Introduction

The following presents a Functional Programming Language named **SK**. The design is closely based on the St. Andrews Static Language (**SASL**) created by David Turner **[DT79]**.   
  
**SASL** influenced design of subsequent Functional Programming languages, such as **Miranda** and **Haskell**, with which **SK** shares syntactic and semantic features.

A language processor for the **SK** language has been implemented in **Common Lisp**.

Notational Conventions

Right arrow (⇒) is used to denote a replacement operation known as “Reduction”. The expression to its left is said to “reduce” to the expression to its right.

Expressions are built up or “Composed” from lists of Terms, as will be explained below. In the context of Reduction, Combinator Theory refers to such an Expression as a “Redex”.

As part of the meta-language used to describe **SK**, Relational Equality (**==**) and its complement (**!=**) appear in syntactic rules to indicate whether or not two meta-symbols σ1 and σ2 refer to the same symbol.

An Equal Sign (=) indicates that the Function appearing to its left is being defined to represent the expression appearing to its right.

An Equivalence Sign ≡ is used to indicate that the expressions to its left and right have equivalent reductions (or evaluations) and are therefore interchangeable with each other.

Matched open and close parentheses are only introduced when necessary to override associativity, as has been their traditional role in mathematics.

Definitions

Symbols: σ have alphanumeric names which begin with a letter.

Terms: τ are either a symbol or a Pair.

Pairs: π are formed via an Internal Function named **Pair**. An infix version of this function is called the “dot operator” (◦). Thus, a pair formed from terms τ1 and τ2 can be written:

τ1 ◦ τ2 ≡ **Pair** τ1 τ2

The first term of a Pair is referred to as its Head and the second term is referred to as its Tail.

The Pairing Operator is *right associative*:

τ1 ◦ τ2 ◦ τ3 ≡ τ1 ◦ (τ2 ◦ τ3)

Lists

An Empty List is represented by the symbol **nil**.

Creating a Pair (via the dot operator) prepends, or “pushes” the Head Element onto the front of an existing List, which becomes the Tail of the newly created List. A List is either the Empty List or a Pair whose Tail is another List.  
  
An Empty List has no elements, and has a Length of Zero. The length of a non-empty List is one greater than the length of its Tail.

Any Non-empty List will have a final Pair. The Head of this Pair will contain the final element, and its Tail is normally the Empty List, **nil**. Such **nil**-terminated lists are said to constitute “Proper Lists”.

If the tail of the last Pair is some symbol other than **nil**, that tail (and any longer list prepended onto it) will be referred to as a “Dotted List.” Note that two distinct Lists may share a common Tail.

Written Representation of Lists

In the same way that the multiplication operator is considered to be *implicit* between adjacent terms in algebra, the Pairing Operator is optional, and may be elided when lists of terms are read or written:

τ1 τ2 ≡ τ1 ◦ τ2 ◦ **nil**τ1 τ2 ◦ σ ≡ τ1 ◦ τ2 ◦ σ

Parentheses are required to distinguish a single-element list containing a term τ1 from the term τ1 itself:

(τ1) ≡ τ1 ◦ **nil**

An empty parenthesized expression is an alternative representation of the Empty List:

() ≡ **nil**

The Head of a Pair can be either a Symbol or a Term, so the elements of a List may contain other Lists.

In this case, parentheses are required to make associativity among the Pairs clear:

(τ1 τ2) τ3 τ4 ≡ (τ1 ◦ τ2 ◦ **nil**) ◦ τ3 ◦ τ4 ◦ **nil**

Abstraction

We designate a reserved keyword lambda (**λ**) to represent the Abstraction Operator.

Abstraction of a symbol from any expression produces a new expression, as follows:

**λ**x x ⇒ **I**

**λ**x y ⇒ **K** y *where x != y*

**λ**x (τ1 τ2) ⇒ **S** (**λ**x τ1) (**λ**x τ2)

While it is expected that implementations should allow use of the full Unicode character set, a Question Mark (**?**) will be accepted as a substitute for **λ** in contexts limited to use of more limited character sets.

Currying

An enhanced form of Abstraction known as Currying is defined, as follows:

**λnil** τ ⇒ K**nil** τ  
**λ** (head ◦ tail) *expr* ≡ **U** (**λ**head (**λ**tail *expr*))

Here, abstraction is performed over an Abstraction List, a.k.a., “Template”, instead of an atomic Symbol.

Note: **U** stands for an operation known as Uncurrying, which will be described in the next section.

A syntactic enhancement allows Abstraction Lists to be expressed using the following square bracket notation:

**[**head ◦ tail**]** *expr* ≡ **λ** (head ◦ tail) *expr***[**head**]** *expr* ≡ **λ** (head) *expr*

A pair of Square Brackets, can be used as an alternative representation for an Empty Abstraction List:

**[]** *expr* ≡ **λnil** *expr*

Uncurrying

The Uncurrying operator **U** reverses the effect of Currying:

**U** *expr* head ◦ tail ⇒ *expr* head tail  
**U** *expr* **nil** ⇒ *expr*

Though **U** is normally referred to as Uncurrying, we can just as well think of it as an “Unpairing” operation which takes a Functional Expression and a Pair as input, and applies the Functional Expression to the **Head** of the **Pair** before finally applying the result of that application to the **Tail** of the **Pair**.

**Note:** The effect of Uncurrying an Abstraction List is very similar to an operation that Lisp referred to as **destructuring-bind**.

Templates represented by an Abstraction must be isomorphic to the form of the input expression to which it is applied and is intended to “destructure”.

Application

Evaluation of expressions is also known as Beta-Reduction (β-Reduction); and is invoked as follows:

β *expr*

Reductions proceed from left to right; and are left associative. When terms appear next to each other, the first term is said to be applied to the second. The result of this reduction will then be applied to the next term, if a third term is present. Otherwise, the partially reduced result is returned.

τ1 τ2 τ3 ≡ (τ1 τ2) τ3

Reduction

Reduction of terms:

**S** τ1 τ2 τ3 ⇒ τ1 τ3 (τ2 τ3)

**K** τ1 τ2 ⇒ τ1

**Knil**, introduced by Currying, works similarly. However, **Knil** τ1 τ2 reduces to τ1 *if and only if* τ2 == **nil**:

**Knil** τ1 **nil** ⇒ τ1

**I** τ ⇒ τ

**Y** τ ⇒ τ (**Y** τ)

Function Definition

Conceptually, Function Definition can be implemented by the Lambda Expression:

*func* x = *expr* ≡ *func* = **λ**x *expr*

Functions are allowed to make recursive reference to themselves. This works because Function Definition is actually implemented using the **Y** operator:

*func* = *expr* ≡ *func* = **Y** (**λ***func* *expr*)

Primitive Functions

All Functions, including **I** and **Y**, can be defined in terms of the two Primitive Functions **S** and **K**:

**I** = **S** **K** **K**

Define a Function named Z:

Z z h = h (z z h)

Using Z, we can implement YTuring which is equivalent to **Y**:

YTuring = Z Z  
  
Reserved Keywords

In the discussion above, a number of “reserved keywords” were introduced:  
  
**I** **S** **K** **Knil** **λ nil** **U** **Y**  
  
Reserved Keywords appear in bold face, owing to their significance in the underlying implementation of the language. While beyond the scope of this brief overview, additional support for Numeric and String literals will be provided.  
  
References

**[DT79]** "A New Implementation Technique for Applicative Languages" by David A. Turner, 1979,

Software-Practice and Experience [vol.9, pp.31-49] John Wiley & Sons, Ltd.