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].

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

Conventions

A right arrow ⇒ is used to denote a “reduction” operation which is essentially a replacement operation. The language processor transforms the expression template on the left into the expression template on the right.

A traditional equal sign = is used in definitions to indicate that the expression to the left is being defined to be the expression on the right.  
  
The equivalence (or identical) 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 reduction.

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 “dot” operator. The pair formed from terms τ1 and τ2 is written:

τ1 . τ2

The dot operator composes pairs using *right associativity*:

τ1 . τ2 . τ3 ≡ τ1 . (τ2 . τ3)

Note: Turner preferred use of a more legible “colon” to the “dot”. We keep to use of the dot here, following its more traditional use by the **Lisp** family of languages.  
  
The first term in a Pair is referred to as its Head and the second term will be referred to as its Tail.

## Lists

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

A Proper List is either the Empty List or a Pair whose Tail is another List. The Head can be any Term (including some other List) which is said to be an Element of the List.

There is an exceptional case where the second element of a Pair can be some symbol other than **nil**. In this case, the Pair and all of its prefixes are said to be “Dotted Lists”. However, when considering Proper Lists, the second element of the final Pair will be **nil**. Proper Lists are thus also said to be nil-terminated.

Printed Representation of Lists

To write any sequence of pairs (whether they may turn out to represent a proper, nil-terminated list or an improper dotted list) one begins by establishing a reference to the “remaining sub-list” of elements. Then the following steps are performed, iteratively:  
  
1) If the remaining sub-list is the symbol nil, writing of the list is complete. [If the remaining sub-list is some other non-nil symbol, writing of the list completes after writing a “dot” followed by writing the non-nil symbol used to terminate the dotted list.]

2) The remaining sub-list must otherwise be a pair; and the head of that sub-list is written. If the head is a symbol, the name of that symbol is written out.

If the head itself represents a list, an open parenthesis is written, followed a recursive application of this method to write out all of the elements of the nested sub-list, followed by a close parenthesis. The parentheses are required to make it clear that the nested list is one of the elements of its parent list.  
  
3) The reference to the “remaining sub-list” of elements is now replaced with the second element of the pair; and the process of writing each sub-list returns to the first step.

For example, the list entered as τ1 . τ2 . nil will be written as:

τ1 τ2

And the dotted list entered as τ1 . τ2 . τ3 will be written as:

τ1 τ2 . τ3

Expressions

An expression: expr is a list of terms or abstractions.

Application

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

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

τ1 τ2 τ3 ≡ (τ1 τ2) τ3

Abstraction

Abstraction of a symbol from a term, produces a term:

λx x ⇒ I

λx y ⇒ K y

λx (τ1 τ2) ⇒ S (λx τ1) (λx τ2)

Currying

Square brackets are “syntactic sugar” for an enhanced form of Abstraction known as Currying:

[head . tail]expr = λ(head . tail) expr  
  
λ(head . tail) τ = U (λhead (λtail τ))  
  
λnil τ ⇒ Knil τ

Note: Currying behaves similarly to the Lisp **destructuring-bind** operation.

Uncurrying

The Uncurrying Combinator U implements (or “realizes”) the semantics of Currying:

U *func* (head . tail) ⇒ *func* head tail

Reduction

Reduction of terms:

S τ1 τ2 τ3 ⇒ τ1 τ3 (τ2 τ3)

K τ1 τ2 ⇒ τ1

A similar combinator, introduced by Currying, only reduces where τ2 is nil:

Knil τ1 nil ⇒ τ1

I τ ⇒ τ

Y τ ⇒ τ (Y τ)

Function Definition

(Recursive) Functions: *func* are defined in terms of expressions.

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

def *func* x expr ≡ def *func* = λx expr

Note that I and Y can be defined in terms of S and K:

I ≡ S K K

And given:

def Z z h = h (z z h)

Y ≡ 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.