

# A Programming Language without Keywords

W. Douglas Maurer<sup>1</sup>

<sup>1</sup> Department of Computer Science, George Washington University, Washington, USA

Correspondence: W. Douglas Maurer, Department of Computer Science, George Washington University, Washington, DC 20052, USA. E-mail: maurer@gwu.edu

Received: June 13, 2012 Accepted: October 20, 2012 Online Published: January 13, 2013

doi:10.5539/cis.v6n1p100

URL: <http://dx.doi.org/10.5539/cis.v6n1p100>

## Abstract

Computer programming today is far too dependent on one particular natural language, namely English. Mathematics is not dependent on English, and programming should not be, either. In order to eliminate the dependence of programming on English, we have constructed a programming language in which the keywords are replaced by symbols available on standard keyboards. Each of these symbols is chosen to suggest the idea of the keyword it replaces. Many of them have more than one use, but always in a way that a compiler can detect. In this way a professional can write a program without ever having to think in English. Our language is called *\**, and it also embodies an innovative approach to programming language safety.

## 1. Introduction

We introduce here a programming language whose name is *\** (pronounced *star* in English, *étoile* in French, *αστερι* in Greek, *estrella* in Spanish, *Stern* in German, and so on) and which has no keywords. The premise of *\** is that, just as mathematics relies entirely on symbolic notation, computer programming should also.

We are all familiar with the fact that mathematics is a universal language. People all over the world, regardless of whether they know English, can read and understand mathematics. One can easily imagine a world in which this was not true — in which one had to use *Integral* instead of an integral sign, *SquareRoot* instead of a radical sign, *Derivative*( $y, x$ ) instead of  $dy/dx$ , and so on. People in non-English-speaking countries could still read mathematics in such a world, but it would be much less universal. Yet this is what people in such countries have to do every day in writing programs. They have to use `while` and `if` and `else` and `for` and `return`, which are just as much a detraction from universality as *Integral* and *SquareRoot* and *Derivative* would be.

If *\** seems, to devotees of other programming languages, to be unduly influenced by C, it is probably because C already took a number of steps in the direction of our premise. English reserved words in other languages, such as `access`, `and`, `array`, `begin`, `end`, `function`, `loop`, `not`, `or`, `procedure`, and `subroutine` were replaced in C and its successor languages by symbols. On the other hand, languages other than these have not been forgotten; for example, *\** uses both `=` and `:=` as assignment symbols, with slightly different meanings (see §4.10).

The idea of having a language without keywords is not new. Indeed, it was used in APL, which even eliminates the words `sin`, `cos`, and `tan`, replacing them, respectively, by  $1^\circ$ ,  $2^\circ$ , and  $3^\circ$ , and similarly for inverse and/or hyperbolic functions (Wiedmann 1974, p. 136). However, the APL design goal was not natural language independence, but rather conciseness: “It is not uncommon for an APL program to require a tenth as many statements as a program in another language” (Wiedmann 1974, p. vii). APL also has its own keyboard, which is different from those of other languages (Wiedmann 1974, p. 16). By contrast:

- All the necessary special characters of *\** are available on standard keyboards.
- *\** is designed for efficiency, as was C.
- Most of the primitive operations of APL are specific to APL, and are not primitive operations in other languages. In *\**, all the primitive operations are found in common algebraic languages.

It has also always been possible to translate the keywords of C into any other language, using C’s `#define` facility. In German, for example, one may write `#define wenn if` and `#define weil while` and the like, and then use `wenn` and `weil` (and so on) in a German-language program. This process is hampered in C, however,

by the lack of letters with diacritical marks, such as the ü in zurückkehren (meaning return). This is not a problem in \*, which uses Unicode (Aliprand et al. 2000) throughout.

In addition to being universal, \* has:

- A novel approach to the concept of safety (see §1.3). This lets managers allow unsafe programming language features for some programs, and parts of programs, while disallowing them for others.
- Extended data structure notation; this is done by combining basic data structures of other languages, such as patterns, sets, resizable arrays, and associative arrays, with the declarators of C (see §4.7).
- Certain features of scripting languages, such as dynamic types (although these are optional in \*), pattern matching, and loops with `else`. On the other hand, \* does not employ features of only one scripting language, such as the special characters at the start of identifiers in Perl, the compiler-sensed indentation in Python, or “everything is an object” as in Ruby. The # character which starts a comment in Perl, Python, and Ruby has other uses in \*, and therefore cannot be used there for that purpose; // is used instead.

All features of \* (with the possible exception of `<=>`; see §2.2 below) correspond to features in other well-known programming languages. The contribution of \* is not the introduction of new language features; it is solely the internationalization of computer programming.

### 1.1 Basic Design Approach

There are many characters and character combinations, such as `->`, which cannot start a statement or a declaration in most programming languages. Many of these can start a \* statement or declaration; these include the following (further expanded upon below):

? (if)	-> (new)
: (else)	<- (delete)
?? (while)	>, (break)
?! (catch)	<, (continue)
>. (return)	^: protected:
<. (exit)	(switch)
>n (forward goto label n)	= (#define)
<n (backward goto label n)	[ (declarations)

All of these are meant to suggest something about the symbols that makes them easy to remember, and that are independent of the programmer’s native language, as illustrated in Table 1. In keeping with the universality of \*, the dollar sign is not a binary or unary operator in \* or any of its future extensions.

Table 1. Operators in \* and what they suggest

Operator	Associated Suggestion
0; (abstract class definition)	0 suggests “no objects”
<n (backward goto n)	Go back (a stylized back arrow)
_ (base class constructor name)	The underscore _ suggests <i>base</i>
>, (break)	Go to the end of the loop (a stylized forward arrow)
>>n- (close file n that was open for read)	No (-), file is no longer open
<<n- (close file n that was open for write)	No (-), file is no longer open
.v (const variable v)	const = unchanging = stopped (like static variables)
<, (continue)	Go to the start of the loop (a stylized back arrow)
= (#define)	#define x y sets x equal to y
<- (delete)	Like new (->) but in the opposite direction

<i>Operator</i>	<i>Associated Suggestion</i>
[0] (dynamic type)	The digit 0 suggests “no fixed type”
: (else)	The ternary <i>if-then-else</i> operator ? :
{: (endless loop)	Anonymous label (:); block becomes a loop
! (exception declarations)	! suggests “Oh, no! That’s wrong!”
<. (exit)	Go back to the operating system (a stylized back arrow)
>! (finally)	> = after; ! = exception ( <i>finally</i> block is done <i>after</i> catching)
,, (friend)	Two friends holding hands
>n (forward goto n)	Go forward (a stylized forward arrow)
? (if)	The ternary <i>if-then-else</i> operator ? :
+, (#include)	All existing source statements <i>plus</i> (+) these new ones
--- (inline)	Suggests a <i>line</i>
!# (NaN)	Not a Number (! = not, # = number)
-> (new)	p = new $\alpha$ ; becomes p = -> $\alpha$ ; (p now <i>points</i> to $\alpha$ )
>>n+ (open file n for read)	Yes (+), file is now open for read
<<n+ (open file n for write)	Yes (+), file is now open for write
^ (overloaded operators)	Up-arrow ^ pointing up, or over ( <i>overloaded</i> )
/ (packed arrays and records)	Slash (/) suggests “cutting down on the size”
-: (private:)	No, outside access is not allowed (also - from UML)
^: (protected:)	“Protected” = roof over one’s head (^ looks like a roof)
+: (public:)	Yes, outside access is allowed (also + from UML)
>. (return)	Go to the end (a stylized forward arrow)
?, (safety statement)	? = “ <i>what</i> unsafe features are to be allowed?”
~{ (set complement, unary)	The bitwise complement operator ~
# (set declarations)	The <i>number</i> of elements in the representation of the set
#(t) (sizeof t)	The size of t is the <i>number of</i> bytes in t
.t (static variable of type t)	Static = stopped (a period <i>stops</i> a sentence)
` (superscripts)	` is <i>above</i> the line (like a superscript)
<=> (swap)	Two quantities; one goes one way, the other goes the other way
(switch)	Switches have alternatives (  means “or” in C)
<! (try)	< = before; ! = exception ( <i>try</i> block is done <i>before</i> catching)
&, (typedef)	All existing type names <i>and</i> (&) this new one
+ (unsigned declarations)	Negative values are not allowed
[] (void)	Void = nothing (nothing between the square brackets)
! (volatile variables)	! suggests “It might fly away!”
?? (while)	Like ? but a repeated test (appearing twice)

### 1.2 Names and Language Independence

Our basic design approach, as outlined so far, is subject to a possible charge of incompleteness. Merely eliminating the usual English-language keywords does not appear to be enough. One can write variable names in French or Chinese or Arabic or Hindi, but what about library names (the names of library functions, classes, exceptions, files, constants such as  $\pi$ , and GUI names)? There are thousands of these, and they are all either English words or based on English words.

One solution to this problem is to assign numerical codes to functions, as was done with some of the APL system functions (Wiedmann 1974, p. 198). This is unacceptable, since it immediately raises the problem of how to remember the codes. Another approach involves translating the specifications of all libraries that a program needs into the programmer's native language. The programmer may use these translated names, and then invoke a simple utility program which inserts whatever definitions are needed. If a German-language program uses `qwz1` (standing for *Quadratwurzel*, meaning "square root"), the definition = `qwz1 sqrt`; (see §2.6) is generated, and uses of `qwz1` become uses of `sqrt`. A more ambitious solution to the same problem, known as the Unibase, is described in §7.3.

### 1.3 Safety

Ever since Fortran IV and Algol 60, there have been "safe" and "unsafe" features of programming languages, starting with range checking for array indices out of bounds. There have been "unsafe" languages like Fortran IV and C, which do no range checking, and which were nevertheless very popular. There have also been "safer" languages like Algol 60, Pascal, and Ada, which do range checking; and some of these, particularly Java, are also very popular. The concept of programming language safety has since expanded to cover a wide variety of features, such as `goto`, `if(x=y)` for `if(x==y)`, type coercion, pointer arithmetic, null pointer checking, switch defaults, array assignment, and deletion from dynamic storage (sometimes called the free store).

In this context, \* is more like C++, where you can have as much safety as you want. If you want to use built-in arrays with no range checking, you can do that; if you want to use array classes with overloaded subscripting operators which do range checking, you can do that too. Safety in \* is different from that in C++, however. In C++, you can make array classes available, but this does not prohibit the use of built-in arrays. In \*, every program has one or more *safety levels*, specified in *safety statements*, which \* enforces, preventing or dissuading programmers from using any of various selected features.

Every potentially unsafe feature of \* is either disallowed, allowed, or allowed with a warning, as specified in a safety statement. This allows programming managers to control safety by requiring programmers to use predesigned safety statements. Safety levels can be stratified, so that system programmers, for example, can be allowed to write more efficient programs that might be theoretically unsafe, while entry-level programmers can have more restrictive safety levels than others, to prevent them from doing too much damage while they are learning. Experimental software can also have lower safety levels than other software; and parts of a program can have their own safety statements (See 5.5).

## 2. Basic Data Types in \*

We start by taking up declarations and operators for integers (signed and unsigned), floating point numbers, Boolean quantities, characters, strings, patterns, integer subtypes, and dynamic types. We then take up variable names and the naming operator, which is applicable to general types.

### 2.1 Numeric Types

Integers and floating point numbers are declared in \* by giving their sizes (in bits, not bytes) explicitly, enclosed in square brackets. There is no confusion in \* as to whether a long integer is 32 bits long or 64 bits long, because the declaration starts with `[32]` or `[64]` (but see also §2.3). This is known, in \*, as a *type specification*; it is followed by a list of *declarators* separated by commas. In the simplest case, as given below, each declarator is just a variable name (see §2.5):

```
[8] α; (8-bit integer called α)
[16] β, γ; (short integers called β and γ)
[32] δ, ε, ζ; (integers called δ, ε, and ζ)
[64] η; (long integer called η)
```

In \*, as in most languages, "constants" such as 3 and 7.5 are called *literals*, while *constants* are variables whose values cannot change, like `const int z=5`; in C++ (or `final int z=5`; in Java). Such a constant is declared, in \*, by writing `.` before its name; thus, for example, `[32] x, y, .z=5`; in \* is like `int x, y; const int z=5`; in a version of C++ in which `int` variables are 32 bits long.

An integer literal in \* can be written as in other languages (a sequence of digits, optionally preceded by a minus sign) but it can also be written in Unicode as an integer in any of various languages (Arabic, Chinese, Thai,

several languages of India, and so on). Note that digit characters from various languages cannot be mixed. The size of an integer in `*` can be arbitrary, and this is useful in defining a packed record (like a `struct` with bit fields in C; see §2.6).

Floating point numbers are given as above, but by writing `[`` instead of simply `[`. This includes `[`32]` for `float` and `[`64]` for `double`. The backquote character ``` is also used in floating point literals, in place of `E` (or `e`), so that `5`6` means  $5 \times 10^6$ , or 5,000,000 (This is not to be confused with `5`6`, which means  $5^6$ , as noted below.) Also, ``6` by itself means  $10^6$  in `*`, whereas `E6` by itself would be a variable name in C. (In all the above cases, ``` suggests a superscript.) Floating point literals can be specified in Arabic, Chinese, and the like in the same way that they would be in those languages. Also, the Unicode character  $\infty$  (for “infinity”) is allowed in `*`, as is  $-\infty$ ; while `!#` is used for Not a Number (NaN).

Unsigned integers are defined like signed integers, but by writing `[+` instead of simply `[` (the `+` suggesting that negative values are not allowed). This includes `[+1]` for single bits and `[+16]` for Unicode characters; these are always unsigned. In particular, `[+1]` also means Boolean (see §2.2).

The usual C arithmetic operators `+`, `-`, `*`, `/`, and `%` may be used in `*`. The Unicode characters  $\cdot$  (for multiplication) and  $\div$  (for division) may also be used. Note that `/` and  $\div$  have the same meaning in `*`; integer division (as in most languages) is used if and only if its operands are both integers. The exponentiation operator in `*` is ``` (this is meant to suggest `**` from Fortran or Ada, although `*` cannot use `**` since `a**b` means `a*(b)`). Here ``` is nonassociative, like `**` in Ada; `a`b`c` is illegal, and it must be given as either `(a`b)`c` or `a`(b`c)`. As in Ada, exponentiation has higher precedence in `*` than multiplication or division.

The ternary `?` operator and the shifting operators `<<` and `>>` are as in C. Right shifts are signed or unsigned, depending on whether they act on signed or unsigned quantities (as in C), so there is no need for `>>>` (from Java). Similarly, the bitwise operators `&`, `|`, `^`, and `!` may be used in `*` with their usual meanings in C. Operator precedence in `*` is like that of C, except that multiplication in `*` has higher precedence than division. Thus writing `a*b/c*d` in `*` produces `ab/cd` (as it would in algebra) and not `(ab/c)*d` (as it would in other programming languages).

## 2.2 Boolean Types and Packed Types

Since a Boolean variable is one bit long, and unsigned, it is given in `*` as `[+1]`; and `true` and `false` are, respectively, `1` and `0` in `*` (as in APL). One may always write `= true 1;` and `= false 0;` (see §2.6) to be able to use `true` and `false` in the usual way. In the same way, however, thinking in German, French, or Greek, one may write `= wahr 1;` or `= vrai 1;` or `= αληθης 1;`.

The usual Boolean-valued relational operators of C (`==` `!=` `<` `<=` `>` `>=`) may be used in `*`. This includes `==` for equality testing (see, however, §3.2). The Unicode characters  $\neq$ ,  $\leq$ , and  $\geq$  may be used in `*` with their usual meanings in mathematics. Inequality testing may be represented either by `!=` (from the C-based languages) or by `<>` (from Basic and Pascal). The logical operators `&&`, `||`, and `~` may be used in `*` with their usual meanings from C. The operator `<=>` means “swap” or “interchange”; this is useful, for example, in sorting.

There are also the *packed types* `[/1]`, `[/2]`, and `[/4]`. These are the same, respectively, as `[+1]`, `[+2]`, and `[+4]` for unsubscripted data; but an array of elements of a packed type is a packed array in the usual sense (see §4.1; the slash suggests “cutting down on the size”). Here `[/2]` is intended to be used for base-pair data, with four base pairs per byte; similarly, `[/4]` may be used for packed decimal data, with two decimal digits per byte. Packed arrays, as usual, save considerable amounts of space, but their processing takes extra time.

## 2.3 Further Basic Types

Further types in `*` include the following:

- *Characters* in Unicode (like `wchar_t` in C++) may be declared by `[']` (which is equivalent to `[+16]`). A character literal in `*` is as in C (including escape characters); but it can also be any Unicode character in single quotes. The use of `\'` or `\"` or `\\` (meaning, respectively, `'` or `"` or `\`) is the same in `*` as in other languages, as are the other usual escape characters. In particular, `\n` is the end-of-line escape character, used in output statements (see §5.1).

- *Strings* are declared by `["]` in `*` (this is like `wstring`, or “wide character string,” in C++). A string `s` consists of characters `s[1]` through `s[n]` for some integer  $n \geq 0$ . If  $n = 0$ , then `s` is the *null string*, denoted by `0`. The substring of `s` from `s[u]` through `s[v]` is denoted by `s[u..v]`, where `u` and `v` may be arbitrary integer expressions; and there are also `s[1..v]` (meaning `s[1..v]`) and `s[u..]` (meaning `s[u..n]`). A string literal in `*` can be as in C (including escape characters), but it can also be any Unicode string in double quotes. (Such

strings may be implemented in compressed form, saving space when a string contains mainly ASCII characters.) The C++ string concatenation operator `+` is allowed in `*`; note that `s1+s2` is always formed by copying both `s1` and `s2`.

We can set up a constant string such as `["] .es = "error";` here the value of `es` is always "error" and cannot change. We can also set up an ordinary string such as `["] s;` here the string value of `s` can become any of various strings during a run. However, *these strings themselves can never change*. Specifically, we cannot set `s[k] = c;` where `c` is a character, although we can set `c = s[k];`. (If you need to set `s[k] = c;` set up a resizable array `a` of characters, and set `a[k] = c;` you can convert `a` to a string `s` later by writing `s = a;`.)

- *Patterns* (see §5.3) are declared with `[/]`. A pattern literal in `*` is a string enclosed in `/` characters, as in Perl. (There is no conflict between this usage of `/`, in statements and initializations, and other usages of `/` in declarations.) Note that the flags of Perl are not in `*`; they are English-oriented (`i` = insensitive, `g` = global, `m` = multiline) and are handled differently in `*` (see §5.3).

- *Integer subtypes* (as in Ada, like integer subranges in Pascal) are declared in `*` by `[m..n]` (meaning the integers  $k$  with  $m \leq k \leq n$ ), for integers  $m$  and  $n$ .

- The declaration `[0]` specifies a *dynamic type*, which resembles the variant types of Visual Basic or the types of “typeless languages” such as Perl, Python, and Ruby. (The digit `0` suggests “no fixed type.”) An implementation of `[0] v;` involves an internal code indicating the current type of `v`.

Thus, for example, we may write:

```
[0] θ; (variable of a variant type called θ)
[+32] ι; (unsigned 32-bit integer called ι)
[`64] κ, λ; (long floating point numbers called κ and λ)
["] μ, ν; (strings called μ and ν)
['] π, ρ; (characters called π and ρ)
[/] σ, τ; (patterns called σ and τ)
[1..100] υ; (subtype called υ of integers k for 1 ≤ k ≤ 100)
```

## 2.4 Type Conversion and Coercion

Type conversion operators in `*` are much as in C; the new type (in its `*` form), enclosed in parentheses, is used as a conversion operator. What is `(int)x` in C becomes `([32])x` in `*`; what is `(double)k` in C becomes `([`64])k` in `*`; what is `(unsigned short)i` in C becomes `([+16])i` in `*`, and so on. For more general type conversions, `*` uses *declarators*, which extend those of C (see the end of §4.9).

Type *coercions*, or implicit conversions, in `*` are an extension of those in C and Java. A conversion from `[m]` to `[n]`, or from `[+m]` to `[+n]`, or from `[+m]` to `[n]`, is a widening conversion in `*` if and only if  $m < n$ . Let  $x$  have type `[m]` and let  $y$  have type `[n]`, where  $m < n$ ; then, in the expression `x op y` or `y op x` for some operator `op`,  $x$  is coerced to type `[n]` before `op` is applied. This holds even if  $m$  and  $n$  are not necessarily powers of 2, as might happen, for example, in a packed record (see §2.6). C is liberal with type coercion; Java disallows some of the coercions in C that are regarded as unsafe. In `*`, the allowed type coercions are either like those of C or like those of Java, depending on the safety level (see §5.5).

## 2.5 Variable Names

Variable names are declared in `*` much as they are in C, with declarators (see §4.6) separated by commas, followed by a semicolon, and preceded by a type. Unicode in the names of variables, functions, classes, and user-defined exceptions in `*` resembles Unicode in literals. Such a name can be a sequence of letters, digits, underscores, *apostrophes*, and *hyphens*, not starting with a digit, apostrophe, or hyphen. This is much like an identifier in other languages except for the apostrophes and hyphens, which are found in words in certain natural languages, such as French. (Note, however, that an apostrophe is not the same as a single quote, and a hyphen is not the same as a minus sign.)

Such a name can also be a sequence of characters from any of various languages (Russian, Arabic, Chinese, other Asian languages, and so on). Such characters may be mixed with Western numerals, but, aside from this, as with integer literals, characters from various languages cannot be mixed. Single-letter variable names are, of course, universal, as in algebra, and here `*` is case-sensitive, so that, for example, `x` and `X` are two different

variables. (The character `_` by itself is an identifier, but a special one, being the name of the base class constructor inside a derived class constructor; see §4.4.)

### 2.6 The Naming Operator

We now consider names other than variable names. The declarations of user-defined type names, except for enumeration type names, have a special syntax in `*` (see §4.8). For names of constants, expressions, enumeration types, records, packed records, unions, classes, generic classes, and exceptions, there is the *naming operator*, which is the unary equals sign. Writing `= v e;` sets `v` to be a name for `e`, which can be any of the following:

- A literal. We can write `= true 1;` and `true` then becomes a name for 1 (see §2.2).
- An expression. We can write `= max(x, y) x>y?x:y;` and `max(x, y)` becomes a name for `x>y?x:y`. This resembles the macro facility of C; but in `*` the actual parameters must be simple, unsubscripted variable names. In order to implement `i=max(j++,k--)` from C in `*`, we have to write something like `u=j++; v=k--; i=max(u,v);`. This avoids a problem associated with `j++` or `k--` being evaluated more than once.
- An enumeration type, resembling those of Ada (not C, because there the syntax of enumerated types resembles that of set literals in `*`; see §4.5). We can write `= dias (lun, mar, mie, jue, vie, sab, dom);` and `dias` becomes a name for an enumeration type having enumeration constants `lun` through `dom`. There is no ambiguity here because `*` has no comma operator (an enumeration type in `*` must have at least two enumeration constants). If `k` is a `*` enumeration constant, then the unary `-` and `+` applied to `k` are like `'PRED` and `'SUCC` (predecessor and successor) in Ada; thus `+mar` is `mie` and `-mie` is `mar` in the example above.
- A record (like `struct` in C). Writing `= complex {[`64] re; [`64] im;};` sets `complex` to be a name for a record type with real and imaginary parts of type `double`. We may later write `complex z1, z2;` and `z1` and `z2` will now have respective parts `z1.re`, `z1.im`, `z2.re`, and `z2.im`, as usual with complex numbers. Note that the semicolon at the end must be present.
- A packed record (like `struct` with bit fields in C). This is defined like a record, except that it starts with `{/;` instead of simply `{`. The `/` character is the same as that used for packed types (see §2.2). There is no need in `*` for syntax like `signif:23` in C for a 23-bit significand, since 23-bit unsigned integers are defined with `[+23]` (see §2.1).
- A union (like `union` in C). This is also defined like a record, except that it starts with `{+;` instead of simply `{` (Note that `+` is also the set union operator.)
- A class (see §4.4). Complex records, as above, cannot be directly added, subtracted, or the like; but they can if we turn `complex` into a class with overloaded operators, rather than a record.
- A derived class. A class `v` derived from `w` is indicated by replacing `v` by `v : w` in the declaration (as in C++; this is like `v extends w` in Java). There is no confusion here with `:` meaning `else`, since that must be in a statement, not a declaration. The terms “base class” and “derived class” are used in `*` rather than “superclass” and “subclass,” thus avoiding the confusion with subclass objects being *larger* than superclass objects, which is counterintuitive.
- A generic class (like template classes in C++). We may define a data structure node, with a pointer to type `con` and a pointer to the next node, or 0 to denote no next node, by writing `= node<con> {(con *)content; (node<con> *) next;};`. We can then use `node<complex>` or `node<[`32]>` or the like.
- An exception class. This is defined like any other class in `*`, except that its name always ends in `!` (see §5.4).

All of these names are intended to be given in the programmer’s native language, removing the necessity for the programmer to think in English.

### 3. Statements in `*`

Statements in `*` are defined much as they are in C. A statement is either simple or compound. The `*` language has simple statements with the logic of assignment, *if*, *else*, *while*, *for*, *break*, *continue*, *return*, *exit*, *switch*, *throw*, *catch*, *try*, *finally*, function call, and forward and backward *goto*. A compound statement, or block, consists of a left curly bracket `{` followed optionally by one or more local declarations, followed by one or more statements (simple or compound) and a right curly bracket `}` at the end.

Whitespace in `*` is like whitespace in C; in particular, line breaks in `*` are whitespace (unlike Basic and other line-oriented languages). The only exception is at the end of a comment in `*`, which, as in C++ and Java, starts with `//` and continues to the end of its line.

In Pascal, semicolons are separators; in C and Ada, semicolons are terminators. In \*, by the above definition of a compound statement, semicolons are terminators, but there is one exception. When ; is followed by } (possibly with intervening whitespace), the ; may be omitted. This works because a statement followed immediately by } has no other meaning than if it were followed by ; } and it is taken to be the same as if it were so followed. Several examples of this are given in §6.2. Note that the Pascal construction `if x>y then z:=x else z:=y` (with no semicolon after `z:=x`) has no counterpart in \*; the semicolon must be present in such a case.

### 3.1 Assignment

There are three kinds of assignment in \*. There is ordinary assignment, `v=e`; as in C. There is clone assignment, `p2:=p1`; which is *only* for pointers (see §4.10) and files (see §5.1). There are then the alternate assignments such as `+=` and `*=` (see Table 3), which are the same in \* as they are in C. They are variations on = (not on :=). As in C, all assignments in \* are operators, and may be used in expressions, producing side effects.

### 3.2 The *if* and *while* Statements

The word `if` becomes ? in \*. This is followed by a condition in parentheses, as in C, and then a statement, optionally followed by : (for *else*) and another statement. (Every statement in \* ends in either a semicolon or a right curly bracket, and a colon following either of these can only mean *else*.) Note that ? *precedes* the condition, rather than following it like the ternary ? operator. The word `while` becomes ?? in \* (a double ? suggesting a *repeated* test). As with ?, the ?? is followed by a condition in parentheses, and then a statement, optionally followed by : (for *else*) and another statement. Here `else` after `while` is as in Python; that is, any *break* (see §3.4) skips over the *else*-part. The usual attempts to use = instead of == in either an *if* statement or a *while* statement (such as ? (a=b) for ? (a==b) or ?? (a=b) for ?? (a==b)) produce warning messages in \*, if this is specified by the safety level (see §5.5). Like the single ?, the double ? precedes the condition.

### 3.3 Labels and the *goto* Statement

There are four kinds of labels in \*:

- An ordinary label (the destination of a *goto*) is an *integer* followed by a colon. (An identifier followed by a colon in \*, at parenthesis level zero, indicates stepping iteration; see §3.7). A \* utility program converts all ordinary labels, and all references to them, so that they become sequential integers starting from the beginning of the program. This makes these labels much easier to find during debugging.
- An anonymous label (see §3.5) immediately follows {.
- A case label in a switch (see §3.6) is of the form (e:), where e is some literal. The parentheses here distinguish case labels from ordinary labels.
- A default label in a switch is of the form (:).

There are two kinds of `goto` in \*, namely `>n`; (forward `goto` label *n*) and `<n`; (backward `goto` label *n*). Each of these is either disallowed, allowed, or allowed with a warning, depending on the safety level (see §5.5). Indeed, this is why there are two kinds of `goto`; it is common to allow only forward `gotos`. Uses for the `goto` in \* include writing very fast code which may be difficult to understand, although this will not matter if the code has been proved correct (see, for example, the 20-statement fast sort in §6.2); a switch in \* for which the C equivalent involves a case that is not ended by a *break* statement (see §3.6); and continue logic in a loop which contains an incrementation statement (see §3.8).

### 3.4 The *break* and *continue* Statements

In \* the word `break` becomes either `>,;` (for an ordinary *break*) or `>,n`; where *n* is a label (for a labeled *break* as in Java). The ordinary *break* statement breaks out of the innermost loop that contains it. (Unlike C and Java, a *break* in \* does not make exit from a switch; see §3.6). Here `>` suggests “go forward” (a stylized forward arrow) while `,` suggests “and keep going” (more to come).

The word `continue` becomes either `<,;` (for an ordinary *continue*) or `<,n`; where *n* is a label (for a labeled *continue* as in Java). The ordinary *continue* statement has the effect, as in other languages, of starting the next iteration of the innermost loop that contains it (but also see §3.8). Here `<` suggests “go back” (a stylized back arrow) while `,` suggests “and keep going” (more to come).

### 3.5 Endless Loops, *repeat-until*, and *do-while*

An endless loop (like `for (;;) {body}` or `while (true) {body}` or `do {body} while (true);` in C, or `Loop body End Loop;` in Ada) could be started in \* with `??(1)` meaning “*while (true)*” (`?? = while, 1 = true`). A better way, however, is to use an *anonymous label*. Starting a block with `{ :` rather than simply `{` turns the block into a loop; as in Ada, such a loop is intended to contain a `>,)` statement. Note that the innermost loop, for either *break* or *continue*, can be a loop consisting of a block with an anonymous label.

In particular, since \* has endless loops, it has no separate *repeat-until* statement, since `repeat L until cond` is equivalent to `{ : L; ?(cond) >, ; }`. Similarly, \* has no separate *do-while* statement, since `do L while cond` is equivalent to `repeat L until not cond`, or `{ : L; ?(! (cond)) >, ; }`. However, an endless loop whose last statement is of the form `?(cond) >, ;` will be statically identified as a *do-while* statement, and any *continue* (`<,)` in such a loop will properly go to this last statement. (A *continue* in an endless loop not of this form goes to the start of the loop.)

An endless loop may be followed by `:` (meaning *else*) and an *else-part*, just like a *while* loop (see §3.2). As with *while* loops, a *break* within an endless loop skips over the *else-part*, if there is one.

### 3.6 Switches and Cases

In \*, the word `switch` from C or Java becomes `|` (suggesting an *or* operation, since switches involve *alternatives*). This is followed by the `switch` expression in parentheses, and then a compound statement containing case labels and, optionally, a default label (see §3.3). It can be made mandatory, in the safety level (see §5.5), to cover every possible case (as in Ada). Note that:

- Switches are more efficient if the case labels are all constants in a small range, but they may be more general constants, variables, or expressions, with no guarantee as to efficiency.
- The use of `break` in C or Java at the end of a case has no parallel in \*; instead, to move from the end of one case to the beginning of another one, use a *goto* (much as in C#).
- A case does not need to be enclosed in its own set of curly brackets, since its statements extend up until the next case, or the default, or the end of the switch (much as in Ada).

### 3.7 Stepping Iteration

There are three kinds of stepping iteration in \*. First of all, what is `for (init; test; incr) body` in C is `( : init; test; incr) body` in \*, where `body` is a (simple or compound) statement. There is no ambiguity here because `init` cannot start with a right parenthesis (note that `( :)` is the default label in \*, see §3.3). As in Java, although `init` and `incr` may contain commas, there is no comma operator in \*. The use of `( :` here is like the use of `{ :` to start an endless loop in \* (see §3.5).

There is also Fortran-style stepping iteration in \*, written `i:e1, e2, body` meaning that `body` is repeated from `i = e1` to `e2`. Variants of this are `i:e1, e2, e3, body` (where `e3` is the step size) and `i:e2, e1, -1, body` (as a special case of this) for a loop *in reverse* in Ada, or a *downto* loop in Pascal. Note that:

- Unless `e3` is constant, `i:e1, e2, e3, body` is inefficient; whenever this is a problem, use a *while-loop* instead.
- Each of the expressions `e1`, `e2`, and `e3` is evaluated just once, at the start of the loop, and that value is then used throughout the loop.
- Here `e1, e2` and `e1, e2, e3` are unambiguous because there is no comma operator in \*, as noted above.
- The use of `i:e1, e2, e3, body` is subject to the usual Fortran caveat about the case `e2`, which will not be done unless `e3` evenly divides `e1-e2`.
- If `e2` is of the form `e-1`, the loop `i:e1, e2, body` is implemented by testing for less than `e` (rather than less than or equal to `e-1`), thus saving time.

There is a *for-each* loop in \* for an associative array  $\alpha$  (see §4.1). This is of the form `i< $\alpha$ : body`, although `i $\in$  $\alpha$ : body` may be used instead. Here  $\alpha$  is to be implemented as a balanced binary search tree, and `i< $\alpha$ :` or `i $\in$  $\alpha$ :` traverses the tree in in-order. It is not allowed to jump into a *for-each* loop or a Fortran-style loop, although it is allowed to jump into other kinds of loops (as in C).

Any kind of stepping iteration may be followed by `:` (for *else*), just as with *while* loops and endless loops (see §3.2). As in a *while* loop, a *break* inside stepping iteration skips over the *else-part*, if there is one. This is another substitute for *goto*, commonly used in linear searching. If the search terminates without a match, the *else-part* is

done (which might be an insertion, or an error indication). This should not be done, and is not done, if a match is found, since then there is a *break*, which skips over the *else*-part.

Like most languages, *\** allows for more than one variable with the same name. If the definition of a variable is in a condition in an *if*-, *while*-, or *for*-statement, that definition holds throughout that statement. Any other definition of a variable holds throughout the innermost block in which it is contained, or the whole program if it is not contained in any block. All this is as in C++.

### 3.8 Incrementation and *continue* Statements

Inside stepping iteration (of any kind; see §3.7), *<*, goes properly to the incrementation. This is also true of a do-while loop in *\**; but inside *while* loops or endless loops, *<*, skips over any incrementation statements that might be there (see §3.5). In such a case, a *goto* which goes to the incrementation should be used in order to start the next iteration properly. All of this is like *continue* in C; for example, when processing all non-null pointers in an array *t* of pointers from *t*[1] through *t*[*n*], if we write, in C (at the left) or in *\** (at the right),

```

for (i = 1; i <= n; i++) {
    if (t[i] == 0) continue;
    . . .
    (process t[i])
    . . .
}

```

```

i:1,n,{? (t[i] == 0) <,;
. . .
(process t[i])
. . .
}

```

the *continue* goes to the *i++*, and therefore properly starts the next iteration; but if we were to write

```

i = 1; while (i <= n) {
    if (t[i] == 0) continue;
    . . .
    (process t[i])
    . . .
    i++;
}

```

```

i = 1; ?? (i <= n) {
    ? (t[i] == 0) <,;
    . . .
    (process t[i])
    . . .
    i++;
}

```

it would not. It would skip over the *i++*, and, instead of starting the next iteration, it would restart the *current* iteration (much like *redo* in Perl), and the loop would be endless. Instead of that, we should use a *goto* (in this particular case, a forward *goto*), like this:

```

i = 1; while (i <= n) {
    if (t[i] == 0) goto incr;
    . . .
    (process t[i])
    . . .
    incr: i++;
}

```

```

i = 1; ?? (i <= n) {
    ? (t[i] == 0) >9;
    . . .
    (process t[i])
    . . .
    9: i++;
}

```

This is one of the few uses of *goto* that is not equivalent to one of the substitutes for *goto* available in both Java and *\** (see §3.3).

### 3.9 Function Definition and Call

Functions in *\** are much like functions in C. Their declarations and definitions start with either [] (for *void*) or the type of the result. This is then followed by a declarator, which, in the simplest case, is a function name (but also see §4.7). This is followed, in parentheses, by parameters separated by commas, each of which is given by

its type in \* followed by its name. This is all followed by a semicolon, for a declaration, or a compound statement representing the body of the function, for a definition.

A function in \* is called, as in C, by giving its name followed by arguments (sometimes called actual parameters) in parentheses, or just the parentheses if there are no arguments, and a semicolon at the end. If an argument is a single unsubscripted variable, then it is passed by value and result; otherwise, it is passed by value. Pass by reference may be simulated either by using pointers, as usual in C, or by using reference types as in C++.

The definition of an inline function is preceded by --- (suggesting a *line*). There is no ambiguity here, since the lexical analyzer reads this as -- - (by the longest-token rule) and the decrement operator -- cannot be legally applied to any expression starting with a minus sign.

### 3.10 The *return* and *exit* Statements

There are two kinds of *return* statement in \*, as in many languages. Write  $>.e;$  to return the expression  $e$ , and write  $&gt.$  to return without returning a value. The  $>$  suggests “go forward” (a stylized forward arrow) and the  $.$  suggests “period, end” — in other words, “go forward to the end of this program.”

An *exit* statement terminates, not only the function in which it occurs, but also the entire program containing that function. In some languages, this is a system function call, but in \* it is  $<.;$  (to simply exit) or  $<.e;$  (to exit and return the value of  $e$ ). Conventionally the returned value is either 0 (for a normal exit) or a small positive integer (for an error exit such as 404 Not Found). Here  $<$  suggests “go back” (to the operating system; a stylized back arrow) and  $.$  suggests “period, end” (end this program).

An easy way to remember the symbols in \* for *break*, *continue*, *return*, *exit*, and forward and backward *goto* is by reference to the following table:

<i>break</i>	Go <i>forward</i> ( $>$ ) and <i>more to come</i> ( $,$ )	$> , ;$
<i>continue</i>	Go <i>back</i> ( $<$ ) and <i>more to come</i> ( $,$ )	$< , ;$
<i>return</i>	Go <i>forward</i> ( $>$ ) and <i>stop this program</i> ( $.$ )	$> . ;$
<i>exit</i>	Go <i>back*</i> ( $<$ ) and <i>stop this program</i> ( $.$ )	$< . ;$
Forward <i>goto</i>	Go <i>forward</i> ( $>$ ) to label $n$	$> n ;$
Backward <i>goto</i>	Go <i>backward</i> ( $<$ ) to label $n$	$< n ;$

\* to the operating system.

## 4. Data Types in \*

Up to now, all the data we have been considering has been of basic types. These include integer and floating point types (see §2.1); Boolean types (see §2.2); character, string, and pattern types, integer subtypes, and dynamic types (see §2.3). In all of these cases, the type description is contained within square brackets. We now take up more general data types in \*. First we consider simple arrays, pointers, references, records, classes, and sets. For more advanced data types, there are declarators, based on those of C, which we first review.

### 4.1 Arrays

In \*, there are several kinds of arrays of elements of some type  $\chi$ :

- An ordinary C array  $\tau$ , extending from  $\tau[0]$  through  $\tau[n-1]$ , for some integer  $n$ , is declared by  $\chi \tau[n]$  as in C. An ordinary Ada or Pascal array  $\tau$ , extending from  $\tau[m]$  through  $\tau[n]$ , is declared by  $\chi \tau[m:n]$ . Array literals, which may appear in initializers, are ordinary literals separated by commas and enclosed in *square* brackets (not curly brackets as in C, since in \* these are for set literals). There is no conflict here with basic type names, since none of these contain commas. (An array literal in \* must have at least two elements.)

- A resizable array  $\tau$  of initial size  $n$  is declared by  $\chi \tau[=n]$ . Such an array is always expanded whenever range checking indicates an out-of-range subscript.

- Packed arrays can be either resizable or not. A packed array is distinguished by being an array of elements of a packed type (see §2.2).

- An associative array  $\tau$  of elements of type  $\phi$  is declared by  $\chi \tau[\phi]$ . Such an array is to be implemented as a balanced binary search tree, so that one can iterate over its elements, as with a *for-each* loop (see §3.7), which would be too inefficient with a hash table implementation.

#### 4.2 Pointers and References

Pointer types are declared in  $*$  much as in C; thus `[32] *pint` declares `pint` to be a pointer to a 32-bit integer. Note that:

- The pointer operators  $*$  and  $\&$  are as in C.
- There are two kinds of pointer literals. One is of the form  $\&k$  where  $k$  is a literal of some type  $t$ ; here  $\&k$  is a pointer literal of type  $*t$ . The other is `0`, which, as in C (and like `null` in Java), is the null pointer literal of any pointer type.
- References in  $*$  are as in C++; what is `int &k`; in C++ is `[32] &k`; in  $*$ .
- The safety level (see §5.5) determines whether range checking is done; whether null pointer checking is done; and whether pointer arithmetic is allowed (in which case it is as in C).
- What is `void *p`; in C (meaning that  $p$  is a pointer to any type) becomes `[] *p`; in  $*$  (using `[]` for `void` as with functions; see §3.9). Do not confuse this with `[0] *p`; meaning that  $p$  is a pointer to something of a dynamic type, having an internal type code (see §2.3).
- There is clone assignment for pointers. If  $p_1$  and  $p_2$  are pointers in  $*$ , `p2:=p1`; sets  $p_2$  to point to a (shallow) copy of  $*p_1$  (see §4.10).

#### 4.3 Multidimensional Arrays

A multidimensional array in  $*$  is an array of pointers to arrays, as in Java. (In C, it is a bare array of arrays, saving small amounts of space while drastically reducing flexibility; for example, it is unusually difficult to write a C program that multiplies two square matrices of arbitrary given size.) The structure of such a bare array of arrays gave rise to the C notation  $\tau[i][j]$  for elements of a double array, and similarly for more general multidimensional arrays. In  $*$  there is no need for this, and the Pascal-style notation  $\tau[i, j]$  is used.

To declare an  $m$ -by- $n$  array of elements of type  $\chi$ , we write  $\chi \tau[m, n]$ ; . Multidimensional resizable arrays are declared by extending this notation to resizable arrays. Thus an array  $\chi \tau[=16, =8]$  is a resizable array, with initial size 16, of pointers to resizable arrays, each of initial size 8, of elements of type  $\chi$ .

#### 4.4 Records and Classes

The  $*$  language is partially object-oriented, like C++, as opposed to being totally object-oriented, like Java or Ruby. There is consequently no need in  $*$  for anything like the static methods of Java. A  $*$  class is given a name by the naming operator `=` (see §2.6; this may include derived class specification). A class definition in  $*$  is enclosed in curly brackets, as in C++ or Java, but is followed by `;` (unlike C++ or Java). It may contain:

- Variable (or member) definitions, which are like other variable definitions in  $*$ . If this is all that a class definition contains, objects of that class are referred to as *records*.
- For an abstract class as in Java, the designation `{0;` instead of simply `{` at the beginning (the `0` here suggests “no objects”).
- Public, private, and protected sections like those of C++ (and unlike Java). Instead of `public:`, `private:`, and `protected:`, we write, respectively, `+`, `-`, and `^`:. (The `+` and `-` designations for `public` and `private`, respectively, are also those used in UML.)
- Static variables, each of which is declared by a period `.` before its type. (Static = stopped; a period *stops* a sentence.) Note that a period before a *type* denotes *static*, while a period before a *variable name* denotes *const*.
- Constructors, like those of C++, except that the identifier `_` (by itself) is the name of the base class constructor inside the definition of a derived class constructor, much like `super` in Java. (Here the underscore `_` suggests *base*.)
- Member function (or method) declarations, which are based on those of C++. In this context, `this` in C++ or Java becomes `..` in  $*$ .
- Overloaded operator definitions. The word *operator* in C++ becomes `^` in  $*$ ; and this is immediately followed, as is *operator* in C++, by the operator symbol to be overloaded. Thus `^+`, `^-`, and `^^` are, respectively, the names of overloaded `+`, `-`, and `^` operators. The rest of the rules for overloaded operators are

like those of C++. There is no confusion between  $\hat{=}$  as a binary alternate assignment and as an overloaded  $=$  operator.

A `*` class may also be accompanied by friend declarations. The word `friend` in C++ becomes `,`, in `*` (the shape of the commas suggests two friends holding hands). The rest of the rules for friends in `*` resemble those of C++. There is no confusion between this use of commas and their use in separating function arguments or parameters (which must be inside parentheses).

#### 4.5 Sets

Sets in `*` are either *restricted* or *general*. If we declare `#32  $\omega$` , then  $\omega$  is a restricted set; it is a 32-bit integer considered as a set of elements in the range from 0 through 31. (The notation `#` suggests the *number* of bits in the representation of the set.) Here union, intersection, and complement are carried out, respectively, by *bit-or*, *bit-and*, and *bit-not*; set difference is *bit-not* followed by *bit-and*. This is generalized in two ways in `*`. We can write, for example, `#8  $\omega$` , using 8-bit instead of 32-bit integers. We can also write, for example, `#256  $\omega$` , and this time  $\omega$  is an internal array of eight 32-bit integers. Operations on sets are now carried out by looping over these arrays. This generalizes an implementation of Pascal in which such internal arrays have maximum size 64.

For general sets, we declare `# $\tau$   $\omega$` , for any type name  $\tau$ . This is to be implemented as a sorted resizable array of elements of the type of  $\tau$ ; this array is either *direct* (specifying all elements in the set) or *complementary* (specifying all elements not in the set). Such an array, having length  $n$ , is preceded internally by either  $n$  (for a direct set) or  $-n-1$  (for a complementary set). In particular, this is 0 for the null set and  $-1$  for the universal set (the complementary set of the null set). For both restricted and general sets, a set literal consists of the elements of the set, separated by commas and enclosed in curly brackets, and optionally preceded by  $\sim$  (for set complement). Thus `{1, 3, 6}` is the set of elements 1, 3, and 6, while `~{1, 3, 6}` is the set of all elements of the universal set except 1, 3, and 6.

Operators on sets (of either kind) are  $+$  or  $\cup$  (union),  $*$  or  $\cap$  (intersection), binary  $\sim$  or  $-$  (set difference), and unary  $\sim$  (set complement). For the subset and superset conditions, we may write  $a \subseteq b$  or  $a \supseteq b$ , but we may also write  $a <= b$  or  $a >= b$ . For set membership conditions, we may write  $k \in a$  or  $k \notin a$ , but we may also write  $k < a$  or  $k /< a$ . Here  $\cup$ ,  $\cap$ ,  $\subseteq$ ,  $\supseteq$ ,  $\in$ , and  $\notin$  are Unicode characters.

#### 4.6 Review of Declarators in C

In some programming languages, such as Ada, pointers are declared directly. If we want  $k_p$  to be a pointer to an integer, we write `kp: pint;` where we have declared `type pint is access integer;` In C, on the other hand, pointers are declared indirectly. If  $k_p$  is a pointer to an integer, then `*kp`, in C notation, is an integer; and this is what is declared. In other words, instead of declaring  $k_p$ , we declare `*kp`, by writing `int *kp;`. One advantage of this is that it may be immediately extended to further levels. Suppose, for example, that we want  $k_{pp}$  to be a pointer to a pointer to an integer. Now `*kpp` is a pointer to an integer, and `**kpp` is an integer; and, again, this is what is declared. We write `int **kpp;` and this, by declaring `**kpp` to be an integer, indirectly declares  $k_{pp}$  to be a pointer to a pointer to an integer.

Constructions such as `*kp` and `**kpp` are known in C as *declarators*. A declaration in C consists of a type followed by one or more declarators (each one optionally followed by an initializer) separated by commas and followed by a semicolon. We may write `int k=10,*kp,**kpp;` for example. The idea of declarators is further extended in C to functions, which are declared directly in other languages. In Ada, if we want  $f$  to be a function of a real argument which returns an integer value, we declare  $f$  directly; we write `function f(x: float) return integer;` In C we declare  $f$  indirectly, by declaring what  $f$  returns. In this case, `f(double x)` (or sometimes just `f(double)`) is an integer, so we write `int f(double x);` or simply `int f(double);`. As before, `f(double x)` and `f(double)` are known as declarators.

Also as before, these can be extended to further levels. If we want  $f_2$  to return a pointer to an integer, rather than just an integer, then `*f2(double x)` is an integer, and this is what is declared, by writing `int *f2(double x)`. All of these can be mixed in a single declaration, such as `int k,*kp,**kpp,f(double x),*f2(double x);`. The idea of indirect declarations also applies to arrays, in an extended sense. If we define `int *t[100];` then it is not `t[100]`, but rather `t[k]` for any  $k$  with  $0 \leq k < 100$ , that is a pointer to an integer, so that `*t[k]` is an integer. Thus `t`, here, is an array of pointers to integers.

Parentheses for grouping can be used in declarators in C. If instead of `int *t[100];` we write `int (*t)[100];` then `(*t)[k]`, for  $0 \leq k < 100$ , is now an integer. Suppose we denote `(*t)` by  $u$ ; then `u[k]`, for  $0 \leq k < 100$ , is an integer, so that  $u$  is an array of 100 integers. Since  $u$  is `(*t)`, we conclude that `t` is a pointer to such an array. In other words, by using declarators, we can now declare `t` to be either an *array of pointers* to integers (with `int`

`*t[100];`) or a *pointer* to an *array* of integers (with `int (*t)[100];`). If parentheses for grouping are not present, the `*` operator has low precedence (see the end of §4.7), so that `int *t[100];` means `int *(t[100]);` and not `int (*t)[100];`.

#### 4.7 Declarators in \*

Declarators in `*` are more general than those of C, because they can use patterns, sets, subtypes, dynamic types, packed arrays, resizable arrays, associative arrays, and general array bounds, none of which are primitives in C. The construction of complex declarators, declarations, and type specifiers in `*` is much like that of C. Thus, in `*`:

- A *type specifier* is either a basic type (see section 4 above) or a type name, set up through the type definition facility (where `&`, in `*` is like `typedef` in C; see §4.8). Thus we might write `&, [32] *pint;` and this sets up `pint` to be the type “pointer to integer.”

- A *declaration* consists of a type specifier followed by a declarator (see below), or by several declarators separated by commas, and all of this followed by a semicolon. Optionally, each declarator here may be followed by an initializer, as in C. Thus we might write `pint p1, .p2, *p3=0;` where `pint` is as above, and this sets up `p1` to be a pointer to an integer; `p2` to be a constant pointer to an integer; and `p3` to be a pointer to a pointer to an integer, with initial value null. (How these determinations are made is taken up in §4.9.)

- A *declarator* is either a variable name (see §2.5) or a *general declarator*, defined recursively as follows. Let  $d$  be a declarator, of either kind; then from  $d$  we derive a new declarator  $d'$ , which may be any of the following:

$*d$	$.d$	$\&d$	$d[=n]$	$d[n]$
$d[m:n]$	$d[r]$	$d()$	$d(u_1, \dots, u_k)$	$(d)$

Here  $m$  and  $n$  are integers;  $r$  is a type; and each  $u_i$ , for  $1 \leq i \leq k$ , is a type, optionally followed by a variable name.

As it stands, this rule is ambiguous; `*t[100]`, for example, could be derived either from `*t` or from `t[100]`. The postfix operators have higher precedence than the prefix operators here, so only the derivation from `t[100]` is legal. (As an alternative, we could formalize precedence by introducing terms and factors for declarators, much as is done for arithmetic expressions in BNF descriptions of programming language syntax.)

#### 4.8 Type Definitions

Declarators are used in `*`, not only in declaring variables, but also in giving names to types. This is done much as in C, where, if we want to set up `alpha` as the name of some type, we first give a definition of `alpha` as a *variable* of that type, and then precede that definition by `typedef`. In `*`, we do the same thing, except that we write `&`, in place of `typedef` (the `&` here suggesting “all existing type names *and* this new one”). This syntax is different from that of the naming operator (see §2.6); thus if we write `typedef type1 *type2` in C, it is `type2`, not `type1`, that becomes a new type name. This is why type definitions in `*` do not use the naming operator.

What is `sizeof(e)` (or `sizeof e`) in C becomes `#(e)` in `*` (`#` suggesting a *number* of bytes); here  $e$  is either an expression or a type. Note carefully that this is a number of *bytes*, not bits. If the size in bits is  $b$ , then the size in bytes is  $1+(b-1)/8$ . This reflects the fact that a variable, outside of a packed array or a packed record, is always kept in an integral number of bytes.

#### 4.9 Declarator and Type Specifier Semantics

A type specifier, as above, always has an associated type. For basic types, the associated type is as follows, where  $m$  and  $n$  are integers:

<code>[n]</code>	Signed integer $k$ with $-2^{n-1} \leq k < 2^{n-1}$ , for $n > 0$
<code>[`n]</code>	IEEE standard floating point number $x$ with $n$ bits total
<code>[+n]</code>	Unsigned integer $k$ with $0 \leq k < 2^n$
<code>[/n]</code>	Same as <code>[+n]</code> except that an array of these is a packed array
<code>[']</code>	Unicode character
<code>["]</code>	String of Unicode characters
<code>[/]</code>	Pattern

$[m..n]$	Element $k$ of integer subtype, with $m \leq k \leq n$
$[0]$	Element of dynamic type (implemented with internal type code)
$\#n$	Restricted set (of elements $k$ with $0 \leq k < n$ )
$\#\tau$	General set (of elements of some type $\tau$ )

For user-defined types, the associated type is the type of the given declarator, as explained below. (Sets, both restricted and general, count as basic types for theoretical reasons set forth below, although they are not basic in an informal sense.)

Within a particular declaration, a declarator also has an associated type. For example, in the declaration `[32] v1, *v2, (*v3) [100];` the type of `v1` is integer, the type of `v2` is “pointer to integer,” and the type of `v3` is “pointer to array of 100 elements of type integer.” (It is possible to abbreviate this last to “pointer to array of 100 integers,” but formalizing that involves a tedious complication. All types would now have to appear in two forms: singular, such as “integer,” and plural, such as “integers”; and each of these would have to be separately defined recursively.)

Every declarator has a *derivation sequence*. Every element of such a sequence, within a particular declaration, has an associated type, which is determined from that of the element derived from this one as in §4.7. Let  $d'$  be derived from  $d$  in a derivation sequence, and let  $d'$  have associated type  $t'$ . Then we determine the associated type  $t$  of  $d$  according to the following rules:

*Rule#*    *If  $d'$  is:*    *Then  $d$  has type  $t$ , where  $t$  is:*

1	$*d$	Pointer to $t'$
2	$.d$	Constant $t'$
3	$\&d$	Reference to $t'$
4	$d[=n]$	Array of elements of type $t'$ (resizable, with initial size $n$ )
5	$d[n]$	Array of $n$ elements of type $t'$
6	$d[m:n]$	Array of elements of type $t'$ , having indices $m$ through $n$
7	$d[r]$	Associative array (indices of type $r$ , elements of type $t'$ )
8	$d()$	Function of no arguments, returning a value of type $t'$
9	$d(u_1, \dots, u_k)$	Function of $k$ arguments, returning a value of type $t'$
10	$(d)$	$t'$

For example, in the declaration `[32] (*v3) [100];` we see that `(*v3) [100]` is derived from `(*v3)`, which is derived from `*v3`, which is derived from `v3`. To obtain the associated type of `v3`, we deduce that:

- `(*v3) [100]` has type integer (since `[32]` has associated type integer);
- `(*v3)` has type array of 100 elements of type integer (by rule 5 above);
- `*v3` has type array of 100 elements of type integer (by rule 10 above);
- `v3` has type pointer to array of 100 elements of type integer (by rule 1 above).

The associated type is what is called an inherited attribute, a concept introduced by Knuth (Knuth, 1968). Here  $d'$ , having type  $t'$ , is derived from  $d$ , having type  $t$ . For a synthesized attribute,  $t'$  would be calculated from  $t$ ; for an inherited attribute, such as this one,  $t$  is calculated from  $t'$ . An example of a synthesized attribute is the *base identifier* of a declarator. For example, the base identifier of `&, ["] *pstr` is `pstr`, and that of `pstr htbl[pstr]` is `htbl`. If a declarator is just an identifier, that is its base identifier; if a declarator  $d'$  is constructed from  $d$  as in the preceding section, then the base identifier of  $d'$  is defined to be the same as that of  $d$ . Using the associated type as an inherited attribute, we may determine the type  $t$  of the base identifier  $id$  of the declarator  $d$ . When we have done this:

- If  $d$  appears in a type definition, then  $id$  becomes a type specifier, which is a name for the type  $t$ .
- If  $d$  appears in a declaration, then  $id$  becomes the name of a variable whose type is  $t$ .
- By removing  $id$  from  $d$  and enclosing the result in parentheses, one obtains a *type conversion operator* (see §2.4). Giving this in an expression, followed by a subexpression  $e$ , gives the result of converting  $e$  to type  $t$ .

Duplication of constant specifications is legal but has no effect. Thus `[32] . . k;` is the same as `[32] . k;` and both mean that `k` is a constant integer.

If `d` is a declarator, and `i` is a literal, then `=i` is an initializer, and an occurrence of `d=i` in a declaration initializes the base identifier `id` of `d`. For this to be legal, the type of `i` must be the type of `id` as defined above.

#### 4.10 Clone Assignment

If `p1` and `p2` point to objects, what happens if we set `p2` equal to `p1`? Does this mean that `p1` and `p2` now point to the same object? If so, these objects are now aliases, and changing part of one of these will (perhaps inappropriately) also change that part of the other one. Or does it mean that `p2` now points to a copy (sometimes called a clone) of what `p1` points to? Now the above problem is avoided, but this is slow, particularly for large objects.

In `*`, both techniques are available; there is `=` for ordinary assignment, and there is `:=` for clone assignment. Both of these act on *pointers* to objects, which are explicit in `*`, as in C++ (and unlike Java). Let `p1` and `p2` be pointers in `*`, and let `p1` point to `z`. If we set `p2=p1`; then `p2` now points to `z`, so that `p1` and `p2` become aliases; if part of `*p2` is now changed, then so is that part of `*p1`. If we set `p2:=p1`; then `p2` now points to a copy of `z`, avoiding the above problem. This is a shallow copy, meaning that pointers inside `z` are not subject to cloning themselves. Here `z` is usually an object, although it may be of any type.

Setting `p2=p1`; as above looks dangerous, although it is quite safe if `*p2` is set up as a constant, which cannot be changed. We can do this by declaring `t . *p2`; where `t` is the type of `z`, as above. Here `. *p2` is of type `t`, so that `*p2` is a constant of type `t`, so that `p2` is a *pointer to constant* (of type `t`); even though `p2` is not itself constant, `*p2` is. Another use for `p2=p1`; arises when `p1` points to an object in a database that is subject to continual updating. Now it is `p2:=p1`; that is dangerous, in a different way. If `*p1` is updated, then `p2` now points to an outdated version of `*p1`, which is normally not what is intended.

Clone assignment is also available for files in `*` (see §5.1).

### 5. Miscellaneous Features of `*`

Here we take up input and output, dynamic storage, pattern matching, exceptions, include files, and volatile variables in the C++ sense. We also give a more explicit description of the treatment of safety in `*` (see §1.3).

#### 5.1 Input and Output

The shifting operators `<<` and `>>` (see §2.1) are overloaded, as in C++, to produce output and input, respectively. Standard input, standard output, and standard error file objects are `>>0`, `<<0`, and `<<!` respectively. Other input file objects are `>>1`, `>>2`, and so on; other output file objects are `<<1`, `<<2`, and so on. The end of an output line is denoted by `\n` in `*` (as in C and Java). Note that the various escape characters (`\n`, `\t`, and so on) are natural-language independent, even though, for example, the `n` in `\n` originally stood for “new line” in English. (This is like the set  $\mathbf{Z}$  of all integers, so named because, in German,  $\mathbf{Z}$  stands for *Zahl*, which means “number.”) Operations on files are notated as follows:

Open file <code>n</code> for reading	<code>&gt;&gt;n+ (some_file_name)</code>
Open file <code>n</code> for writing	<code>&lt;&lt;n+ (some_file_name)</code>
Close file <code>n</code> for reading	<code>&gt;&gt;n-</code>
Close file <code>n</code> for writing	<code>&lt;&lt;n-</code>
Copy file <code>n</code> to file <code>m</code>	<code>&gt;&gt;m:=&gt;&gt;n;</code> (or the like)
Compare files <code>m</code> and <code>n</code>	<code>? (&gt;&gt;m!=&gt;&gt;n) (error action)</code>
Redesignation	<code>&gt;&gt;n=&lt;&lt;m;</code>

Here *redesignation* refers to redesignating an output file as an input file, so that it can be read. If we have written to `<<1` and closed it, then by writing `>>1=<<1`; we can redesignate `<<1` as the input file `>>1` and read it. When we do this, we are reading the same characters that we just wrote to `<<1`. This is different from writing `>>1:=<<1`; since this would copy the file `<<1`, reading in all its data and writing it out. By contrast, `>>1=<<1`; takes almost no additional time, since it is just a redesignation, not a file copy. Note that file copy and file compare do *not* automatically open and close the given files; these must be done by the program.

The `#include` feature of C is given by `+`, in `*` (the `+` suggesting “*additional* statements to be included here”), followed by the name of the include-file. This may be used in `*` wherever `#include` would be used in C, but it has a special usage with safety statements (see §5.5).

### 5.2 Dynamic Storage, *new*, and *delete*

The keyword `new` in most programming languages, or the `malloc` function in C, becomes `->` in `*`. Thus `p = new Alpha;` becomes `p = ->Alpha;` suggesting that `p` now points to an object of class `Alpha`. The keyword `delete` in C++, or the `free` function in C, becomes `<-` in `*` (suggesting the opposite of `->`); thus `delete p;` becomes `<-p;`. Array deletion, what is `delete[]` in C++, becomes `<-[]` in `*`. Garbage collection may be specified in the safety level (see §5.5), in which case `<-` and `<-[]` are not used and not allowed.

### 5.3 Regular Expressions and Pattern Matching

*Patterns* in `*` (see §2.3) generalize regular expressions in the Perl sense. They may contain individual Unicode characters as well as character classes from any of various Unicode alphabets; for example, `α-ω` means “any lower-case Greek character.” There are visually ambiguous character classes such as `A-Z` which can mean either “any upper-case Roman character” or “any Greek character from upper-case `α` through upper-case `ζ`.” These are not really ambiguous, however, since the Greek `A` and `Z` (with character codes 0391 and 0396) are different from the Latin `A` and `Z` (with codes 0041 and 005A).

The character `€` (or any other currency symbol) may be used in place of `$` in any pattern. Pattern matching which ignores case is generalized to pattern matching which ignores diacritical marks. If `s` and `r` are strings, `p` is a pattern, and `t` is a resizable array of strings, then:

- `s == p` is a condition which is 1 (*true*) if `s` matches `p`, and 0 (*false*) otherwise; in particular, we may write `if (s == p)` with the usual meaning. The negation of `(s == p)` is `(s != p)` (or `(s <> p)`).
- `p < s` (or `s > p`) is `-1` if `s` does not match `p`; otherwise, it is the index of the start of the (first, if there are more than one) substring of `s` that matches `p`.
- `-p` is a pattern that behaves like `p` except that case is ignored in matching; for example, `Fortran` matches `FORTTRAN`.
- ``p` is a pattern that behaves like `p` except that diacritical marks are ignored in matching; for example, `résumé` matches `resume`.
- `+p` is a pattern that behaves like `p` except for multiline matching. Any or all of these unary operators `-`, ```, and `+` may be applied to the same pattern `p`.
- `s = r + p` sets `s` to the (first, if there are more than one) substring of `r` that matches `p`. If there is no match, `s` is set to 0 (the null string).
- `t = r + p` sets `t` to the array of all substrings of `r` which match `p`. If there are none, then `t` is set to 0.
- `s = p + r` sets `s` to a string which is the result of replacing the matched portion of `s` by `r` within `s`. If `s` does not match `p`, then it remains unchanged.
- `s = p * r` is like `s = p + r`, except that `s = p * r` does a global replacement.
- `s = t * p` sets `s` to the string formed by joining together the strings in `t` with the delimiter `p` (which must be a single character) between each two.
- `t = s / p` sets `t` to the array of strings obtained by splitting `s` at the delimiter `p`. Here `p` is often a single character, but it can be a more general pattern.

Unlike other languages, `*` allows pointers to patterns, arrays of patterns, functions which return a pattern, and the like (see §4.9).

### 5.4 Exceptions

In order to help find what caused an error when it happens, we normally need to know the current values of certain variables `vi`. If exceptions are used, the `vi` may be moved to an object `e` of an *exception class* `Z`. In C++, we write `throw Z(v1, v2, ...)`; where `Z` has members `xi` and a constructor `Z(a1, a2, ...)` which sets `x1 = a1, x2 = a2, ...`, effectively setting `x1 = v1, x2 = v2, ...`, in this case. We catch this exception by writing `catch (Z e)` (or `catch (Y e)` where `Z` is derived from `Y`, directly or indirectly). This is followed by a block `B` in which every `vi` that we need is now `e.xi`, and we can look at these, inside `B`, to try to find the cause of the error. If we do not need to look at any `vi` at all, then `Z` is allowed to have no members `xi`, so that its objects are empty. Here `Z` still

has a constructor (with no arguments); this doesn't construct anything, but can simply (for example) print out the fact that an error has occurred. In `*`, exceptions are quite similar to this; however, in `*`:

- The name of an exception class in `*` always ends in `!` (suggesting “Oh, no! That's wrong!”); and it is declared with the naming operator (see §2.6). There is no confusion here with the unary `!` operator. (The character `!` is also at the end of the standard error file name `<<! in *`; see §5.1.)

- *Using an exception class constructor is itself a throw statement.* There is no need in `*` for the symbolic equivalent of a keyword like `throw` or `raise`. A re-throw (like `throw;` in C++, or `throw e;` in Java) is `!;` in `*`, but, aside from this, a throw in `*` *always* throws an exception class constructor. There is no equivalent, in `*`, of `throw 17;` or the like in C++.

- *Specifying an exception class in an if-statement turns that statement into a catch-clause.* There is no need in `*` for the symbolic equivalent of `catch`.

A try-block in `*` starts with `<!` (suggesting “before the catch”). A finally-block in `*` starts with `>!` (suggesting “after the catch”). The general logic of *try*, *catch*, and *finally* resembles that of Java. If some catch-clause applies, it is executed, followed by the finally-block, and control returns to the next function down the dynamic chain. Otherwise, the finally-block is done; the current function is terminated; and an attempt is made to find an applicable catch-clause one level down, unless there are no more levels, in which case a system error message is generated and the program terminates.

Mathematical function errors such as overflow and division by zero are not caught as errors in many programming languages. In `*`, whether they are caught depends on the safety level (see §5.5). This even applies to integer overflow, which is typically not caught by the hardware. Specifying integer overflow checking in the safety level in `*` implies that extra code is compiled for every operation which might produce integer overflow; if it does, then an exception is thrown.

Derived exceptions in `*` are like those in C++ and Java (although, unlike Java, an exception in `*` is not required to be derived). Thus catching an exception will catch any exception derived from it, either directly or indirectly.

### 5.5 Safety Statements

In `*`, maximum safety is the default. If no safety statements are given in a program, then every potentially unsafe feature of `*` is disallowed; in order to allow such features, include a safety statement (as described below). Such a statement is normally the first statement of a `*` program, but it may also be given at the start of a block, in which case the specified safety level is for that block only. This allows the writing of programs which are mainly safe but which contain certain parts that are not.

The presence of safety statements in `*` raises an immediate question as to whether programmers could get around safety restrictions simply by including their own safety statements. The answer is that, again as the default, a safety statement in `*` must be part of a system-level include-file whose name resembles a password. Users are given these names, but no other information; and there are several such files, each corresponding to a safety level which a particular user is authorized to use. Such a file name is to be treated like a password; it is not to be revealed by its user to anyone else. It is possible, in a safety statement in an include-file, to specify that users can give their own safety statements.

A safety statement starts with `?`, (the `?` suggesting “*what* unsafe features are to be allowed?”). This is followed by any number of codes, separated by blanks, associated with specific unsafe features of `*`. Such a feature is indicated as being allowed, unless the code follows `.` in which case the feature is disallowed, or unless the code follows `!` in which case the feature is allowed but with a warning. The codes and their corresponding features are:

- `.` All features are allowed except those which follow.
- `!` Any features which follow this are allowed but with warning messages
- `[]` Array references without range checking
- `<` Backward `gotos`
- `<-` Deletion (as opposed to garbage collection)
- ``!` Floating point arithmetic with unchecked overflow
- `>` Forward `gotos`

- == `if (x=y)` for `if (x==y)` (and similarly with `while`)
- =! Integer arithmetic with unchecked overflow
- >+ Pointer arithmetic
- > Pointers without null pointer checking
- \* Safety statements defined outside of include files
- | Switches which do not cover all possible cases
- ( ) Type coercions in the C++ style (as opposed to the Java style)

As an example, the safety statement

```
?, =! ->+ ! > == -> |
```

specifies that integer overflow is unchecked; that pointer arithmetic is to be allowed; that forward `gotos`, `if (x=y)` for `if (x==y)`, pointers without null pointer checking, and switches which do not cover all possible cases produce warning messages when they are used; that all array references are range checked; that only type coercions in the Java style, not the C style, are to be allowed; that floating point overflow is checked; and that backward `gotos`, safety statements defined outside of include files, and deletion are not allowed (so that garbage collection is used).

### 5.6 Volatile Variables

The name of a volatile variable, in the C++ sense, is preceded, in its definition in `*`, by `!` (suggesting “It might fly away!”). Thus `int x; const int y; volatile int clock;` in C++ becomes [32] `x, .y, !clock;` in `*`. This is very rare, but there is no conflict between this use of `!` in definitions and other uses of `!` in executable statements and initializations.

## 6. Tables and Examples

### 6.1 Uses of Characters

Tables 2 and 3 illustrate the various uses in `*` of the special characters appearing on a standard keyboard. Table 2 is arranged with digits first, and then according to the positions of characters on the keyboard, from top to bottom and, within this, from left to right. Table 3 is arranged alphabetically.

The `*` language has been carefully constructed in such a way that `*` programs have unambiguous syntax; and this should be clear from Table 2. Non-ambiguity follows here from several considerations. Every expression must start with either a letter, a digit (in any natural language), a unary operator (`+ - & * ! ~`), or one of the characters `. ' " ` (`. Therefore, if the first character after a (binary or unary) operator is not one of these, it can have another meaning; and this is common throughout `*`. Also, compilers have long distinguished between binary and unary `+` or `-` or `*` or `&`. In `*` there are also `=` and `!` and `<` as both binary and unary operators; and these are distinguished similarly.

As we have seen, the Unicode characters `∞ · ÷ ≠ ≤ ≥ ∪ ∩ ⊆ ⊇ ∈ ∉` (as well as others, for names) may also be used in `*`. None of these appear in our tables, however, as they are entered in different ways on different keyboards.

Table 2. Special characters on the standard keyboard, and their uses in `*`

0;	abstract class definition	~	set difference	# (t)	sizeof t
0	false	``	exponentiation	#	set declarations
0	null pointer	`	floating point	%=	alternate assignment
0	null string	!#	NaN	%	remainder
1	true	!=	unequal	^=	alternate assignment
1	in boolean declaration	!;	re-throw (C++ <code>throw;</code> )	^:	protected:
~{	unary set complement	!	bitwise <i>not</i>	^	bitwise exclusive <i>or</i>
~c	destructor for class <code>c</code>	!	end of exception name	^	overloaded operators
~	logical <i>not</i> (unary)	!	volatile	&&=	alternate assignment

&&	logical <i>and</i>	{ and }	set literals	<>	unequal
&=	alternate assignment	[ and ]	array literals	<	less than
&,	typedef	[ and ]	array usage	<	<i>for-each</i> loop
&	address-of (unary)	[ and ]	in declarations	<	generic class declaration
&	bitwise <i>and</i>	[]	void	<	set membership
*=	alternate assignment	=	alternate assignment	. <i>t</i>	static variable, type <i>t</i>
*	multiplication	=	alternate assignment	. <i>v</i>	const variable <i>v</i>
*	pointer (unary)		logical <i>or</i>	..	subtype (binary)
*	set intersection		bitwise <i>or</i>	..	substrings
( <i>e</i> :)	case <i>e</i>		switch (unary)	..	this
(:)	default	\	for escape characters	.	floating point literals
(:	stepping iteration	;	terminator	> <i>n</i>	forward goto
()	grouping	:=	clone assignment	>!	finally
()	function calls	:	else	>=	greater than or equal
()	type conversion	:	derived class definition	>=	set contains
()	enumeration constants	:	Fortran-style iteration	>, ;	break
---	inline	:	<i>for-each</i> loop	>, <i>n</i> ;	labeled break
--	decrement operators	:	label	>. ;	return
--=	alternate assignment	'	character literals	>. <i>e</i> ;	return <i>e</i>
-:	private:	"	string literals	>> <i>n</i> +	open file <i>n</i>
->	new	,,	friend	>> <i>n</i> -	close file <i>n</i>
-	unary minus	,	Fortran-style iteration	>>=	alternate assignment
-	subtraction	,	set literals	>>	right shift
-	set difference	,	separating parameters	>>	input
_	base class constructor	< <i>n</i>	backward goto	>	greater than
_	super (from Java)	<!	try	//	comments
==	equality testing	<-	delete	/=	alternate assignment
=	initializers	<-[]	delete []	/;	packed records
=	assignment	<=>	swap (interchange)	/<>	not a member of a set
=	naming operator	<=	less than or equal	/	pattern literals
++	increment operators	<=	set contained	/	division
+=	alternate assignment	<, ;	continue	?,	safety statement
+:	public:	<, <i>n</i> ;	labeled continue	??	while
+,	#include	<<=	alternate assignment	?	if
+	unsigned declarations	<< <i>n</i> +	open file <i>n</i> for write	?	catch
+	unary plus	<< <i>n</i> -	close file <i>n</i>		
+	addition	<<	left shift		
+	set union	<<	output		
{:	endless loop	<. ;	exit		
{ and }	blocks	<. <i>e</i> ;	exit <i>e</i>		

Table 3. Keywords and concepts in C++/Java and their equivalents in \*

abstract class	{0;	else	:	NaN	!#
address-of (unary)	&	end of line	\n	new	->
alternate assignments:		endless loop	{:	<i>not</i> , bitwise	!
+= -= *= /= %=		enumeration constants	()	<i>not</i> , logical	~
&=  = ^= &&=   =		equality testing	==	null pointer	0
<<= >>=		escape characters	\	null string	0
<i>and</i> , bitwise	&	exception names end in !		open file <i>n</i>	>> <i>n</i> +, << <i>n</i> +
<i>and</i> , logical	&&	exclusive <i>or</i> , bitwise	^	operator	^
anonymous label	{:	exit	<.;	<i>or</i> , bitwise	
array literals	[ , ]	exit <i>e</i>	<.e;	<i>or</i> , logical	
array usage	[ ]	exponentiation	``	output	<<
assignment	=, :=	false	0	overloaded operators	<<
base class constructor	_	finally	>!	packed records	/;
blocks	{ }	float	[`32]	pattern literals	/
bool	[+1]	floating point	`	plus	+
break	>, ;	<i>for-each</i> loop	:	pointer (unary)	*
break, labeled	>, <i>n</i> ;	<i>for-each</i> loop	<	private:	-:
case <i>e</i>	( <i>e</i> :)	<i>for</i> -loop	(:	protected:	^:
catch	?	friend	,,	public:	+:
character (Unicode)	[']	function calls	()	remainder	%
character (Unicode)	[+16]	generic class declaration	<	repeat ( <i>see endless loops</i> )	
character literals	'	goto, backward	< <i>n</i>	re-throw ( <i>throw</i> ;	!;
cin	>>0	goto, forward	> <i>n</i>	return	>.;
class	unary =	greater than	>	return <i>e</i>	>.e;
clone assignment	:=	greater than or equal	>=	right shift	>>
close file <i>n</i>	>> <i>n</i> -, << <i>n</i> -	grouping	()	safety statement	?,
comments	//	if	?	set complement	~
const variable <i>v</i>	. <i>v</i>	increment operators	++	set contained	<=
continue	<, ;	initializers	=	set contains	>=
continue, labeled	<, <i>n</i> ;	#include	+,	set declarations	#
cout	<<0	inline	---	set difference	~ <i>or</i> -
decrement operators	--	input	>>	set intersection	*
default	(:)	int	[32]	set literals	{ , }
#define	=	iteration	: , ,	set membership	<
delete	<-	label	:	set non-membership	/<
delete []	<-[]	left shift	<<	set union	+
derived class definition	:	less than	<	short	[16]
destructor	~	less than or equal	<=	signed char	[16]
divide	/	long	[64]	sizeof <i>t</i>	#( <i>t</i> )
do ( <i>see endless loops</i> )		loop	{:	static variable, type <i>t</i>	. <i>t</i>
double	[`64]	minus	-	string	[""]
dynamic type	[0]	naming operator	=	string literals	"

struct	unary =	throw (like function call)	unsigned	[+32]
substrings	..	times	* unsigned int	[+32]
subtype	[m..n]	true	1 unsigned long	[+64]
super (from Java)	_(	try	<! unsigned short	[+16]
swap (interchange)	<=>	type conversion	() void	[]
switch		typedef	&, volatile	!
template class	unary =	unequal	!=, <>	wchar_t [']
this	..	union	{+;	while ??

## 6.2 Programming Examples

The following examples of C++/Java code and its equivalent in \* are intended to illustrate the ease of learning \*. These are all actually written in C++ except that some of them contain certain Java statements, such as `finally`, labeled `break`, and labeled `continue`.

It was never our intention that \* be easier to read than C++ or Java for a programmer whose first language is English. Indeed, such a programmer might consider it easier to do mathematics using *Integral* and *SquareRoot* and *Derivative* (see the Introduction above) than by using symbols. Mathematicians use symbols because mathematics is universal, not because they make it easier to understand. Our contention is that, by making programming universal, more people will be able to write programs and computer science should gain thereby, just as mathematics has gained by being universal.

It is also not our intention, however, that programmers fluent in C++ or Java should necessarily program in \* instead, if their first language is English. They can always program, and complete the debugging process, in C++ or Java, and then translate into \* using our Table 3 (ideally this translation should be done by a simple utility program). If `goto OPEN_FILE_FAILED;` is easier to understand, for such a programmer, than `goto 2;` then it should be used until debugging is complete. (Although ease of understanding, here, might be an illusion; *which* file failed to open? If one of them did, does the program go, here, to a place which assumes that a different one did?) Note:

- The use, in many of the \* examples here, of semicolons as separators rather than as terminators (although the omitted semicolons at the end need not be actually omitted, if the programmer so desires).
- The replacement of arrays with fixed upper bounds, such as `t[100]`, by resizable arrays with initial size 16, such as `t[=16]`. (Both C++ and Java have resizable arrays, but only by using named classes, such as `Vector`.)
- Each of these examples is assumed to be contained in a program whose safety level (see §5.5) is such that every feature of \* which is used is allowed.
- It is assumed that an `int` in C++ is 32 bits long, and that a `short` is 16 bits long.
- Note that the `next` field of a `DLL32node` in \*, although it is declared as pointing to an `SLL32node`, actually points to an `DLL32node`, and this is allowed in \* (as in C++ and Java) because the class `DLL32node` is derived from the class `SLL32node`.

C++/Java

\*

- Even/odd test:

<code>int k;</code>	<code>[32] k;</code>
<code>cout &lt;&lt; "Enter a number:\n";</code>	<code>&lt;&lt;0 &lt;&lt; "Enter a number:\n";</code>
<code>cin &gt;&gt; k;</code>	<code>&gt;&gt;0 &gt;&gt; k;</code>
<code>if (k%2 == 0) cout &lt;&lt; k &lt;&lt;</code>	<code>? (k%2 == 0) &lt;&lt;0 &lt;&lt; k &lt;&lt;</code>
<code>  " is even.\n";</code>	<code>  " is even.\n";</code>
<code>if (k%2 != 0) cout &lt;&lt; k &lt;&lt;</code>	<code>? (k%2 != 0) &lt;&lt;0 &lt;&lt; k &lt;&lt;</code>
<code>  " is odd.\n";</code>	<code>  " is odd.\n";</code>

- Greatest common divisor:

```

int gcd(int m, int n) {
    int i,j;
    i = m; j = n;
    while (i != j) {
        if (i > j) i-=j; else j-=i; }
    return i;
}

[32] gcd([32] m, [32] n) {
    [32] i,j;
    i = m; j = n;
    ?? (i != j) {
        ? (i > j) i-=j; : j-=i; }
    >.i;
}

```

- Search for k in table t of length n:

```

bool search(int t[100],
    const int n, int k) {
    int j;
    for (j = 0; j < n; j++)
        if (k == t[j]) return true;
    return false;
}

[+1] search([32] t[=16], .n,k){
    [32] j;
    j:0,n-1,? (k == t[j]) >.1;
    >.0
}

```

- Real absolute value function:

```

inline double abs(double x) {
    if (x >= 0) return x; return -x; }

--- [^64] abs([^64] x) {
    ? (x >= 0) >.x; >.-x; }

```

- Search and insert with goto:

```

int j,k,n,t[100];
for (j = 0; j < n; j++)
    if (k == t[j]) goto done;
t[n] = k; n++;
done: (next statement)

[32] j,k,n,t[=16];
j:0,n-1,? (k == t[j]) >9;
t[n] = k; n++;
9: (next statement)

```

- Search and insert with break and else-logic (for-loop):

```

int j,k,n,t[100];
for (j = 0; j < n; j++) {
    if (k == t[j]) break; }
else {t[n] = k; n++;}

[32] j,k,n,t[=16];
j:0,n-1,
{ ? (k == t[j]) >,; }
: {t[n] = k; n++;}

```

- Search and insert with break and else-logic (while-loop):

```

int j,k,n,t[100];
j = 0;
while (j < n) {
    if (k == t[j]) break;
    j++; }
else {t[n] = k; n++;}

[32] j,k,n,t[=16];
j = 0;
?? (j < n) {
    ? (k == t[j]) >,; j++; }
: {t[n] = k; n++;}

```

- Polynomial evaluation:

```

int i; const int n;
double x,value,a[100];
value = a[0];
for (i = 1; i<=n; i++)
    value += a[i]*pow(x,(double)i);

```

```

[32] i,.n;
[64] x,value,a[=16];
value = a[0];
i:1,n,value += a[i]*x`i;

```

- Searching a sorted array and returning an index, or -1 meaning “not found”:

```

int k,n,t[100];
for (int j = 0; j < n; j++)
    if (k > t[j]) continue;
    if (k < t[j]) return -1;
    return j;
}

```

```

[32] k,n,t[=16];
(:[32] j = 0; j < n; j++) {
    ? (k > t[j]) <,;
    ? (k < t[j]) >.-1;
    >.j;
}

```

- Interchange sort:

```

int i,n,t[100],z; bool more;
do {
    more = false;
    for (i = 1; i < n; i++)
        if (t[i-1] > t[i]) {
            z = t[i-1]; t[i-1] = t[i];
            t[i] = z; more = true;
        }
} while (more);

```

```

[32] i,n,t[=16]; [+1] more;
{:
    more = 0;
    i:1,n-1,
    ? (t[i-1] > t[i]) {
        t[i-1] <=> t[i];
        more = 1
    }
? (!more) >,; }

```

- Calculator with switch, case, and default:

```

class CalcError {};
char c,s[256];
do {
    int i = 0, j = 0,
        r = s[0]-'0';
    c = s[++j];
    if (c == '=') break;
    i = s[++j]-'0';
    if (i < 0 || i > 9)
        throw CalcError();
    case '+': r += i; break;
switch (c) {
    case '-': r -= i; break;
    case '*': r *= i; break;
    case '/': r /= i; break;
    default: throw CalcError();
}

```

```

= CalcError! {};
[+16] c,s[=16];
{:
    [32] i = 0, j = 0,
        r = s[0]-'0';
    c = s[++j];
    ? (c == '=') >,;
    i = s[++j]-'0';
    ? (i < 0 || i > 9)
        CalcError!();
    | (c) {
        ('+':) r += i;
        ('-':) r -= i;
        ('*':) r *= i;
        ('/':) r /= i;
        (:) CalcError!();
}

```

```

    }
} while(true);
cout << r << '\n';

```

```

    }
}
<<0 << r << '\n';

```

- Display Fibonacci numbers < 100:

```

void fib() {
    short i = 0, j = 1, oldi;
    cout << '0';
    while (j < 100) {
        cout << " " << j;
        oldi = i; i = j;
        j = oldi+j;
    }
}

```

```

[] fib() {
    [16] i = 0, j = 1, oldi;
    <<0 << '0';
    ?? (j < 100) {
        <<0 << " " << j;
        oldi = i; i = j;
        j = oldi+j;
    }
}

```

- Grade point averages and a case that falls through:

```

class gpaError {char badc;};
int k,n; double t,gpa;
char s[256];
n = 0; t = 0;
for (k = 0; s[k] != 0; k++) {
    switch (s[k]) {
        case 'A': t += 4; n++; break;
        case 'B': t += 3; n++; break;
        case 'C': t += 2; n++; break;
        case 'D': t += 1;
        case 'F': n++; break;
        case '+': t += 0.3; break;
        case '-': t -= 0.3; break;
        default: throw gpaError(s[k]);
    }
}
gpa = t/n;

```

```

= gpaError! {'' badc};
[32] k,n; [64] t,gpa;
['] s[=16];
n = 0; t = 0;
(:k = 0; s[k] != 0; k++) {
    | (s[k]) {
        ('A:') t += 4; n++;
        ('B:') t += 3; n++;
        ('C:') t += 2; n++;
        ('D:') t += 1; >1;
        ('F:') 1: n++;
        ('+:') t += 0.3;
        ('-:') t -= 0.3;
        (:) gpaError!(s[k])
    }
}
gpa = t/n;

```

- String length with pointers and type definition:

```

typedef pint *int;
int length; pint s;
pint p = s;
while (*p != 0) p++;
length = p-s;

```

```

&, p32 *[32];
[32] length; p32 s;
p32 p = s;
?? (*p != 0) p++;
length = p-s;

```

- Intersection of two subsets of {0, 1, 2, ..., 31}:

```
int a,b,c;
```

```
#32 a,b,c;
```

```
c = a & b;
```

```
c = a*b;
```

- Union of two subsets of  $\{0, 1, 2, \dots, 31\}$ :

```
int a,b,c;
```

```
#32 a,b,c;
```

```
c = a | b;
```

```
c = a+b;
```

- Set difference of two subsets of  $\{0, 1, 2, \dots, 31\}$ :

```
int a,b,c;
```

```
#32 a,b,c;
```

```
c = a & !b;
```

```
c = a~b;
```

- Test whether  $k$ , with  $0 \leq k \leq 31$ , is contained in a subset  $c$  of  $\{0, 1, 2, \dots, 31\}$ :

```
int c,k;
```

```
#32 c; [32] k;
```

```
if (((1 << k) & c) == 0)
```

```
? (k /< c) NotFound!(k, c);
```

```
throw NotFound(k, c);
```

- Test whether a subset of  $\{0, 1, 2, \dots, 31\}$  is contained in another such subset:

```
int a,b;
```

```
#32 a,b;
```

```
if (a & !b == 0) (some statement)
```

```
? (a<=b) (some statement)
```

- $k = 63-k$ ; for  $0 \leq k \leq 63$  (the subtype which (Stroustrup 1997, p. 275) calls `Tiny`):

```
Tiny k; k = 63-k;
```

```
[0..63] k; k = 63-k;
```

- Intersection  $a_3$  of general sets  $a_1$  and  $a_2$ , having respective sizes  $n_1$  and  $n_2$ :

```
int k1,k2,k3,n1,n2;
```

```
#[`64] a1,a2,a3;
```

```
double a1[100],a2[100],a3[100];
```

```
a3 = a1*a2;
```

```
k1 = 0, k2 = 0, k3 = 0; for (;;)
```

```
if (a1[k1] == a2[k2]) {
```

```
    a3[k3] = a1[k1];
```

```
    k1++; if (k1 >= n1) return;
```

```
    k2++; if (k2 >= n2) return;
```

```
    k3++;}
```

```
else if (a1[k1] < a2[k2]) {
```

```
    k1++; if (k1 >= n1) return;}
```

```
else {k2++; if (k2 >= n2) return;}
```

```
}
```

- Binary search of  $t[0]$  through  $t[n-1]$ , in ascending order (note: the current subarray is  $t[\text{first}]$  through  $t[\text{last}-1]$ , not  $t[\text{first}]$  through  $t[\text{last}]$ ):

```
int n,first,middle,last;
```

```
[32] n,first,middle,last;
```

```
double x,t[100];
```

```
[`64] x,t[=16];
```

```
first = 0; last = n;
```

```
first = 0; last = n;
```

```
do {
```

```
{:
```

```
    middle = (first+last)/2;
```

```
    middle = (first+last)/2;
```

```

    if (first == middle) break;
    if (x < t[middle])
        last = middle;
    else first = middle;
} while (true);
if (x == t[first])
    return first;
else return -1;

```

```

? (first == middle) >,;
? (x < t[middle])
    last = middle;
    : first = middle
}
? (x == t[first])
>.first; : >.-1;

```

- Search a double array for an element x, using a Java labeled break:

```

int i,j,x,t[100][100];
alpha: for (i = 0; i < 100; i++) {
    for (j = 0; j < 100; j++) {
        if (x == t[i][j]) break alpha;
    }
}
return (i == 100)?false:true;

```

```

[32] i,j,x,t[100,100];
25: i:0,99,{
    j:0,99,{
        ? (x == t[i,j]) >,25
    }
}
return (i == 100)?0:1;

```

- Search a double array for a row of all zeroes, using a Java labeled continue:

```

int i,j,x,t[100][100];
alpha: for (i = 0; i < 100; i++) {
    for (j = 0; j < 100; j++) {
        if (t[i][j] != 0)
            continue alpha;
    } return true;
} return false;

```

```

[32] i,j,x,t[100,100];
25: i:0,99,{
    j:0,99,{
        ? (t[i,j] != 0) <,25
    } >.1;
} >.0;

```

- Singly linked list (SLL) node with integer content:

```

class SLLintnode {public:
    int content;
    static int ncalls = 0;
    SLLintnode *next;
    SLLintnode(int c,
        SLLintnode n) {ncalls++;
        content = c; next = n;}
};

```

```

= SLL32node {+:
    [32] content;
    .[32] ncalls = 0;
    SLL32node *next;
    SLL32node([32] c,
        SLL32node n) {ncalls++;
        content = c; next = n;}
};

```

- Doubly linked list (DLL) node with integer content:

```

class DLLintnode : SLLintnode {
    public:
    DLLintnode *prev;
    DLLintnode(int c,
        DLLintnode nextn,

```

```

= DLL32node : SLL32node {
    +:
    DLL32node *prev;
    DLL32node([32] c,
        DLL32node nextn,

```

```

        DLLintnode prevn) :                DLL32node prevn)
    SLLintnode(c, nextn)                    {_(c,nextn); prev=prevn;
    { prev = prevn; }                        }
};                                           };

```

• Doubly linked list with integer content, using doubly linked list nodes:

```

class DLLint {public:                        = DLL32 {+:
    DLLintnode *first;                       DLL32node *first;
    DLLint() {first = 0;}                   DLL32() {first = 0}
    DLLint operator+=                       DLL32 ^+=
        (DLLint &list, int k) {              (DLL32 &list, [32] k) {
        newnode = new DLLintnode             newnode = ->DLL32node
            (k, list, 0);                    (k, list, 0);
        newnode->next->prev                   newnode->next->prev
            = newnode;                       = newnode;
        list.first = newnode;                list.first = newnode;
    }                                         list.ncalls++;
        list.ncalls++;                       }
    ~DLLint() {                              ~DLL32() {
        for(DLLintnode *p = first;           (:DLL32node *p = first;
        p != 0; p = p->next)                 p != 0; p = p->next)
        delete p;                             <-p;
    }                                         }
};                                           };

```

• File copy:

```

istream file1(file_name_1);                >>1+(file_name_1);
ostream file2(file_name_2);                <<1+(file_name_2);
char c;                                    <<1:=>>1;
while (!file1.eof()) {                     >>1-; <<1-;
    file1 >> c; file2 << c;
}
file1.close(); file2.close();

```

• File compare, using finally from Java:

```

class notTheSame {char ch1,ch2;};          = notTheSame! {['] ch1,ch2;};
istream file1(file_name_1);                >>1+(file_name_1);
istream file2(file_name_2);                >>2+(file_name_2);
try {char c1,c2;                            <! {
    while (!file1.eof()) {                   ? (>>1!=>>2)
        file1 >> c1; file2 >> c2;           notTheSame!(c1,c2);
        if (c1 != c2)                       }
        throw notTheSame(c1,c2); }          >! { >>1-; >>2-; }

```

```

}
finally {
    file1.close(); file2.close();
}

```

• 20-statement fast sort (based on heap sort):

```

int i,j,k,l,m,n,t[100];
j = n;
i = n>>1;
13: i--;
11: l = i+1;
    k = t[l];
    goto loop;
17: t[l] = t[m];
    l = m;
loop:  m = l<<1;
    if (m > j) goto 16;
    if (m == j) goto 15;
    if (t[m+1] > t[m]) m++;
15: if (t[m] > k) goto 17;
16: t[l] = k;
    if (i != 0) goto 13;
    l = t[j];
    t[j] = t[l];
    t[l] = l;
    j--;
    if (j != 1) goto 11;

[32] i,j,k,l,m,n,t[=16];
j = n;
i = n>>1;
1:  i--;
2:  l = i+1;
    k = t[l];
    >4;
3:  t[l] = t[m];
    l = m;
4:  m = l<<1;
    ? (m > j) >6;
    ? (m == j) >5;
    ? (t[m+1] > t[m]) m++;
5:  ? (t[m] > k) <3;
6:  t[l] = k;
    ? (i != 0) <1;
    t[j] <=> t[l];
j--;
? (j != 1) <2;

```

• Complex class with public members and sample overloaded operators:

```

class complex {
public: double re; double im;
    complex(double x, double y)
        { re = x; im = y; }
    complex operator+(complex z1, z2)
        { return complex(z1.re+z2.re,
            z1.im+z2.im); }
    complex operator-(complex z) {
        return complex(-z.re,-z.im); }
};

= complex {
+: [^64] re; [^64] im;
    complex([ ^64] x, [ ^64] y)
        { re = x; im = y }
    complex ^+(complex z1, z2)
        { >.complex(z1.re+z2.re,
            z1.im+z2.im); }
    complex ^-(complex z) {
        >.complex(-z.re,-z.im); }
};

```

• Complex class with private members and sample overloaded friend operators:

```

class complex {
private: double re; double im;

= complex {
-: [^64] re; [^64] im;

```

```

public:
    complex(double x, double y)
        { re = x; im = y; }
};
friend complex operator+
    (complex z1, z2);
friend complex operator-
    (complex z);

+:
    complex([`64] x, [`64] y)
        { re = x; im = y; }
};
,,complex ^+(complex z1, z2);
,,complex ^-(complex z);

```

• Converting a float to an int, using a union and bit fields:

```

struct floatfields {
    unsigned int sign:1;
    unsigned int expon:8;
    unsigned int signif:23;
};
union floatff {
    float as_float;
    floatfields as_fields;
}
int float2int(float x) {
    int y; floatff z;
    z.as_float = x;
    if (z.as_fields.expon < 127)
        return 0;
    y = (z.as_fields.signif+(1<<23))
        >> (151-z.as_fields.expon);
    if (z.as_fields.sign != 0)
        y = -y;
    return y;
}

= floatfields {/;
    [+1] sign;
    [+8] expon;
    [+23] signif;
};
= floatff {+;
    [`32] u32;
    floatfields ub32;
};
[32] f32to32([`32] x) {
    [32] y; floatff z;
    z.u32 = x;
    ? (z.ub32.expon < 127) >.0;
    y = (z.ub32.signif+(1<<23))
        >> (151-z.ub32.expon);
    ? (z.ub32.sign != 0)
        y = -y;
    >.y;
}

```

• Constructing strings and writing them to a file (with finally from Java):

```

class FormatError {};
ostream file1(some_file_name);
int nchars; wstring s;
try {
    while (true) {
        if (noMoreStrings()) break;
        s = constructString();
        nchars += sizeof s;
        file1.write(s);
    }
}

= FormatError! {};
<<1+(some_file_name);
[32] nchars; ["] s;
<! {
    {
        ? (noMoreStrings()) >,;
        s = constructString();
        nchars += #(s);
        <<1 << s;
    }
}

```

```

catch (FormatError) {
    cerr << "Format error\n";
}
finally {
    file1.close();
    cout << nchars <<
        " characters written\n";
}
}

? (FormatError!()) {
    <<! << "Format error\n";
}
>! {
    <<1-;
    <<0 << nchars <<
        " characters written\n";
}
}

```

### 6.3 Non-English Examples

Since \* has no keywords, the only non-English words in \* programs, as we have seen, are names, and these can be easily translated into other languages. Looking for the moment at our last example above, of constructing strings and writing them to a file, we see that a programmer whose native language is French or Greek could render this \* program as

French	Greek
= ErreurDeFormat! {};	= ΣφάλμαΜορφοποίησης! {};
<<1+( <i>nom de fichier</i> );	<<1+( <i>ονομα του αρχειου</i> );
[32] ncaractères; ["] s;	[32] αΧαρακτηρων; ["] σ;
<! {	<! {
{:	{:
? (lesCordesNePlus()) >,;	? (πλεονΧορδες()) >,;
s = construisezUneCorde();	σ = κΑκολουθια();
ncaractères += #(s);	αΧαρακτηρων += #(σ);
<<1 << s;	<<1 << σ;
}	}
}	}
? (ErreurDeFormat!()) {	? (ΣφάλμαΜορφοποίησης!()) {
<<! << "Erreur de format\n";	<<! << "Σφάλμα μορφοποίησης\n";
}	}
>! {	>! {
<<1-;	<<1-;
<<0 << ncaractères <<	<<0 << αΧαρακτηρων <<
" caractères écrites\n";	" χαρακτηρων γραπτη\n";
}	}

and similarly for any other natural language.

### 7. Future Work

In the terminology which arose in the design of Ada (Whitaker 1993), this paper is to be regarded as a "Strawman" document. Only the basic features of existing languages have their counterparts in \* as here described, and more advanced features may be added to \* at a later time. (Also to be introduced at a later time is a formal definition of \*. Just as the Ada community did, we prefer to wait with this. The Ada Strawman document was not accompanied by a formal definition of Ada; that came later).

Indeed, it is not intended for \* to be a static language, never modifiable (like Algol 60 or PL/I or standard Pascal or Ada 83). Rather, it is intended to be a language (like C or Perl when they were first introduced) which grows as new features are perceived to be needed. Three of these are briefly discussed below.

### 7.1 Parallel and Concurrent Programming

At present there are no features of `*` for parallel or concurrent programming. Partly this is because there is far less agreement among programming languages as to how these features should be implemented than there is with respect to other language features. By far the most ambitious approach is taken by Ada; other approaches, however, are taken by Java, Python, and Lua. Much of what can be done with such features in one of these languages would be difficult or impossible in any of the others; it is thus unclear to this author how to implement such features in `*`. Nevertheless, we have reserved certain syntax in `*` for tasks, concurrent processes, threads, and coroutines. These are all intended to have names starting with `||` (suggesting *parallel* lines).

### 7.2 Further Pattern Generalizations

The English-oriented character sequence `\d` (`d` for digit) in a pattern needs to be generalized to digits in other natural languages. Once this is done, there is a more general question of how to generalize the other English-oriented character sequences in patterns, such as `\D`, `\w`, `\W`, `\s`, `\S`, `\b`, and `\B`. All the features of patterns in `*` need to be accompanied by worked examples.

### 7.3 The Unibase

In §1.2 it was suggested that libraries be translated from English into other languages. Ideally, however, this should be done by an independent organization, much as Unicode was formulated by a consortium (Aliprand et al. 2000). One can imagine a database, called the Unibase, which contains library names in all these languages. The Unibase folder, or directory, would contain several files. First there is the English-language Unibase file, which contains library names as these are expressed in English. Every other Unibase file is for a specific natural language, and it contains these same names, in the same order, translated into that language.

Every such name starts with the `@` character. Every such name `@N` has a unique index  $i$ , which is the same in every Unibase file. You can always write `@N` in your program, but `*` will replace this by `@i`. The first line of every `*` program would consist of the characters `*`, followed by a native language name (`English`, `français`, `deutsch`, `español`, `Ελληνικά`, or the like). The corresponding Unibase file is then used in printing out, or displaying on the screen, that program in the specified natural language, using the stored indices.

This has the consequence that no programmer is ever forced to think in English when working with a `*` program. The only words in a `*` program that are in the programmer's native language are, first, the names of variables, functions, classes, files, user-defined exceptions, and GUI names; and then the comments and documentation, which, as in any programming language, should be extensive. There is an analogy here with a mathematical paper written in a language other than English. The mathematics itself is universal, but the paper has to be translated into English in order for most English-speaking mathematicians to be able to read it.

The names `sin`, `cos`, `tan`, and `log` have become part of standard mathematics, and are preserved in `*`. The names `cot`, `sec`, and `csc` are not used in any programming language, including `*`; one uses `1.0/tan(x)`, `1.0/cos(x)`, and `1.0/sin(x)` instead. Other mathematical function names would be in the Unibase. Note that there are still unsolved problems connected with the Unibase, which are intended to be the subject of future research, such as the question of how it should handle abbreviations (such as `ptr` for `pointer`) and run-on phrases (such as `intptr` meaning "pointer to an integer").

### 7.4 Processors and Textbooks

Finally, and most obviously, this document is intended as a spur to compiler writers, to produce compilers, interpreters, and the like for `*`. Once these are written, elementary `*` textbooks can be written in various languages, allowing students to learn programming even if they know no English at all. Meanwhile, even without compilers or textbooks, `*` may be used as a formal notation, much as APL was during the first six years of its existence (Wiedmann 1974, p. vii). Unlike Ada, `*` does not discourage subset compilers; indeed, these might uncover unexpected difficulties in compiling `*`, giving rise to modifications to the design of `*`.

Note that compiling `*` might require more input character backup than compiling other languages, due to the reuse of special characters. This should not be a problem if a compiler uses a resizable array in which to keep characters over which backup might be necessary. Also, for the same reason, a lexical analyzer for `*` might have to be written by hand, instead of being produced by a lexical analyzer generator. This should be at most a minor annoyance to compiler writers.

### 7.5 Responses to Reviewers' Concerns

Does this paper resemble more a programming language user manual, rather than a research paper? If `*` were a language with an actual compiler, this would be a valid criticism. All the syntactic features of the language

should be in a user manual, and then there should be a paper, separate from the user manual, explaining the significance of the language and referring the reader, for details, to the user manual. But \*, like APL or like Ada, is being initially presented from a theoretical standpoint only. If there are to be compilers, these will come later, in all these cases (over five years later, for APL and also for Ada). In order for these to be written, a complete description of the language must be available, and in a theoretical journal, since \* is theoretical only, at this point. This is why we have written the paper the way we have.

In one reviewer's opinion, we could have dedicated less space to mentioning syntactic features of the language, and more space to discussing its efficiency, speed, memory space, and other technical aspects to those of other languages. Again, if there were an actual compiler, these would be valid criticisms; but, without a compiler, we can only speculate, and not demonstrate, concerning speed and memory space.

What is a programmer supposed to gain from a language that is full of symbol sequences, rather than well-known keywords that are easy to understand even for non-English speakers? The advantage is not to the individual programmers, but rather to the organizations for which they work. Once a large program is written in \*, it can be maintained by people who can think in their own natural languages, rather than having to think, at least partially, in English.

### References

- Aliprand, J., Allen, J., Becker, J., Davis, M., Everson, M., Freytag, A., ... Whistler, K. (2000). *The Unicode Standard Version 3.0*. Addison-Wesley.
- Knuth, D. E. (1968). Semantics of context-free languages. *Mathematical Systems Theory*, 2(2), 127-145. <http://dx.doi.org/10.1007/BF01692511>
- Stroustrup, B. (1997). *The C++ programming language* (3rd ed.). Addison-Wesley.
- Wiedmann, C. (1974). *Handbook of APL Programming*. New York: Petrocelli Books.
- Whitaker, W. (1993). Ada - The Project, The DoD High Order Language Working Group. *ACM SIGPLAN Notices*, 28(3), 299-331. <http://dx.doi.org/10.1145/155360.155376>