

# Assignment #2:

## Compiling to the Combinator Calculus

Roger Wolff, Salil Joshi for 15-312: Principles of Programming Languages

Out: Wednesday, January 25th  
Due: Tuesday, February 7th 11:59pm

### 1 Introduction

This assignment consists of a theoretical section and an implementation section.

For the theoretical component, we present a language called  $\mathcal{L}\{SKI\}$  that is simpler than  $\mathcal{L}\{\lambda\}$  and yet just as powerful. You will be asked to prove that these two languages are equally powerful by embedding each into the other.

For the implementation, you have to write interpreters for both  $\mathcal{L}\{\lambda\}$  and  $\mathcal{L}\{SKI\}$ , and a compiler that implements the embeddings that you discovered in the theoretical part. In addition, you must also write an interpreter for an extended  $\lambda$ -calculus called  $\mathcal{L}\{Dyn\}$  that has natural numbers and nullary and binary tuples as primitives.

#### 1.1 Submission

To submit your solutions place your `assn2.pdf` and `*.sml` files in your `handin` directory:

`/afs/andrew.cmu.edu/course/15/312/handin/<yourandrewid>/assn2/`

### 2 Theory

#### 2.1 Combinator Calculus

As counterintuitive as it may be to believe that  $\mathcal{L}\{\lambda\}$  is Turing complete, we present another combinator calculus language,  $\mathcal{L}\{SKI\}$ , that is even simpler and yet just as powerful.

Expression	$A ::= x$	variable
	$A_1 A_2$	application
	$C$	combinator

Combinator  $C ::= S | K | I$

Variables will only be used later for an encoding from  $\mathcal{L}\{\lambda\}$ , so  $\mathcal{L}\{SKI\}$  is really a language consisting of 3 constants,  $S$ ,  $K$ , and  $I$ . And if we'd like to be a bit pedantic,  $I$  is really simply syntactic

sugar for  $\mathcal{S} \mathcal{K} \mathcal{K}$ , as we shall see later.

The statics of the language are so trivial as to not need specification. Any string of combinators is allowed, and parentheses, as typical, are used simply to specify grouping.

The dynamics of  $\mathcal{L}\{SKI\}$  are defined as follows, using an equational definition as with  $\mathcal{L}\{\lambda\}$ . We use  $\equiv^c$  instead of  $\equiv$  to syntactically differentiate which relation we are referring to, but in essence they mean the same thing.

$$\frac{}{A \equiv^c A} (\equiv^c \text{REF}) \qquad \frac{A \equiv^c A' \quad A' \equiv^c A''}{A \equiv^c A''} (\equiv^c \text{TRANS})$$

$$\frac{A \equiv^c A'}{A' \equiv^c A} (\equiv^c \text{SYM}) \qquad \frac{A_1 \equiv^c A'_1 \quad A_2 \equiv^c A'_2}{A_1 A_2 \equiv^c A'_1 A'_2} (\equiv^c \text{APP})$$

$$\frac{}{S A B C \equiv^c (A C)(B C)} (\equiv^c \mathcal{S}) \qquad \frac{}{\mathcal{K} A B \equiv^c A} (\equiv^c \mathcal{K}) \qquad \frac{}{\mathcal{I} A \equiv^c A} (\equiv^c \mathcal{I})$$

The relation  $\equiv^c$  is an *equivalence* relation, meaning it is reflexive, transitive, and symmetric. The rule ( $\equiv^c \text{APP}$ ) is as one would expect. The interesting rules are how the combinators work.

As we did when defining,  $\Rightarrow$ , we will first prove some basic correctness of the relation  $\equiv^c$ , namely that substitution preserves  $\equiv^c$ .

**Sublemma 1.** *If  $B \equiv^c B'$  then  $[B/x]A \equiv^c [B'/x]A$ .*

*Proof.* Induction on the form of  $A$ . □

**Sublemma 2.** *If  $A \equiv^c A'$  then  $[B/x]A \equiv^c [B/x]A'$ .*

*Proof.* Induction on the derivation of  $A \equiv^c A'$ . □

**Lemma 1.** *If  $A \equiv^c A'$  and  $B \equiv^c B'$  then  $[B/x]A \equiv^c [B'/x]A'$ .*

*Proof.* Follows directly from Sublemmas 1,2, and the transitivity of  $\equiv^c$ . □

## 2.2 Embedding $\mathcal{L}\{SKI\}$ in $\mathcal{L}\{\lambda\}$

Eventually, we will prove that  $\mathcal{L}\{SKI\}$  and  $\mathcal{L}\{\lambda\}$  are equally powerful. We will do this by providing a pair of embeddings. A  $\mathcal{L}\{SKI\}$  expression,  $A$ , can be embedded into  $\mathcal{L}\{\lambda\}$  by  $\widehat{A}$ . And a  $\mathcal{L}\{\lambda\}$  expression,  $M$ , can be embedded into  $\mathcal{L}\{SKI\}$  by  $\overline{M}$ .

The first embedding,  $\widehat{A}$  is defined as:

$$\begin{aligned}
\widehat{A_1 A_2} &= \widehat{A_1} \widehat{A_2} \\
\widehat{\mathcal{S}} &= \lambda(x) \lambda(y) \lambda(z) (x z)(y z) \\
\widehat{\mathcal{K}} &= \lambda(x) \lambda(y) x \\
\widehat{\mathcal{I}} &= \lambda(x) x
\end{aligned}$$

**Task 2.1** [2 points]:

Compute  $\widehat{\mathcal{S} \mathcal{K} \mathcal{K} a}$ , and evaluate it to normal form (show all transitions).

### 2.3 Embedding $\mathcal{L}\{\lambda\}$ in $\mathcal{L}\{SKI\}$

The second embedding,  $\overline{M}$  is defined as:

$$\begin{aligned}
\overline{x} &= x \\
\overline{M N} &= \overline{M} \overline{N} \\
\overline{\lambda(x) M} &= \lambda^\dagger(x) \overline{M}
\end{aligned}$$

The definition of  $\overline{M}$  depends on the auxiliary definition of  $\lambda^\dagger(x) M$ .

$$\begin{aligned}
\lambda^\dagger(x) x &= \mathcal{I} \\
\lambda^\dagger(x) y &= \mathcal{K} y \\
\lambda^\dagger(x) M N &= \mathcal{S} (\lambda^\dagger(x) M) (\lambda^\dagger(x) N) \\
\lambda^\dagger(x) \mathcal{C} &= \mathcal{K} \mathcal{C}
\end{aligned}$$

**Task 2.2** [1 points]:

Compute  $\overline{(\lambda(x) x) a}$ , and evaluate it to normal form (show all transitions).

We now prove that substitution preserves  $\equiv^{\mathcal{C}}$  in the  $\lambda^\dagger$  encoding.

**Lemma 2.**  $(\lambda^\dagger(x) A) B \equiv^{\mathcal{C}} [B/x]A$ .

**Task 2.3** [4 points]:

Prove 2 by induction on the form of  $A$ .

We are almost ready to show that  $\mathcal{L}\{SKI\}$  is just as powerful as  $\mathcal{L}\{\lambda\}$ . We will do this by proving an encoding from  $\mathcal{L}\{SKI\}$  to  $\mathcal{L}\{\lambda\}$  to show that  $\mathcal{L}\{\lambda\}$  can compute anything that  $\mathcal{L}\{SKI\}$  can. And we will provide an encoding in the opposite direction as well.

Unfortunately, we will not be able to. The reason is that our definitions of  $\equiv$  and  $\equiv^{\mathcal{C}}$  are not powerful enough. These equivalences only compare the *structure* of expressions, and not their *functionality*. Take the expression  $\mathcal{S} \mathcal{K} \mathcal{K} a$ , for example. we see that  $\mathcal{S} \mathcal{K} \mathcal{K} a \equiv^{\mathcal{C}} (\mathcal{K} a)(\mathcal{K} a) \equiv^{\mathcal{C}} a$ . This behaves precisely like the expression  $\mathcal{I} a \equiv^{\mathcal{C}} a$ . That is, they both behave like the identity function. No matter what argument,  $a$  is given to it, it will evaluate to  $a$ .

This will eventually motivate the use of types. If we know that an expression is a mathematical function,

then we would like to use that information to our advantage. Two mathematical functions are equivalent if they behave the same on all inputs.

To encode this information in  $\mathcal{L}\{SKI\}$ , we must add the following rule:

$$\frac{A x \equiv^c B x \quad x \notin A, B}{A \equiv^c B} (\equiv^c_{\text{EXT}})$$

This language feature is called *extensionality*.

Similarly, we will need to expand the rules for  $\mathcal{L}\{\lambda\}$ . We add the following:

$$\frac{x \notin M}{\lambda(x) (M x) \equiv M} (\equiv \eta)$$

From the  $(\equiv \eta)$  rule, the following extensionality rule is *derivable*.

$$\frac{M x \equiv N x \quad x \notin M, N}{M \equiv N} (\equiv_{\text{EXT}})$$

*Proof.* We give an abbreviated derivation below.

$$(\equiv \eta) \frac{\frac{x \notin M}{M \equiv \lambda(x) (M x)} \quad \frac{M x \equiv N x}{\lambda(x) (M x) \equiv \lambda(x) (N x)} (\equiv \lambda) \quad \frac{x \notin N}{\lambda(x) (N x) \equiv N} (\equiv \eta)}{M \equiv N} (\equiv_{\text{TRANS}})$$

□

Now that we have changed the definitions of equivalence, we are required to amend the proofs of Lemmas 1, 2. We will spare you the suspense, and tell you that they indeed still hold.

Finally, we are ready to prove that the embeddings are faithful representations of the original expressions. We will first show that our embedding of  $\mathcal{L}\{SKI\}$  in  $\mathcal{L}\{\lambda\}$  preserves equivalence.

**Theorem 1.** *If  $A \equiv^c B$  then  $\widehat{A} \equiv \widehat{B}$ .*

*Proof.* Induction on the derivation of  $A \equiv^c B$ . The reflexivity, transitivity, symmetry and application are trivial, using the the induction hypothesis and the corresponding equivalence rule of  $\mathcal{L}\{\lambda\}$ .

Other cases are almost as simple. Take  $(\equiv^c K)$ , for instance.  $\widehat{\mathcal{K} A B} = \widehat{\mathcal{K}} \widehat{A} \widehat{B} = (\lambda(x) \lambda(y) x) \widehat{A} \widehat{B} \equiv \widehat{A}$ , by  $(\equiv \beta)$ .

□

From Theorem 1, we see that expressions of  $\mathcal{L}\{SKI\}$  can faithfully be embedded into  $\mathcal{L}\{\lambda\}$ . This is not so surprising, once you understand the universality of  $\mathcal{L}\{\lambda\}$ . The more surprising result is that our embedding of  $\mathcal{L}\{\lambda\}$  into  $\mathcal{L}\{SKI\}$  preserves equivalence as well.

We will need one more lemma, though, before we proceed.

**Lemma 3.** *If  $A \equiv^c A'$ , then  $\lambda^\dagger(x) A \equiv^c \lambda^\dagger(x) A'$ .*

**Task 2.4** [8 points]:

Lemma 3 can be proved by induction on the derivation of  $A \equiv^c A'$ . Show the cases for  $(\equiv^c \mathcal{K})$ ,  $(\equiv^c \mathcal{S})$ , and  $(\equiv^c \mathcal{I})$ . Hint: Use  $(\equiv^c EXT)$ , by applying each function with a generic argument,  $a$ .

**Theorem 2.** *If  $M \equiv N$  then  $\overline{M} \equiv^c \overline{N}$ .*

**Task 2.5** [6 points]:

Theorem 2 can be proven by induction on the derivation of  $M \equiv N$ . Prove this theorem for cases  $(\equiv \lambda)$  and  $(\equiv \beta)$ .

Together, Theorems 2 and 1 show that  $\mathcal{L}\{SKI\}$  and  $\mathcal{L}\{\lambda\}$  are computationally equivalent, as one can be faithfully embedded within the other.

## 3 Implementation

### 3.1 Description

For the implementation, you are required to implement the following modules:

- For  $\mathcal{L}\{\lambda\}$ , `Lambda:LAMBDA` and `LambdaUtil:LAMBDA_UTIL`. Part of this second structure has been written for you in `lambda-util.sml` [15 points]
- For  $\mathcal{L}\{SKI\}$ , `Combinator:COMBINATOR` [10 points]
- The compiler that translates between  $\mathcal{L}\{\lambda\}$  and  $\mathcal{L}\{SKI\}$ , `Compiler:COMPILER` [10 points]
- For  $\mathcal{L}\{Dyn\}$ , `Dyn:DYN` and `DynUtil:DYN_UTIL`. Again, part of the second structure is already in `dyn-util.sml` [20 points]
- Provide a test file in the REPL syntax (described later). Do not write trivial tests. Credit will be given if your test file breaks other people's code [10 points]

Implementing these requires some infrastructure to represent and manipulate abstract syntax, particularly for  $\mathcal{L}\{\lambda\}$  and  $\mathcal{L}\{Dyn\}$ , because you have to deal with variable binding.

The following sections will describe the existing codebase and give you some hints on how to deal with variable binding.

A parser and REPL has already been created for you. These are described in the last section.

### 3.2 Variables

A variable is a placeholder for a fixed but unspecified term. You will not need to implement the `Variable` module, but you will need to utilize it. The `Variable` module implements the `VARIABLE` signature, which is shown here.

```

signature VARIABLE =
sig
  type t

  (* creates a new, globally unique variable *)
  val newvar : string -> t

  (* tests whether two variables are equal *)
  val equal  : (t * t) -> bool

  (* compares two variables.  This is used to allow
     variables as keys into a hash table *)
  val compare : (t * t) -> order

  (* provides a string representation of the globally
     unique variable *)
  val toString : t -> string

  (* provides the string used to create the variable *)
  val toUserString : t -> string
end

```

### 3.3 Variable binding

Here is one of the signatures that you have to implement:

```

signature LAMBDA =
sig
  type t

  datatype 'a view =
    ` of Variable.t
  | \ of Variable.t * 'a
  | $ of 'a * 'a

  val into : t view -> t
  val out  : t -> t view

  (* aequiv e1 e2 = true iff e1 and e2 are alpha equivalent *)
  val aequiv : t * t -> bool

  (* freevars e returns a duplicate-free list containing
     * all the variables that are free in e *)
  val freevars : t -> Variable.t list

```

```

(* subst s e substitutes for the free variables in e using the list s *)
val subst : (Variable.t * t) list -> t -> t
end

```

The first thing to notice here is the abstract type `t`. This is the type you will use in your internal representation of lambda graphs. However, externally, there are no simple functions to construct any values of type `t`. Instead, access to the representation type `t` is mediated by a *view*, which is a non-recursive type `'a view`, and a pair of functions `into` and `out`.

The idea behind a view is that it is a type whose values represent a one-step unfolding of the abstract type `t`, and that `out` unfolds the lambda graph one step, and `into` puts a view on a term (ie, a one-step unfolding) back together. This is a version of an interface to syntax that has been developed here at CMU over numerous compiler development efforts, and which is designed to help users avoid running into some very common errors.

Using these views, we will be able to use pattern matching to write code to manipulate lambda terms, without knowledge of the underlying representation. For example, this is how we wrote the parser without knowing your implementation of `t`.

- Suppose that the term `e` of type `t` represents the lambda term  $x$ . Then, a call `out e` would return some value ``x`, where the value `x` of type `Variable.t` represents the variable  $x$ .
- Likewise, suppose a term `e` represents a lambda term  $(e_1 e_2)$ . Then a call `out e` should return `$ (e1, e2)`, where `e1` and `e2` are terms of type `t` representing the lambda terms  $e_1$  and  $e_2$  respectively.
- Finally, if a term `e` represents a lambda abstraction  $\lambda x.e'$  then the call `out e` will return `(y, e''')` where `y` is some new variable  $y$  and `e''` is a value representing the lambda term  $[y/x]e'$ . Notice that the `out` operation renames the variable bound by a lambda abstraction whenever it unpacks it! This is one of the essential features of this interface, and doing this correctly will save you many hours tracing down strange  $\alpha$ -conversion issues.

The function `into` has type `t view -> t`, and folds a one-step unfolding back into an  $\lambda$ -term.

- Suppose `x` is a term of type `Variable.t`, which represents the variable  $x$ . Then `into (`x)` will return the lambda term consisting of the variable occurrence  $x$ .
- If `e1` and `e2` are terms of type `t` representing the lambda terms  $e_1$  and  $e_2$ . Then `into ($ (e1, e2))` will return a term representing  $e_1 e_2$
- If `x` is a term of type `Variable.t`, which represents the variable  $x$  and `e` is a term of type `t` representing the lambda  $e$ , then `into(\(x, e))` will return a term representing the lambda term  $\lambda x.e$ .

However, `into` is not quite the inverse of `out`. If `e` is a term of type `t`, then `e` and `in(out(e))` will not in general represent equal terms – but they will, however, always represent  $\alpha$ -equivalent terms!

We can test  $\alpha$ -equivalence with the `aequiv` operation, which returns `true` if its two arguments are  $\alpha$ -equivalent, and `false` otherwise. This means that `aequiv(in(out(e)), e)` is always `true`.

The interfaces for `DYN` and `COMBINATOR` are similar, so you should be able to figure out what's going on there. `COMBINATOR` is particularly simple since it does not involve any variable binding. `DYN` is just an extension of `LAMBDA`

In the next section we will show you one way of implementing the internal representation `t`.

### 3.3.1 Implementing Variable Binding

One of the inconveniences of using a straightforward representation of lambda terms is that  $\alpha$ -equivalent terms can have multiple representations –  $\text{ABS}(x, \text{FV } x)$  and  $\text{ABS}(y, \text{FV } y)$  are  $\alpha$ -equivalent terms even when  $x$  and  $y$  are different variables.

We will look at a more sophisticated representation, called *locally nameless form*, which avoids this problem, so that each  $\alpha$ -equivalence class is represented with a single data value and  $\alpha$ -equivalence can be tested for with a simple structural traversal.

The basic idea is to observe that variables in a lambda term can serve two roles. First, they can appear free – that is, they can be a *name* for a hypothesis in the hypothetical context.

Second, they can be a bound variable, in which case the variable occurrences *point back* to the location of the binding abstraction. For example, in the ML program

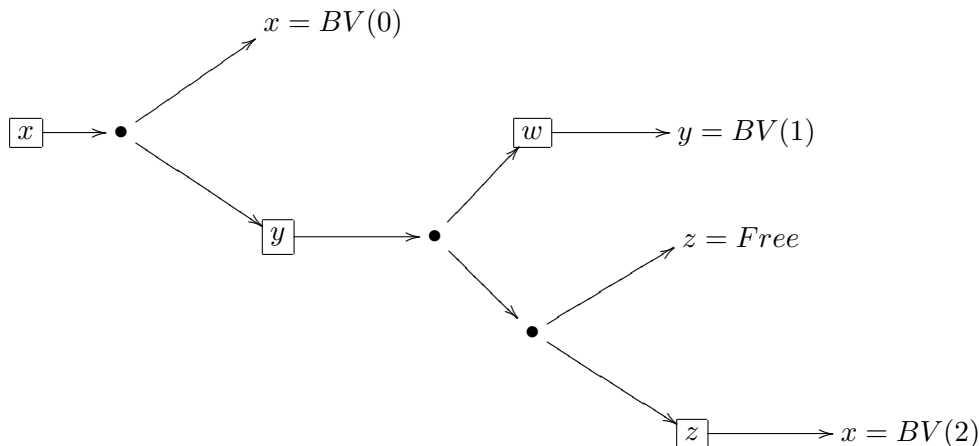
```
fn x => fn y => (x, fn z => z + x + y)
```

The bound variable  $x$  is a pointer back to the first argument, and the bound variable  $y$  points back to the second argument. The only reason we need these names is to distinguish one abstraction site from another – the variables themselves are irrelevant. (This is just another way of saying that we want to identify terms up to  $\alpha$ -equivalence, of course.)

In a locally nameless representation, we distinguish these two roles in our data structure, in order to make  $\alpha$ -equivalence implementable as structural equality on terms.

The trick in this representation is to notice that in a lambda term we have a *unique* path from a lambda abstractor to each occurrence of the variable it binds. Furthermore, since we're only interested in the binder sites, we can compress this path to a single number, which tells us how many binders we have to hop over before we reach the one we're interested in.

Consider the following diagram of the  $\lambda$ -term:  $\lambda x.(x (\lambda y.((\lambda z.x) z) (\lambda w.y)))$



We've put a box around every  $\lambda$  abstraction, used  $\bullet$  to represent application, and labelled each bound variable with its bound variable number. We can calculate the bound variable number by looking at each path from an abstraction to its use sites, and count the number of abstractions crossed along the way:

Path	Variable #
$\boxed{x} \rightarrow \bullet \rightarrow x$	0
$\boxed{y} \rightarrow \bullet \rightarrow \boxed{w} \rightarrow y$	1
$\boxed{x} \rightarrow \bullet \rightarrow \boxed{y} \rightarrow \bullet \rightarrow \bullet \rightarrow \boxed{z} \rightarrow x$	2

An important fact to notice about these paths is that even for the same binder, each of its bound variables can have a *different* bound variable number, depending on the number of abstractions we crossed over to reach that variable occurrence.

You will need to implement the structure `Lambda` in file `lambda.sml` (and similarly `Combinator` and `Dyn`) using this locally nameless representation.

Your implementation of `Lambda.t` will look something like this:

```
datatype t =
  FV of Variable.t
| BV of int
| ABS of t
| APP of t * t
```

*Hint:* When implementing this structure, you will find it helpful to define two functions `bind` and `unbind`.

- `bind` should take a term  $e$  and a free variable  $x$ , and return a new term corresponding to  $\lambda x. e$ .
- `unbind` should take a term  $\lambda x. e$ , and return a variable  $x'$ , and a term corresponding to  $[x'/x]e$ .

*Bonus hint:* Note that one of the invariants of the locally nameless representation is that the case `BV n` only occurs once you've gone beneath a binder. So it can't happen at the top level of a term.

*Bonus bonus hint:* Your implementation of `aequiv` is a structural equality comparison.

While writing your code, you will notice that the code for the functions `into`, `out`, and later `subst` is almost exactly the same for all three languages, and writing it out three times seems like a bad idea. From the next assignment onwards, we will solve this problem by using a data structure called an Abstract Binding Tree (ABT) which allows us to implement these functions in a way that is completely independent of the underlying language. ABTs are a more abstract representation (and thus a little harder to use at first) but get rid of some of the code duplication that you will find in this assignment.

### 3.4 The core of the interpreter

For all three languages, you need to implement the following three functions:

```
val subst : (Variable.t * t) list -> t -> t
val reduce : t -> t
val normalize : t -> t
```

- `subst` is a substitution function. So `subst [(x1, e1), ..., (xn, en)] e` will substitute  $e_1$  for  $x_1$ , and so on in  $e$ .

- `reduce` carries out parallel reduction using as much parallelism as possible. For the lambda calculus, use the rules defined in the previous assignment. For the combinator calculus you have to define your own reduction rules using the equivalence rules defined in the theory part of the assignment. We will explain the language  $\mathcal{L}\{Dyn\}$  and its reduction rules in a later section.

Note that by “parallel” reduction we mean that it should be *conceptually* parallel, you are not expected to write code that actually *runs* in parallel.

- `normalize` simply involves a repeated reduction until a normal form is reached. Since not every term *has* a normal form, this function might not terminate. However, if term does have a normal form then it is unique, as you proved in the previous assignment. A similar theorem can also be proved about  $\mathcal{L}\{SKI\}$  (but you need not prove it). For  $\mathcal{L}\{Dyn\}$ , uniqueness is trivial since the dynamics we have given are deterministic (although again, there are terms with no normal form).

The `reduce` and `normalize` functions for  $\mathcal{L}\{\lambda\}$  and  $\mathcal{L}\{Dyn\}$  are declared in `LAMBDA_UTIL` and `DYN_UTIL` respectively. You need to use the view interface defined earlier to write them (which is a good thing. Writing these using the internal representation is error prone).

### 3.5 From $\lambda$ to SKI

In the theory part of this assignment, you learnt how to embed the  $\lambda$  calculus into the combinator calculus, and vice versa. The signature `COMPILE` asks you to implement these embeddings. It consists of just two functions:

```
signature COMPILE =
sig
  val lambdaToCombinator : Lambda.t -> Combinator.t
  val combinatorToLambda : Combinator.t -> Lambda.t
end
```

### 3.6 Extending the untyped $\lambda$ calculus

The language  $\mathcal{L}\{Dyn\}$  extends the lambda calculus by providing a more direct representation for natural numbers and nullary and binary tuples. It is based on the Dynamic PCF language defined in Chapter 18 of PFPL.

In the  $\lambda$  calculus, there is only one *class* of values: everything is a function. In  $\mathcal{L}\{Dyn\}$  we introduce three new classes of values: numerals, binary tuples and nullary tuples. Thus we now have four classes of values:

- *Functions* which have the form  $\lambda x.e$  as before.
- *Numerals* which have the form  $\bar{n}$  where  $n$  is a natural number.
- *Nullary tuples*, with only one possible value: *nil*
- *Binary tuples* which have the form  $cons(e_1, e_2)$

*nil* and *cons* together allows us to sort of - but not quite - code up lists of values. The reason that this doesn't quite give you lists is because you can have weird terms like  $\text{cons}(\bar{1}, \bar{2})$ , which is not really a list (it should be:  $\text{cons}(\bar{1}, \text{cons}(\bar{2}, \text{nil}))$ )

We also have some primitive operations on values from each of these classes:

- Functions:
  - As before, we can *apply* one term to another:  $e_1 e_2$
- Numerals:
  - We can *add* two terms:  $e_1 + e_2$ . If  $e_1$  and  $e_2$  evaluate to the numerals  $\bar{n}_1$  and  $\bar{n}_2$  respectively, then  $e_1 + e_2$  evaluates to  $\overline{n_1 + n_2}$
  - We can *branch*:  $\text{ifz}(e_1, e_2, e_3)$ . If  $e_1$  evaluates to  $\bar{0}$  then  $\text{ifz}(e_1, e_2, e_3)$  evaluates to  $e_2$ . If  $e_1$  evaluates to a number greater than 0, then the entire term evaluates to  $e_3$
- Nullary tuples:
  - We have another conditional:  $\text{cond}(e_1, e_2, e_3)$ . This evaluates to  $e_2$  if  $e_1$  evaluates to *anything other than nil*. If  $e_1$  evaluates to *nil* then the conditional evaluates to  $e_3$ . Thus *nil* represents the boolean falsehood, and all other values represent boolean truth.
  - A *predicate* to test if a value belongs to this class:  $\text{isnil}(e)$ . Thus  $\text{isnil}(e)$  evaluates to *nil* (boolean falsehood) if  $e$  does *not* evaluate to *nil*, and to any other non-nil value otherwise.
- Binary tuples:
  - We can take the first and second *projections* of a pair:  $\text{car}(e)$  or  $\text{cdr}(e)$ . Thus  $\text{car}(\text{cons}(e_1, e_2)) \equiv e_1$  and  $\text{cdr}(\text{cons}(e_1, e_2)) \equiv e_2$
  - A *predicate* to test if a value belongs to this class:  $\text{iscons}(e)$ . Thus  $\text{iscons}(e)$  evaluates to *nil* if  $e$  does *not* evaluate to  $\text{cons}(e_1, e_2)$ , and to any other non-nil value otherwise.

Note that not all operations make sense for all classes of values. For example, what should  $\text{car}(\lambda x.x)$  be? Or  $\bar{1} \text{cons}(e_1, e_2)$ ?

We take the viewpoint that such terms result in a *runtime error* (i.e. your program crashes). Your interpreter must throw an exception called `RuntimeError` if it encounters such meaningless terms while performing reductions. So for example,  $e_1 + e_2$  makes sense only if both  $e_1$  and  $e_2$  evaluate to numerals. If they don't, then this is a runtime error (note that  $e_1$  and  $e_2$  need not literally *be* numerals; we only require that they *evaluate* to numerals).

The statics and parallel reduction rules for this language are given in Appendix B

As you write the code, you will notice that:

1. Your values are tagged in some way to indicate what *class* they belong to.
2. These tags need to be constantly checked at runtime (which is why we need judgements like *is\_num* and *isnt\_num* in the dynamic semantics in Appendix B).
3. Using boolean predicates to distinguish classes does *not* negate the need for run time checks because the boolean test does flow any useful information into the branches. For example, in the term  $\text{cond}(\text{iscons}(e), \text{car}(e), \dots)$ , the  $\text{car}(e)$  *still* has to perform the runtime check, even though  $\text{iscons}(e)$  has already checked that  $e$  is a binary pair.

- Terms need to be de-tagged and re-tagged. For example in the final rule for addition, we have to check the tags for  $e_1$  and  $e_2$ , strip off these tags to get at the underlying numbers  $n_1$  and  $n_2$ , add then retag the number  $n_1 + n_2$  to get the final term  $\overline{n_1 + n_2}$

Some of these computations may be obscured by ML's pattern matching, but they are being carried out under the hood.

All of this overhead is *inherent to every dynamically typed language*. It cannot be eliminated unless the language can statically express and enforce certain invariants i.e. unless the language is statically typed.

### 3.7 The parser and REPL

We have given you a parser and REPL that works with all three languages. This infrastructure has already been written for you, and is meant solely to help you test your code. You do not need to implement anything here.

The concrete syntax for all the languages is really simple. We describe it briefly below. In all the three languages, variables are written as a string of alphanumeric characters (including underscores and primes), starting with a letter. The following keywords may not be used as variable names:

`S, K, I, val, compile, normalize, reduce, lam, ski, dyn, ifz, nil, cons, cond, car, cdr, isnil, iscons, assert_true, assert_false, aequiv_lam, aequiv_ski, aequiv_dyn, reduce_ex, normalize_ex, load.`

#### 3.7.1 $\lambda$ calculus

- The syntax for a variable  $x$  is just `x`
- The syntax for a lambda abstraction  $\lambda x.e$  is `\x.e`
- Application  $e_1 e_2$  is `e1 e2`. It associates to the left.
- Parantheses can be used to group terms as usual

For example, the term  $(\lambda x.(xy))(\lambda x.x)$  can be written: `(\x.x y) \x.x`

#### 3.7.2 Combinator calculus

- A variable  $x$  is just `x`
- The three combinators available are `S, K` and `I`
- Application  $e_1 e_2$  is `e1 e2`. It associates to the left.
- Again, parantheses can be used to group terms

For example, the term  $(SKK)(KS)$  is written: `(S K K) (K S)`

### 3.7.3 $\mathcal{L}\{Dyn\}$

- The rules for variables, lambda abstraction and application are the same as for the  $\lambda$  calculus.
- The natural number  $n$  is just  $n$
- The syntax for addition is  $e1 + e2$ . It associates to the left.
- Branching is  $ifz(e1, e2, e3)$
- The nullary tuple is  $nil$
- Consing is  $cons(e1, e2)$
- The conditional is  $cond(e1, e2, e3)$
- $car$  is  $car(e)$
- $cdr$  is  $cdr(e)$
- $isnil$  is  $isnil(e)$
- $iscons$  is  $iscons(e)$

The `Parser` structure provides three functions that you might find useful when writing tests:

```
val parseLam : string -> Lambda.t option
val parseSKI : string -> Combinator.t option
val parseDyn : string -> Dyn.t option
```

`parseLam` takes a string which it tries to parse as a term in the  $\mathcal{L}\{\lambda\}$ , returning a `Lambda.t option`. `parseSKI` and `parseDyn` try to parse terms in  $\mathcal{L}\{SKI\}$  and  $\mathcal{L}\{Dyn\}$  respectively.

### 3.7.4 The REPL

We have provided a REPL that may be useful for testing and debugging. You can start the REPL by running `TopLevel.repl()` in the SML REPL. This section describes the commands that the REPL understands.

Note that in everything that follows,  $M$  stands for a term in any one of the three languages, prefixed by a keyword (`lam` for  $\mathcal{L}\{\lambda\}$ , `ski` for  $\mathcal{L}\{SKI\}$  and `dyn` for  $\mathcal{L}\{Dyn\}$ ) to indicate which language you mean. i.e., everytime we use  $M$ , we mean something of the form `lam e` or `ski e` or `dyn e`.

- First, entering simply  $M$  (i.e. `lam e` or `ski e` or `dyn e`) will parse the term according to the specified language, and bind it to the variable `it`.

For example, entering: `ski S K;`

gives: `val it = ski (S K)`

- `val x = M` binds the term  $M$  to the variable  $x$ . After entering this in the REPL, all future uses of the variable  $x$  will be replaced by  $M$  using the `subst` function for the language. This is only true for terms in the same language as  $M$

For example, entering:

```
val x = lam \x.x;
lam \y.x y;
```

gives: `val it = lam \y.((\x.x) y)` but entering `ski x`; after this still gives `val it = ski x`. The `x` is not substituted in here because there is no binding for it in  $\mathcal{L}\{SKI\}$ .

You can do the same thing with the other two languages as well. For example, for the combinator calculus:

```
val z = ski K K;
ski S z;
```

gives: `val it = ski (S (K K))`

Note that binding something else to `z` later will shadow this binding in terms defined later.

- `reduce M` will call the `reduce` function of the corresponding language.

For example,

```
reduce lam (\x.x) y;
```

will use `Lambda.reduce` and the REPL will print: `val it = lam y`

- `Similarly normalize M` will call the `normalize` function of the corresponding language.

Note that to `normalize` (or `reduce`) and bind the result to a variable `x`, you have to first `normalize` (or `reduce`) and then bind the variable `it` to `x`.

For example:

```
-> normalize lam (\x.x) y;
val it = lam y
-> val foo = it;
```

i.e. you *cannot* say: `val foo = normalize lam (\x.x) y`

- `compile lam e` will call `Compile.lambdaToCombinator e` and `compile ski e` will call `Compile.combinatorToLamda e`.  
`compile dyn e` is a syntax error
- We have two assertions `assert_true` and `assert_false`.

They run one of the following tests. `assert_true` passes if the test succeeds, and fails otherwise. `assert_false` passes if the test *doesn't* succeed and fails otherwise.

- `aequiv_lam (e1, e2)` tests to see if the two lambda calculus terms `e1` and `e2` are  $\alpha$ -equivalent by calling the `Lambda.aequiv` function on them.

- `aequiv_ski (e1, e2)` does the same thing for the combinator calculus
- `aequiv_dyn (e1, e2)` does the same thing for  $\mathcal{L}\{Dyn\}$ .
- `reduce_ex e`. Here `e` must be a term in  $\mathcal{L}\{Dyn\}$ . This test will call `Dyn.reduce` on `e` and succeed if a `RuntimeError` is raised.
- `normalize_ex e`. This test will call `Dyn.normalize` on `e` and succeed if a `RuntimeError` is raised.

Here are some examples of assertions at work:

This:

```
assert_true (aequiv_lam (\x.x, \y.y));
```

gives: Ok because it is true that the two  $\lambda$ -terms are  $\alpha$ -equivalent.

whereas this:

```
assert_false (aequiv_lam (\x.x, \y.y));
```

gives: Assertion failed

As another example:

```
assert_true (reduce_ex ((\x.x) + 1));
```

gives: Ok because a `RuntimeError` is indeed thrown on reducing the  $\mathcal{L}\{Dyn\}$  term `x.x + 1`.

- `load "filename"` will read in REPL commands from the file `filename` and execute them in the current environment.

If the commands in the file can all be executed successfully (meaning there are no syntax errors) then all variable bindings created in `filename` will be added to the current environment. If not, then the REPL will print out the syntax error, and no new bindings will be created. Note that `Dyn.RuntimeErrors` do not cause loading to abort.

If the file you loaded had assertions in it, then after loading the REPL will print out a list of all *failed* assertions (if any).

The `TopLevel` structure provides another function you may find useful:

```
val evalFile : string -> (TopLevelCommands.assertions list * Environment.env)
```

`evalFile` is similar to the REPL's `load` command. It takes a filename as an argument, and executes the REPL commands in the file. If everything works, it returns an a list of all failed assertions in the file, and an environment with the value bindings specified in the file. If there is a syntax error in the file, it raises a `Parser.Error` exception.

## A $\lambda$ -calculus

### A.1 Syntax

Expression  $u ::= x$  variable  
 $\lambda(x) u$   $\lambda$ -abstraction  
 $u_1 u_2$  application

### A.2 Statics

$$\frac{}{\Gamma, x \text{ ok} \vdash x \text{ ok}} \text{ (VAR)} \quad \frac{\Gamma \vdash u_1 \text{ ok} \quad \Gamma \vdash u_2 \text{ ok}}{\Gamma \vdash u_1 u_2 \text{ ok}} \text{ (APP)} \quad \frac{\Gamma, x \text{ ok} \vdash u \text{ ok}}{\Gamma \vdash \lambda(x.u) \text{ ok}} \text{ (\lambda)}$$

### A.3 Equational Dynamics

$$\frac{}{\Gamma, u \text{ ok} \vdash u \equiv u} \text{ (\equiv REF)} \quad \frac{\Gamma, x \text{ ok} \vdash u \equiv u'}{\Gamma \vdash \lambda(x.u) \equiv \lambda(x.u')} \text{ (\equiv \lambda)} \quad \frac{\Gamma \vdash u \equiv u' \quad \Gamma \vdash u' \equiv u''}{\Gamma \vdash u \equiv u''} \text{ (\equiv TRANS)}$$

$$\frac{\Gamma \vdash u' \equiv u}{\Gamma \vdash u \equiv u'} \text{ (\equiv SYM)} \quad \frac{\Gamma \vdash e_1 \equiv e'_1 \quad \Gamma \vdash e_2 \equiv e'_2}{\Gamma \vdash e_1 e_2 \equiv e'_1 e'_2} \text{ (\equiv APP)} \quad \frac{\Gamma, x \text{ ok} \vdash e_2 \text{ ok} \quad \Gamma \vdash e_1 \text{ ok}}{\Gamma \vdash \lambda(x.e_2) e_1 \equiv [e_1/x]e_2} \text{ (\equiv \beta)}$$

## B $\mathcal{L}\{Dyn\}$

### B.1 Syntax

Expression  $u ::= x$  variable  
 $\lambda(x) u$   $\lambda$ -abstraction  
 $u_1 u_2$  application  
 $\bar{n}$  natural numbers  
 $u_1 + u_2$  addition  
 $ifz(u_1, u_2, u_3)$  branch on zero  
 $nil$  nullary tuple  
 $cons(u_1, u_2)$  binary tuples  
 $cond(u_1, u_2, u_3)$  branch on  $nil$   
 $car(u)$  first projection  
 $cdr(u)$  second projection  
 $isnil(u)$  test if  $u$  is in class  $nil$   
 $iscons(u)$  test if  $u$  is in  $cons$

## B.2 Statics

$$\begin{array}{c}
\frac{}{\Gamma, x \text{ ok} \vdash x \text{ ok}} \quad \frac{\Gamma \vdash u_1 \text{ ok} \quad \Gamma \vdash u_2 \text{ ok}}{\Gamma \vdash u_1 u_2 \text{ ok}} \quad \frac{\Gamma, x \text{ ok} \vdash u \text{ ok}}{\Gamma \vdash \lambda(x.u) \text{ ok}} \\
\\
\frac{n \in \mathbb{N}}{\Gamma \vdash \bar{n} \text{ ok}} \quad \frac{\Gamma \vdash u_1 \text{ ok} \quad \Gamma \vdash u_2 \text{ ok}}{\Gamma \vdash u_1 + u_2 \text{ ok}} \quad \frac{\Gamma \vdash u_1 \text{ ok} \quad \Gamma \vdash u_2 \text{ ok} \quad \Gamma \vdash u_3 \text{ ok}}{\Gamma \vdash \text{ifz}(u_1, u_2, u_3) \text{ ok}} \\
\\
\frac{}{\Gamma \vdash \text{nil} \text{ ok}} \quad \frac{\Gamma \vdash u_1 \text{ ok} \quad \Gamma \vdash u_2 \text{ ok}}{\Gamma \vdash \text{cons}(u_1, u_2) \text{ ok}} \quad \frac{\Gamma \vdash u_1 \text{ ok} \quad \Gamma \vdash u_2 \text{ ok} \quad \Gamma \vdash u_3 \text{ ok}}{\Gamma \vdash \text{cond}(u_1, u_2, u_3) \text{ ok}} \\
\\
\frac{\Gamma \vdash u \text{ ok}}{\Gamma \vdash \text{car}(u) \text{ ok}} \quad \frac{\Gamma \vdash u \text{ ok}}{\Gamma \vdash \text{cdr}(u) \text{ ok}} \quad \frac{\Gamma \vdash u \text{ ok}}{\Gamma \vdash \text{isnil}(u) \text{ ok}} \quad \frac{\Gamma \vdash u \text{ ok}}{\Gamma \vdash \text{iscons}(u) \text{ ok}}
\end{array}$$

## B.3 Reduction

First we have a value judgement for terms that cannot reduce any further:

$$\frac{}{\lambda x.e \text{ val}} \quad \frac{}{\bar{n} \text{ val}} \quad \frac{}{\text{nil} \text{ val}} \quad \frac{}{\text{cons}(e_1, e_2) \text{ val}}$$

Then we have some judgements to check the class of a value:

$$\begin{array}{c}
\frac{}{\lambda x.e \text{ is\_fun } x.e} \quad \frac{}{\bar{n} \text{ isnt\_fun}} \quad \frac{}{\text{nil} \text{ isnt\_fun}} \quad \frac{}{\text{cons}(e_1, e_2) \text{ isnt\_fun}} \\
\\
\frac{}{\bar{n} \text{ is\_num } n} \quad \frac{}{\lambda x.e \text{ isnt\_num}} \quad \frac{}{\text{nil} \text{ isnt\_num}} \quad \frac{}{\text{cons}(e_1, e_2) \text{ isnt\_num}} \\
\\
\frac{}{\text{nil} \text{ is\_nil}} \quad \frac{}{\bar{n} \text{ isnt\_nil}} \quad \frac{}{\lambda x.e \text{ isnt\_nil}} \quad \frac{}{\text{cons}(e_1, e_2) \text{ isnt\_nil}} \\
\\
\frac{}{\text{cons}(M_1, M_2) \text{ is\_cons } \langle M_1, M_2 \rangle} \quad \frac{}{\bar{n} \text{ isnt\_cons}} \quad \frac{}{\lambda x.e \text{ isnt\_cons}} \quad \frac{}{\text{nil} \text{ isnt\_cons}}
\end{array}$$

Finally, we can define reduction:

$$\begin{array}{c}
\frac{M \text{ is\_fun } x.M'}{M N \Rightarrow [N/x]M'} \quad \frac{M \text{ isnt\_fun}}{M N \text{ err}} \quad \frac{M \Rightarrow M'}{M N \Rightarrow M' N} \\
\\
\frac{M_1 \Rightarrow M'_1}{M_1 + M_2 \Rightarrow M'_1 + M_2} \quad \frac{M_1 \text{ is\_num } n \quad M_2 \Rightarrow M'_2}{M_1 + M_2 \Rightarrow M_1 + M'_2} \quad \frac{M_1 \text{ is\_num } n_1 \quad M_2 \text{ is\_num } n_2}{M_1 + M_2 \Rightarrow n_1 + n_2} \\
\\
\frac{M_1 \text{ isnt\_num}}{M_1 + M_2 \text{ err}} \quad \frac{M_2 \text{ isnt\_num}}{M_1 + M_2 \text{ err}} \\
\\
\frac{M_1 \Rightarrow M'_1}{\text{ifz}(M_1, M_2, M_3) \Rightarrow \text{ifz}(M'_1, M_2, M_3)} \quad \frac{M_1 \text{ is\_num } 0}{\text{ifz}(M_1, M_2, M_3) \Rightarrow M_2} \quad \frac{M_1 \text{ is\_num } n \quad n > 0}{\text{ifz}(M_1, M_2, M_3) \Rightarrow M_3} \\
\\
\frac{M_1 \text{ isnt\_num}}{\text{ifz}(M_1, M_2, M_3) \text{ err}} \\
\\
\frac{M_1 \Rightarrow M'_1}{\text{cond}(M_1, M_2, M_3) \Rightarrow \text{cond}(M'_1, M_2, M_3)} \quad \frac{M_1 \text{ is\_nil}}{\text{cond}(M_1, M_2, M_3) \Rightarrow M_3} \quad \frac{M_1 \text{ isnt\_nil}}{\text{cond}(M_1, M_2, M_3) \Rightarrow M_2} \\
\\
\frac{M \Rightarrow M'}{\text{car}(M) \Rightarrow \text{car}(M')} \quad \frac{M \text{ is\_cons } \langle M_1, M_2 \rangle}{\text{car}(M) \Rightarrow M_1} \quad \frac{M \text{ isnt\_cons}}{\text{car}(M) \text{ err}} \\
\\
\frac{M \Rightarrow M'}{\text{cdr}(M) \Rightarrow \text{cdr}(M')} \quad \frac{M \text{ is\_cons } \langle M_1, M_2 \rangle}{\text{cdr}(M) \Rightarrow M_2} \quad \frac{M \text{ isnt\_cons}}{\text{cdr}(M) \text{ err}}
\end{array}$$

For the predicates below, the term  $\overline{42}$  is completely arbitrary since these predicates are meant to be used with *cond*. Any non-*nil* term could be used there.

$$\begin{array}{c}
\frac{M \Rightarrow M'}{\text{isnil}(M) \Rightarrow \text{isnil}(M')} \quad \frac{M \text{ is\_nil}}{\text{isnil}(M) \Rightarrow \overline{42}} \quad \frac{M \text{ isnt\_nil}}{\text{isnil}(M) \Rightarrow \text{nil}} \\
\\
\frac{M \Rightarrow M'}{\text{iscons}(M) \Rightarrow \text{iscons}(M')} \quad \frac{M \text{ is\_cons } \langle M_1, M_2 \rangle}{\text{iscons}(M) \Rightarrow \overline{42}} \quad \frac{M \text{ isnt\_cons}}{\text{iscons}(M) \Rightarrow \text{nil}}
\end{array}$$

Finally, we have some rules to propagate errors upwards through the syntax tree. You do not need to code these rules in your `reduce` function, since throwing a `RuntimeError` takes care of this automatically. We present these rules here only for completeness.

$$\begin{array}{cccccc}
\frac{M \text{ err}}{\lambda x.M \text{ err}} & \frac{M_1 \text{ err}}{M_1 M_2 \text{ err}} & \frac{M_2 \text{ err}}{M_1 M_2 \text{ err}} & \frac{M_1 \text{ err}}{M_1 + M_2 \text{ err}} & \frac{M_2 \text{ err}}{M_1 + M_2 \text{ err}} & \frac{M_1 \text{ err}}{\text{ifz}(M_1, M_2, M_3) \text{ err}} \\
\frac{M_2 \text{ err}}{\text{ifz}(M_1, M_2, M_3) \text{ err}} & \frac{M_3 \text{ err}}{\text{ifz}(M_1, M_2, M_3) \text{ err}} & \frac{M_1 \text{ err}}{\text{cons}(M_1, M_2) \text{ err}} & \frac{M_2 \text{ err}}{\text{cons}(M_1, M_2) \text{ err}} & & \\
\frac{M_1 \text{ err}}{\text{cond}(M_1, M_2, M_3) \text{ err}} & \frac{M_2 \text{ err}}{\text{cond}(M_1, M_2, M_3) \text{ err}} & \frac{M_3 \text{ err}}{\text{cond}(M_1, M_2, M_3) \text{ err}} & \frac{M \text{ err}}{\text{car}(M) \text{ err}} & \frac{M \text{ err}}{\text{cdr}(M) \text{ err}} & \\
\frac{M \text{ err}}{\text{isnil}(M) \text{ err}} & & \frac{M \text{ err}}{\text{iscons}(M) \text{ err}} & & & 
\end{array}$$