Comfort

John David Stone

October 14, 2013

1 Overview

Comfort is a notation for expressing algorithms, adapted from the Joy programming language designed by Manfred von Thun. It is based on a functional, stack-based model of computation. As compared to Joy, it is quite limited, but provides a convenient way to implement and test many non-trivial algorithms. However, its primary purpose is to provide a suitably challenging implementation exercise for undergraduate students of compiler construction.

After a survey of its lexical structure, the following sections introduce the various elements and constructions of Comfort and its unusual processing model. Syntactic rules in Backus–Naur form define these constructions formally, but are accompanied by informal semantic explanations and discussions of grammatical constraints that are not readily expressible in BNF.

2 Lexical structure

The tokens of Comfort are numerals, identifiers, and the symbols `@`, `==`, `;`, `,`, `[`, and `]`. Numerals, identifiers, and the symbol `==` must be separated from one another by whitespace characters (spaces, tabs, or newlines) or comments, but other symbols need not be separated from adjacent tokens.

A numeral is a sequence of one or more digits, optionally preceded by a hyphen-minus character and optionally followed by a fractional part (consisting of a decimal point and one or more digits), an exponent indicator (consisting of the letter `e`, in either case; an optional sign, `+` or `-`; and a sequence of one or more digits), or both (provided that the fractional part precedes the exponent).

An identifier is a sequence of one or more letters, digits, underscores, and hyphen-minus characters that does not begin with a digit or with a

hyphen-minus character followed by a digit. By convention, we also include the symbols for arithmetic operations and relations as identifiers.

The *reserved identifiers* of Comfort have predefined meanings that cannot be changed or overridden. They are: `+`, `-`, `*`, `/`, `=`, `!=`, `<`, `<=`, `>`, `>=`, `abs`, `acos`, `all`, `and`, `app1`, `app11`, `app12`, `asin`, `at`, `atan`, `atan2`, `binary`, `binrec`, `boolean`, `branch`, `case`, `ceil`, `choice`, `cleave`, `compare`, `concat`, `cond`, `cons`, `construct`, `cos`, `cosh`, `dip`, `div`, `drop`, `dup`, `dupd`, `enconcat`, `equal`, `exp`, `false`, `filter`, `first`, `float`, `floor`, `fold`, `frexp`, `genrec`, `has`, `i`, `id`, `ifte`, `in`, `infra`, `integer`, `ldexp`,`linrec`, `list`, `log`, `log10`, `map`, `max`, `min`, `modf`, `neg`, `not`, `null`, `nullary`, `of`, `or`, `pop`, `popd`, `pred`, `pow`, `primrec`, `rem`, `rest`, `rolldown`, `rolldownd`, `rollup`, `rollupd`, `rotate`, `rotated`, `sign`, `sin`, `sinh`, `size`, `small`, `some`, `split`, `sqrt`, `stack`, `succ`, `swap`, `swapd`, `swons`, `tailrec`, `take`, `tan`, `tanh`, `ternary`, `times`, `treegenrec`, `treerec`, `treestep`, `true`, `trunc`, `unary`, `unary2`, `unary3`, `unary4`, `uncons`, `unstack`, `unswons`,`while`, `x`, and `xor`.

A comment in Comfort begins with the character-pair `(*` and continues through the next following occurrence of the character-pair `*)`.

3 Stack-based execution

When executed, a Comfort program carries out its computations by pushing values onto a stack and popping values from it. Every numeral and identifier in Comfort effectively designates a command that has some effect on the stack.

The stack can contain values of four types: integers, reals, Booleans, and *quotations*, which are list-like structures that can also be used as executable procedures.

An *operand* is an expression that generates a value and pushes it onto the stack. Numerals are operands, as are the reserved identifiers `true` and `false`.

An *operator* is an expression that pops its arguments from the stack, performs some computation with them, and pushes the result or results onto the stack. For instance, the `+` operator pops two arguments, each of which must be a number, from the stack, finds their sum, and pushes the sum onto the stack.

Some operators are executed solely for their operations on the stack. For instance, the `swap` operator pops two values from the stack and pushes them back on immediately, but in the opposite order; and the `pop` operation pops a value from the stack and discards it, pushing nothing.

A *combinator* is expression that pops one or more values from the stack, performs some computation with them that involves invoking at least one of them as an executable procedure, and pushes the results. Typically, at least one of the values on which a combinator operates is a quotation, and the combinator determines whether the procedure expressed in that quotation should be invoked at all and, if so, how often and under what conditions. For instance, the `ifte` combinator pops three quotations from the stack and executes the one that was bottommost of the three, which should leave a Boolean value on top of the stack. If this Boolean value is `true`, `ifte` then executes the middle quotation; if `false`, `ifte` executes the top quotation. This is the analogue of an "if ... then ... else ..." construction in more conventional programming languages.

4 Expressions and expression sequences

An expression in Comfort, then, is either a numeral, an identifier, or a quotation. A quotation is formed by enclosing zero or more whitespace-separated subexpressions in square brackets.

expression_sequence → ϵ
  | *expression expression_sequence*
expression → *numeral*
  | *identifier*
  | `[` *expression_sequence* `]`

A *list* is a quotation that contains only numerals, the operands `true` and `false`, and other lists. It is possible to execute a list; the effect is to push the value of each of its elements onto the stack, in order. This is consistent with the general rule for executing a quotation, which is to execute each of its elements, in order. (Recall that "executing" an operand amounts to pushing its value onto the stack.)

Most quotations, however, are not lists; they contain identifiers that are operators or combinators. "Executing" such an identifier means performing the associated stack operations and computation.

This is, in fact, the entire computational model of Comfort. A program is essentially an expression list, and one executes the program by executing each of the expressions in the list in turn. When a quotation is encountered in an expression list, it is treated as a structured operand; its elements are collected (but not evaluated or executed), and the collection, treated as a unit, is pushed onto the stack. This structure is not executed unless and

until a combinator that is subsequently applied to it calls for its execution. (In this respect, a quotation is somewhat similar to a `lambda`-expression in a more conventional functional language.)

5 Definitions and programs

Apart from `true` and `false`, all of the reserved identifiers listed above are built-in operators and combinators that have specified computational effects. Comfort programmers can also define their own operators and combinators by combining the built-ins, according to the following syntax:

definition → *identifier* `==` *expression_list*

Recursive and mutually recursive definitions are allowed and will not result in errors provided that every identifier has been defined before it is executed.

Definitions can be collected in a *definitions section* that begins with the symbol `@` and ends with the symbol `.` (the full-stop character). Successive definitions must be separated from one another by semicolons.

definition_section → ϵ
  | `@` *definition_sequence* `.`
definition_sequence → *definition more_definitions*
more_definitions → ϵ
  | `;` *definition more_definitions*

A Comfort program consists of a definition section, an expression list, and a closing full-stop character. The execution of the program causes the expression list to be executed on an initially empty stack. At the end of the program, the value on top of the stack, if there is one, is printed.

program → *definition_section expression_list* `.`

6 A sample program

```
(* Compute the greatest common divisor of two numbers
   using Euclid's algorithm.
*)

@ gcd == [dup rotate dup swapd !=]
         [[dup rotate dup swapd <] [swap] [] ifte
             dup rotate swap -]
         while .

1216 1152 gcd .
```