How to Design Programming Language Syntax

Adrian King
11 August 2014

or

How Not to Design Programming Language Syntax

Adrian King
11 August 2014

(your choice)

The Dawn of History: A Language without Precedent

```
PROGRAM Compute_Factorial
!-------------------------------------------------------------------------
!
!  This program computes n! using a recursive function
!
!  Uses:  FUNCTION Factorial(n)
!
!-------------------------------------------------------------------------
IMPLICIT NONE

INTERFACE
   FUNCTION Factorial(n)
     INTEGER :: Factorial
     INTEGER, INTENT(IN) :: n
   END FUNCTION Factorial
END INTERFACE

! Declare local variables
INTEGER :: n

!  Prompt user for radius of circle
write(*, '(A)', ADVANCE = "NO") "Enter n for computing n!:  "
read(*,*) n

! Write out value of factorial using function call
write(*,100) n, "factorial is ", Factorial(n)
100 format (I3, 2x, A, 2x, I12)

END PROGRAM Compute_Factorial

!-----Factorial-----------------------------------------------------
!
!  Function to calculate factorials recursively
!
!-------------------------------------------------------------------------
RECURSIVE FUNCTION Factorial(n)  RESULT(Fact)

IMPLICIT NONE
INTEGER :: Fact
INTEGER, INTENT(IN) :: n

IF (n == 0) THEN
   Fact = 1
ELSE
   Fact = n * Factorial(n-1)
END IF

END FUNCTION Factorial
```

COBOL:
Concision Is the Enemy

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. FACTORIAL.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       77 N PIC 9(4).
       77 A PIC S9(4) VALUE 0.
       77 F PIC 9(4) VALUE 1.
       PROCEDURE DIVISION.
       PARA.
           DISPLAY "ENTER A NUMBER.".
           ACCEPT N.
           PERFORM PARA1 UNTIL A = N.
           DISPLAY "THE FACTORIAL IS".
           DISPLAY F.
           STOP RUN.
       PARA1.
           ADD 1 TO A.
           COMPUTE F = F * A.
```

Lisp Is the Best Programming Language Ever

```
(define (factorial n)
  (if (zero? n)
    1
    (* n
     (factorial (- n 1)))))
```

Homoiconicity

- In Lisp, the shape of a program and the shape of the most commonly used data structure are very similar.
- It's easy to create and manipulate programs from within a program.
- It's easy to write eval and understand how it works.

(Lisp (Is (the (Worst (Programing (Language (Ever))))))

- Lisp syntax is easy to describe and manipulate, but hard to read.
- Human beings can easily understand infix notation, but Lisp has only prefix notation.
- Too many parentheses!

Algol:
Welcome to the Modern World

```
PROC facto = (INT n) INT:
  BEGIN
     INT a := 1;
     FOR i FROM 1 TO n DO
        a := a * i;
     OD
  END;
```

Keywords, Infix Operators, and Recursive Grammars

Most modern programming languages are designed along the same lines as the Algols:

- A modest number of keywords (compared with COBOL, anyway).
- Infix syntax for mathematical operators.
- Some degree of regularity in allowing constructs to nest.
- Homoiconicity? Fuhgeddaboudit.

Simplicity and Understandability Are Not Always the Aim

```
++++++++++++++++++++++++++++++++
>++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++
>++++++++++
>+++++++
>
>+
<<
[
>+++++++++++++++++++++++++++++++++++++++++++++++++
--------------------------------------------------
<<<<.-.>.<.+

>>>>>
>
>++++++++++
<<
[->+>-[>+>>]>[+[-<+>]>+>>]<<<<<<]
>[<+>-]
>[-]

>>
>++++++++++
<
[->-[>+>>]>[+[-<+>]>+>>]<<<<<]
>[-]
>>[++++++++++++++++++++++++++++++++++++++++++++++++++.[-]]
<[++++++++++++++++++++++++++++++++++++++++++++++++++.[-]]
<<<++++++++++++++++++++++++++++++++++++++++++++++++++.[-]

<<<<<<.
>>+

>[>>+<<-]
>>
[
<<<[>+>+<<-]
>>[<<+>>-]
>-
]
<<<<-
]
```

- Some languages eschew mundane goals like usability.
- On the left is a Brainfuck implementation of factorial.
- (See also C++.)

Haskell and the MLs: the Revenge of the Math Nerds

- The ML/Haskell aesthetic descends from Algol's: keywords, infix operators, regularity in nesting expressions.
- But the emphasis is on constructs that have a sound basis in type theory.
- The end (we can hope) of implementing languages before designing them.

Unaries and Binaries

- Most non-keyword symbols in ML and Haskell can be treated either as (possibly unapplied) unary prefix operators or binary infix operators.
- Unary operators have higher precedence than binary operators and are left-associative.
- Binary operators have individually specified precedence and associativity. So:

  a b c + d e * f g h

  is understood as:

  ((+) ((a b) c) ((*) (d e) ((f g) h)))

  but at a great savings in parentheses.

Punctuation-Saving Tricks of the Great Languages

- Haskell, like some other languages, uses indentation level to decide which lines continue previous constructs.
- The rule is defined to infer locations of curly brace enclosures and semicolon separators, which can usually be omitted.

- Scala normally uses semicolons to separate consecutive declarations or statements, but between curly braces, you can omit semicolons at ends of lines where Scala sees that a semicolon is legal.

```
{
  val a = 1 ;
  val b = 2 ;
  println(a + b)
}
```

```
size s = length (stkToLst s)  where
          {stkToLst  Empty         = []
           ;stkToLst (MkStack x s)  = x:xs where {xs = stkToLst s}}
```

Whitespace: the Final Frontier

- People sometimes use spaces within an expression to make operator precedence more visible to the reader:

```
a + b*c - d
```

- But what would happen if you wrote:

```
a+b * c-d
```

Uh

Rules for Uh (Uncomplicated language for Humans):

- Non-delimiter tokens are divided into unary and binary operators.
- Binaries start with punctuation character. Relative precedence is hardwired and determined by first character; associativity by last character.
- A binary operator with no whitespace before or after it has higher precedence than unary operators; otherwise, lower.
- Token rules are determined by surrounding *delimiters*.
- Predefined delimiters are start/end of input, { and }, and (and).
- Except within (and), Scala-style semicolon inference between a unary at end of one line and another unary at start of the next line.

Some Uh

```
fac ~ n, n =< 1 ?? 1 !! n * (fac n-1)

println (fac 42)
```

- or

```
fac: Int-:Int ~
  n: Int, n =< 1 ?? 1 !! n * (fac n-1)

println (fac 42)
```

- Uh compiles to the optionally dependently typed lambda calculus (implemented in a virtual machine named Um).
- Previous rules describe how to construct function applications.
- Built-in macros translate to other terms in the calculus, like comma for function abstraction.
- Some macro bindings are unconventional for conciseness or to accommodate the oddities of dependent types.

Some Um

```
fac ~ n, n =< 1 ?? 1 !! n * (fac n-1)
println (fac 42)
```

translates to:

```
App(
 Lam(
  fac,Om,
  App(
   Var(println),
   App(
    Var(fac),Var(42)))),
    Fix(
     fac,Om,
      Lam(
       n,Om,
       If(
        App(App(Var(=<),Var(n)),Var(1)),
        Var(1),
        App(
         App(Var(*),Var(n)),
         App(Var(fac),App(App(Var(-),Var(n)),Var(1))))))))
```

- Om is the name of the omitted (dynamic or inferred) type.

I'd Like My Homoiconicity Back Now, Please

- Translation from Uh to Um is not as simple to describe as Lisp translation to untyped lambda calculus.
- But it's not too horrible.
- Makes programming built-in macros easy enough.
- Whether or not you think a language *should* have macros or eval, it would be nice to believe the language is simple enough that you *could*.

Further Explorations

- Delimited mixfix operators:

  if[itsOk]then[sayIt]else[justShutUp]

- Interbinary lambda inference:

  (* 3 + -) =>

  λ *a b c*. *a* * 3 + *b* - *c*

- This already works, but only at the start and end of a group (*a* and *c* above).