt{clos}(\lambda x.M, E) \mid E \mid K \,\rangle \\
\langle\, W \mid E_1 \mid (\bigcirc\; M\; E_2)\,,\, K \,\rangle & \longrightarrow & \langle\, M \mid E_2 \mid (W\, \bigcirc)\,,\, K \,\rangle \\
\langle\, W \mid E_1 \mid (\mathtt{clos}(\lambda x.M, E_2)\, \bigcirc)\,,\, K \,\rangle & \longrightarrow & \langle\, M \mid E_2[x \mapsto W] \mid K \,\rangle
\end{array}
$$

CEK steps explained 1

1. If the current code is a variable x, we must look it up in the environment E.
2. If the current code is an application $M_1 M_2$, then we proceed to evaluate the M_1 in the operator position. M_2 is pushed onto the continuation stack. Note that M_2 may contain free variables. For looking them up later, we push the current environment E along with M_2.
3. If the current code is a lambda abstraction, we package it up with the current environment to form a closure. That closure is the value of the expression.

CEK steps explained 2

4. If the hole in the top frame is on the left, we pop of the term M and its associated environment E, and proceed to evaluate M relative to E. We also push on a new frame that tells us we still have to apply W to the result we hope to get from the evalutation of M.
5. If the hole in the top frame is on the right, we pop off the frame, which should contain a closure. We apply the closure by evaluating its body in the environment extended with the value W for the argument of the closure. If there is no closure in the frame, then the machine gets stuck and there is no transition step. That situation arises if we get a run-time type error from ill-typed code such as applying a constant to something, as in $7\,(\lambda x.x)$.

CEK evaluation

We say that a term M evaluates to a value W if there is a sequence of steps:

$$\langle\, M \;\mid \emptyset \mid \blacksquare \,\rangle \xrightarrow{*} \langle\, W \;\mid E \mid \blacksquare \,\rangle$$

The end state must have a value in the C position and an empty stack/final continuation $K = \blacksquare$.
The environment need not be empty.
Note: the machine is done when its stack is empty, like the LL machine.
The CEK machine is deterministic.
The CEK machine may get stuck.

Example: function as result and function inside another

$((\lambda x.\lambda y.x)\,1)\,2$ evaluates to 1

$$\langle ((\lambda x.\lambda y.x)\,1)\,2 \;\mid \emptyset \mid \blacksquare \rangle$$

Example: function as result and function inside another

$((\lambda x.\lambda y.x)\,1)\,2$ evaluates to 1

$$\begin{array}{rl} & \langle\, ((\lambda x.\lambda y.x)\,1)\,2 \;\mid \emptyset \mid \blacksquare \,\rangle \\ \longrightarrow & \langle\, (\lambda x.\lambda y.x)\,1 \;\mid \emptyset \mid (\bigcirc\; 2\; \emptyset) \,\rangle \end{array}$$

Example: function as result and function inside another

$((\lambda x.\lambda y.x)\,1)\,2$ evaluates to 1

$$
\begin{array}{rl}
 & \langle\, ((\lambda x.\lambda y.x)\,1)\,2 \;\mid \emptyset \mid \blacksquare \,\rangle \\
\longrightarrow & \langle\, (\lambda x.\lambda y.x)\,1 \;\mid \emptyset \mid (\bigcirc\; 2\; \emptyset) \,\rangle \\
\longrightarrow & \langle\, (\lambda x.\lambda y.x) \;\mid \emptyset \mid (\bigcirc\; 1\; \emptyset)\,,(\bigcirc\; 2\; \emptyset) \,\rangle
\end{array}
$$

Example: function as result and function inside another

$((\lambda x.\lambda y.x)\,1)\,2$ evaluates to 1

$$
\begin{array}{rl}
 & \langle\, ((\lambda x.\lambda y.x)\,1)\,2 \;\mid \emptyset \mid \blacksquare \,\rangle \\
\longrightarrow & \langle\, (\lambda x.\lambda y.x)\,1 \;\mid \emptyset \mid (\bigcirc\ 2\ \emptyset) \,\rangle \\
\longrightarrow & \langle\, (\lambda x.\lambda y.x) \;\mid \emptyset \mid (\bigcirc\ 1\ \emptyset)\,,(\bigcirc\ 2\ \emptyset) \,\rangle \\
\longrightarrow & \langle\, \mathtt{clos}((\lambda x.\lambda y.x), \emptyset) \;\mid \emptyset \mid (\bigcirc\ 1\ \emptyset)\,,(\bigcirc\ 2\ \emptyset) \,\rangle
\end{array}
$$

Example: function as result and function inside another

$((\lambda x.\lambda y.x)\,1)\,2$ evaluates to 1

$$\begin{array}{rl}
 & \langle\, ((\lambda x.\lambda y.x)\,1)\,2 \;\mid \emptyset \mid \blacksquare \,\rangle \\
\longrightarrow & \langle\, (\lambda x.\lambda y.x)\,1 \;\mid \emptyset \mid (\bigcirc\ 2\ \emptyset) \,\rangle \\
\longrightarrow & \langle\, (\lambda x.\lambda y.x) \;\mid \emptyset \mid (\bigcirc\ 1\ \emptyset)\ ,(\bigcirc\ 2\ \emptyset) \,\rangle \\
\longrightarrow & \langle\, \mathtt{clos}((\lambda x.\lambda y.x),\emptyset) \;\mid \emptyset \mid (\bigcirc\ 1\ \emptyset)\ ,(\bigcirc\ 2\ \emptyset) \,\rangle \\
\longrightarrow & \langle\, 1 \;\mid \emptyset \mid (\mathtt{clos}((\lambda x.\lambda y.x),\emptyset)\,\bigcirc)\ ,(\bigcirc\ 2\ \emptyset) \,\rangle
\end{array}$$

Example: function as result and function inside another

$((\lambda x.\lambda y.x)\,1)\,2$ evaluates to 1

$$
\begin{array}{rl}
 & \langle\, ((\lambda x.\lambda y.x)\,1)\,2 \;\mid \emptyset \mid \blacksquare \,\rangle \\
\longrightarrow & \langle\, (\lambda x.\lambda y.x)\,1 \;\mid \emptyset \mid (\bigcirc\; 2\; \emptyset) \,\rangle \\
\longrightarrow & \langle\, (\lambda x.\lambda y.x) \;\mid \emptyset \mid (\bigcirc\; 1\; \emptyset)\,,(\bigcirc\; 2\; \emptyset) \,\rangle \\
\longrightarrow & \langle\, \texttt{clos}((\lambda x.\lambda y.x),\emptyset) \;\mid \emptyset \mid (\bigcirc\; 1\; \emptyset)\,,(\bigcirc\; 2\; \emptyset) \,\rangle \\
\longrightarrow & \langle\, 1 \;\mid \emptyset \mid (\texttt{clos}((\lambda x.\lambda y.x),\emptyset)\,\bigcirc)\,,(\bigcirc\; 2\; \emptyset) \,\rangle \\
\longrightarrow & \langle\, \lambda y.x \;\mid x \mapsto 1 \mid (\bigcirc\; 2\; \emptyset) \,\rangle
\end{array}
$$

Example: function as result and function inside another

$((\lambda x.\lambda y.x)\,1)\,2$ evaluates to 1

$$
\begin{array}{ll}
 & \langle\, ((\lambda x.\lambda y.x)\,1)\,2 \;\mid \emptyset \mid \blacksquare \,\rangle \\
\longrightarrow & \langle\, (\lambda x.\lambda y.x)\,1 \;\mid \emptyset \mid (\bigcirc\; 2\; \emptyset) \,\rangle \\
\longrightarrow & \langle\, (\lambda x.\lambda y.x) \;\mid \emptyset \mid (\bigcirc\; 1\; \emptyset)\,,(\bigcirc\; 2\; \emptyset) \,\rangle \\
\longrightarrow & \langle\, \mathtt{clos}((\lambda x.\lambda y.x), \emptyset) \;\mid \emptyset \mid (\bigcirc\; 1\; \emptyset)\,,(\bigcirc\; 2\; \emptyset) \,\rangle \\
\longrightarrow & \langle\, 1 \;\mid \emptyset \mid (\mathtt{clos}((\lambda x.\lambda y.x), \emptyset)\,\bigcirc)\,,(\bigcirc\; 2\; \emptyset) \,\rangle \\
\longrightarrow & \langle\, \lambda y.x \;\mid x \mapsto 1 \mid (\bigcirc\; 2\; \emptyset) \,\rangle \\
\longrightarrow & \langle\, \mathtt{clos}((\lambda y.x), x \mapsto 1) \;\mid x \mapsto 1 \mid (\bigcirc\; 2\; \emptyset) \,\rangle
\end{array}
$$

Example: function as result and function inside another

$((\lambda x.\lambda y.x)\,1)\,2$ evaluates to 1

$$
\begin{array}{rl}
 & \langle\, ((\lambda x.\lambda y.x)\,1)\,2 \;\mid \emptyset \mid \blacksquare \,\rangle \\
\longrightarrow & \langle\, (\lambda x.\lambda y.x)\,1 \;\mid \emptyset \mid (\bigcirc\; 2\; \emptyset) \,\rangle \\
\longrightarrow & \langle\, (\lambda x.\lambda y.x) \;\mid \emptyset \mid (\bigcirc\; 1\; \emptyset)\,,(\bigcirc\; 2\; \emptyset) \,\rangle \\
\longrightarrow & \langle\, \texttt{clos}((\lambda x.\lambda y.x),\emptyset) \;\mid \emptyset \mid (\bigcirc\; 1\; \emptyset)\,,(\bigcirc\; 2\; \emptyset) \,\rangle \\
\longrightarrow & \langle\, 1 \;\mid \emptyset \mid (\texttt{clos}((\lambda x.\lambda y.x),\emptyset)\,\bigcirc)\,,(\bigcirc\; 2\; \emptyset) \,\rangle \\
\longrightarrow & \langle\, \lambda y.x \;\mid x \mapsto 1 \mid (\bigcirc\; 2\; \emptyset) \,\rangle \\
\longrightarrow & \langle\, \texttt{clos}((\lambda y.x), x \mapsto 1) \;\mid x \mapsto 1 \mid (\bigcirc\; 2\; \emptyset) \,\rangle \\
\longrightarrow & \langle\, 2 \;\mid \emptyset \mid (\texttt{clos}(\lambda y.x, x \mapsto 1)\,\bigcirc) \,\rangle
\end{array}
$$

Example: function as result and function inside another

$((\lambda x.\lambda y.x)\,1)\,2$ evaluates to 1

$$
\begin{array}{ll}
 & \langle\, ((\lambda x.\lambda y.x)\,1)\,2 \;\mid \emptyset \mid \blacksquare \,\rangle \\
\longrightarrow & \langle\, (\lambda x.\lambda y.x)\,1 \;\mid \emptyset \mid (\bigcirc\; 2\; \emptyset) \,\rangle \\
\longrightarrow & \langle\, (\lambda x.\lambda y.x) \;\mid \emptyset \mid (\bigcirc\; 1\; \emptyset)\,,(\bigcirc\; 2\; \emptyset) \,\rangle \\
\longrightarrow & \langle\, \mathtt{clos}((\lambda x.\lambda y.x), \emptyset) \;\mid \emptyset \mid (\bigcirc\; 1\; \emptyset)\,,(\bigcirc\; 2\; \emptyset) \,\rangle \\
\longrightarrow & \langle\, 1 \;\mid \emptyset \mid (\mathtt{clos}((\lambda x.\lambda y.x), \emptyset)\, \bigcirc)\,,(\bigcirc\; 2\; \emptyset) \,\rangle \\
\longrightarrow & \langle\, \lambda y.x \;\mid x \mapsto 1 \mid (\bigcirc\; 2\; \emptyset) \,\rangle \\
\longrightarrow & \langle\, \mathtt{clos}((\lambda y.x), x \mapsto 1) \;\mid x \mapsto 1 \mid (\bigcirc\; 2\; \emptyset) \,\rangle \\
\longrightarrow & \langle\, 2 \;\mid \emptyset \mid (\mathtt{clos}(\lambda y.x, x \mapsto 1)\, \bigcirc) \,\rangle \\
\longrightarrow & \langle\, x \;\mid x \mapsto 1, y \mapsto 2 \mid \blacksquare \,\rangle
\end{array}
$$

Example: function as result and function inside another

$((\lambda x.\lambda y.x)\,1)\,2$ evaluates to 1

$$\begin{array}{ll}
 & \langle\, ((\lambda x.\lambda y.x)\,1)\,2 \;\mid \emptyset \mid \blacksquare \,\rangle \\
\longrightarrow & \langle\, (\lambda x.\lambda y.x)\,1 \;\mid \emptyset \mid (\bigcirc\ 2\ \emptyset) \,\rangle \\
\longrightarrow & \langle\, (\lambda x.\lambda y.x) \;\mid \emptyset \mid (\bigcirc\ 1\ \emptyset)\,,(\bigcirc\ 2\ \emptyset) \,\rangle \\
\longrightarrow & \langle\, \mathtt{clos}((\lambda x.\lambda y.x), \emptyset) \;\mid \emptyset \mid (\bigcirc\ 1\ \emptyset)\,,(\bigcirc\ 2\ \emptyset) \,\rangle \\
\longrightarrow & \langle\, 1 \;\mid \emptyset \mid (\mathtt{clos}((\lambda x.\lambda y.x), \emptyset)\ \bigcirc)\,,(\bigcirc\ 2\ \emptyset) \,\rangle \\
\longrightarrow & \langle\, \lambda y.x \;\mid x \mapsto 1 \mid (\bigcirc\ 2\ \emptyset) \,\rangle \\
\longrightarrow & \langle\, \mathtt{clos}((\lambda y.x), x \mapsto 1) \;\mid x \mapsto 1 \mid (\bigcirc\ 2\ \emptyset) \,\rangle \\
\longrightarrow & \langle\, 2 \;\mid \emptyset \mid (\mathtt{clos}(\lambda y.x, x \mapsto 1)\ \bigcirc) \,\rangle \\
\longrightarrow & \langle\, x \;\mid x \mapsto 1, y \mapsto 2 \mid \blacksquare \,\rangle \\
\longrightarrow & \langle\, 1 \;\mid x \mapsto 1, y \mapsto 2 \mid \blacksquare \,\rangle\ ☺
\end{array}$$

Example 2: function as parameter

$(\lambda f . f\,2)\,(\lambda x . x)$ evaluates to 2

$$\langle\, (\lambda f . f\,2)\,(\lambda x . x) \;\mid \emptyset \mid \blacksquare \,\rangle$$

Example 2: function as parameter

$(\lambda f . f\,2)\,(\lambda x . x)$ evaluates to 2

$$\begin{array}{rl} & \langle (\lambda f . f\,2)\,(\lambda x . x) \;\mid \emptyset \mid \blacksquare \rangle \\ \longrightarrow & \langle (\lambda f . f\,2) \;\mid \emptyset \mid (\bigcirc\;(\lambda x . x)\;\emptyset)\,,\blacksquare \rangle \end{array}$$

Example 2: function as parameter

$(\lambda f.f\,2)\,(\lambda x.x)$ evaluates to 2

$$\begin{array}{ll}
 & \langle\,(\lambda f.f\,2)\,(\lambda x.x)\ \mid \emptyset \mid \blacksquare\rangle \\
\longrightarrow & \langle\,(\lambda f.f\,2)\ \mid \emptyset \mid (\bigcirc\ (\lambda x.x)\ \emptyset)\ , \blacksquare\rangle \\
\longrightarrow & \langle\,\texttt{clos}((\lambda f.f\,2), \emptyset)\ \mid \emptyset \mid (\bigcirc\ (\lambda x.x)\ \emptyset)\ , \blacksquare\rangle
\end{array}$$

Example 2: function as parameter

$(\lambda f.f\,2)\,(\lambda x.x)$ evaluates to 2

$$\begin{array}{rl}
 & \langle\, (\lambda f.f\,2)\,(\lambda x.x) \;\mid \emptyset \mid \blacksquare \rangle \\
\longrightarrow & \langle\, (\lambda f.f\,2) \;\mid \emptyset \mid (\bigcirc\;(\lambda x.x)\;\emptyset)\,,\blacksquare \rangle \\
\longrightarrow & \langle\, \mathtt{clos}((\lambda f.f\,2),\emptyset) \;\mid \emptyset \mid (\bigcirc\;(\lambda x.x)\;\emptyset)\,,\blacksquare \rangle \\
\longrightarrow & \langle\, \lambda x.x \;\mid \emptyset \mid (\mathtt{clos}((\lambda f.f\,2),\emptyset)\,\bigcirc)\,,\blacksquare \rangle
\end{array}$$

Example 2: function as parameter

$(\lambda f.f\,2)\,(\lambda x.x)$ evaluates to 2

$$\begin{array}{rl}
& \langle (\lambda f.f\,2)\,(\lambda x.x) \mid \emptyset \mid \blacksquare \rangle \\
\longrightarrow & \langle (\lambda f.f\,2) \mid \emptyset \mid (\bigcirc\ (\lambda x.x)\ \emptyset)\ ,\blacksquare \rangle \\
\longrightarrow & \langle \texttt{clos}((\lambda f.f\,2),\emptyset) \mid \emptyset \mid (\bigcirc\ (\lambda x.x)\ \emptyset)\ ,\blacksquare \rangle \\
\longrightarrow & \langle \lambda x.x \mid \emptyset \mid (\texttt{clos}((\lambda f.f\,2),\emptyset)\ \bigcirc)\ ,\blacksquare \rangle \\
\longrightarrow & \langle \texttt{clos}(\lambda x.x,\emptyset) \mid \emptyset \mid (\texttt{clos}((\lambda f.f\,2),\emptyset)\ \bigcirc)\ ,\blacksquare \rangle
\end{array}$$

Example 2: function as parameter

$(\lambda f.f\,2)\,(\lambda x.x)$ evaluates to 2

$$\begin{array}{ll}
 & \langle (\lambda f.f\,2)\,(\lambda x.x) \;\mid \emptyset \mid \blacksquare \rangle \\
\longrightarrow & \langle (\lambda f.f\,2) \;\mid \emptyset \mid (\bigcirc\;(\lambda x.x)\;\emptyset)\,,\blacksquare \rangle \\
\longrightarrow & \langle \mathtt{clos}((\lambda f.f\,2), \emptyset) \;\mid \emptyset \mid (\bigcirc\;(\lambda x.x)\;\emptyset)\,,\blacksquare \rangle \\
\longrightarrow & \langle \lambda x.x \;\mid \emptyset \mid (\mathtt{clos}((\lambda f.f\,2), \emptyset)\,\bigcirc)\,,\blacksquare \rangle \\
\longrightarrow & \langle \mathtt{clos}(\lambda x.x, \emptyset) \;\mid \emptyset \mid (\mathtt{clos}((\lambda f.f\,2), \emptyset)\,\bigcirc)\,,\blacksquare \rangle \\
\longrightarrow & \langle f\,2 \;\mid f \mapsto \mathtt{clos}(\lambda x.x, \emptyset) \mid \blacksquare \rangle
\end{array}$$

Example 2: function as parameter

$(\lambda f . f\, 2)\,(\lambda x . x)$ evaluates to 2

$$\begin{array}{rl}
& \langle\, (\lambda f . f\, 2)\,(\lambda x . x) \;\mid \emptyset \mid \blacksquare \rangle \\
\longrightarrow & \langle\, (\lambda f . f\, 2) \;\mid \emptyset \mid (\bigcirc\; (\lambda x . x)\; \emptyset)\,,\, \blacksquare \rangle \\
\longrightarrow & \langle\, \mathtt{clos}((\lambda f . f\, 2), \emptyset) \;\mid \emptyset \mid (\bigcirc\; (\lambda x . x)\; \emptyset)\,,\, \blacksquare \rangle \\
\longrightarrow & \langle\, \lambda x . x \;\mid \emptyset \mid (\mathtt{clos}((\lambda f . f\, 2), \emptyset)\, \bigcirc)\,,\, \blacksquare \rangle \\
\longrightarrow & \langle\, \mathtt{clos}(\lambda x . x, \emptyset) \;\mid \emptyset \mid (\mathtt{clos}((\lambda f . f\, 2), \emptyset)\, \bigcirc)\,,\, \blacksquare \rangle \\
\longrightarrow & \langle\, f\, 2 \;\mid f \mapsto \mathtt{clos}(\lambda x . x, \emptyset) \mid \blacksquare \rangle \\
\longrightarrow & \langle\, f \;\mid f \mapsto \mathtt{clos}(\lambda x . x, \emptyset) \mid (\bigcirc\; 2\; f \mapsto \mathtt{clos}(\lambda x . x, \emptyset))\,,\, \blacksquare \rangle
\end{array}$$

Example 2: function as parameter

$(\lambda f.f\,2)\,(\lambda x.x)$ evaluates to 2

$$
\begin{array}{ll}
 & \langle\, (\lambda f.f\,2)\,(\lambda x.x) \;\mid \emptyset \mid \blacksquare \rangle \\
\longrightarrow & \langle\, (\lambda f.f\,2) \;\mid \emptyset \mid (\bigcirc\ (\lambda x.x)\ \emptyset)\ , \blacksquare \rangle \\
\longrightarrow & \langle\, \texttt{clos}((\lambda f.f\,2), \emptyset) \;\mid \emptyset \mid (\bigcirc\ (\lambda x.x)\ \emptyset)\ , \blacksquare \rangle \\
\longrightarrow & \langle\, \lambda x.x \;\mid \emptyset \mid (\texttt{clos}((\lambda f.f\,2), \emptyset)\,\bigcirc)\ , \blacksquare \rangle \\
\longrightarrow & \langle\, \texttt{clos}(\lambda x.x, \emptyset) \;\mid \emptyset \mid (\texttt{clos}((\lambda f.f\,2), \emptyset)\,\bigcirc)\ , \blacksquare \rangle \\
\longrightarrow & \langle\, f\,2 \;\mid f \mapsto \texttt{clos}(\lambda x.x, \emptyset) \mid \blacksquare \rangle \\
\longrightarrow & \langle\, f \;\mid f \mapsto \texttt{clos}(\lambda x.x, \emptyset) \mid (\bigcirc\ 2\ f \mapsto \texttt{clos}(\lambda x.x, \emptyset))\ , \blacksquare \rangle \\
\longrightarrow & \langle\, \texttt{clos}(\lambda x.x, \emptyset) \;\mid f \mapsto \texttt{clos}(\lambda x.x, \emptyset) \mid (\bigcirc\ 2\ f \mapsto \texttt{clos}(\lambda x.x, \emptyset))\ , \blacksquare \rangle
\end{array}
$$

Example 2: function as parameter

$(\lambda f.f\,2)\,(\lambda x.x)$ evaluates to 2

$$
\begin{array}{ll}
& \langle\, (\lambda f.f\,2)\,(\lambda x.x) \;\mid \emptyset \mid \blacksquare \rangle \\
\longrightarrow & \langle\, (\lambda f.f\,2) \;\mid \emptyset \mid (\bigcirc\ (\lambda x.x)\ \emptyset)\ ,\blacksquare \rangle \\
\longrightarrow & \langle\, \mathtt{clos}((\lambda f.f\,2),\emptyset) \;\mid \emptyset \mid (\bigcirc\ (\lambda x.x)\ \emptyset)\ ,\blacksquare \rangle \\
\longrightarrow & \langle\, \lambda x.x \;\mid \emptyset \mid (\mathtt{clos}((\lambda f.f\,2),\emptyset)\bigcirc)\ ,\blacksquare \rangle \\
\longrightarrow & \langle\, \mathtt{clos}(\lambda x.x,\emptyset) \;\mid \emptyset \mid (\mathtt{clos}((\lambda f.f\,2),\emptyset)\bigcirc)\ ,\blacksquare \rangle \\
\longrightarrow & \langle\, f\,2 \;\mid f \mapsto \mathtt{clos}(\lambda x.x,\emptyset) \mid \blacksquare \rangle \\
\longrightarrow & \langle\, f \;\mid f \mapsto \mathtt{clos}(\lambda x.x,\emptyset) \mid (\bigcirc\ 2\ f \mapsto \mathtt{clos}(\lambda x.x,\emptyset))\ ,\blacksquare \rangle \\
\longrightarrow & \langle\, \mathtt{clos}(\lambda x.x,\emptyset) \;\mid f \mapsto \mathtt{clos}(\lambda x.x,\emptyset) \mid (\bigcirc\ 2\ f \mapsto \mathtt{clos}(\lambda x.x,\emptyset))\ ,\blacksquare \rangle \\
\longrightarrow & \langle\, 2 \;\mid f \mapsto \mathtt{clos}(\lambda x.x,\emptyset) \mid (\mathtt{clos}(\lambda x.x,\emptyset)\bigcirc)\ ,\blacksquare \rangle
\end{array}
$$

Example 2: function as parameter

$(\lambda f.f\,2)\,(\lambda x.x)$ evaluates to 2

$$\begin{array}{ll}
& \langle (\lambda f.f\,2)\,(\lambda x.x) \;\mid \emptyset \mid \blacksquare \rangle \\
\longrightarrow & \langle (\lambda f.f\,2) \;\mid \emptyset \mid (\bigcirc\;(\lambda x.x)\;\emptyset)\,,\blacksquare \rangle \\
\longrightarrow & \langle \mathtt{clos}((\lambda f.f\,2),\emptyset) \;\mid \emptyset \mid (\bigcirc\;(\lambda x.x)\;\emptyset)\,,\blacksquare \rangle \\
\longrightarrow & \langle \lambda x.x \;\mid \emptyset \mid (\mathtt{clos}((\lambda f.f\,2),\emptyset)\,\bigcirc)\,,\blacksquare \rangle \\
\longrightarrow & \langle \mathtt{clos}(\lambda x.x,\emptyset) \;\mid \emptyset \mid (\mathtt{clos}((\lambda f.f\,2),\emptyset)\,\bigcirc)\,,\blacksquare \rangle \\
\longrightarrow & \langle f\,2 \;\mid f \mapsto \mathtt{clos}(\lambda x.x,\emptyset) \mid \blacksquare \rangle \\
\longrightarrow & \langle f \;\mid f \mapsto \mathtt{clos}(\lambda x.x,\emptyset) \mid (\bigcirc\;2\;f \mapsto \mathtt{clos}(\lambda x.x,\emptyset))\,,\blacksquare \rangle \\
\longrightarrow & \langle \mathtt{clos}(\lambda x.x,\emptyset) \;\mid f \mapsto \mathtt{clos}(\lambda x.x,\emptyset) \mid (\bigcirc\;2\;f \mapsto \mathtt{clos}(\lambda x.x,\emptyset))\,,\blacksquare \rangle \\
\longrightarrow & \langle 2 \;\mid f \mapsto \mathtt{clos}(\lambda x.x,\emptyset) \mid (\mathtt{clos}(\lambda x.x,\emptyset)\,\bigcirc)\,,\blacksquare \rangle \\
\longrightarrow & \langle x \;\mid x \mapsto 2 \mid \blacksquare \rangle
\end{array}$$

Example 2: function as parameter

$(\lambda f.f\,2)\,(\lambda x.x)$ evaluates to 2

$$
\begin{array}{ll}
 & \langle (\lambda f.f\,2)\,(\lambda x.x) \mid \emptyset \mid \blacksquare \rangle \\
\longrightarrow & \langle (\lambda f.f\,2) \mid \emptyset \mid (\bigcirc\ (\lambda x.x)\ \emptyset)\ ,\blacksquare \rangle \\
\longrightarrow & \langle \mathtt{clos}((\lambda f.f\,2),\emptyset) \mid \emptyset \mid (\bigcirc\ (\lambda x.x)\ \emptyset)\ ,\blacksquare \rangle \\
\longrightarrow & \langle \lambda x.x \mid \emptyset \mid (\mathtt{clos}((\lambda f.f\,2),\emptyset)\ \bigcirc)\ ,\blacksquare \rangle \\
\longrightarrow & \langle \mathtt{clos}(\lambda x.x,\emptyset) \mid \emptyset \mid (\mathtt{clos}((\lambda f.f\,2),\emptyset)\ \bigcirc)\ ,\blacksquare \rangle \\
\longrightarrow & \langle f\,2 \mid f \mapsto \mathtt{clos}(\lambda x.x,\emptyset) \mid \blacksquare \rangle \\
\longrightarrow & \langle f \mid f \mapsto \mathtt{clos}(\lambda x.x,\emptyset) \mid (\bigcirc\ 2\ f \mapsto \mathtt{clos}(\lambda x.x,\emptyset))\ ,\blacksquare \rangle \\
\longrightarrow & \langle \mathtt{clos}(\lambda x.x,\emptyset) \mid f \mapsto \mathtt{clos}(\lambda x.x,\emptyset) \mid (\bigcirc\ 2\ f \mapsto \mathtt{clos}(\lambda x.x,\emptyset))\ ,\blacksquare \rangle \\
\longrightarrow & \langle 2 \mid f \mapsto \mathtt{clos}(\lambda x.x,\emptyset) \mid (\mathtt{clos}(\lambda x.x,\emptyset)\ \bigcirc)\ ,\blacksquare \rangle \\
\longrightarrow & \langle x \mid x \mapsto 2 \mid \blacksquare \rangle \\
\longrightarrow & \langle 2 \mid x \mapsto 2 \mid \blacksquare \rangle\ ☺
\end{array}
$$

Example 3: machine gets stuck ☹

$5\,(\lambda x.x)$ is nonsense. You cannot apply a number to a function.

$$\langle 5\,(\lambda x.x) \;\mid \emptyset \mid \blacksquare \rangle$$

Example 3: machine gets stuck ☹

$5\,(\lambda x.x)$ is nonsense. You cannot apply a number to a function.

$$\begin{array}{rl} & \langle 5\,(\lambda x.x) \mid \emptyset \mid \blacksquare \rangle \\ \longrightarrow & \langle 5 \mid \emptyset \mid (\bigcirc\,(\lambda x.x)\,\emptyset)\,,\blacksquare \rangle \end{array}$$

Example 3: machine gets stuck ☹

$5\,(\lambda x.x)$ is nonsense. You cannot apply a number to a function.

$$\begin{array}{ll}
 & \langle 5\,(\lambda x.x) \;\mid \emptyset \mid \blacksquare \rangle \\
\longrightarrow & \langle 5 \;\mid \emptyset \mid (\bigcirc\,(\lambda x.x)\,\emptyset)\,,\blacksquare \rangle \\
\longrightarrow & \langle (\lambda x.x) \;\mid \emptyset \mid (5\,\bigcirc)\,,\blacksquare \rangle
\end{array}$$

Example 3: machine gets stuck ☹

$5\,(\lambda x.x)$ is nonsense. You cannot apply a number to a function.

$$
\begin{array}{ll}
 & \langle 5\,(\lambda x.x) \mid \emptyset \mid \blacksquare \rangle \\
\longrightarrow & \langle 5 \mid \emptyset \mid (\bigcirc\,(\lambda x.x)\,\emptyset)\,,\blacksquare \rangle \\
\longrightarrow & \langle (\lambda x.x) \mid \emptyset \mid (5\,\bigcirc)\,,\blacksquare \rangle \\
\longrightarrow & \langle \mathtt{clos}((\lambda x.x),\emptyset) \mid \emptyset \mid (5\,\bigcirc)\,,\blacksquare \rangle ☹
\end{array}
$$

Does not match the rule, as 5 is not a closure.

$$\langle\, W \mid E_1 \mid (\mathtt{clos}(\lambda x.M, E_2)\,\bigcirc)\,, K\,\rangle \longrightarrow \langle\, M \mid E_2[x \mapsto W] \mid K\,\rangle$$

In ML, it would be rejected by the type checker. In Lisp/Scheme, you get a runtime error.

Example 4: machine gets stuck ☹

A free variable that does not occur in the environment makes the machine get stuck.

$$\langle x \mid \emptyset \mid \blacksquare \rangle$$ ☹

In ML, this would be a compile-time error and in Lisp/Scheme a run-time error.
The type system of ML prevents all such run-time errors;
Milner's slogan for ML, "Well-typed programs do not go wrong." ☺

Examples as above in OCaml

```
# (fun x -> fun y -> x) 1 2;;
- : int = 1

# (fun f -> f 2) (fun x -> x);;
- : int = 2

# 5 (fun x -> x);;
Error: This expression is not a function;
it cannot be applied

# x;;
Error: Unbound value x
```

CEK implementation ideas in C++

In C/C++, it is not hard to implement the CEK machine. The machine state consists of 3 pointers:

- the C component is implemented as a pointer into the abstract syntax tree of the lambda term
- the E component is implemented as a heap-allocated linked list, so that each of the list elements contains a variable name and the corresponding value
- the K component is implemented as a stack (= array and a stack pointer) of frames, where frames are represented as a tagged union
- closures are represented as pairs of pointers in a heap-allocated struct

The machine is then a function from states to states, corresponding to $\longrightarrow$.
It may raise an exception when the state is stuck.
The step function is run in a loop to give $\xrightarrow{*}$ until an accepting state is reached.

Extended exercise 2

Implement the CEK machine in one of: C, C++, Java, OCaml, Haskell.
You do not need write a parser. It is sufficient if you can build terms from constructor calls, e.g.

```
App(Lambda("x", Var("x")), Const(2))
```

The implementation should run the transition function until a value is reached and then print that value.

CEK exercise

Explain what sequence of steps the CEK machine performs starting from the following configuration:

$$\langle\, (\lambda x.xx)\,(\lambda x.xx) \;\mid \emptyset \mid \blacksquare \rangle$$

In particular, does one get a W at the end?

CEK exercise

Suppose we do not build closures at all, and we just pass around terms like $\lambda x . M$ instead. How would that affect the semantics of the language?

Optimizations, lambda calculus and the CEK machine

- Let's revisit optimizations in the light of what we have learned about lambda and CEK
- The CEK machine gives us a formal model in which we can prove that optimizations are correct.
- Formally: optimizations preserve contextual equivalence.
- Constant propagation is like a beta reduction.
- Function inlining is like two beta reductions.
- On the other hand, the CEK machine is too high-level to do tail call optimization; need to make jumps explicit via continuations
- Current (2010–) research: verify optimizations in LLVM ☺
- Problem: some common C++ optimizations are not valid due to concurrency ☹

Encoding let as lambda

Naming intermediate values with `let` is useful for optimizations.
Recall that a local let binding

$$\texttt{let}\ x = M_1\ \texttt{in}\ M_2$$

is encoded as

$$(\lambda x.M_2)\ M_1$$

Idea: evaluate M_1, bind the result to x and then evaluate M_2.

Optimization example 1: constant propagation

$$\texttt{let}\ x = V\ \texttt{in}\ \ldots x \ldots$$

can be optimized to

$$\texttt{let}\ x = V\ \texttt{in}\ \ldots V \ldots$$

This is easier in functional than imperative languages. Consider

```
int x = 42;
x = x + 2;
y = x
```

Optimization example 2: function inlining

$$\texttt{let}\ f = \lambda x.M\ \texttt{in}\ \ldots f\ V \ldots$$

can be optimized to

$$\texttt{let}\ f = \lambda x.M\ \texttt{in}\ \ldots M[x \mapsto V] \ldots$$

where $M[x \mapsto V]$ is the substitution of the actual parameter V for the formal parameter x on the body M of the function f.

Function inlining from Clang lectures revisited

```
long s(long x)
{
  return x * x;
}

long f(long y)
{
  return s(y + 1);
}
```

Function inlining from Clang lectures revisited

```
long s(long x)
{
  return x * x;
}

long f(long y)
{
  return s(y + 1);
}
```

$\texttt{let}\ s = \lambda x.x * x\ \texttt{in}$
$\texttt{let}\ f = \lambda y.s(y + 1)\ \texttt{in}$
M

Function inlining from Clang lectures revisited

```
long s(long x)
{
  return x * x;
}

long f(long y)
{
  return s(y + 1);
}
```

$\texttt{let}\ s = \lambda x.x * x\ \texttt{in}$
$\texttt{let}\ f = \lambda y.s(y+1)\ \texttt{in}$
M

$$\texttt{let}\ s = \lambda x.x * x\ \texttt{in}\ \texttt{let}\ f = \lambda y.s(y+1)\ \texttt{in}\ M$$

Function inlining from Clang lectures revisited

```
long s(long x)
{
  return x * x;
}

long f(long y)
{
  return s(y + 1);
}
```

$\mathtt{let}\ s\ =\ \lambda x.x * x\ \mathtt{in}$
$\mathtt{let}\ f = \lambda y.s(y+1)\ \mathtt{in}$
M

$$\begin{aligned} & \mathtt{let}\ s = \lambda x.x * x\ \mathtt{in}\ \mathtt{let}\ f = \lambda y.s(y+1)\ \mathtt{in}\ M \\ = \quad & (\lambda s.(\lambda f.M)\ (\lambda y.s(y+1)))\ (\lambda x.x * x) \end{aligned}$$

Function inlining from Clang lectures revisited

```
long s(long x)
{
  return x * x;
}

long f(long y)
{
  return s(y + 1);
}
```

let $s = \lambda x.x * x$ in
let $f = \lambda y.s(y+1)$ in
M

$$\begin{array}{rl} & \texttt{let}\ s = \lambda x.x * x\ \texttt{in}\ \texttt{let}\ f = \lambda y.s(y+1)\ \texttt{in}\ M \\ = & (\lambda s.(\lambda f.M)\,(\lambda y.s(y+1)))\,(\lambda x.x*x) \\ \to_\beta & (\lambda f.M)\,(\lambda y.(\lambda x.x*x)\,(y+1)) \end{array}$$

Function inlining from Clang lectures revisited

```
long s(long x)
{
  return x * x;
}

long f(long y)
{
  return s(y + 1);
}
```

$\texttt{let}\ s\ =\ \lambda x.x * x\ \ \texttt{in}$
$\texttt{let}\ f = \lambda y.s(y+1)\ \texttt{in}$
M

$$
\begin{array}{rl}
 & \texttt{let}\ s = \lambda x.x * x\ \texttt{in}\ \texttt{let}\ f = \lambda y.s(y+1)\ \texttt{in}\ M \\
= & (\lambda s.(\lambda f.M)\,(\lambda y.s(y+1)))\,(\lambda x.x * x) \\
\to_\beta & (\lambda f.M)\,(\lambda y.(\lambda x.x * x)\,(y+1))) \\
\to_\beta & (\lambda f.M)\,(\lambda y.(y+1) * (y+1))
\end{array}
$$

Function inlining from Clang lectures revisited

```
long s(long x)
{
  return x * x;
}

long f(long y)
{
  return s(y + 1);
}
```

$\texttt{let}\ s\ =\ \lambda x.x * x\ \ \texttt{in}$
$\texttt{let}\ f\ =\ \lambda y.s(y+1)\ \texttt{in}$
M

$$\begin{array}{rl}
& \texttt{let}\ s = \lambda x.x * x\ \texttt{in}\ \texttt{let}\ f = \lambda y.s(y+1)\ \texttt{in}\ M \\
= & (\lambda s.(\lambda f.M)\ (\lambda y.s(y+1)))\ (\lambda x.x * x) \\
\to_\beta & (\lambda f.M)\ (\lambda y.(\lambda x.x * x)\ (y+1))) \\
\to_\beta & (\lambda f.M)\ (\lambda y.(y+1) * (y+1)) \\
= & \texttt{let}\ f = \lambda y.(y+1) * (y+1)\ \texttt{in}\ M
\end{array}$$

Beta reduction as an optimizaton

Models of Computation: Church Rosser and confluence: you can beta reduce a lambda anywhere inside an expression, without changing the result
In call-by-value with effects (like assignment or control), we need to be more careful. Unrestricted beta reduction is not a sound optimization in the CEK machine!
We cannot optimize

$$(\lambda x.5)\,((\lambda x.xx)\,(\lambda x.xx))$$

into

$$5$$

Need to be careful about effects and use beta-value instead.

Advanced programming language features

Many advanced language constructs first appeared in functional languages.
closures: first ML and Scheme, now also C++11 and Java
garbage collection: first in Lisp, then other functional languages, became mainstream in Java
exceptions: C++ adapted them from ML, now mainstream

CEK machine with control operators here and go

We extend the programming language with two control operators:

$$\begin{array}{rcl} M & ::= & \ldots \\ & \mid & \texttt{go}\, n \\ & \mid & \texttt{here}\, M \end{array}$$

Intuitively, go jumps to the nearest enclosing `here`.
This in an idealized version of exceptions in Ocaml, C++, Java, ...

Compare exceptions in OCaml

The control we have added to the CEK machine is like a single exception of integer type in OCaml, such that all exception handlers are the identity.

$$
\begin{array}{rcl}
 & & \texttt{exception E of int;;} \\
(\mathrm{go}\, n) & = & \texttt{raise (E } n\texttt{)} \\
(\mathbf{here}\, M) & = & \texttt{try } M \texttt{ with E x -> x}
\end{array}
$$

Stack frames extended for control

To give meaning to the control operators, we add a new kind of frame. It just marks a position on the stack:

$$F ::= \dots \mid \blacktriangleright\!\blacktriangleright$$

The CEK+go machine with control has these additional transition steps:

$$\begin{aligned}
\langle \mathtt{here}\, M \mid E \mid K \rangle &\longrightarrow \langle M \mid E \mid \blacktriangleright\!\blacktriangleright, K \rangle && (1)\\
\langle \mathtt{go}\, n \mid E \mid K_1, \blacktriangleright\!\blacktriangleright, K_2 \rangle &\longrightarrow \langle n \mid E \mid K_2 \rangle && (2)\\
\langle W \mid E \mid \blacktriangleright\!\blacktriangleright, K \rangle &\longrightarrow \langle W \mid E \mid K \rangle && (3)
\end{aligned}$$

Here K_1 does not contain $\blacktriangleright\!\blacktriangleright$, that is,

$$K_1 \neq K_3, \blacktriangleright\!\blacktriangleright, K_4$$

for any K_3 and K_4.

The here and go rules explained

1. A term `here` M is evaluated by pushing the marker onto the stack and continuing with evaluating M.
2. A term go M is evaluated by erasing the stack up to the nearest marker, and then proceeding with evaluating M.
3. When a value W is produced by the evaluation, any marker on the top of the stack is popped off and ignored.

CEK machine with control operators here and go

$$
\begin{array}{rcl}
\langle x \mid E \mid K \rangle & \longrightarrow & \langle \texttt{lookup}\, x\, \texttt{in}\, E) \mid E \mid K \rangle \\
\langle M_1\, M_2 \mid E \mid K \rangle & \longrightarrow & \langle M_1 \mid E \mid (\bigcirc\ M_2\ E)\,,K) \rangle \\
\langle \lambda x.M \mid E \mid K \rangle & \longrightarrow & \langle \texttt{clos}(\lambda x.M, E) \mid E \mid K \rangle \\
\langle W \mid E_1 \mid (\bigcirc\ M\ E_2)\,,K \rangle & \longrightarrow & \langle M \mid E_2 \mid (W\,\bigcirc)\,,K \rangle \\
\langle W \mid E_1 \mid (\texttt{clos}(\lambda x.M, E_2)\,\bigcirc)\,,K \rangle & \longrightarrow & \langle M \mid E_2[x \mapsto W] \mid K \rangle
\end{array}
$$

$$
\begin{array}{rcl}
\langle \texttt{here}\, M \mid E \mid K \rangle & \longrightarrow & \langle M \mid E \mid \blacktriangleright\!\!\blacktriangleright\,,K \rangle \\
\langle \texttt{go}\, n \mid E \mid K_1\,,\blacktriangleright\!\!\blacktriangleright\,,K_2 \rangle & \longrightarrow & \langle n \mid E \mid K_2 \rangle \\
\langle W \mid E \mid \blacktriangleright\!\!\blacktriangleright\,,K \rangle & \longrightarrow & \langle W \mid E \mid K \rangle
\end{array}
$$

Example 1 of jumping with here and go

$$\langle \mathtt{here}\,((\lambda x.2)\,(\mathtt{go}\,5)) \;\mid \emptyset \mid \blacksquare \rangle$$

Example 1 of jumping with `here` and `go`

$$
\begin{array}{rl}
 & \langle \texttt{here}\,((\lambda x.2)\,(\mathsf{go}\,5)) \;\mid \emptyset \mid \blacksquare \rangle \\
\longrightarrow & \langle (\lambda x.2)\,(\mathsf{go}\,5) \;\mid \emptyset \mid \blacktriangleright\!\!\blacktriangleright, \blacksquare \rangle
\end{array}
$$

Example 1 of jumping with `here` and `go`

$$
\begin{array}{ll}
 & \langle \texttt{here}\,((\lambda x.2)\,(\mathsf{go}\,5)) \mid \emptyset \mid \blacksquare \rangle \\
\longrightarrow & \langle (\lambda x.2)\,(\mathsf{go}\,5) \mid \emptyset \mid \blacktriangleright\!\!\blacktriangleright, \blacksquare \rangle \\
\longrightarrow & \langle (\lambda x.2) \mid \emptyset \mid (\bigcirc\,(\mathsf{go}\,5)\,\emptyset), \blacktriangleright\!\!\blacktriangleright, \blacksquare \rangle
\end{array}
$$

Example 1 of jumping with `here` and go

$$
\begin{array}{ll}
 & \langle \texttt{here}\,((\lambda x.2)\,(\mathsf{go}\,5)) \mid \emptyset \mid \blacksquare \rangle \\
\longrightarrow & \langle (\lambda x.2)\,(\mathsf{go}\,5) \mid \emptyset \mid \blacktriangleright\!\blacktriangleright, \blacksquare \rangle \\
\longrightarrow & \langle (\lambda x.2) \mid \emptyset \mid (\bigcirc\,(\mathsf{go}\,5)\,\emptyset), \blacktriangleright\!\blacktriangleright, \blacksquare \rangle \\
\longrightarrow & \langle \texttt{clos}(\lambda x.2, \emptyset) \mid \emptyset \mid (\bigcirc\,(\mathsf{go}\,5)\,\emptyset), \blacktriangleright\!\blacktriangleright, \blacksquare \rangle
\end{array}
$$

Example 1 of jumping with `here` and `go`

$$
\begin{array}{rl}
 & \langle\, \mathtt{here}\,((\lambda x.2)\,(\mathsf{go}\,5)) \;\mid \emptyset \mid \blacksquare \rangle \\
\longrightarrow & \langle\, ((\lambda x.2)\,(\mathsf{go}\,5) \;\mid \emptyset \mid \blacktriangleright\!\!\blacktriangleright , \blacksquare \rangle \\
\longrightarrow & \langle\, ((\lambda x.2) \;\mid \emptyset \mid (\bigcirc\;(\mathsf{go}\,5)\;\emptyset) , \blacktriangleright\!\!\blacktriangleright , \blacksquare \rangle \\
\longrightarrow & \langle\, \mathtt{clos}(\lambda x.2, \emptyset) \;\mid \emptyset \mid (\bigcirc\;(\mathsf{go}\,5)\;\emptyset) , \blacktriangleright\!\!\blacktriangleright , \blacksquare \rangle \\
\longrightarrow & \langle\, ((\mathsf{go}\,5) \;\mid \emptyset \mid (\mathtt{clos}(\lambda x.2, \emptyset)\,\bigcirc) , \blacktriangleright\!\!\blacktriangleright , \blacksquare \rangle
\end{array}
$$

Example 1 of jumping with `here` and `go`

$$
\begin{array}{rl}
 & \langle \texttt{here}\,((\lambda x.2)\,(\mathsf{go}\,5)) \mid \emptyset \mid \blacksquare \rangle \\
\longrightarrow & \langle (\lambda x.2)\,(\mathsf{go}\,5) \mid \emptyset \mid \blacktriangleright\!\blacktriangleright , \blacksquare \rangle \\
\longrightarrow & \langle (\lambda x.2) \mid \emptyset \mid (\bigcirc\ (\mathsf{go}\,5)\ \emptyset) , \blacktriangleright\!\blacktriangleright , \blacksquare \rangle \\
\longrightarrow & \langle \texttt{clos}(\lambda x.2, \emptyset) \mid \emptyset \mid (\bigcirc\ (\mathsf{go}\,5)\ \emptyset) , \blacktriangleright\!\blacktriangleright , \blacksquare \rangle \\
\longrightarrow & \langle (\mathsf{go}\,5) \mid \emptyset \mid (\texttt{clos}(\lambda x.2, \emptyset)\,\bigcirc) , \blacktriangleright\!\blacktriangleright , \blacksquare \rangle \\
\longrightarrow & \langle 5 \mid \emptyset \mid \blacksquare \rangle
\end{array}
$$

The go 5 has deleted the $(\lambda x.2)$.

Example 2 of jumping with `here` and `go`

$$\langle \texttt{here}\,((\texttt{go}\,2)\,(\texttt{go}\,5)) \;\mid \emptyset \mid \blacksquare \rangle$$

Example 2 of jumping with `here` and `go`

$$
\begin{aligned}
& \langle \texttt{here}\,((\mathrm{go}\,2)\,(\mathrm{go}\,5)) \;\mid \emptyset \mid \blacksquare \rangle \\
\longrightarrow\;\; & \langle (\mathrm{go}\,2)\,(\mathrm{go}\,5) \;\mid \emptyset \mid \blacktriangleright\!\blacktriangleright , \blacksquare \rangle
\end{aligned}
$$

Example 2 of jumping with `here` and go

$$
\begin{array}{rl}
 & \langle \mathtt{here}\,((\mathrm{go}\,2)\,(\mathrm{go}\,5)) \;\mid \emptyset \mid \blacksquare \rangle \\
\longrightarrow & \langle (\mathrm{go}\,2)\,(\mathrm{go}\,5) \;\mid \emptyset \mid \blacktriangleright\!\blacktriangleright , \blacksquare \rangle \\
\longrightarrow & \langle (\mathrm{go}\,2) \;\mid \emptyset \mid (\bigcirc\ (\mathrm{go}\,5)\ \emptyset) , \blacktriangleright\!\blacktriangleright , \blacksquare \rangle
\end{array}
$$

Example 2 of jumping with `here` and go

$$
\begin{array}{rl}
& \langle \texttt{here}\,((\mathsf{go}\,2)\,(\mathsf{go}\,5)) \;\mid \emptyset \mid \blacksquare \rangle \\
\longrightarrow & \langle (\mathsf{go}\,2)\,(\mathsf{go}\,5) \;\mid \emptyset \mid \blacktriangleright\!\!\blacktriangleright, \blacksquare \rangle \\
\longrightarrow & \langle (\mathsf{go}\,2) \;\mid \emptyset \mid (\bigcirc\,(\mathsf{go}\,5)\,\emptyset), \blacktriangleright\!\!\blacktriangleright, \blacksquare \rangle \\
\longrightarrow & \langle 2 \;\mid \emptyset \mid \blacksquare \rangle
\end{array}
$$

The go 2 happens first and deletes the go 5 due to left to right evaluation.

Example 3 of jumping with `here` and `go`

A `go` without a surrounding `here` is stuck.

$$\langle\,(\mathsf{go}\; n)\;\mid E \mid K\,\rangle$$

This corresponds to an uncaught exception in ML. The type system does not prevent this.
In Java, `throws` clauses try to minimize uncaught checked exceptions at compile time.

Compare exceptions in OCaml

The control we have added to the CEK machine is like a single exception of integer type in OCaml, such that all exception handlers are the identity.

```
exception E of int;;
```

$$\begin{array}{rcl} (\mathsf{go}\, n) & = & \texttt{raise (E } n\texttt{)} \\ (\texttt{here}\, M) & = & \texttt{try } M \texttt{ with E x -> x} \end{array}$$

Exceptions in OCaml and evaluation order

We can observe that OCaml does right-to-left evaluation:

```
exception E of int;;
```

```
(raise (E 1)) (raise (E 2));;
```

Or, since uncaught exceptions are grungy, we could add a handler:

```
try (raise (E 1)) (raise (E 2)) with E x -> x;;
```

This gives

```
- : int = 2
```

Exercise: more than one here

Evaluate the following term in the CEK machine

$$(\lambda f\,.\,\mathtt{here}\,((\lambda x.1)\,(f\,2)))\,(\mathtt{here}\,(\lambda y\,.\,\mathrm{go}\,y))$$

You should pay particular attention to which of the two occurrences of `here` is jumped to. The one on the left is *dynamically enclosing* (when the go is evaluated), whereas the one on the right is *statically enclosing* (where the go is defined).

Compare exceptions in OCaml: static vs dynamic

```
exception E;;

let f =
    try
        fun x -> raise E
    with E -> fun y -> "static"
in
    try
        f 0
    with E -> "dynamic"
;;
```

Exercise: translate this example using lambda into C++ or Java.

Intermediate language in the middle end

$$\begin{tikzcd}
C \arrow[r] & IR \arrow[r] & IR' \arrow[r] \arrow[dr] \arrow[ddr] & x86 \\
C++ \arrow[ur] & & & ARM \\
Haskell \arrow[uur] & & & SPARC
\end{tikzcd}$$

IR = intermediate representation, used for optimizations

Intermediate languages

- Intermediate languages are used for optimizations
- All intermediate values are named.
- All control flow is made explicit with jumps and labels.
- This format is good for optimizations and code generation:
- named values $\mapsto$ registers
- labels $\mapsto$ code location
- Both SSA and CPS do this.
- SSA = static single assignment, used in LLVM
- CPS = continuation passing style, used in functional languages

Intermediate languages and let bindings

Let bindings are useful for breaking up larger expressions:

$$\text{compile}(M_1\ M_2) =$$
$$\texttt{let}\ f = \text{compile}(M_1)\ \texttt{in}\ \texttt{let}\ x = \text{compile}(M_2)\ \texttt{in}\ f\ x$$

Thus $M_1\ M_2$ is broken down into chunks of code:

1. compute the value of M_1 and store it as f
2. compute the value of M_2 and store it as x
3. perform the application $f\ x$ (e.g. a `call` f instruction)

The compiler can put f and x into registers

Static Single Assignment (SSA) basic idea

- Suppose we have some code:
  x = 5;
  x = x + 4;
  y = x * 2;
  Can we replace x by 5 everywhere?
- In SSA, each variable is assigned only once in the code.
  $x_1 = 5$;
  $x_2 = x_1 + 4$;
  $y = x_2 * 2$;
- Many optimizations become easier in SSA
- SSA is like `let` in functional programming
- The different x_i may be put into different registers
- LLVM uses SSA in its Intermediate Representation

SSA example: starting code

```
x = 5;
x = x + 4;
y = x * 2;
```

SSA example: convert to SSA by renaming variables

$x_1 = 5;$
$x_2 = x_1 + 4;$
$y = x_2 * 2;$

SSA example: propagate $x_1 = 5$

$x_1 = 5;$
$x_2 = 5 + 4;$
$y = x_2 * 2;$

SSA example: eliminate useless $x_1 = 5$

$x_2 = 5 + 4;$
$y = x_2 * 2;$

Verifying compiler optimizations

If the compiler optimizes M_1 into M_2, then they should be contextually equivalent, so the user never sees any difference between the optimized and unoptimized program.
Example: a let binding (of SSA assignment)

$$\texttt{let}\ x = V\ \texttt{in}\ M$$

is optimized by replacing x by V in M:

$$M[x \mapsto V]$$

This may seem obvious, but is not so easy to prove (simulation proof).
Verifying compiler optimizations, including LLVM, is an active research area.

Contextual equivalence: two programs behave the same

For the CEK machine, we write $M \Downarrow n$ if

$$\langle\, M \mid \emptyset \mid \blacksquare \,\rangle \longrightarrow \cdots \longrightarrow \langle\, n \mid E \mid \blacksquare \,\rangle$$

A context C is a "term with a hole":

$$\begin{array}{rcl} C & ::= & \bigcirc \\ & \mid & M\,C \\ & \mid & C\,M \\ & \mid & \lambda x.C \end{array}$$

We write $C[M]$ for the term that we get by plugging M into the hole position $\bigcirc$ in C.
M_1 and M_2 are *contextually equivalent* if for all contexts C, and integers n

$$C[M_1] \Downarrow n \text{ if and only if } C[M_2] \Downarrow n$$

Contextual equivalence and optimizations

If M_1 and M_2 are contextually equivalent, then the compiler can replace M_1 by M_2 if the latter is better is some way (e.g. runs faster)

One can prove that this works for constant propagation and function inlining

Some current research in compilers

- There is lots of research, but here are some areas that I am aware of.
- Semantics of intermediate languages, like LLVM IR using abstract machines.
- Verification of optimizations.
- SSA = static single assignment, used in LLVM
- CPS = continuation passing style, used in functional languages
- If you are interested in a PhD on compilers, feel free to ask me.

Further reading

[ABDM03] gives an overview of abstract machines and simple compilers derived from them
[Ken07] gives an overview of modern CPS compilation

References

Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard.
From interpreter to compiler and virtual machine: a functional derivation.
BRICS, Department of Computer Science, University of Aarhus, 2003.

Andrew Kennedy.
Compiling with continuations, continued.
In *ACM SIGPLAN Notices*, volume 42, pages 177–190. ACM, 2007.

Peter J. Landin.
The mechanical evaluation of expressions.
The Computer Journal, 6(4):308–320, January 1964.

Structure of the module

Parsing ✓

- Progression from: Language + Logic, Models of Computation
- abstract machines, formal, “mathy”

Compiling C with Clang ✓

- Progression from: Computer Systems + Architecture, C/C++
- not so formal, by example, x86 machine code

Implementing functional languages ✓

- Progression from: functional programming
- builds on abstract machines and C stack
- the most research-led section, goes beyond Dragon Book