The CHATTERTON Reference Manual

Richard A. O’Keefe

2010

1 The Game and the Name

The game is to conduct one or more software engineering experiments. For years debate has raged over the One True Way to write identifiers in programs. Many programmers have strong feelings that their Way is so much better than all the others that anyone who does not follow it should be ashamed of themselves. Nobody appears to be able to point to any evidence.

More precisely, there is evidence that words that are visibly separated are easier to read than words that are run together and that text with internal upper casing (like iPhoNe, say) is harder to read than lower case text or text with initial capital letters. But all the evidence of that kind which I have been able to find relates to reading natural language text, not programming text. For years I have looked in journals and on the Web and have asked in mailing lists for evidence that bears on programming and nobody has given me any.

I am sick of it. I know what style I prefer, but I am not so arrogant as to mistake that for evidence that it is the best. The sign on my door says “to be less wrong than yesterday”, and if there is a style that is measurably better than what I happen to like, I want to know so that I may switch to it.

What I’ve done is to write three short programs, which I want you to read, answer some simple questions about, and fix. Each program has been converted to three dialects of CHATTERTON, so you will be asked to work with one program in each dialect. I want to know whether there is any detectable difference in

- how long it takes you to read a (short) program to the point where you think you understand it
- how long it takes you to find answers to questions like you might have when reading someone else’s code to understand it well enough to work with later
- how long it takes you to find a couple of simple mistakes and fix them.

It may well be that the nature of the materials, the size of the class, and the amount of variation between people, make it impossible to detect any difference. There is nothing much I can do about this except to repeat the experiment with more people.
It may well be that the effect of familiarity outweighs anything else, so that what you know determines what you find easy to read. I work on a daily basis with code in such a range of styles that I count all the dialects listed here “familiar”, yet that does not mean I find them all equally easy to cope with. The way I have tried to deal with this is by using a programming language that you aren’t familiar with. It’s not so different that you should be confused by it, but it’s different enough that I hope (only hope!) that it will unsettle your expectations about identifier style.

It may be that there is clear evidence that one style is best. That won’t make me switch this year, because to keep the workload for this assignment down, you will only be exposed to three of the six styles listed here. One of the other three might be better still.

1.1 The Name

I needed a new language for this experiment. Since I was to be the only person writing programs in it, and since they were supposed to be slightly buggy anyway, there is no need to run them. So there isn’t any compiler for Chatterton.

A notation that looks like a programming language but can’t be run is sometimes called a pseudo-code.

Thomas Chatterton was an 18th century English writer who not only wrote pseudo-mediæval poetry, but did so in an English dialect of his own devising, these days described as “Rowleian jargon” after the fictious poet Thomas Rowley claimed to have been the true author.

1.2 The Dialects

Chatterton is a family of languages, suitable for experiments with different aspects of programming language readability. The members of the family are named Chatterton-ids-side-semis-field

*ids* can be C, I, P, S, or U, described below.

*side* can be B, M, or E. It is B if identifiers are at the beginning of declarations, *e.g.,*

\[
\text{sorted \ var \ [] \ str := \{\}}
\]

It is M if identifiers are in the middle of declarations, *e.g.,*

\[
\text{var \ sorted \ : \ [] \ str := \{}}
\]

It is E if identifiers are at the end of declarations, *e.g.,*

\[
\text{var \ [] \ str :: \ sorted := \{}}
\]

*semi* can be O, L, or R. It is O if semicolons are optional and usually omitted. It is L if semicolons are used when and only when the last token of one construct and the first token of the next are on the same line. It is R if semicolons are always required.
field can be F, D, M, or X. It is F if fields of algebraic data types can only accessed using function call syntax. It is D if they can only be accessed using dot syntax. It is M if dot and function call syntax can be mixed freely. It is X if you don’t care because algebraic data types and their fields are not allowed.

In all the dialects of Chatterton, an identifier may be one or more words, allowing any letter in the ISO Latin 1 character set, but not regarding alphabetic case as significant, so that “coupe” and “Coupe” and “COUPE” and “coUpE” are all the same identifier, but “coupé” is a different one. The ids part of a dialect name says how that dialect separates the words of a multi-word identifier.

A Named for Algol 60, this dialect separates words with spaces. This makes “can not” and “cannot” different identifiers, and indeed they are subtly different in English. Only spaces are allowed; there may not be a tab or newline inside an identifier. Historically, Algol 60 actually ignored spaces. Chatterton-A treats any non-empty run of white space characters as a significant separator.

C Named for COBOL 61, this dialect separates words with hyphens, just like English does. Computers (like the typewriters they copied) use the same character for hyphen and minus. To disambiguate, the infix operator for subtraction must have at least one space on each side, in all dialects.

I Named for InterLisp, this dialect separates words with dots. The statistics programming environment S (with free version R) does this too.

P Named for PL/I (roughly 1965), this dialect separates words with underscores. PL/I is where /* */ comments come from.

S Named for Smalltalk 80, this dialect runs words together, but capitalises the first letter of all words after the first. Most people don’t know that this style was not thought to be superior to the PL/I style, but was forced by the use of a character set that simply didn’t have an underscore character or anything like it. The character code that is now used for underscore was used for ←, which was used for assignment. Some other languages used the same convention for precisely the same reason, and when those languages were used on systems with less impoverished character sets, programmers continued the same now obsolete style.

U Named for Unix, this dialect just runs words together. The style is not peculiar to Unix. It goes back to the 1950s. The Unix V7 linker had a limit of 8 significant characters in an identifier, and the compiler added an underscore to the beginning of identifiers. (This still happens in Mac OS X, for example, though like other modern Unix systems it allows much much longer identifiers.) This meant that your global variables had to be different in the first 7 letters, which encouraged you to squeeze hard. (But not as hard as the IBM 1130, which allowed only 5 letters, upper case.)
To keep your work during this experiment to a manageable level, the only dialects considered are CHATTERTON-P-L-O-X, CHATTERTON-S-L-O-X, and CHATTERTON-U-L-O-X. Here are some examples in each of the styles you will see.

P  time_of_day  count_on_left
S  timeOfDay   countOnLeft
U  timeofday   countonleft

1.3 Use of styling

CHATTERTON distinguishes comments, identifiers, and keywords by character style. A CHATTERTON source file is an HTML file containing bulk commentary in the text and source code in ⟨PRE CLASS=code⟩ sections, in which comments are marked up using ⟨I⟩, keywords are marked up using ⟨B⟩, and identifiers are not marked up manually, but we’d expect to have a tool that wrapped them in ⟨A⟩ so that you could easily navigate to definitions.

2 Example

Here is a small example of CHATTERTON code, presented in each of the three ways you will see. The program fits a straight line to a collection of points using simple linear regression.
PL/I style

mean proc (x in [] real) ⇒ (m out real)
  assert size x > 0
  m := 0.0
  loop for i in [0,size x)
    m +=: x[i]
  end loop
  m /:= float(size x)
end proc

covariance proc (x in [] real, x_mean in real, y in [] real, y_mean in real) ⇒ (c out real)
  assert size x > 1 and size x = size y
  c := 0.0
  loop for i in [0,size x)
    c +=: (x[i] - x_mean) * (y[i] - y_mean)
  end loop
  c /:= float(size x - 1)
end proc

x var [] real := {}  independent variable, empty to start
y var [] real := {}  dependent variable, empty to start

loop
  t_x var real
  t_y var real
  while get (t_x, t_y)
    x @:= {t_x}  push new element on right of x
    y @:= {t_y}  push new element on right of y
  end loop
mean_of_x const = mean(x)
mean_of_y const = mean(y)
slope const = covariance(y, mean_of_y, x, mean_of_x)  
/ covariance(x, mean_of_x, x, mean_of_x)
intercept const = mean_of_y - slope * mean_of_x
print "slope =", slope, "intercept =", intercept
Smalltalk style

mean proc (x in [] real) ⇒ (m out real)
  assert size x > 0
  m := 0.0
  loop for i in [0,size x)
    m +:= x[i]
  end loop
  m /= float(size x)
end proc

covariance proc (x in [] real, xMean in real, y in [] real, yMean in real) ⇒ (c out real)
  assert size x > 1 and size x = size y
  c := 0.0
  loop for i in [0,size x)
    c +:= (x[i] − xMean) * (y[i] − yMean)
  end loop
  c /= float(size x − 1)
end proc

x var [] real := {} \hspace{1cm} independent variable, empty to start
y var [] real := {} \hspace{1cm} dependent variable, empty to start
loop
tX var real
tY var real
  while get (tX, tY)
    x @:= {tX} \hspace{1cm} push new element on right of x
    y @:= {tY} \hspace{1cm} push new element on right of y
  end loop
meanOfX const = mean(x)
meanOfY const = mean(y)
slope const = covariance(y, meanOfY, x, meanOfX)
  / covariance(x, meanOfX, x, meanOfX)
intercept const = meanOfY − slope * meanOfX
print "slope =", slope, "intercept =", intercept
UNIX style

mean proc (x in [] real) ⇒ (m out real)
   assert size x > 0
   m := 0.0
   loop for i in [0, size x)
      m +:= x[i]
   end loop
   m /:= float(size x)
end proc

covariance proc (x in [] real, xmean in real, y in [] real, ymean in real) ⇒ (c out real)
   assert size x > 1 and size x = size y
   c := 0.0
   loop for i in [0, size x)
      c +:= (x[i] - xmean) * (y[i] - ymean)
   end loop
   c /:= float(size x - 1)
end proc

x var [] real := {}  independent variable, empty to start
y var [] real := {}  dependent variable, empty to start
loop
   tx var real
   ty var real
   while get (tx, ty)
      x @:= {tx}  push new element on right of x
      y @:= {ty}  push new element on right of y
   end loop
meanofx const = mean(x)
meanofy const = mean(y)
slope const = covariance(y, meanofy, x, meanofx) / covariance(x, meanofx, x, meanofx)
intercept const = meanofy - slope * meanofx
print "slope =", slope, "intercept =", intercept
3 None of this is new

Chatterton is an imperative programming language that works exclusively with values. The names of the primitive types (bool, char, int, real) are taken from Algol 68, as is the idea of making it an expression language, where things you might think of as statements, like assignment statements, if statements, and loop statements, have values. The use of while and until inside loops is copied from a Lisp dialect reported in an article in BIT years ago.

The idea of having each kind of composite statement control a whole block of statements, not just a single statement, comes from Algol 68. This means that we don’t need anything like C’s curly braces, so we can entirely avoid arguments about where to put them. Algol 68 used if . . . fi and do . . . od; the use of end if and end loop and so on is copied from Ada 81.

The form of procedure headings is ultimately due to Algol W, but the idea of having in, in out, and out parameters comes from Ada 81. This lets us avoid Pascal “var” (C++ “reference”) parameters, so that Chatterton has absolutely no idea of exposing a pointer to part of a variable. The idea of working exclusively with values, so that there are (or at least appear to be) no mutable substructures shared by different variables comes partly from SETL and partly from S.

The idea of putting the identifier declared as the first thing in a declaration comes from Pascal, but with Ada 81 influence as well. The way strings (and arrays) are sliced owes something to Algol 68 and Fortran 77. The way intervals are notated is common mathematics, but putting them in a programming language comes from MESA. The idea of making concatenation apply to arrays as well as strings comes from Ada 81. Multidimensional arrays as rows of rows comes from Algol 68, Burroughs Algol, APL, and Euler. The idea of making arrays be flexible (stretchy) comes from Algol 68. Dijkstra’s notation also does this, and I gave serious thought to adopting his array operations. The decision to use 0-origin indexing, to make the data structures more familiar, meant that this didn’t work out. The idea of growing an array by just assigning to the next unused element comes from a Lisp dialect.

The idea of using tuples rather than records comes from functional languages like SML and Haskell, although they have records as well as tuples. Having functions return multiple results by returning tuples is the norm in functional programming languages; MESA did this in an imperative language.

Sinister function calls come from Pop-2, SETL, and S. This is when you can define what f(x):= means. Dotted calls are from Pop-2.

A policy that led to some perhaps unfamiliar choices is that if an operation applies to only one type, it might be written using function call syntax or it might be written as an operator, depending on which is more traditional, but an operation that applies to more than one type must be written as an operator. Thus we have sqrt(x) but abs y, because abs i is also allowed. This is why max and min are infix operators.
4 Data types

All the primitive types are ordered, so $a < b$, $a = b$, and so on are all allowed whenever $a$ and $b$ are the same type. $a \max b$ (also written $a \lor b$) is whichever of the two is the greater; $a \min b$ (also written $a \land b$) is whichever of the two is the smaller.

We may test whether something is in an interval. For example,

$x \in [L,U)$

is true when precisely when

$L \leq x \text{ and } x < U$

except that $x$, $L$, and $U$ are all evaluated exactly once. There is more about intervals in the subsection on arrays.

4.1 bool

The Boolean type is called bool. The values are false and true. They are ordered so that false < true. Apart from comparison, the operations on bool are not, and, and or. The and and or operators are lazy, like C’s && and ||.

Should you want strict operators that evaluate both arguments, the standard ordering on Booleans means that $\min$ (or $\land$) is a strict “and”, and $\max$ (or $\lor$) is a strict “or”.

4.2 char

The character type is called char. Character values are Unicode and the order between two characters is determined by the numeric order of the code points that Unicode assigns to those characters. Character literals are written between single quotation marks and are exactly like C.

Apart from comparison, the operations on characters are ord(c), which converts a character to its integer code, chr(i), which converts an integercode to the corresponding character, val(c), which converts a character ‘0’–’9’, ‘a’–’z’, or ‘A’–’Z’ to the corresponding integer 0–9, 10–35, or 10–35, and dig(i), which converts an integer 0–35 to an ASCII character ‘0’–’9’ or ‘a’–’z’.

The C character classification functions are available with names is_alphanumeric (isalnum), is_letter (isalpha), is_blank, is_control (iscntrl), is_digit, is_graph, is_lower, is_print, is_punctuation (ispunct), is_space, is_upper. It doesn’t make a whole lot of sense to try to convert Unicode strings from one case to another one character at a time, so there are no character functions to do that.

4.3 int

There is one integer type, called int. Values of int are integers. There is no bizarre limit on the size of integers (like 16, 32, or even 64 bits), and arithmetic operations on integers either deliver correct results or stop the program with
an error message. Integer literals may be written as decimal numbers, or as a
decimal number 2–36 immediately followed by the letter “r” immediately
followed by digits in the given base, where A=a=10 and Z=z=35. (Hence the
definitions of dig(i) and val(c).)

In CHATTERTON-P digits may be grouped into blocks separated by under-
scores. It’s much easier to be sure that a million is not 10 million or 100 thousand
when you can write it as 1,000,000.

Apart from comparison, the operations on integers are exponentiation, written
as m**n, floored division m div n (take the rational number m/n and
truncate towards negative infinity), the remainder of floored division m mod n,
product m*n, sum m+n, difference m−n, absolute value abs n, and negation
~n.

It is unusual for negation to be given a different symbol from subtraction,
but not unheard of. The main reason for it in CHATTERTON is that semicolons
are completely optional, so that using a symbol as both a prefix and an infix
operator would be ambiguous.

In the current state of the language there are no bitwise operations.

4.4 real

There is one floating-point numeric type, called real. It corresponds to the
double type in C, C++, and Java. Values of real are IEEE double-precision
floats. They are written in the usual fashion, except that a scaling factor that is
a negative point of 10 must be written with a negation (~) sign, not a subtraction
(−) sign.

In CHATTERTON-P digits may be grouped into blocks separated by under-
scores.

Comparison is done according to IEEE rules, on the understanding the “un-
defined” outcomes are run time errors. For the purposes of max and min, “0.0
is taken to be less than 0.0; for the purposes of the comparison operators, they
are equal. The absolute value of ~0.0 is 0.0.

Apart from comparison, the operations on reals are exponentiation to an int
power, written as x**n, exponentiation to a real power, written as x^n, division
x/y, product x*y, sum x+y, difference x−y, absolute value abs x, and negation
~x.

There is no mixed mode arithmetic. If you want to combine an int and a
real you must convert one to the same type as the other using float(n), floor(x),
ceil(x), trunc(x), or round(x).

The usual C math functions sin(x), asin(x), cos(x), acos(x), tan(x), atan(x),
atan2(y, x), sinh(x), asinh(x), cosh(x), acosh(x), tanh(x), atanh(x), exp(x),
expm1(x), log(x), log1p(x), sqrt(x), cbt(x), hypot(x, y), and copysign(x, y) are
available, as are the classification functions is_finite(x), is_infinite(x), is_NaN(x),
is_normal(x)

10
4.5 \textbf{str}

In \textsc{Chatterton} strings are nothing other than arrays of characters. String literals are written between double quotation marks and look exactly like C.

The operations of concatenation ($s@t$), repetition ($s@@n$), indexing, and slicing apply to all arrays in \textsc{Chatterton}, not just to strings.

Because \textsc{Chatterton} has an explicit concatenation operator, it does not have C’s “string pasting”, which is restricted to literals.

In addition to the operations available on all arrays, there are some special functions for strings. These include \texttt{to_lower(s)}, \texttt{to_upper(s)}, and \texttt{to_title(s)} for case conversion. These functions should be locale-sensitive, so that \texttt{to_lower(“Î”) = “ı”} in a Turkish locale.

There are also operations for converting between strings and numbers:

- $n := \texttt{oct(s)}$ \quad $s := \texttt{to\_oct(n)}$ \quad base 8 (octal)
- $n := \texttt{dec(s)}$ \quad $s := \texttt{to\_dec(n)}$ \quad base 10 (decimal)
- $n := \texttt{hex(s)}$ \quad $s := \texttt{to\_hex(n)}$ \quad base 16 (hexadecimal)
- $x := \texttt{num(s)}$ \quad $s := \texttt{to\_num(x, n)}$ \quad reals (like \texttt{\%ng})
- $s := \texttt{to\_sci(x, n)}$ \quad reals (like \texttt{\%ne})
- $s := \texttt{to\_fix(x, n)}$ \quad reals (like \texttt{\%nf})

There is a search operator that looks for the first occurrence of a substring. The definition is

$$s\%t = \min i | s[i, i + \textbf{size } t) = t$$

or

$$s\%t = \textbf{size } s$$

if there is no such $i$. Remember source\%target and $s$ precedes $t$.

It is important that this should return \texttt{size } $s$ on failure for the sake of things like

```
line[|line65|"#",.] := "" \textit{delete sh-style comment}
```

There are two functions for splitting a line of input into pieces. Data files tend to come in two forms in UNIX. Here’s an entry from /etc/passwd:

```
_svn:*:73:73:SVN Server:/var/empty:/usr/bin/false
```

Calling \texttt{fields(line, ':')} on such a line would give you

```
{"_svn","*","73","73","SVN Server","/var/empty","/usr/bin/false"}
```

Another popular field separator character is TAB.

The other style separates fields with runs of white space. Here an example from the Unicode character mappings:

```
0x79 0x0079 # LATIN SMALL LETTER Y
```

Calling \texttt{tokens(line)} on such a line would give you

```
{"0x79","0x0079","#","LATIN","SMALL","LETTER","Y"}
```
There is a function to “undo” both tokens() and fields: join(a, s) concatenates an array of strings a, putting a separating string s between adjacent elements.

The functions happen to be built in but could have been defined thus in Chatterton:

**fields**

```chatterton
proc (s in str, c in char) ⇒ (r out []str)
p var := 0
r := {}loop for i in [0, size s)
  if s[i] = c then
    r @:= {s[p,i]}
    p := i+1
  end if
  i +:= 1
end loop
r @:= {s[p,)}
end proc
```

**tokens**

```chatterton
proc (s in str) ⇒ (r out []str)
i var := 0
loop while i < size s and s[i] ≤ ' '
i +:= 1
end loop
r := {}loop while i < size s
  j var := i
  loop while j < size s and s[j] > ' '
    j +:= 1
  end loop
  r @:= {s[i,j]}
loop while j < size s and s[j] ≤ ' '
  j +:= 1
end loop
i := j
end loop
end proc
```

**join**

```chatterton
proc (a in []str, s in str) ⇒ (r out str)
if size a = 0 then
  r := ""
else
  nr := a[0]
  loop for i in [1, size a)
    r @:= s @ a[i]
  end loop
```
end if
end proc

4.6 arrays

If \( t \) is a type, then \([t] \) is a type. This type may be pronounced “array of \( t \)” or “row of \( t \)'s. An array of \( t \) is a sequence of values of type \( t \), where the first is element 0, the second element 1, and so on. This is very like arrays in C and Java and Scheme.

Array values may be written as \( \{v_0, \ldots, v_{n-1}\} \), where all of the values belong to the same type. As long as there is at least one element, it is obvious what the type of the whole is. If there are no elements, the type must be supplied by the context.

Comparison is done lexicographically. If one array is a prefix of another, the shorter one is lesser.

The operations on arrays are determining the size, indexing, slicing, concatenation, and deletion.

The size of an array is the number of elements it has, which is one more than the last index. It is written \texttt{size} \( a \).

Indexing, \( a[i] \), returns the \( i \)th element of \( a \), counting from 0. It is an error if \( i \) does not refer to an existing element.

Indexed assignment, \( a[i] := e \), replaces the \( i \)th element of \( a \) if \( i \) refers to an existing element. It is an error if \( i \) does not refer to an existing element.

Slicing indexes an array with an interval. There are four kinds of intervals:

\[ \begin{align*}
\text{closed} & : [i,j) \{k | i \leq k \leq j\} \\
\text{half-open on the right} & : (i,j) \{k | i \leq k < j\} \\
\text{half-open on the left} & : (i,j] \{k | i < k \leq j\} \\
\text{open} & : (i,j) \{k | j < k < j\}
\end{align*} \]

This is standard mathematical notation in the English speaking world. (Some cultures use \( ] \) in place of \( ( \). Such is the perversity of humankind.)

A slice \( a[i,j] \) — or using any other kind of interval — returns an array containing just those elements \( a[k] \) for which \( k \in [i,j) \). If the interval is empty, the result is an empty array. If the interval has one element, such as \( a[i,i] \), the result is still an array. The upper bound of the interval may be omitted if the interval is open on the right; the missing bound is the size of the array. So \( a[0,p) \) extracts the first \( p \) elements of \( a \) and \( a[p,) \) the remainder. This is a common convention for string slicing.

A slice may be assigned to. This replaces the selected elements by the elements of the right hand side, which may have more or fewer elements than those it replaces. For example, to insert a space after the \( p \)th character of a string, you would write \( s[p,p) := \texttt{" "} \). The value of a slice assignment is the right hand side.

Slice assignment can be used to make an array grow and shrink. For example,

\[
x[0,0) := \{12\} \quad \text{insert 12 at the bottom of } x
\]
\[
x[\text{size } x,) := \{\texttt{-42}\} \quad \text{append -42 at the top of } x
\]
x[size x-1.) := {}  remove the top element of x
x[0,1) := {}  remove the bottom element of x

This document was written using an Emacs-like text editor using the “buffer gap” technique for representing the working copy of a file, and it allows just such insertions and deletions tolerably efficiently.

Programming languages have used a wide range of symbols for concatenation: || in PL/I, // in Fortran, ` or @ in SML, ++ in Haskell, & in Ada, . in PERL, and of course the execrable and wholly unjustifiable + in BASIC. Since C doesn’t use @ for anything, and Java only uses it for annotations, that symbol may be least likely to confuse you by suggesting something inappropriate.

The idiomatic way to add a new element \( x \) at the high end of array \( a \) is
\[
a @:= \{ x \}
\]
because it is easier to get right.

There’s an operation related to concatenation that we have a use for: repeatedly concatenating something with itself. It’s helpful if the structure of the symbols reinforces the analogy:

- repeated concatenation \( @@ \) is to concatenation \( @ \)
- as repeated multiplication \( ** \) is to multiplication \( * \)

The parallel includes the fact that \( s@@n \) and \( m**n \) are both defined only when \( n \geq 0 \).

Many programming languages, including APL, PL/I, Fortran, S, and Matlab, offer operations on arrays of one sort or another. This is usually done by overloading the usual arithmetic operators. That works well for addition, subtraction, and multiplication by numbers, because arrays can be seen as belonging to vector spaces. However, elementwise multiplication is not matrix multiplication. CHATTERTON is intended to help in the experimental study of readability; deliberately making things confusing is not one of its aims. So the CHATTERTON operations on arrays are different but transparently derived from operations on numbers and so on.

If \( \phi : T_1 \rightarrow T_2 \) is a unary operator and \( e : []T_1 \) then \( \phi.e : []T_2 \). That is, writing a dot immediately to the right of a unary operator turns it into a unary operator that applies to arrays. The value is given by the rule \( (\phi.e)[i] = \phi e[i] \).

Thus `.` negates all the numbers in an array of arrays of numbers.

If \( \theta : T_1 \times T_2 \rightarrow T_3 \) is a binary operator, \( x : T_1 \), \( e : []T_1 \), \( y : T_2 \), and \( f : []T_2 \), then
- \( x\theta.f : []T_3 \) where \( (x\theta.f)[i] = x\theta f[i] \)
- \( e\theta.f : []T_3 \) where \( (e\theta.f)[i] = e[i]\theta f[i] \)
- \( e\theta y : []T_3 \) where \( (e\theta y)[i] = x[i]\theta y \)

### 4.7 tuples

If \( t_1, t_2, \ldots, t_n \) are types, then \( (t_1,t_2,\ldots,n) \) is a type. As usual, the tuple \( (t_1) \) is exactly the same thing as \( t_1 \).

Values of a tuple type are tuples \( (v_1,v_2,\ldots,v_n) \), where each \( v_i \) is a value of the corresponding \( t_i \).
You may assign a tuple of values to a tuple of variables. For example, two
swap two variables you may write (x,y) := (y,x). If a procedure returns a tuple
of results, you may assign that to a single tuple-valued variable, or to a tuple
of variables. For example, we could write

\[
\text{fitted \ line proc (x in [] real, y in [] real) ⇒ (slope out real, intercept out real)}
\]
\[
\text{mean of}_x \text{ const} = \text{mean(x)}
\]
\[
\text{mean of}_y \text{ const} = \text{mean(y)}
\]
\[
\text{slope} := \text{covariance(y, mean of}_y, x, \text{mean of}_x)
\]
\[
/ \text{covariance(x, mean of}_x, x, \text{mean of}_x)
\]
\[
\text{intercept} := \text{mean of}_y - \text{slope} \ast \text{mean of}_x
\]
\end proc

\[
\text{this line := fitted \ line(x, y) \ variable that is a tuple}
\]
\[
\text{(this_a, this_b) := fitted \ line(x, y) \ tuple of variables}
\]

Tuples are compared lexicographically. To compare (x,y,z) with (u,v,w),
start by comparing x with u. If they are equal, compare y with v. If those are
equal too, compare z with w.

### 4.8 algebraic data types

Tuples are rather like C \texttt{struct}s. Algebraic data types are rather like a cross
between C \texttt{enum}s and C \texttt{union}s, only safe. When you define an algebraic data
type, you define

- a new named type
- one or more constructor functions for that type
- for each constructor, one or more projection functions

An algebraic type declaration uses the keyword \texttt{data} and lists the constructors:

\[
\text{identifier data [] constructor [] constructor} \ldots
\]

The declaration of a constructor looks rather like a procedure heading.

\[
[\text{identifier}]([\text{field}, \text{field}] \ldots ) ) \ [\text{where expr}]
\]

If one of the constructor identifiers is missing, it is taken to be the same as
the name of the new type. Only one constructor identifier may be missing.

It is appropriate that a constructor declaration should resemble a procedure
declaration, because it does define a function with that name and those (input)
arguments, whose effect is to construct a value of the type. In effect, it allocates
a box, fills it in, and labels it to say which of the constructors was used.

A field looks a bit like a variable declaration, but there is no initial value:

\[
\text{identifier [const|var] type}
\]

A field declaration results in a projection function

\[
\text{identifier proc(it in datatype) ⇒ (result type)}
\]
that extracts the field in question. Chatterton uses function call syntax for field selection.

For a field declared **const** — the default — that’s all you get. A field declared **var** gets an update function as well:

\[
\text{identifier proc(it in out datatype) := (replacement type)}
\]

Note that the argument being in out means that you cannot use an update function to build a value of such a type, only to update an already complete one. The only way to get a complete value is to use a constructor function.

The classic example is a binary search tree.

\[\text{tree data}\]

\[
| \text{empty()} \\
| \text{node(key str, val var info, left var tree, right var tree)}
\]

In addition to defining a new “tree” type, this defines the following functions:

\[\text{empty proc () \Rightarrow (r out tree)}\]
\[r := \text{a new record with nothing in it}\]

end proc

\[\text{node proc (key in str, val in info, left in tree, right in tree) \Rightarrow (r out tree)}\]
\[r := \text{a new record containing key, val, left, right}\]

end proc

\[\text{key proc (it in tree) \Rightarrow (r out str)}\]
\[\text{run time error if it’s not a “node” record}\]
\[r := \text{the “key” part of the “node”}\]

end proc

\[\text{val proc (it in tree) \Rightarrow (r out info)}\]
\[\text{run time error if it’s not a “node” record}\]
\[r := \text{the “val” part of the “node”}\]

end proc

\[\text{val proc (it in tree) := (n in info)}\]
\[\text{run time error if it’s not a “node” record}\]
\[\text{set the “val” part of the record to n}\]

end proc

\[\text{left proc (it in tree) \Rightarrow (r out tree)}\]
\[\text{run time error if it’s not a “node” record}\]
\[r := \text{the “left” part of the “node”}\]

end proc

\[\text{left proc (it in tree) := (n in tree)}\]
\[\text{run time error if it’s not a “node” record}\]
set the “left” part of the record to n

end proc

right proc (it in tree) ⇒ (r out tree)
run time error if it’s not a “node” record
r := the “right” part of the “node”
end proc

right proc (it in tree) := (n in tree)
run time error if it’s not a “node” record
set the “right” part of the record to n
end proc

For example, we might have

weight proc (t in tree) => (w int)
case t of
  | empty() w := 0
  | node(k,v,l,r) w := 1 + weight(l) + weight(r)
end case
end proc

t var tree := node("D", no_info(), empty(), empty())
info(t) := some_info()

Values of algebraic data types may be compared. If two values were made using different constructors, the one whose constructor was declared earlier is the lesser. If two values were made using the same constructor, and all corresponding fields are equal, they are equal. Otherwise, take the leftmost field that differs in the two values, and the order of the two values is the order of those two fields. For this example, it is as if we had

"=" proc (t1 in tree, t2 in tree) => (r bool)
case t1 of
  | empty()
    case t2 of
      | empty() b := true
      | node(_,_,_,_) b := false
    end case
  | node(k1,v1,l1,r1)
    case t2 of
      | empty() b := false
      | node(k2,v2,l2,r2) b := k1=k2 and v1=v2 and l1=l2 and r1=r2
    end case
  end case
end proc
4.9 procedures

Because they are not needed for this experiment, the current version of CHATTERTON does not have procedure literals (lambda-expressions), procedure valued expressions, procedure valued variables, or procedure valued parameters. Fortran already had procedure valued parameters back in the late 1950s, so this is regrettable.

5 Declarations

There are four kinds of declaration. Arguably, specifications of procedure parameters and results are also declarations. An important structural property is that the identifier being declared is the first token of the declaration (at least in Chatterton-ids-B-semis), and should be the first token on its line. Another property is that the identifier being declared is immediately followed by a keyword saying what kind of declaration it is, although these keywords are optional in procedure headings.

A fundamental rule of CHATTERTON is that you must declare an identifier before its first use in the program. This would create a problem with recursive types if CHATTERTON had them. It does create a problem when there is a cycle of mutually recursive procedures, which is solved by forward declarations, an idea copied from Pascal.

The scope of a type, constant, or variable declaration is from just after its declaration to the end of the containing serial clause.

A procedure identifier may be used inside a procedure (itself or another) from just after the heading of the procedure or forward declaration for it, to the end of the containing clause. No call to it may happen until after its declaration is complete.

5.1 Type declarations

\[
\text{identifier type type [where expr]}
\]

defines the identifier on the list to be an alias for the type on the right. This is basically a C “typedef”. But there is one more part to it than that. A value \(v\) belongs to the new type if and only if it belongs to the underlying type and the expr is true when \(v\) is substituted for the identifier ‘it’.

For example, suppose we want to talk about day of week numbers. What range are we talking about, and what do they mean? All is clear if we write

\[
\begin{align*}
\text{Sunday const} & = 0 \\
\text{Saturday const} & = 6 \\
\text{day type int where it in [Sunday,Saturday]}
\end{align*}
\]

If we want to constrain an array type, we usually want to say something about every element. There are two kinds of expression that you might not have seen before. One of them comes from APL. If \(\theta\) is an associative operator
with left and right identity \( e \), then \( \theta/\{x_1, \ldots, x_n\} \) is \( e \theta x_1 \ldots \theta x_n \). If \( \theta \) is an associative operator without identity, then \( \theta/\{x_1, \ldots, x_n\} \) is \( x_1 \theta x_2 \theta \ldots \theta x_n \) and is undefined when \( n = 0 \). The only such operators in Chatterton are

\[
\begin{array}{cccc}
\theta & e & \text{name} & \text{derived} & \text{meaning} \\
+ & 0 & \text{add} & +/ & \text{sum of all} \\
* & 1 & \text{multiply} & */ & \text{product of all} \\
@ & "" & \text{append} & @/ & \text{concatenation of all} \\
\wedge & \text{N/A} & \text{lesser} & \wedge/ & \text{least of all} \\
\lor & \text{N/A} & \text{greater} & \lor/ & \text{greatest of all} \\
\land & \text{true} & \text{and} & \land/ & \text{all, forall} \\
\lor & \text{false} & \text{or} & \lor/ & \text{any, exists} \\
\end{array}
\]

Note that when applied to Booleans and characters, min and max do have identities. When applied to strings, max has "" as identity. Min and max do not otherwise have identities.

For example, if we want to say “array of non-negative integers” we can write

\[
\square \text{int where min/it} >= 0 
\]

although that doesn’t work if there aren’t any elements.

The second kind of expression used here is called a comprehension. It is based on list comprehensions in functional languages, which are in turn based on set comprehensions in mathematics. The form is \{ \text{e for} \ldots \}, where you can use \text{for} and \text{where} just like in a loop. In fact this is a loop. To say “array of non-negative integers” in a way that works for empty arrays, we can write

\[
\square \text{int where not loop for i in} [0, \text{size it}) \text{ until it}[i] < 0 \text{ end loop}
\]

which is clumsy, or we can write

\[
\square \text{int where \wedge/\{it[i] \geq 0 for i in} [0, \text{size it})}\}
\]

expressing what we want to be true rather than what we hope isn’t.

Constrained types are only used in one of the examples, and for that one you can rely on an intuitive reading of the stuff after \text{where}.

5.2 Constant declarations

\text{identifier const = expression} \\
\text{identifier const type = expression}

These define the identifier to be read-only and to have \text{expression} as its value. The type of the variable is the type of the expression; only if that is not complete is it necessary to mention the type in the initialisation.

It is important to understand here that named constants are \textit{not} names for variables that you happen not to be allowed to reassign. They are names for \textit{values}. If you declare

\[
n \text{const} = 42
\]

you are asserting that \textit{n equals} 42. Whenever the name \textit{n} appears within its scope, it can be replaced by 42.
Do not talk about “const variables” or “constant variables”, because there is no such thing. Constants are just named values; in this example $n$ is no more a variable than 42 is. Do not talk about or think about a const declaration as “assigning” a value to the name; it is illegal to assign to constants, named or anonymous, always and everywhere. In a dialect of Chatterton using American “=” and “==” instead of European “:=” and “=”, constant declarations would use “==”, because they are all about equality, not assignment.

### 5.3 Variable declarations

\[
\text{identifier var} := \text{expression} \\
\text{identifier var type} := \text{expression} \\
\text{identifier var type}
\]

These define the identifier to be a variable that may be (re)assigned. The first two provide an initial value. Only if that includes something like an empty array is it necessary to say what the type is. The last form is used when the variable will be initialised later.

Note that even when a variable is initialised, it is still a variable, not a constant. We do not use an “=” sign here, because the initialising assignment for a variable really is an assignment, whether it appears in the same declaration, in the following statement, or 500 pages later.

### 5.4 Procedure declarations

A procedure declaration has a header giving the interface of the procedure and a body, which is a serial clause:

\[
\text{name proc rest of heading;} \\
\text{serial clause} \\
\text{end proc}
\]

The heading of a procedure may take three forms.

**no result tuple**  \text{name proc (arguments)}

This is used either for a procedure that has no results or returns them using out parameters. This is the equivalent of a C function/Java method returning “void”.

**result tuple**  \text{name proc (arguments) ⇒ (results)}

The result part lists one or more identifiers with their types. The keyword \text{out} is optional in the \text{results} but it is implied even if not there. Each of those identifiers is a variable that the procedure is obliged to initialise. The usual case is where there is only one result, but there’s no reason why you can’t have more. For example, you might have a procedure

\[
\text{decode_date proc (julian_day_number int) ⇒ (year int, month int, day int)}
\]
which returns three values.

**sinister** name proc (arguments) := (extra input)

This is a procedure that can be called on the left hand side of an assignment statement. For example, suppose you want to keep track of which department each person is in. If people were identified by small integers, you might use

```plaintext
... department[person]
...
department[person] := ...
```

But if people were identified by large integers, or by strings, you would want to use procedures

```plaintext
... department(person)
...
set_department(person, ...)
```

which would be so much more like the array code if you could write

```plaintext
... department(person)
...
department(person) := ...
```

With sinister function calls (so named because they are on the left) you can. This idea was found in Pop-2 and SETL and is used a very great deal in S. Strictly speaking, “department” and “department:=” are different names of different but related procedures.

The usual parameters of a procedure come in four kinds:

**in** Input parameters are passed by value. In principle this involves making a copy. Such a parameter may not be changed by the procedure. It’s a local **const**, not a local **var**. So a compiler would normally be able to avoid copying things like arrays. The actual parameter may be any expression of the right type.

**in out** Input parameters are local variables whose value is copied in on entry to the procedure and copied back out on return from it. The actual parameter must be a variable of the right type, including an indexed or sliced one, but is not changed while the procedure is running. (Although a smart compiler might cheat if it couldn’t be caught out.) A const identifier may not be passed as an in out parameter, with or without indexing or slicing. A var identifier that is passed as an in out parameter must be initialised before the procedure is entered, and may not be passed as any other parameter, not even an in one.
Output parameters are local variables which are uninitialised on entry, and whose values are copied out when the procedure returns. The procedure must initialise such parameters before returning. The actual parameter must be a variable of the right type. It is not changed while the procedure is running.

A const identifier may not be passed as an in out parameter, with or without indexing or slicing. A var parameter that is passed as an out parameter need not be initialised before the procedure is entered. It may not be passed as any other parameter, not even an in one.

Result parameters (after ⇒) are like out parameters except that in assignments \( v := f(\ldots) \) or \( (v_1, \ldots, v_n) := f(\ldots) \) the left hand side do not count as out parameters, so \( x := f(x) \) is legal.

A Chatterton program may not use an identifier until after its declaration. This applies to procedures as well as variables. There is no problem with a procedure calling itself, because such a call must follow the procedure heading. There is a problem when there are procedures \( f \) and \( g \) such that \( f \) calls \( g \) and \( g \) calls \( f \). Chatterton solves this the same way that Pascal does: with forward declarations. A forward declaration provides the interface of a procedure without any body:

\[
\begin{align*}
\text{name proc rest of heading } &; \text{forward} \\
\end{align*}
\]

A forward declaration must be completed by an ordinary procedure declaration in the same serial clause. The only language constructs allowed between a forward declaration and its completion are other procedure and forward declaration.

There are no artificial restrictions on what kinds of declarations may occur inside others. In particular, any serial clause may contain procedure declarations, even the serial clause that is the body of a procedure.

6 Control structures

6.1 Procedure calls

Procedure calls are conventional, except that there may not be any characters at all, not even white space, between a procedure name and the left parenthesis that follows it. The actual parameter corresponding to an “in” formal parameter may be any expression of the right type. The actual parameter corresponding to an “out” parameter may be any variable of the right type. The actual parameter corresponding to an “in out” parameter may be any variable of the right type that has definitely been initialised.

Here “variable” includes slices. For example, here is quicksort.
sort proc (a in out [str])
  n const = size a
  if n > 1 then
    pivot const := a[0]
    e var := 0  a[0,e) all = pivot
    l var := 1  a[e,l) all < pivot
    g var := n  a[g,n) all > pivot
    loop while l < g
      x const = a[l]
      if x < pivot then
        l +:= 1
      else if x > pivot then
        g -:= 1
        (a[l], a[g]) := (a[g], x)
      else
        e +:= 1
        l +:= 1
        (a[e], a[l]) := (x, a[e])
      end if
    end loop
    loop while e > 0
      l ::= 1
      e ::= 1
      (a[e], a[l]) := (a[l], a[e])
    end loop
    sort(a[0,l))
    sort(a[g,n))
  end if
end proc

There is one unusual feature copied from Pop-2. A procedure call \( f(e_1, e_2, \ldots, e_n) \) may also be written \( e_1.f(e_2, \ldots, e_n) \), and as a special exception, \( f(e) \) may be written \( e.f \) without trailing parentheses. This means that projection and update functions of algebraic data types may be written using the familiar dot notation. Thus in

date data date(year var int, month var int, day var int)
today var := date(2010, 8, 14)
today.day := today.month*2

the last line is equivalent to

day(today) := month(today)*2

This makes it possible to use CHATTERTON for experiments to find out whether dot notation is simply more popular than function call notation or whether it is actually better.
6.2 Assignments

Assignments are expressions and have values. The value of an assignment is always the value that is assigned to the variable. Chatterton does not include mixed mode arithmetic, so there is never any invisible conversion from one type to another in assignments or elsewhere.

In a plain assignment \( lhs := rhs \) the left hand side may be any variable, simple, indexed, or sliced, provided it has the right type. The replacement for a slice need not be the same size as what it replaces.

When the right hand side has a tuple value, the left hand side may be a whole variable with that type, or may be a tuple of variables, and so on recursively. So to swap two variables we may write

\[
(x, y) := (y, x)
\]

Chatterton includes update assignments using \( +:=, -:=, *:=, /:=, \text{div}:=, \text{mod}:=, ^:=, **:=, @:=, @@:=, \text{max}:=, \text{min}:= \). In such assignments the left hand side must already have a value. The value of an update is the new value assigned to the variable. If \( m = 2 \) then the value of \( m+:= 1 \) is 3.

6.3 Serial clauses and blocks

A block or compound statement has the form

\[
\begin{align*}
\text{begin} & \\
\{ & \text{declaration ;} \\
| & \text{statement ;}\} \\
\text{end}
\end{align*}
\]

This is very like a block in C or Java.

A serial clause is exactly the same thing but without the \text{begin} or \text{end}. Compound control structures have serial clauses as parts, like Algol 68 or Ada, not single statements, like C.

The only time you would ever use a block is when you want to declare something for only part of a larger serial clause.

In the Chatterton dialects you will see, semicolons are completely optional. You can use them as separators (as in Pascal) or terminators (as in Ada); you have include them or exclude them; and it makes no difference. This is very like the programming language Eiffel, but there are no situations in Chatterton where semicolons are actually necessary.

Most statements are expressions with a value as well as an effect. Only \text{assert}, \text{put}, and \text{print} never have values. If the last thing in a serial clause is an expression, the serial clause has a value.

6.4 If statements

\[
\text{if test expr then}
\]
An if statement (or if expression; same thing) is executed by evaluating the test expressions from first to last until one is true, then evaluating the corresponding serial clause. If no test expression is true, the else part is evaluated if there is one, or nothing is done if there is not.

In a context where no value is required, there need not be an else part, and the other arms need not have values of the same type. In a context where a value is required, an else part is required, all the serial clauses must have values, and they must all be of the same type.

### 6.5 Case statements

Case statements select one of several alternatives based on the value of a single expression. There are two slightly different kinds: ones for pattern matching and ones for ordering.

```
case <expr> of
  { <guard>}+ <serial clause>+ 
  [else <serial clause>]
end case
```

<guard> ::= 
  ( <relop> <expr> | in <interval> | <pattern> ) 
  [where <expression>]

<pattern> ::= 
  wildcard 
  | identifier 
  | constant 
  | pattern tuple 
  | array tuple 
  | record constructor(pattern...)

### 6.6 Loop statements

The outer brackets of loops are copied from Ada. The way that `while` and `until` are used is copied from a Lisp dialect. The `present` and `absent` parts are inspired by Charles Zahn’s “situation case”, but rather more limited. The form of the `for` parts is adapted from MESA.
loop
  [  for identifier [down] in interval[:]
     {  for identifier [down] in interval[:]
            | where expression[:]
       }...]
     {  declaration[:]
            | statement[:]
            | while expression[:]
            | until expression[:]
       }...
  [  present
       serial clause ]
  [  absent
       serial clause ]
end loop

The basic idea is that you loop while there is hope until you have found
what you are looking for.

If the expression of a while evaluates to true, the loop continues, otherwise
the search has failed, and control is transferred to the absent section. The loop
has not found what we are seeking. In the absent section, none of the variables
introduced inside the loop exists any longer.

If the expression of an until evaluates to false, the loop continues, otherwise
the search has succeeded, and control is transferred to the present section. In
the present section, all of the variables introduced in the loop before the first
until are still available. (Variables introduced after that point are still visible,
hiding any others of the same name outside, but may not be used.)

If both the present and the absent are left out, it is as if you had written

present
  true
absent
  false

If a context where no value is required, either or both of these parts may be left
out. In a context where a value is required, both must be there or both omitted;
if both are there then both must have values of the same type.

The for parts introduce identifiers that range over intervals of integers.
Other kinds of sequence may be introduced if there is need for them in ex-
periments. A where part is basically a very disciplined continue statement,
but expressed positively.

As an example, suppose we are given an array of strings a and want to find
the first element of it that begins with a string s. We could write

loop for i in [0,size a] where size a[i] ≥ sizes
     until a[i][0,size s) = s
present
print "Found", a[i], "at", i

absent
print "Not found"
end loop

See how the present part uses i. It is allowed to. The absent part is not.

6.7 Input/Output

For present purposes, CHATTERTON needs to be able to express reading from standard input and writing to standard output. It does not any more general facilities, so it has none.

Input/output in CHATTERTON is a mishmash of Algol 68, Pascal, a soupçon of PL/I, and a little bit of AWK. There is even a reminiscence of Burroughs Algol. But it is a principled mishmash that is consistent with the rest of the language.

There are four constructions.

• get var

   This considers the input stream as island “words” embedded in a sea of white space characters. To consume a word, first leading white space characters are skipped, then the word is read until the next white space character, that character is discarded, and the word is converted to the appropriate type. If skipping white space leaves us at the end of standard input with no word, get returns false. If there is a word, and it is not the right shape, that’s a run time error. If reading was successful, get returns true.

   char if var is a character variable, the word should contain one non-blank character. That character is the result. If you are familiar with C, this is like %1s, not like %c.

   bool the word should be “true”, “T”, “yes”, “y”, or “1”, or “false”, “f”, “no”, “n”, or ‘0’. Alphabetic case is ignored.

   int the word should be a decimal integer, possibly with embedded or underscores, and possibly with a leading +, -, or ~ sign. It may also be written in a non-decimal base using the same convention as CHATTERTON source code.

   item the word should be a decimal floating point number, possibly with embedded underscores. The number’s sign and the scaling factor’s sign may be written using +, -, or -. The word may also be “±NaN”, “±Inf”, or “±Infinity”. Alphabetic case is ignored.

   str the word is the result, unchanged.

• tuple get (v₁, . . . , vₙ) is equivalent to get v₁ and . . . and get vₙ. The var may also be a variable of tuple type, in which case its fields are read just like this. For example,
ymd var (int,int,int)
hms var (int,int,int)
get (ymd, hms)

would read six integers, the first three being the (y,m,d) components of ymd and the last three being the (h,m,s) components of hms.

Returning a Boolean value is so that you can write a loop like this:

```
loop while get (x, y, z)
body
end loop
```

• get_line(s)

The get operator is an operator because it applies to many types. This one applies to strings only. The get_line procedure considers the input stream to be a sequence of lines terminated by newlines. If we are already positioned at the end of the input stream, get_line returns false, having set the string to empty. Otherwise, it reads characters from the input until a newline has been read, sets the string to the sequence of characters preceding the newline, and returns true.

• put expressions

This writes elementary items to standard output separated by spaces. It does not generate newline characters unless they are part of the data.

- bool Boolean values are written as “true” or “false”.
- char Characters are written without quotes or escapes.
- int Integers are written in decimal with the fewest possible digits, using “~” for negation if necessary.
- real Floating point numbers are written in something like C’s \%g format, with enough digits to read back.
- tuple the elements of the tuple are written from left to right, with one space between each adjacent pair. You need not write the outer parentheses of a tuple.
- str the characters of a string are written in ascending index order with no quotes or escapes and no separating spaces.
- array the elements of the array are written in ascending index order, with one space between each adjacent pair, except with the element type is char.
- set sets are written like arrays; the order of the elements is not defined.

• print expressions

This is exactly the same as put expressions followed by put "\n". This is basically the same as the default behaviour of the print statement in AWK.

28
6.8 Assertions

An assertion has the form `assert expression` and has no value. It expresses the programmer’s intent that the expression should be true whenever control reaches that point.

7 On Assignment and Equality

One student argued that using `=` for assignment and `==` for equality would make Chatterton better.

There are three approaches to the handling of equality and assignment in programming languages, which we may call the Fortran way, the Algol way, and the PL/I way.

The Fortran way is to use `==` for assignment and some other symbol for equality. Fortran used `.LT.` `.GE.` `.GT.` `.LE.` `.NE.` and `.EQ.` for comparison. Fortran couldn’t use `:=` for assignment or even use `<` or `>` in comparisons because those characters simply were not available to it in ancient character codes.

This flagrant abuse of the mathematical symbol for equality caused great confusion amongst beginning programmers.

The Algol way is to use `:=` for assignment and `=` for equality. After all, `=` just is the equality symbol, and if you have any choice at all, it would be pretty perverse to use it for anything else. In Algol, all the comparison operators were technically single characters. Now that we have Unicode, `<`, `≥`, `>`, `≤`, `=`, and `≠` are again single characters.

The PL/I way is to be even more perverse than Fortran. PL/I uses `=` for assignment and it uses `==` for equality comparison. Lest you think that too grotesque to survive, that convention survives to this day in Visual Basic, where `X = Y = Z` means “test whether Y equals Z and assign the Boolean result to X”.

When I listed the languages that followed the Algol convention, such as Pascal, Modula 2, and Ada, it was interesting to notice that most of them were invented in Europe, or were very close relatives of European programming languages. Fortran and PL/I are of course American programming languages. Tracing the ancestry of Java, Algol 60 → CPL → BCPL → B → C → C++ → Java, we notice that the languages up to BCPL are European ones using `:=` for assignment, and the languages from B on are American ones using `=` for assignment.

It seems as though this is a readability issue that should be studied experimentally too. Of course the Chatterton family could be enlarged with another parameter, Atlantic, with values European (`:=` for assignment including updates and `=` for equality) and American (`=` for assignment including updates and `==` for equality).

The same student suggested, if I understand him correctly, that a distinction between initialisation and (re)assignment might be a good idea. That idea is
found in Dijkstra’s notation where

\[ x \text{ \texttt{vir} int} := 1 \]

initialises a variable and

\[ x := 2 \]

reassigns it. It’s also found, sort of, in the language Go, where

\[ x := 0 \]

declares-and-initialises \( x \) to have the same type and current value as the expression 0, whereas

\[ x = 0 \]

reassigns 0 to \( x \). Possibly one might use \( = \) or \( := \) for a reassignment, but I don’t actually have notations I’m happy with yet.

8 Data Flow

This section describes the next revision of Chatterton. Instead of \texttt{in}, \texttt{in out}, and \texttt{out} keywords copied from Ada, we might use \texttt{rd}, \texttt{rw}, and \texttt{wr}, where \texttt{rd} means “may read”, \texttt{rw} means “may read and may write”, and \texttt{wr} means “must write and then may read or write”. These correspond to “\texttt{con}”, “\texttt{var}”, and “\texttt{vir}” in Dijkstra’s notation.

A procedure heading may be followed by read, read-write, and write lists. Indeed, any block may include such lists, with the default being to infer them from use.

We can define \( \mathit{Rd}(e) \), \( \mathit{Rw}(e) \), and \( \mathit{Wr}(e) \) as follows:

- constant \( c \): \( \mathit{Rd} = \mathit{Rw} = \mathit{Wr} = \{\} \)
- identifier \( i \): \( \mathit{Rd} = \{i\}, \mathit{Rw} = \mathit{Wr} = \{\} \)
- indexed \( v[e] \): \( \mathit{Rd} = \mathit{Rd}(v) \cup \mathit{Rd}(e), \mathit{Rw} = \mathit{Rw}(v) \cup \mathit{Rw}(e), \mathit{Wr} = \mathit{Wr}(v) \cup \mathit{Wr}(e) \)
- sliced \( v[b,e] \): \( \mathit{Rd} = \mathit{Rd}(v) \cup \mathit{Rd}(b) \cup \mathit{Rd}(e), \mathit{Rw} = \mathit{Rw}(v) \cup \mathit{Rw}(b) \cup \mathit{Rw}(e), \mathit{Wr} = \mathit{Wr}(v) \cup \mathit{Wr}(b) \cup \mathit{Wr}(e) \), and similarly for the other kinds of slice
- call \( f(e_1,\ldots,e_n) \): let \( E \) be all the \texttt{rd} arguments, all the \texttt{rw} arguments, and all the indices of the \texttt{wr} arguments; let \( M \) be the top level variables of the \texttt{rw} arguments and the \texttt{wr} arguments; let \( D \) be the \texttt{wr} arguments that are whole variables. Then \( \mathit{Rd} = \mathit{Rd}(f) \cup \bigcup_{e \in E} \mathit{Rd}(e), \mathit{Rw} = \mathit{Rw}(f) \cup \bigcup_{e \in E} \mathit{Rw}(e) \cup M, \) and \( \mathit{Wr} = \mathit{Wr}(f) \cup D. \)
- tuple \( e_1,\ldots,e_n \): \( \mathit{Rd} = \bigcup_{i=1}^{n} \mathit{Rd}(e_i), \mathit{Rw} = \bigcup_{i=1}^{n} \mathit{Rw}(e_i), \mathit{Wr} = \bigcup_{i=1}^{n} \mathit{Wr}(e_i) \).
- unary \( \phi e \): \( \mathit{Rd} = \mathit{Rd}(e), \mathit{Rw} = \mathit{Rw}(e), \mathit{Wr} = \mathit{Wr}(e) \).
\[e_1 \text{ and } e_2: \quad \text{Rd} = \text{Rd}(e_1) \cup \text{Rd}(e_2), \quad \text{Rw} = \text{Rw}(e_1) \cup \text{Rw}(e_2), \quad \text{Wr} = \text{Wr}(e_1).\]

\[e_1 \text{ or } e_2: \quad \text{Rd} = \text{Rd}(e_1) \cup \text{Rd}(e_2), \quad \text{Rw} = \text{Rw}(e_1) \cup \text{Rw}(e_2), \quad \text{Wr} = \text{Wr}(e_1).\]

\[\text{binary } e_1 \psi e_2: \quad \text{Rd} = \text{Rd}(e_1) \cup \text{Rd}(e_2), \quad \text{Rw} = \text{Rw}(e_1) \cup \text{Rw}(e_2), \quad \text{Wr} = \text{Wr}(e_1).\]

\[\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ fi: } \quad \text{Rd} = \text{Rd}(e_1) \cup \text{Rd}(e_2) \cup \text{Rd}(e_3), \quad \text{Rw} = \text{Rw}(e_1) \cup \text{Rw}(e_2) \cup \text{Rw}(e_3), \quad \text{Wr} = \text{Wr}(e_1) \cup (\text{Wr}(e_2) \cap \text{Wr}(e_3)).\]

\[\text{and so on.}\]

This would be used for two purposes.

8.1 Non-interference checking

In a tuple \((e_1, \ldots, e_n), \text{Rw}(e_i) \cap (\text{Rd}(e_j) \cup \text{Rw}(e_j)) = \emptyset\) unless \(i = j\).

In a function call \(f(e_1, \ldots, e_n), \text{Rw}(e_i) \cap (\text{Rd}(e_j) \cup \text{Rw}(e_j)) = \emptyset\) unless \(i = j, \text{Rw}(e_j) \cap (\text{Rd}(f) \cup \text{Rw}(f)) = \emptyset\).

With a binary operation \(e_1 e_2\) or \(e_1 \psi e_2, \text{Rw}(e_i) \cap (\text{Rd}(e_j) \cup \text{Rw}(e_j)) = \emptyset\) unless \(i = j\).

That is, if one operand may write a variable, no other operand may read or write that variable.

8.2 Definedness checking

The Wr sets keep track of what variables must be completely assigned to. You may not use a variable until it has been completely assigned to. An assignment

\[a[i] := e\]

is considered to be an abbreviation for

\[a := a \ (i, e)\]

and as such a use of “a” as well as writing to it, so “a” must already be defined before this is done.

We may consider two variants of Chatterton: one where this restriction is enforced, and one where it is not (though it happens not to be violated in the examples). Here is something where a characteristic of a language with no visible correlate in the source code ought to have an effect on readability. If you are reliably informed that a program cannot contain an interaction like

\[\text{inc}(i) \ - \ is \ i \ 1 \ or \ 0?\]

\[\text{inc}(x) \ - \ is \ i \ 2 \ or \ 0?\]

then it ought to be easier to read than a program where you don’t know this and have to watch out for it.
9 YACC Grammar

9.1 Keywords

abs absent and assert begin const div down elif else end for get if in loop max min mod not or out present print proc put size then type until var where while

9.2 Other Operators

+ + := − − := @ @ := @@ @@ := ** ** := * * := / := \* := div := mod :=

min := max := × > > = " = < <= :=

9.3 Built-in functions

acos acos asin asinh atan atan2 cbrt chr copy sign cos cosh dec dig exp expm1 fields get_line hex hypot is_alphanumeric is_blank is_control is_digit is_finite is_graph is_infinite is_letter is_lower is_NaN is_normal is_print is_punctuation is_space is_upper join log log1p oct ord sin sinh sqrt tan tanh to_dec to_hex to_lower to_oct to_title to_upper tokens val

9.4 Yacc grammar

Here is a grammar for Chatterton ready to be processed by the UNIX parser generator Yacc.

```yacc
%token VALUE ID
%token IDLP LP RP COMMA LB RB SEMI LC RC
%token BEGIN END IF THEN ELIF ELSE LOOP FOR DOWN WHERE WHILE UNTIL
%token ASSERT CONST VAR PROC OUT FORWARD OUTPUT
%right GETS GIVES GIVES0 /* := and + := − := @ @ := @@ @@ := ** ** := * * := / := \* := div := mod :=

min := max := × > > = " = < <= :=

%left NOT /* not (logical operators are spelled out) */
%left RELOP EQUAL IN /* < <= >= ~= = in */
%nonassoc MAX MIN /* max min */
%left PLUS MINUS CAT NEG /* + − @ */
%left DIV MOD SLASH TIMES REP /* div mod / * @@ */
%right POW /* ** \* */
%nonassoc SIZE GET /* size, get */
%

program
    : serial
    ;
```
expression
  : VALUE /* false, true, 'x', 12, 3.4, "foo" */
  | simple_var
  | indexed_var
  | sliced_var
  | call
  | BEGIN serial END
  | LP expressions RP /* tuple */
  | LC RC /* empty array */
  | LC expressions RC /* non-empty array */
  | GET expression
  | SIZE expression
  | if_body END IF
  | loop_body END LOOP
  | expression POW expression
  | expression DIV expression
  | expression MOD expression
  | expression SLASH expression
  | expression TIMES expression
  | expression REP expression
  | NEG expression
  | expression PLUS expression
  | expression MINUS expression
  | expression CAT expression
  | expression MAX expression
  | expression MIN expression
  | expression RELOP expression
  | expression EQUAL expression
  | expression IN interval
  | NOT expression
  | expression AND expression
  | expression OR expression
  | expression GETS expression
  ;

simple_var
  : ID
  ;

indexed_var
  : simple_var index
  | indexed_var index
  ;

index
  : LB expression RB
  ;
sliced_var
  : simple_var slice
  | indexed_var slice
  ;

slice /*I'd like all four kinds of interval, but...*/
  : LB opt_expression COMMA opt_expression RB
  | LB opt_expression COMMA opt_expression RP
  ;

opt_expression
  : /* EMPTY */
  | expression
  ;

call
  : ID IDLP RP
  | ID IDLP expressions RP
  ;

expressions
  : expression COMMA expressions
  | expression
  ;

if_body
  : if_head ELSE serial
  | if_head
  ;

if_head
  :      IF serial THEN serial
  | if_head ELIF serial THEN serial
  ;

loop_body
  : loop_body WHILE expression opt_semi
  | loop_body UNTIL expression opt_semi
  | loop_body expression opt_semi
  | loop_head expression opt_semi
  | loop_body LOOP
  | LOOP
  ;

loop_head
  : loop_head WHERE expression opt_semi
  | loop_head FOR ID IN interval opt_semi
  | loop_head FOR ID DOWN IN interval opt_semi
  | LOOP FOR ID IN interval opt_semi
  | LOOP FOR ID DOWN IN interval opt_semi
  ;
interval
  : LB expression COMMA expression RB
  | LB expression COMMA expression RP
  | LP expression COMMA expression RP
  | LP expression COMMA expression RB
  ;

serial
  : serial_body statement opt_semi
  ;

statement
  : expression
  | ASSERT expression
  | OUTPUT expressions
  ;

serial_body
  : /* EMPTY */
  | serial_body declaration opt_semi
  | serial_body statement opt_semi
  ;

opt_semi
  : /* EMPTY */
  | SEMI
  ;

declaration
  : ID CONST opt_type EQUAL expression
  | ID VAR opt_type GETS expression
  | ID VAR type
  | ID PROC opt_arguments opt_results opt_semi serial END PROC
  | ID PROC opt_arguments opt_results opt_semi FORWARD
  ;

opt_arguments
  : LP RP
  | LP arguments RP
  ;

arguments
  : arguments COMMA argument
  | argument
  ;

argument
opt_results : /* EMPTY */
| GIVES LP results RP
| GETS LP rhs RP
| GIVES0 type
;

results : results COMMA result
| result
;

result : ID OUT type
| ID type
;

rhs : ID IN type
| ID type
;

type : interval /* LP types RP */
| LC types RC /* tuple */
| LB RB type
| ID /* bool, char, int, real, str */
;

opt_type : /* EMPTY */
| type
;

types : types COMMA type
| type
;
10 Afterthoughts to be integrated elsewhere

10.1 Things to vary

- European (:= =) or American (= ==) assignment and equality.
- Multiword identifiers: bring back A. Add F. U runs words together in lower case, F runs them together in upper case.
- Which side of a declaration does the identifier go? The main text already has sides B (id var [type] [:= initial]), M (var id [: type] [:= initial]), and E (var [type] id [:= initial]). Add J (type id [:= initial | = constval]) with [] suffix on types.
- Semicolons: +Z (required, may only be last token on line), +D (required, may only be first token on line).
- Keyword case: L(ower), T(itle), U(pper). Which is better:

  is_leap_year proc (year int) => (l bool)
  l := year mod 4 = 0 and (year mod 100 ~= 0 or year mod 400 = 0)
  end proc

  or

  is_leap_year Proc (year int) => (l bool)
  l := year Mod 4 = 0 And (year Mod 100 ~= 0 Or year Mod 400 = 0)
  End Proc

  or

  is_leap_year PROC (year int) => (l bool)
  l := year MOD 4 = 0 AND (year MOD 100 ~= 0 OR year MOD 400 = 0)
  END PROC

10.2 Variables

A variable is

- an identifier declared var, out, or in out.
- a variable with an index
- a variable with a slice
- a field access v.f or f(v) where v is a variable
• a (tuple) whose elements are variables or wildcards

• an {array} whose elements are variables or wildcards

A wildcard is \_; it’s like a predefined variable but accepts values of any type and discards them.