Introduction

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

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

Conventions

A right arrow (⇒) is used to denote a “Reduction” which is essentially a replacement operation. This means that the expression template on the left can be reduced to the expression template on the right.

The Relational Equality operator (**==**) and its logical complement (**!=**) will be used to distinguish cases whether two symbols σ1 and σ2 are identical to each other.

A simple Equal Sign (=) will be used to indicate that a Function expression appearing to the left is being defined to represent the expression to its right.

The equivalence sign ≡ is used to indicate that the expressions to the left and to the right are operationally equivalent to each other. The two expressions have the same behavior and are interchangeable with each other.

Matched Open and Close parentheses will be used to fill their traditional mathematical role – to override associativity. They are only introduced when necessary to ensure proper order of evaluation.

Definitions

Symbols: σ are represented by alphanumeric names which begin with a letter.

Terms: τ are either a symbol or a Pair.

Pairs: π are formed via an infix Pairing Operator (◦) also known as the “dot operator”. A pair formed from terms τ1 and τ2 is written:

τ1 ◦ τ2

The Pairing Operator is *right associative*:

τ1 ◦ τ2 ◦ τ3 ≡ τ1 ◦ (τ2 ◦ τ3)

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

Lists

The Empty List is represented by the symbol **nil**. The Empty List has no elements and is said to have a Length of Zero.

An Element may be prepended to the front of an existing List to make a longer List, by introducing a Pair (via the dot operator) whose Head is the new Element whose Tail is the existing List.

Thus, a List is either the Empty List or a Pair whose Tail is another List. The length of any non-empty List is one greater than the length of its Tail.

In a non-empty List, there will exist a final Pair, the head of which will contain the final element and the tail of which is the Empty List, represented by **nil**. Such “Proper Lists” are thus said to be **nil**-terminated.

An exceptional case can arise when the second element of the final Pair is some symbol other than **nil**. Such a Pair, and any Elements prepended onto them, are referred to as “Dotted Lists”.

Written Representation of Lists

The multiplication operator is often implied among adjacent terms in algebra. Similarly, the “dot” operator may be elided when reading or writing a List. Though invisible, it is still present; and can be optionally made explicit on input.

τ1 ◦ τ2 ◦ σ ≡ τ1 τ2 ◦ σ

The Empty List contains no elements and need not be written explicitly following a final element.

τ1 ◦ τ2 ◦ **nil** ≡ τ1 τ2

Because the Head of a Pair may be either a Symbol or a Term, Lists may contain other Lists as elements.

Lists are comprised of Pairs. Parentheses may be required to make associativity among the Pairs clear:

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

Parentheses are used to distinguish between a term τ1 and the List containing that term:

(τ1) ≡ τ1 ◦ **nil**

Two Parentheses can be used as an alternative representation of the Empty List:

() ≡ **nil**

Composition

Lists (and their constituent Pairs) provide for composition of more complex terms, built up from simpler, constituent terms.

Though a List of terms is fundamentally a Pair, we will use also the terminology Expression: *expr* to refer to such Lists.

Abstraction

We designate the special symbol lambda **λ** to represent an Abstraction Operator.

Abstraction of a symbol from any term produces a term, as follows:

**λ**x x ⇒ I

**λ**x y ⇒ K 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:

**λ** (head ◦ tail) *expr* ≡ U (**λ**head (**λ**tail *expr*))

**λnil** τ ⇒ K**nil** τ  
  
In these cases, abstraction is performed using a “template” which is passed using a List, rather than an atomic Symbol. These templates are referred to as an Abstraction Lists.

Note: The U operator stands for Uncurrying which will be described in the next section.

An experimental “syntactic enhancement” allows currying to be expressed using the following square bracket notation:

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

As with Parentheses, Square Brackets can be used to distinguish between a term τ1 and the Abstraction List containing that term:

**[**τ**]** *expr* ≡ **λ** (τ) *expr*

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

**[]** *expr* ≡ **λnil** *expr*

Uncurrying

The Uncurrying operator U reverses the effect of Currying:

U *func* head ◦ tail ⇒ *func* head tail

Though U is normally referred to as Uncurrying, we might as well think of it as an “Unpairing” operation which takes a Functional Expression and a Pair as input, then 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 applying a Curried Abstraction to an expression is very similar to an operation that Lisp referred to as **destructuring-bind**.

The template represented by an Abstraction must match the form of the input expressions to which it is applied and which it is intended to “destructure”.

Application

Evaluation of an expression is performed in using Beta Reduction, which is invoked as follows:

beta *expr*

Applications are left associative. When two terms appear next to each other, the first term is said to be applied to the second:

τ1 τ2 τ3 ≡ (τ1 τ2) τ3

Reduction

Reduction of terms:

S τ1 τ2 τ3 ⇒ τ1 τ3 (τ2 τ3)

K τ1 τ2 ⇒ τ1

A similar Function, introduced by Currying, only reduces to τ1 where τ2 is **nil**:

K**nil** τ1 **nil** ⇒ τ1

I τ ⇒ τ

Y τ ⇒ τ (Y τ)

Function Definition

A Function: *func* is defined in terms of Lambda Expressions:

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

Functions may make recursive reference to themselves:

*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

If we define a Function named Z:

Z z h = h (z z h)

Then we can define Y in terms of Z:

Y**Turing** = Z Z

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.