Introduction

The following presents a Functional Programming Language named **SK**. The design was influenced by the St. Andrews Static Language (**SASL**) created by David Turner **[DT79]**. **SASL** has been a source of influence for Functional Programming languages, such as **Miranda** and **Haskell**, with which **SK** has many features in common.

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

Notational Conventions

The Double Right Arrow (⇒) denotes 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 described below.

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

An Equal Sign (=) indicates that the symbol appearing to its left is being given a “Function Definition” 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

The Pair (or dot) operator results in a new List where the Head Expression is prepended (or “pushed”) 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. Empty Lists are represented by the symbol **nil**.

An Empty List has a Length of Zero. The length of a non-empty List is one greater than the length of its Tail. Distinct Lists may share a common Tail.

Non-empty Lists will have a final Pair. The Head of this Pair contains the “last” element of the list. If the Tail of the final Pair is **nil**, the List is said to be **nil**-terminated. **nil**-terminated lists are called “Proper Lists”. If the tail of the final Pair is any symbol other than **nil**, the list is called a “Dotted List.”

Written Representation of Lists

In the same way that the multiplication operator is considered *implicit* between adjacent terms in an algebraic expression, the Pairing Operator is implicit, and may be elided from lists of terms when they 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 pair of parentheses 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)

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 Abstraction Lists, a.k.a., “Templates”, instead of atomic Symbols.

Note: **U** stands for an operation, described in the next section, which is called Uncurrying.

A syntactic enhancement for Abstraction Lists allows them 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** refers to Uncurrying, we might just as well say that it refers to an “Unpairing” operation that takes a Functional Expression and a Pair as input, and applies the Functional Expression to the **Head** of the **Pair** before applying the expression resulting from that application to the **Tail** of the **Pair**.

Uncurrying an Abstraction List is very similar to an operation referred to as **destructuring-bind** in Lisp.

**Note:** Abstraction Templates must be isomorphic to the expressions they are used to destructure.

Application

Evaluation of expressions (a.k.a., β-Reduction) is invoked as follows:

β *expr*

Reductions proceed from left to right; and are left associative. When two terms appear next to each other, the first term is said to be applied to the second. The result of this reduction is then applied to the next term. Combinator Theory refers to the result of reducing one sub-expression to another as a “Redex”. If sufficient arguments are not provided, it may not be possible to fully reduce an expression.

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

The following is a list of “reserved keywords” for the **SK** language introduced in preceding sections:  
  
**I** **S** **K** **Knil** **λ nil** **U** **Y**  
  
Reserved Keywords appear in bold face, to distinguish them from ordinary Function Definitions. These symbols are defined in a separate package (named SK.) Though beyond the scope of this brief overview, additional support for Numeric and String literals is also provided.  
  
Note: Implementations should support use of the Unicode character for the Greek Letter **λ**. In contexts limited to more limited character sets, Question Mark (**?**) will be accepted as a substitute for **λ**.

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.