University of Toronto

CSC 324F Programming Languages — **Fall 1996**

Assignment 1 [10% of mark] — **Out: October 8, 1996**
Due: October 22, 1996, 11:59 p.m.

Objective

In this project, you will learn how to evaluate arbitrary λ-expressions (both pure and applied) using S and K combinators.

In particular, we will supply you with a function `compile` that translates λ-expressions into SKI expressions. You will write Scheme functions `reduce-pure` and `reduce-applied` to interpret SKI expressions: `(reduce-pure (compile` E`))` will be the result of evaluating E in the context of the pure λ-calculus; `(reduce-applied (compile` E`))` will be the result of evaluating E in the context of an applied λ-calculus.

Background

A *combinator* is a λ-expression without free variables. Here are three combinators:

$$\begin{aligned} \mathsf{I} &= \lambda x.x \\ \mathsf{S} &= \lambda x.\lambda y.\lambda z.xz(yz) \\ \mathsf{K} &= \lambda x.\lambda y.x \end{aligned}$$

This project is based on $\mathsf{S\,K}$-combinator theory, invented as a basis for mathematical logic in the 1920's by Schöfinkel and independently in the 1930's by Curry. (Remember currying - a technique for producing functions taking just one argument from functions taking two or more arguments? That's the same guy!) The $\mathsf{S\,K}$-combinator theory defines computation over (possibly infinite) input strings using only *two* operators: S and K. However, for this project we will use three operators, S, K and I, and thus talk about SKI expressions.

Here's the definition of SKI expressions.

- Any Scheme symbol or number is a SKI expression.
- If E is a SKI expression, then so is (E).
- Symbols S, K, and I, are SKI expressions.
- If E and F are SKI expressions, then so is $E\,F$.

Nothing else is an SKI expression.

The last rule shows how to write function application — it is the same as in λ-calculus. As before, application associates to the left, so that $E\,F\,G$ is equal to $(E\,F)\,G$, which is not always the same as $E\,(F\,G)$.

Given an SKI expression, we may rewrite its *prefix*[1] as follows:

$$\begin{aligned} (E) &\mapsto E \\ \mathsf{S}\,F\,G\,X &\mapsto F\,X\,(G\,X) \\ \mathsf{K}\,X\,Y &\mapsto X \\ \mathsf{I}\,X &\mapsto X \end{aligned}$$

[1] A prefix of string α is an initial substring of α.

Here, E, F, G, X, and Y stand for arbitrary SKI subexpressions. [Note that $\mathsf{K}\,b$ is *not* a subexpression of $\mathsf{S\,K}\,b\,c$ because of left-associativity: $\mathsf{S\,K}\,b\,c = (((\mathsf{S\,K})\,b)\,c)$.]

The first rule means parentheses can be dropped when not necessary.

The second rule means that the S combinator expresses function composition (F composed with G), passing argument X to both functions (and therefore duplicating information). You can prove this rewrite rule for yourself using the definition of S.

The third rule means that operator K expresses both preservation and destruction of information — it preserves the first argument, removes the second, and preserves all expressions thereafter. Since the application associates on the left,

$$\begin{array}{rcl} \mathsf{K}\,X\,Y\,Z & = & (\mathsf{K}\,X\,Y)\,Z \\ & \mapsto & X\,Z \end{array}$$

Finally, I is the identity combinator. I has the same effect as S K K:

$$\begin{array}{rcl} \mathsf{S\,K\,K}\,x & \mapsto & \mathsf{K}\,x\,(\mathsf{K}\,x) \\ & \mapsto & x \end{array}$$

So, I is not really necessary but is used anyway to make expressions more concise. These expressions can be used to express any computable function from strings to strings. For example, expression S I I makes a duplicate of its first argument:

$$\begin{array}{rcl} \mathsf{S\,I\,I}\,x & \mapsto & \mathsf{I}\,x\,(\mathsf{I}\,x) \\ & \mapsto & x\,(x) \\ & \mapsto & x\,x \end{array}$$

Notice that we reduced the first subexpression in parentheses, I x, just as if it were a top level expression. In another example, the expression $\mathsf{S\,(S\,I\,I)\,I}\,x$ reduces to three copies of x:

$$\begin{array}{rcl} \mathsf{S\,(S\,I\,I)\,I}\,x & \mapsto & \mathsf{S\,I\,I}\,x\,(\mathsf{I}\,x) \\ & \mapsto & \mathsf{I}\,x\,(\mathsf{I}\,x)\,(\mathsf{I}\,x) \\ & \mapsto & x\,x\,x \end{array}$$

Project Description

It is a remarkable fact that there is a function `compile` which takes an arbitrary (pure or applied) λ-calculus expression and converts it into an equivalent SKI expression. For example,

```
(compile '(\ x x))
      ->  i

(compile '((\ x x x) 4))
      -> (s i i 4)
```

The syntax of lambda terms is as usual, except for the following differences.

1. Applications at the top level must be surrounded by parentheses. For example, write $E\,F$ as $(E\,F)$. But you may still write $E\,F\,G$ as $(E\,F\,G)$. Remember that application is left-associative, so that $(E\,F\,G)$ has the same value as $((E\,F)\,G)$.
2. Use a backslash \ in place of the Greek letter λ. (The backslash is supposed to look a bit like λ.)
3. Spaces must appear between the lambda and the variable being abstracted.
4. Spaces must appear between the variable being abstracted and the body. Do not use a dot to separate them.

5. Abstractions must be surrounded by parentheses, unless they appear as the body of another abstraction.

Here are some examples of this syntax:
Write $\lambda x. + x\, x$ as `(\ x + x x)`.
Write $\lambda x.\lambda y.x$ as `(\ x (\ y x))` or as `(\ x \ y x)`
Write $\lambda x.x(\lambda y.x)$ as `(\ x x (\ y x))`.

Your goal is to write functions that apply SKI rewrite rules to reduce any SKI expression to its "normal" form (i.e., the expression cannot be reduced any further). When coupled with the `compile` function, you will have an evaluator for the (pure or applied) λ-calculus.

In Scheme, we will represent top-level SKI applications with lists. For example, the application S K K is represented in Scheme by `(s k k)`. Note that in the ordinary SKI rules, an application that is itself an argument in an enclosing application must always be surrounded by parentheses in order to override left-associativity.

Stage 1

Implement the rewrite rules. We have four rewrite rules, and thus there will be four functions.

1. Function `reduce-paren` removes unnecessary parentheses around the prefix of its argument. If its argument is an SKI expression that is not an application, i.e. its argument is a list of one element, then it removes those parentheses:

   ```
   (reduce-paren '((X ...) E ...))
         -> (X ... E ...)
   (reduce-paren '(x))
         -> x
   ```

   It will only be called when its argument actually has such unnecessary parentheses. Example:

   ```
   (reduce-paren '((a b c d) a b))
         -> (a b c d a b)
   (reduce-paren '(42))
         -> 42
   ```

2. Function `reduce-i` will always be given a list of at least two elements, the first of which is the name `i`. It returns the result of applying the I rewrite rule:

   ```
   (reduce-i '(i x ...))
         -> (x ...)
   ```

   Example:

   ```
   (reduce-i '(i x k s y))
        -> (x k s y)
   ```

3. Function `reduce-k` will always be given a list of at least three elements, the first of which is the name `k`. It returns the result of applying the K rewrite rule:

   ```
   (reduce-k '(k x y ...))
       -> (x ...)
   ```

   Example:

   ```
   (reduce-k '(k (x 2) q z f))
       -> ((x 2) z f)
   ```

4. Function `reduce-s` will always be given a list of at least four elements, the first of which is the name `s`. It returns the result of applying the S rewrite rule:

```
(reduce-s '(s f g x ...)
   -> (f x (g x) ...)
```

Example:

```
(reduce-s '(s s i x i x))
   -> (s x (i x) i x)
```

Stage 2

Write function `reduce-pure` which reduces an arbitrary SKI to its normal form in the context of the pure λ-calculus. These are some of the auxiliary functions for `reduce-pure`:

1. Function `reduce-pure-outer` is given an SKI expression as its one argument. If none of the rewrite rules apply to its argument at the top level, then it returns its argument unchanged. Otherwise, it returns the result of applying the appropriate rewrite rule (i.e. one of `reduce-paren`, `reduce-i`, `reduce-k`, or `reduce-s`). For example, if its argument is a list of at least two elements, the first of which is the symbol `i`, then it returns the result of passing its argument to the `reduce-i` function. Examples:

```
(reduce-pure-outer '(k (i y) x z (i 4)))
      -> ((i y) z (i 4))
(reduce-pure-outer '((x y)))
      -> (x y)
(reduce-pure-outer '(s x))
      -> (s x)
```

   In the last example, the S rule could not be applied because there weren't enough arguments to the S function.

2. Function `reduce-pure-whnf` calls `reduce-pure-outer` until no more rules can be applied. The expression is now in what is called "Weak Head Normal Form".
   Hint: No more reductions are possible if result of `reduce-pure-outer` for some expression `E` equals `E`. Use operation `equal?` to test for equality. So, you need to stop when

```
(equal? (reduce-pure-outer E) E)
```

   Example:

```
(reduce-pure-whnf '(k i (x y) z (i 42)))
     -> (z (i 42))
```

3. Finally, function `reduce-pure` applies `reduce-pure-whnf` to its argument. If the result is a list, then it applies itself to all the elements of that list. If the result was not a list, then that is returned. The resulting value is in "Normal Form". Example:

```
(reduce-pure '(s (s i i) i x))
    -> (x x x)
```

As described earlier, `(reduce-pure (compile` E`))` for any lambda expression E will be the normal form of E if one exists. (However, if the normal form of E still contains λ abstractions, then they will be expressed in SKI form.)

Stage 3

We will now extend our λ-calculus evaluator to an applied λ-calculus by adding rewrite rules for some constants. The list of constants is: **if**, **iszero**, **pred**, **succ**, **fix**, and **plus**. We also have constants **true**, and **false**, although they have no rewrite rules associated with them. (Note that these are the same constants as described in Section 14.1 of Sethi's book, except that we've added **plus** as a primitive operation for the sake of efficiency.) The extensions will build a function `reduce-applied` which evaluates an applied SKI expression in the context of this applied λ-calculus.

First, you will write functions that implement reductions for these constants.

- Function `reduce-if` will always be given a list of at least four elements: `(if` *cond* ***then-branch else-branch*** `...)`. It reduces *cond* to normal form (using `reduce-applied`). If the result is the constant `true`, then the list `(`***then-branch*** `...)` is returned. On the other hand, if the result is the constant `false`, then the list `(`***else-branch*** `...)` is returned. Function `reduce-if` does *not* reduce either *then-branch* or *else-branch*.

  ```
  (reduce-if '(if true x y z))
        -> (x z)
  (reduce-if '(if (i false) t (i e) foo bar)
        -> ((i e) foo bar)
  ```

  In the first example, **if** chose the "then" branch and appended the rest of the expression (i.e., `z`) to it. In the second example, it had to reduce the conditional to normal form before seeing that it had value `false`; it then chose the "else" branch.

- Function `reduce-iszero` takes a list of at least two arguments `(iszero` ***expr*** `...)`. It reduces *expr* to normal form (using `reduce-applied`). If the result is the number 0, then the list `(true ...)` is returned. Otherwise, the list `(false ...)` is returned. (Note that the rest of the list, i.e. ... may be empty.) For example,

  ```
  (reduce-iszero '(iszero 0 x))
        -> (true x)
  (reduce-iszero '(iszero (i 0))
        -> (true)
  (reduce-iszero '(iszero 64))
        -> (false)
  (reduce-iszero '(iszero s))
        -> (false)
  ```

- Function `reduce-pred` takes a list of at least two arguments `(pred` ***expr*** `...)`. It reduces *expr* to normal form and and returns the list `(`***expr-1*** `...)`, where *expr-1* is the result of subracting 1 from the normal form of *expr*. The result is undefined if the normal form of *expr* is not a number.

  ```
  (reduce-pred '(pred 2 x))
        -> (1 x)
  (reduce-pred '(pred (i 42) x))
        -> (41 x)
  ```

- Function `reduce-succ` works the same way as `reduce-pred` but adds 1 to the second argument. For example,

  ```
  (reduce-succ '(succ 2 x))
        -> (3 x)
  ```

- Function `reduce-plus` takes a list of at least three arguments `(plus` ***exp1 exp2*** `...)`. It returns the list `(`***exp1+exp2*** `...)` where *exp1+exp2* is the sum of the normal forms of expressions *exp1* and *exp2*.

```
(reduce-plus '(plus 3 5 6))
      -> (8 6)
(reduce-plus '(plus (plus (i 3) (i 2)) 1)
      -> (6)
```

- Function `reduce-fix` takes a list of at least two elements `(fix` f `...)` and returns `(`f `(fix` f`) ...)`. Important: `reduce-fix` does *not* evaluate f! Example:

```
(reduce-fix '(fix (if true 1) y))
     -> ((if true 1) (fix (if true 1)) y)
```

Second, you will write functions `reduce-applied-outer`, `reduce-applied-whnf`, and `reduce-applied` which are modifications of `reduce-pure-outer`, `reduce-pure-sequence`, and `reduce-pure`, respectively.

- Function `reduce-applied-outer` is given an SKI expression as its one argument. If none of the pure or applied rewrite rules apply to its argument at the top level, then it returns its argument unchanged. Otherwise, it returns the result of applying the appropriate rewrite rule.

```
(reduce-applied-outer '(succ 1 (i x))
     -> (2 (i x))
(reduce-applied-outer '(4))
     -> 4
(reduce-applied-outer '42)
     -> 42
```

- Function `reduce-applied-whnf` applies `reduce-applied-outer` to its argument until a fixed point is reached, i.e. calling `reduce-applied-outer` on argument E results in a value equal to E.
- Function `reduce-applied` calls `reduce-applied-whnf` on its one argument. If the result is not a list, then that is returned. Otherwise, it calles itself on each of the elements of the that list. The result is in normal form.

Submission (electronic)

Submit a tar file with

1. Well-documented source code. All the code for the pure reducer should be in a file called `pure.s`. The extra code needed for the applied reducer should be in a file called `pure.s`. (For example, the definition of `reduce-s` should appear in `pure.s` but not in `applied.s`.)
2. Test program `test.s`. This will load `compile.s`, `pure.s` and and `applied.s` and then run and display test cases. We will supply you with the source file `compile.s` that contains the `compile` function. Include enough test cases to test your functions. Don't forget border cases.
3. Output showing the operation of your program.
4. (Optional) README file with instructions on how to run your program.

Note that we will rerun your program with, so please **do not change the output of your program!!!!!**

Grading

The marking scheme will be as follows:

Testing	10	(Has the program been tested accordingly?)
Style	25	(The program should be written using functional programming style.)
Documentation	10	(Explanation for each module and the entire program.)
Correctness	55	

David Neto will grade this assignment. He will have office hours on Friday October 11, 1996, from 4-5pm in SF3207, and on Monday October 21, 1996, from 3-4pm in SF3207.

Notes

- You may write any auxiliary functions you find necessary, or borrow any from `compile.s`.
- Severe penalties in style will be taken if you use assignments and loops. All processing should be done using purely functional paradigm, using recursion where appropriate.
- The minimum that you need to implement for your project to be considered "working" is to implement `reduce-pure` correctly.
- Code for `compile` and sample test cases will be posted separately.
- Read `ut.cdf.csc324h` for updated information, test cases, etc.

Acknowledgment

This project was suggested by David Neto. Implementation of function `compile` is also due to him.