the Small booklet

April 2001

ITB CompuPhase

Typeset with TEX in the "Computer Modern" and "Pandora" typefaces at a base size of 11 points.

Introduction

"Small" is a simple, typeless, 32-bit extension language with a C-like syntax. Execution speed, stability, simplicity and a small footprint were essential design criterions for both the language and the interpreter/abstract machine that a Small program runs on.

An application or tool cannot do or be *everything* for *all* users. This not only justifies the diversity of editors, compilers, operating systems and many other software systems, it also explains the presence of extensive configuration options and macro or scripting languages in applications. My own applications have contained a variety of little languages; most were very simple, some were extensive. And most needs could have been solved by a general purpose language with a special purpose library.

The Small language was designed as a flexible language for manipulating objects in a host application. The tool set (compiler, abstract machine) were written so that they were easily extensible and would run on different software/hardware architectures.

Many years ago, I retyped the "Small C" compiler from Dr. Dobb's Journal, by Ron Cain and James Hendrix. Having just grasped the basics of the C language, working on the Small C compiler was a learning experience of its own. The compiler, as published, generated code for a 8080 assembler. The first modification I needed to make was to adapt it to the 8086 processor. Through the years that I used it (to write low level system software) I expanded the compiler with new features and fixed many details. Eventually, as I was moving towards bigger applications in more conventional environments, the Small C compiler was replaced by main-stream development environments.

In early 1998, I was looking for a scripting language for an animation toolkit. Among the languages that I evaluated were Lua, BOB, Scheme, REXX, Java, ScriptEase and Forth. None of these languages covered my requirements completely. I have always felt that the C language is a flexible and convenient language whose basics can be mastered in a week. While experimenting with Quincy (from Al Stevens), I decided that a simplified C would probably be a good fit. I dusted off Small C.

Small is a descendent of the original Small C, which at its turn was a subset of C. Some of the modifications that I did to Small C, e.g. the removal of the

type system and the substitution of pointers by references, were so fundamental that I could hardly call my language a "subset of C" or a "C dialect" anymore. Therefore, I stripped off the "C" from the title and kept the name "Small".

I am indebted to Ron Cain and James Hendrix (and more recently, Andy Yuen), and to Dr. Dobb's Journal to get this ball rolling. Although I must have touched nearly every line of the original code multiple times, the Small C origins are still clearly visible.

A detailed treatise of the design goals and compromises is in appendix A; here I would like to summarize a few key points. As written in the previous paragraphs, Small is for customizing applications, not for writing applications. Small is weak on data structuring because Small programs are intended to manipulate objects (text, sprites, streams, queries, ...) in the host application, but the Small program is, *by intent*, denied direct access to any data outside its abstract machine. The only means that a Small program has to manipulate objects in the host application is by calling subroutines —so called "native functions"— that the host application provides.

Small is flexible in that key area: calling functions. Small supports default values for any of the arguments of a function (not just the last), call-by-reference as well as call-by-value, and "named" as well as "positional" function arguments. Small does not have a "type checking" mechanism, by virtue of being a typeless language, but it *does* offer in replacement a "classification checking" mechanism, called "tags". The tag system is especially convenient for function arguments because each argument may specify multiple acceptable tags.

For any language, the power (or weakness) lies not in the individual features, but in their combination. For Small, I feel that the combination of default values for function arguments in combination with named arguments blend together to a very convenient way to call functions —and indirectly, to manipulate objects in the host application.

This booklet tries to unite two books:
◇ a manual for the Small compiler and the abstract machine that I wrote;
◇ a definition of the Small language, independent of the current implementation.

These goals reflect the two main parts of the booklet entitled: "Small: the language" and "Small: the compiler". The third part of the booklet, "Appendices", provides relevant supplementary information and a rationale for the design.

In the language specification, the term "parser" refers to any implementation that reads and operates on conforming Small programs. A parser refers to both interpreters or compilers.

Small: the language

Small is a simple programming language with a syntax reminiscent to the "C" programming language. A Small program consists of a set of functions and a set of variables. The variables are data objects and the functions contain instructions (called "statements") that operate on the data objects or that perform tasks.

For compiling instructions, see page 69

The first program in almost any computer language is one that prints a simple string; printing "Hello world" is a classic example. In Small, the program would look like:

```
#include <console>

main()
    printf("Hello world^n")
```

Small separates the language from the function library. Since Small is designed to be an *extension language* for applications, the function set that a Small program has at its disposal depends on the implementation. As a result, the Small language has no intrinsic knowledge of *any* function; a program must declare every function that it uses. In this first example, the `printf` function must be declared, either by writing the definition (the function's *prototype*) somewhere near the top of the source file, or by including a text file that contains the required definition —along, perhaps, with definitions of constants and of other functions. The "Hello world" example uses the latter approach, as its first line exhibits.

A stand-alone Small program starts execution with function `main`. Here, the function `main` contains only a single instruction, which is printed at the line below the function head itself. Line breaks and indenting are insignificant; the invocation of the function `printf` could equally well be on the same line as the head of function `main`.

The arguments of a function are always enclosed in parentheses. If a function does not have any arguments, like function `main`, the opening and closing parentheses are still present. The single argument of the `printf` function is a string, which must be enclosed in double quotes.

String literals: 41

The characters `^n` near the end of the string form a *control character*, in this case they indicate a "newline" symbol. When `printf` encounters the newline control character, it advances the cursor to the first column of the next line. One has to use the `^n` control character to insert a "newline" into the string, because a string may not wrap over multiple lines.

Control characters: 41

Small is a "case sensitive" language: upper and lower case letters are considered to be different letters. It would be an error to spell the function `printf` in the above example as "`PrintF`".

This first example also reveals a few differences between Small and the C language:
- ◇ semicolons are optional, except when writing multiple statements on one line;
- ◇ when the body of a function is a single instruction, the braces (for a compound instruction) are optional;
- ◇ "escape characters" are called "control characters" in Small, and they start with a caret ("`^`") rather than a backslash ("\"), but see also page 56 or page 69 to change this special character.

❖

Fundamental elements of most programs are calculations, decisions (conditional execution), iterations (loops) and variables to store input data, output data and intermediate results. The next program example illustrates many of these concepts. The program calculates the greatest common divisor of two values using an algorithm invented by Euclides.

```
/* the greatest common divisor of two values, using Euclides' algorithm */
#include <console>

main()
    {
    print("Input two values^n")
    new a = getvalue()
    new b = getvalue()
    while (a != b)
        if (a > b)
            a = a - b
        else
            b = b - a
    printf("The greatest common divisor is %d^n", a)
    }
```

When the body of a function contains more than one statement, these statements must be embodied in braces —the "`{`" and "`}`" characters. This groups the instructions to a single *compound statement*. The notion of grouping statements in a compound statement applies as well to the bodies of `if`–`else` and loop instructions.

Compound statement: 52

The `new` keyword creates a variable. The name of the variable follows `new`. It is common, but not imperative, to assign a value to the variable already at the moment of its creation. Variables must be declared before they are used in an

Data declarations are covered in detail starting at page 18

expression. The `getvalue` function (also part of the "console" function set) reads in a value from the keyboard and returns the result. Note that Small is a *typeless* language, all variables are numeric cells that can hold a signed integral value.

"while" loop: 55
"if–else": 54

Loop instructions, like `while`, repeat a single instruction as long as the loop condition, the expression between parentheses, is "true". To execute multiple instructions in a loop, again, requires one to group these in a compound statement. The `if`–`else` instruction has one instruction for the "true" clause and one for the "false".

Relational operators: 48

The loop condition for the `while` loop is "`(a != b)`"; the symbol `!=` is the "not equal to" operator. That is, the `if`–`else` instruction is repeated until `a` equals `b`. It is good practice to indent the instructions that run under control of another statement, as is done in the preceding example.

The call to `printf`, near the bottom of the example, differs from how it was used in the first example (page 4). Here it prints literal text and the value of a variable (in a user-specified format) at the same time. The `%d` symbol in the string is a token that indicates the position and the format that the subsequent argument to function `printf` should be printed. At run time, the token `%d` is replaced by the value of variable `a` (the second argument of `printf`).

❖

Next to *simple* variables with a size of a single cell, Small supports arrays and symbolic constants, as exemplified in the program below. It displays a series of prime numbers using the well known "sieve of Eratosthenes".

```
/* Print all primes below 100, using the "Sieve of Eratosthenes" algorithm */
#include <console>

main()
    {
    const max_primes = 100
    new series[max_primes] = { true, ... }

    for (new i = 2; i < max_primes; ++i)
        if (series[i])
            {
            printf("%d ", i)
            /* filter all multiples of this "prime" from the list */
            for (new j = 2 * i; j < max_primes; j += i)
                series[j] = false
            }
    }
```

When a program or sub-program has some fixed limit built-in, it is good practice create a symbolic constant for it. In the preceding example, the symbol `max_primes` is a constant with the value 100. The program uses the symbol `max_primes` three times after its definition: in the declaration of the variable `series` and in both `for` loops. If we were to adapt the program to print all primes below 500, there is now only one line to change.

Constant declaration: 42

Like simple variables, arrays may be initialized upon creation. Small offers a convenient shorthand to initialize all elements to a fixed value: all hundred elements of the "`series`" array are set to `true` —without requiring that the programmer types in the word "`true`" a hundred times. The symbols `true` and `false` are predefined constants.

Progressive initiallers: 20

When a simple variable, like the variables `i` and `j` in the primes sieve example, is declared in the first expression of a `for` loop, the variable is valid only inside the loop. Variable declaration has its own rules; it is not a statement —although it looks like one. One of those rules is that the first expression of a `for` loop may contain a variable declaration.

"for" loop: 53

Both `for` loops also introduce new operators in their third expression. The `++` operator increments its operand by one; that is, `++i` is equal to `i = i + 1`. The `+=` operator adds the expression on its right to the variable on its left; that is, `j += i` is equal to `j = j + i`.

An overview of all operators: 45

The first element in the `series` array is `series[0]`, if the array holds `max_primes` elements, the last element in the array is `series[max_primes-1]`. If `max_primes` is 100, the last element, then, is `series[99]`. Accessing `series[100]` is invalid.

Larger programs separate tasks and operations into functions. Using functions increases the modularity of programs and functions, when well written, are portable to other programs. The following example implements a function to calculate numbers from the Fibonacci series.

The Fibonacci sequence was discovered by Leonardo "Fibonacci" of Pisa, an Italian mathematician of the 13th century—whose greatest achievement was popularizing for the Western world the Hindu-Arabic numerals. The Fibonacci numbers describe a surprising variety of natural phenomena. For example, the two or three sets of spirals in pineapples, pine cones and sunflowers usually have consecutive Fibonacci numbers between 5 and 89 as their number of spirals. The numbers that occur naturally in branching patterns (e.g. that of plants) are indeed Fibonacci numbers. Finally, although the Fibonacci sequence is *not* a geometric sequence,

the further the sequence is extended, the more closely the ratio between successive terms approaches the *golden ratio*, of 1.6188...that appears so often in art and architecture.

"assert" statement: 52

The `assert` instruction at the top of the `fibonacci` function deserves explicit mention; it guards against "impossible" or invalid conditions.

```
/* Calculation of Fibonacci numbers by iteration */
#include <console>

main()
    {
    print("Enter a value: ")
    new v = getvalue()
    printf("The value of Fibonacci number %d is %d^n",
           v, fibonacci(v) )
    }

fibonacci(n)
    {
    assert n > 0

    new a = 0, b = 1
    for (new i = 2; i < n; i++)
        {
        new c = a + b
        a = b
        b = c
        }
    return a + b
    }
```

Functions: properties & features: 24

The implementation of a user-defined function is not much different than that of function `main`. Function `fibonacci` shows two new concepts, though: it receives an input value through a parameter and it returns a value (it has a "result").

Function parameters are declared in the function header; the single parameter in this example is `n`. Inside the function, a parameter behaves as a local variable, but one whose value is passed from the outside at the *call* to the function.

The `return` statement ends a function and sets the result of the function. It need not appear at the very end of the function; early exits are permitted.

Dates are a particularly rich source of algorithms and conversion routines, because the calenders that a date refers to have known such a diversity, through time and around the world.

The "Julian Day Number" is attributed to Josephus Scaliger[1] and it counts the number of days since November 24, 4714 BC (proleptic Gregorian calendar). Scaliger chose that date because it marked the coincidence of three well-established cycles: the 28-year Solar Cycle (of the old Julian calendar), the 19-year Metonic Cycle and the 15-year Indiction Cycle (periodic taxes or governemental requisitions in ancient Rome), and because no literature or recorded history was known to predate that particular date in the remote past. Scaliger used this concept to reconcile dates in historic documents, later astronomers embraced it to calculate intervals between two events more easily.

Julian Day numbers (sometimes denoted with unit "JD") should not be confused with Julian Dates (the number of days since the start of the *same* year), or with the Julian calendar that was introduced by Julius Caesar.

Below is a program that calculates the Julian Day number from a date in the (proleptic) Gregorian calendar, and vice versa. Note that in the proleptic Gregorian calendar, the first year is 1 AD (Anno Domini) and the year before that is 1 BC (Before Christ): year zero does not exits! The program uses negative year values for BC years and positive (non-zero) values for AD years. The Gregorian calendar was decreed to start on 15 October 1582 by pope Gregory XIII, which means that earlier dates do not really exist in the Gregorian calendar. When extending the Gregorian calendar to days before 15 October 1582, we refer to the *proleptic* Gregorian calendar.

```
/* calculate Julian Day number from a date, and vice versa */
#include <console>

main()
    {
    new d, m, y, jdn

    print("Give a date (dd-mm-yyyy): ")
    d = getvalue(_, '-', '/')
    m = getvalue(_, '-', '/')
    y = getvalue()

    jdn = DateToJulian(d, m, y)
    printf("Date %d/%d/%d = %d JD^n", d, m, y, jdn)

    print("Give a Julian Day Number: ")
    jdn = getvalue()
    JulianToDate(jdn, d, m, y)
```

[1] There is some debate on exactly *what* Josephus Scaliger invented and *who* or *what* he called it after.

```
    printf("%d JD = %d/%d/%d^n", jdn, d, m, y)
    }

DateToJulian(day, month, year)
    {
    /* The first year is 1. Year 0 does not exist: it is 1 BC (or -1) */
    assert year != 0
    if (year < 0)
        year++

    /* move January and February to the end of the previous year */
    if (month < 2)
        year--, month += 12
    new jdn = 365*year + year/4 - year/100 + year/400
              + (153*month - 457) / 5
              + day + 1721119
    return jdn
    }

JulianToDate(jdn, &day, &month, &year)
    {
    jdn -= 1721119

    /* approximate year, then adjust in a loop */
    year = (400 * jdn) / 146097
    while (365*year + year/4 - year/100 + year/400 < jdn)
        year++
    year--

    /* determine month */
    jdn -= 365*year + year/4 - year/100 + year/400
    month = (5*jdn + 457) / 153

    /* determine day */
    day = jdn - (153*month - 457) / 5

    /* move January and February to start of the year */
    if (month > 12)
        month -= 12, year++

    /* adjust negative years (year 0 must become 1 BC, or -1) */
    if (year <= 0)
        year--
    }
```

Function `main` starts with creating variables to hold the day, month and year, and the calculated Julian Day number. Then it reads in a date —three calls to `getvalue`— and calls function `DateToJulian` to calculate the day number. After calculating the result, `main` prints the date that you entered and the Julian Day number for that date. Now, let us focus on function `DateToJulian`...

Near the top of function `DateToJulian`, it increments the `year` value if it is negative; it does this to cope with the absence of a "zero" year in the proleptic Gregorian calendar. In other words, function `DateToJulian` modifies its function

arguments (later, it also modifies `month`). Inside a function, an argument behaves like a local variable: you may modify it. These modifications remain local to the function `DateToJulian`, however. Function `main` passes the values of `d`, `m` and `y` into `DateToJulian`, who maps them to its function arguments `day`, `month` and `year` respectively. Although `DateToJulian` modifies `year` and `month`, it does not change `y` and `m` in function `main`; it only changes local copies of `y` and `m`. This concept is called "call by value".

"Call by value" versus "call by reference": 25

The example intentionally uses different names for the local variables in the functions `main` and `DateToJulian`, for the purpose of making the above explanation easier. Renaming `main`'s variables `d`, `m` and `y` to `day`, `month` and `year` respectively, does not change the matter: then you just happen to have two local variables called `day`, two called `month` and two called `year`, which is perfectly valid in Small.

The remainder of function `DateToJulian` is uninteresting arithmetic.

Returning to the second part of the function `main` we see that it now asks for a day number and calls another function, `JulianToDate`, to find the date that matches the day number. Function `JulianToDate` is interesting because it takes one input argument (the Julian Day number) and needs to calculate three output values, the day, month and year. Alas, a function can only have a single return value —that is, a `return` statement in a function may only containe *one* expression. To solve this, `JulianToDate` specifically requests that changes that it makes to some of its function arguments are copied back to the variables of the caller of the function. Then, in `main`, the variables that must hold the result of `JulianToDate` are passed as arguments to `JulianToDate`.

Function `JulianToDate` marks arguments individually for the purpose of "copying back to caller" by prefixing the arguments with an `&` symbol. Arguments with an `&` are copied back, arguments without is are not. "Copying back" is actually not the correct term. An argument tagged with an `&` is passed to the function in a special way that allows the function to directly modify the original variable. This is called "call by reference" and an argument that uses it is a "reference argument".

In other words, if `main` passes `y` to `JulianToDate` —who maps it to its function argument `year`— and `JulianToDate` changes `year`, then `JulianToDate` *really* changes `y`. Only through reference arguments can a function directly modify a variable that is declared in a different function.

To summarize the use of call-by-value versus call-by-reference: if a function has

one output value, you typically use a `return` statement; if a function has more output values, you use reference arguments. You may combine the two inside a single function, for example in a function that returns its "normal" output via a reference argument and an error code in its return value.

As an aside, many desktop application use conversions to and from Julian Day numbers (or varieties of it) to conveniently calculate the number of days between to dates or to calculate the date that is 90 days from now —for example.

Small has no intrinsic "string" type; character strings are stored in arrays, with the convention that the array element behind the last valid character is zero. Working with strings is therefore equivalent with working with arrays.

Among the simplest of encryption schemes is the one called "ROT13" —actually the algorithm is quite "weak" from a cryptological point of view. It is most widely used in public electronic forums (BBSes, Usenet) to hide texts from casual reading, such as the solution to puzzles or riddles. ROT13 simply "rotates" the alphabet by half its length, i.e. 13 characters. It is a symmetric operation: applying it twice on the same text reveals the original.

```
/* Simple encryption, using ROT13 */
#include <console>

main()
    {
    printf("Please type the string to mangle: ")

    new str[100]
    getstring(str, sizeof str)
    rot13(str)

    printf("After mangling, the string is: ^"%s^"^n", str)
    }

rot13(string[])
    {
    for (new index = 0; string[index]; index++)
        if ('a' <= string[index] <= 'z')
            string[index] = (string[index] - 'a' + 13) % 26 + 'a'
        else if ('A' <= string[index] <= 'Z')
            string[index] = (string[index] - 'A' + 13) % 26 + 'A'
    }
```

In the function header of `rot13`, the parameter "`string`" is declared as an array, but without specifying the size of the array —there is no value between the square brackets. When you specify a size for an array in a function header, it must match the size of the *actual* parameter in the function call.Omitting the array size specification in the function header removes this restriction and allows the function to be called with arrays of any size. You must then have some other means of determining the (maximum) size of the array. In the case of a string parameter, one can simply search for the zero terminator.

The `for` loop that walks over the string is typical for string processing functions. Note that the loop condition is "`string[index]`". The rule for true/false conditions in Small is that any value is "true", except zero. That is, when the array cell at `string[index]` is zero, it is "false" and the loop aborts.

The ROT13 algorithm rotates only letters; digits, punctuation and special characters are left unaltered. Additionally, upper and lower case letters must be handled separately. Inside the `for` loop, two `if` statements filter out the characters of interest. The way that the second `if` is chained to the "else" clause of the first `if` is noteworthy, as it is a typical method of testing for multiple non-overlapping conditions.

Another point of interest are the conditions in the two `if` statements. The first `if`, for example, holds the condition "`'a' <= string[index] <= 'z'`", which means that the expression is true if (and only if) both `'a' <= string[index]` *and* `string[index] <= 'z'` are true. In the combined expression, the relational operators are said to be "chained", as they chain multiple comparisons in one condition.

Relational operators: 48

Finally, note how the last `printf` in function `main` uses the control character `^"` to print a double quote. Normally a double quote ends the literal string; the control character inserts a double quote into the string.

Control characters: 41

❖

In a typeless language, we might assign a different purpose to some array elements than to other elements in the same array. Small supports enumerated constants with an extension that allows it to mimic some functionality that other languages implement with "structures" or "records".

The example to illustrate enumerations and arrays is longer than previous Small programs, and it also displays a few other features, such as global variables and named parameters.

```
/* Priority queue (for simple text strings) */
#include <core>
#include <console>

enum message
    {
    text : 40 char,
    priority
    }

main()
    {
    new msg[message]

    /* insert a few items (read from console input) */
    printf("Please insert a few messages and their priorities; \
            end with an empty string^n")
    for ( ;; )
        {
        printf("Message:  ")
        getstring(.string = msg[text], .maxlength = 40, .pack = true)
        if (strlen(msg[text]) == 0)
            break
        printf("Priority: ")
        msg[priority] = getvalue()
        if (!insert(msg))
            {
            printf("Queue is full, cannot insert more items^n")
            break
            }
        }

    /* now print the messages extracted from the queue */
    printf("^nContents of the queue:^n")
    while (extract(msg))
        printf("[%d] %s^n", msg[priority], msg[text])
    }

const queuesize = 10
new queue[queuesize][message]
new queueitems = 0

insert(const item[message])
    {
    /* check if the queue can hold one more message */
    if (queueitems == queuesize)
        return false            /* queue is full */

    /* find the position to insert it to */
    new pos = queueitems        /* start at the bottom */
    while (pos > 0 && item[priority] > queue[pos-1][priority])
        --pos                   /* higher priority: move one position up */

    /* make place for the item at the insertion spot */
    for (new i = queueitems; i > pos; --i)
```

```
            queue[i] = queue[i-1]

        /* add the message to the correct slot */
        queue[pos] = item
        queueitems++

        return true
        }

    extract(item[message])
        {
        /* check whether the queue has one more message */
        if (queueitems == 0)
            return false            /* queue is empty */

        /* copy the topmost item */
        item = queue[0]
        --queueitems

        /* move the queue one position up */
        for (new i = 0; i < queueitems; ++i)
            queue[i] = queue[i+1]

        return true
        }
```

Near the top of the program listing is the declaration of the enumeration `message`. This enumeration defines two constants: `text`, which is zero, and `priority`, which is 11 (assuming a 32-bit cell). The idea behind an enumeration is to quickly define a list of symbolic constants without duplicates. By default, every constant in the list is 1 higher than its predecessor and the very first constant in the list is zero. However, you may give an *extra* increment for a constant so that the successor has a value of 1 plus that extra increment. The `text` constant specifies an extra increment of `40 char`. In Small, `char` is an operator, it returns the number of cells needed to hold a packed string of the specified number of characters. Assuming a 32-bit cell and a 8-bit character, 10 cells can hold 40 packed characters.

"enum" statement: 42

"char" operator: 50

Immediately at the top of function `main`, a new array variable is declared with the size of `message`. The symbol `message` is the name of the enumeration. It is also a constant with the value of the last constant in the enumeration list plus the optional extra increment for that last element. So in this example, `message` is 11. That is to say, array `msg` is declared to hold 11 cells.

Further in `main` are two loops. The `for` loop reads strings and priority values from the console and inserts them in a queue. The `while` loop below that extracts element by element from the queue and prints the information on the screen. The point to note, is that the `for` loop stores both the string and the priority number (an integer) in the same variable `msg`; indeed, function `main` declares only a single variable. Function `getstring` stores the message text that you type starting at

array `msg[text]` while the priority value is stored (by an assignment a few lines lower) in `msg[priority]`. The `printf` function in the `while` loop reads the string and the value from those positions as well.

At the same time, the `msg` array is an entity on itself: it is passed in its entirity to function `insert`. That function, near the end, says "`queue[queueitems] = item`", where `item` is an array with size `message` and `queue` is a two-dimensional array that holds `queuesize` elements of size `message`. The declaration of `queue` and `queuesize` are just above function `insert`.

The example implements a "priority queue". You can insert a number of messages into the queue and when these messages all have the same priority, they are extracted from the queue in the same order. However, when the messages have different priorities, the one with the highest priority comes out first. The "intelligence" for this operation is inside function `insert`: it first determines the position of the new message to add, then moves a few messages one position upward to make space for the new message. Function `extract` simply always retrieves the first element of the queue and shifts all remaining elements down by one position.

Note that both functions `insert` and `extract` work on two shared variables, `queue` and `queueitems`. A variable that is declared inside a function, like variable `msg` in function `main` can only be accessed from within that function. A "global variable" is accessible by *all* functions, and that variable is declared outside the scope of any function. Variables must still be declared before they are used, so `main` cannot access variables `queue` and `queueitems`, but both `insert` and `extract` can.

Function `extract` returns the messages with the highest priority via its function argument `item`. That is, it changes its function argument by copying the first element of the `queue` array into `item`. Function `insert` copies in the other direction and it does not change its function argument `item`. In such a case, it is advised to mark the function argument as "`const`". This helps the Small parser to both check for errors and to generate better (more compact, quicker) code.

Named parameters: 26

A final remark on this latest sample is the call to `getstring` in function `main`: note how the parameters are attributed with a decsription. The first parameter is labeled "`.string`", the second "`.maxlength`" and the third "`.pack`". Function `getstring` receives "named parameters" rather than positional parameters. The order in which named parameters are listed is not important. Named parameters are convenient in specifying —and deciphering— long parameter lists.

If you know the C programming language, you will have seen many concepts that you are familiar with, and a few new ones. If you don't know C, the pace of this introduction has probably been quite high. Whether you are new to C or experienced in C, I encourage you to read the following pages carefully. This booklet attempts to be both an informal introduction and a (more formal) language specification at the same time, perhaps succeeding at neither. Since it is also the only book on Small, the focus of this booklet is on being accurate and complete, rather than being easy to grasp.

The double nature of this section of the booklet shows in the order at which it presents the subjects. The larger conceptual parts of the language, variables and functions, are covered first. The operators, the statements and general syntax rules follow later; not that they are less important, but they are easier to learn, to look up, or to take for granted.

Data and declarations

Small is a typeless language. All data elements are of type "cell", and a cell can hold an integral number. The size of a cell (in bytes) is system dependent —usually, a cell is 32-bits.

A new variable is declared with the keywords `new`, `static` or `public`. A simple variable declaration creates a variable that occupies one "cell" of data memory. Unless it is explicitly initialized, the value of the new variable is zero.

A variable declaration may occur:
◇ at any position where a statement would be valid —local variables;
◇ at any position where a function declaration (native function declarations) or a function implementation would be valid —global variables;
◇ in the first expression of a `for` loop instruction —also local variables.

Local declarations

A local declaration appears inside a compound statement. A local variable can only be accessed from within the compound statement, and from nested compound statements. A declaration in the first expression of a `for` loop instruction is also a local declaration.

Global declarations

A global declaration appears outside a function and a global variable is accessible to any function. Global data objects can only be initialized with constant expressions.

• Static local declarations

A local variable is destroyed when the execution leaves the compound block in which the variable was created. Local variables in a function only exist during the run time of that function. Each new run of the function creates and initializes new local variables. When a local variable is declared with the keyword `static` rather than `new`, the variable remains in existence after the end of a function. This means that static local variables provide private, permanent storage that is accessible only from a single function (or compound block). Like global variables, static local variables can only be initialized with constant expressions.

• Public declarations

Global "simple" variables (no arrays) may be declared "public" in two ways:
◇ declare the variable using the keyword `public` instead of `new`;

◇ start the variable name with the "@" symbol.

Public variables behave like global variables. The abstract machine, however, has a special interface to access public variables (read and write). As such, a host program (which includes the abstract machine) may require that you declare a variable with a specific name as "public" for special purposes —such as the most recent error number, or the general program state.

The Abstract Machine interface: 83

• Constant variables

It is sometimes convenient to be able to create a variable that is initialized once and that may not be modified. Such a variable behaves much like a symbolic constant, but it still is a variable.

Symbolic constants: 42

To declare a constant variable, insert the keyword `const` between the keyword that starts the variable declaration —`new`, `static` or `public`— and the variable name.

Examples:

```
new const address[4] = { 192, 0, 168, 66 }
public const status              /* initialized to zero */
```

Three typical situations where one may use a constant variable are:

◇ To create an "array" constant; symbolic constants cannot be indexed.

◇ For a public variable that should be set by the host application, and *only* by the host application. See the preceding section for public variables.

◇ A special case is to mark array arguments to functions as `const`. Array arguments are always passed by reference, declaring them as `const` guards against unintentional modification. Refer to page 26 for an example of `const` function arguments.

• Arrays (single dimension)

The syntax `name[constant]` declares `name` to be an array of "`constant`" elements, where each element is a single cell. The `name` is a placeholder of an identifier name of your choosing and `constant` is a positive non-zero value; `constant` may be absent. If there is no value between the brackets, the number of elements is set equal to the number of initiallers —see the example below.

See also "multi-dimensional arrays", page 20

The array index range is "zero based" which means that the first element is at `name[0]` and the last element is `name[constant-1]`.

• Initialization

Constants: 40

Data objects can be initialized at their declaration. The initialler of a global data object must be a constant. Arrays, global or local, must also be initialized with constants.

Uninitialized data defaults to zero.

Examples:

```
new i = 1
new j                           /* j is zero */
new k = 'a'                     /* k has character code for letter 'a' */

new a[] = {1,4,9,16,25}         /* a has 5 elements */
new s1[20] = {'a','b'}          /* the other 18 elements are 0 */

new s2[] = "Hello world..."     /* a unpacked string */
```

Examples of **invalid** declarations:

```
new c[3] = 4                    /* an array cannot be set to a value */
new i = "Good-bye"              /* i must be an array for this initialler */
new q[]                         /* unknown size of array */
new p[2] = { i + j, k - 3 }     /* array initiallers must be constants */
```

• Progressive initiallers for arrays

The ellipsis operator continues the progression of the initialisation constants for an array, based on the last two initialized elements. The ellipsis operator (three dots, or "...") initializes the array up to its declared size.

Examples:

```
new a[10] = { 1, ... }          /* sets all ten elements to 1 */
new b[10] = { 1, 2, ... }       /* sets: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 */
new c[8] = { 1, 2, 40, 50, ... } /* sets: 1, 2, 40, 50, 60, 70, 80, 90 */
new d[10] = { 10, 9, ... }      /* sets: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 */
```

• Multi-dimensional arrays

Multi-dimensional arrays are arrays that contain references to the sub-arrays.[2] That is, a two-dimensional array is an "array of single-dimensional arrays". Below are a few examples of declarations of two-dimensional arrays.

[2] The current implementation of the Small compiler supports only arrays with up to two dimensions.

```
new a[4][3]
new b[3][2] = { { 1, 2 }, { 3, 4 }, { 5, 6 } }
new c[3][3] = { { 1 }, { 2, ...}, { 3, 4, ... } }
new d[2][5] = { !"agreement", !"dispute" }
new e[2][]  = { "OK", "Cancel" }
```

As the last declaration (variable "e") shows, the final dimension may have an unspecified length, in which case the length of each sub-array is determined from the related initializer. Every sub-array may have a different size; in this particular example, "e[1][5]" contains the letter "l" from the word "Cancel", but "e[0][5]" is *invalid* because the length of the sub-array "e[0]" is only three cells (containing the letters "O", "K" and a zero terminator).

• Tag names

A tag is a label that denotes the objective of —or the meaning of— a variable, a constant or a function result. Tags are entirely optional, their only purpose is to allow a stronger compile-time error checking of operands in expressions, of function arguments and of array indices.

Tags should not be confused with variable *types* in C and other languages. A tagged variable is still a cell, which holds an integral number.
◇ a type specifies the memory layout and range of variables and function results
◇ a tagname labels the purpose of variables, constants and function results

A tag consists of a symbol name followed by a colon; it has the same syntax as a label. A tag precedes the symbol name of a variable, constant or function. In an assignment, only the right hand of the "=" sign may be tagged.

Label syntax: 52

Examples of valid tagged variable and constant definitions are:

```
new bool:flag = true        /* "flag" can only hold "true" or "false" */

const error:success = 0
const error:fatal= 1
const error:nonfatal = 2

error:errno = fatal
```

The sequence of the constants success, fatal and nonfatal could more conveniently be declared using an enum instruction, as illustrated below. The enumeration instruction below creates four constants, success, fatal, nonfatal and error, all with the tag error.

"enum" statement: 42

```
enum error {
    success,
```

```
    fatal,
    nonfatal,
}
```

A typical use of "tagged" `enum`'s is in conjunction with arrays. If every field of an array has a distinct purpose, you can use a tagged `enum` to declare the size of an array and to add tag checking to the array usage in a single step:

```
enum rectangle
    {
    left,
    top,
    right,
    bottom
    }

new my_rect[rectangle]          /* array is declared as having 4 cells */

my_rect[left] = 10
my_rect[top] = 5
my_rect[right] = 30
my_rect[bottom] = 12

for (new i = 0; rectangle:i < rectangle; ++i)
    my_rect[rectangle:i] *= 2
```

After the declaration of "`my_rect`" above, you can access the second field of `my_rect` with "`my_rect[top]`", but saying "`my_rect[1]`" will give a parser diagnostic (a warning or error message). A tag override (or a tag *cast*) adjusts a function, constant or variable to the desired tag name. The `for` loop at the last two lines in the preceding example depicts this: the loop variable `i` is a plain, untagged cell, an it must be cast to the tag `rectangle` before as an index in the array `my_rect`. Note that the `enum` construct has created both a constant and a tag with the name "`rectangle`".

Tag names intruduced so far started with a lower case letter; these are "weak" tags. Tag names that start with an upper case letter are "strong" tags. The difference between weak and strong tags is that weak tags may, in a few circumstances, be dropped implicitly by the Small parser —so that a weakly tagged expression becomes an untagged expression. The tag checking mechanism verifies the following situations:

◇ When the expressions on both sides of a binary operator have a different tag, or when one of the expressions is tagged and the other is not, the compiler issues a "*tag mismatch*" diagnostic. There is no difference between weak and strong tags in this situation.

- ◇ There is a special case for the assignment operator: the compiler issues a diagnostic if the variable on the left side of an assignment operator has a tag, and the expression on the right side either has a different tag or is untagged. However, if the variable on the left of the assignment operator is untagged, it accepts an expression (on the right side) with a *weak* tag. In other words, a weak tag is dropped in an assignment when the *lvalue* is untagged.
- ◇ Passing arguments to functions follows the rule for assignments. The compiler issues a diagnostic when the *formal* parameter (in a function definition) has a tag and the *actual* parameter (in the function call) either is untagged or has a different tag. However, if the formal parameter is untagged, it also accepts a parameter with any *weak* tag.
- ◇ An array may specify a tag for every dimension, see the "`my_rect`" example above. Tag checking array indices follows the rule of binary operator tag checking: there is no difference between weak and strong tags.

"lvalue": the variable on the left side in an assigment, see page 45

Functions

A function declaration specifies the name of the function and, between parentheses, its formal parameters. A function may also return a value. A function declaration must appear on a global level (i.e. outside any other functions) and is globally accessible.

If a semicolon follows the function declaration (rather than a statement), the declaration denotes a forward declaration of the function.

The preferred way to declare forward functions is at page 31

The `return` statement sets the function result. For example, function `sum` (see below) has as result the value of both its arguments added together. The `return` expression is optional for a function, but one cannot use the value of a function that does not return a value.

```
sum(a, b)
    return a + b
```

Arguments of a function are (implicitly declared) local variables for that function. The *function call* determines the values of the arguments.

Another example of a complete definition of the function `leapyear` (which returns `true` for a leap year and `false` for a non-leap year):

```
leapyear(y)
    return y % 4 == 0 && y % 100 != 0 || y % 400 == 0
```

The logical and arithmetic operators used in the `leapyear` example are covered on pages 48 and 45 respectively.

Usually a function contains local variable declarations and consists of a compound statement. In the following example, note the `assert` statement to guard against negative values for the exponent.

"assert" statement: 52

```
power(x, y)
    {
    /* returns x raised to the power of y */
    assert y >= 0
    new r = 1
    for (new i = 0; i < y; i++)
        r *= x
    return r
    }
```

• Function arguments (call-by-value versus call-by-reference)

The "`faculty`" function in the next program has one parameter which it uses in a loop to calculate the faculty of that number. What deserves attention is that the function modifies its argument.

```
/* Calculation of the faculty of a value */
#include <console>

main()
    {
    print("Enter a value: ")
    new v = getvalue()
    new f = faculty(v)
    printf("The faculty of %d is %d^n", v, f)
    }

faculty(n)
    {
    assert n >= 0

    new result = 1
    while (n > 0)
        result *= n--

    return result
    }
```

Whatever (positive) value that "`n`" had at the entry of the `while` loop in function `faculty`, "`n`" will be zero at the end of the loop. In the case of the `faculty` function, the parameter is passed "by value", so the change of "`n`" is local to the `faculty` function. In other words, function `main` passes "`v`" as input to function `faculty`, but upon return of `faculty`, "`v`" still has the same value as before the function call.

Arguments that occupy a single cell can be passed by value or by reference. The default is "pass by value". To create a function argument that is passed by reference, prefix the argument name with the character `&`.

Example:

```
swap(&a, &b)
    {
    new temp = b
    b = a
    a = temp
    }
```

To pass an array to a function, append a pair of brackets to the argument name. You may optionally indicate the size of the array; doing so improves error checking of the parser.

Example:

```
addvector(a[], const b[], size)
    {
    for (new i = 0; i < size; i++)
        a[i] += b[i]
    }
```

Constant variables: 19

Arrays are always passed by reference. As a side note, array `b` in the above example does not change in the body of the function. The function argument has been declared as `const` to make this explicit. In addition to improving error checking, it also allows the Small parser to generate more efficient code.

To pass an array of literals to a function, use the same syntax as for array initiallers: a literal string or the series of array indices enclosed in braces (see page 41; the ellipsis for progressive initiallers cannot be used). Literal arrays can only have a single dimension.

The following snippet calls `addvector` to add five to every element of the array "`vect`":

```
new vect[3] = { 1, 2, 3 }

addvector(vect, {5, 5, 5}, 3)

/* vect[] now holds the values 6, 7 and 8 */
```

"Hello world" program: 4

The invocation of function `printf` with the string `"Hello world^n"` in the first ubiquitous program is another example of passing a literal array to a function.

• Named parameters versus positional parameters

In the previous examples, the order of parameters of a function call was important, because each parameter is copied to the function argument with the same sequential position. For example, with the function `weekday` (which uses Zeller's congruence algorithm) defined as below, you call `weekday(12,31,1999)` to get the week day of the last day of this century.

```
weekday(month, day, year)
    {
    /* returns the day of the week: 0=Saturday, 1=Sunday, etc. */
    if (month <= 2)
        month += 12, --year
    new j = year % 100
```

```
    new e = year / 100
    return (day + (month+1)*26/10 + j + j/4 + e/4 - 2*e) % 7
    }
```

Date formats vary according to culture and nation. While the format *month/day/year* is common in the United States of America, European countries often use the *day/month/year* format, and technical publications sometimes standardize on the *year/month/day* format. In other words, no order of arguments in the `weekday` function is "logical" or "conventional". That being the case, the alternative way to pass parameters to a function is to use "named parameters", as in the next examples (the three function calls are equivalent):

```
new wkday1 = weekday( .month = 12, .day = 31, .year = 1999)

new wkday2 = weekday( .day = 31, .month = 12, .year = 1999)

new wkday3 = weekday( .year = 1999, .month = 12, .day = 31)
```

With named parameters, a period (".") precedes the name of the function argument. The function argument can be set to any expression that is valid for the argument. The equal sign ("=") does in the case of a named parameter not indicate an assignment; rather it links the expression that follows the equal sign to one of the function arguments.

One may mix positional parameters and named parameters in a function call with the restriction that all positional parameters must precede any named parameters.

• Default values of function arguments

A function argument may have a default value. When the function call specifies an argument placeholder instead of a valid argument, the default value applies. The argument placeholder is the underscore character (`_`). The argument placeholder is only valid for function arguments that have a default value.

If the rightmost argument placeholder may simply be stripped from the function argument list. For example, if function `increment` is defined as:

```
increment(&value, incr=1) value += incr
```

the following function calls are all equivalent:

```
increment(a)
increment(a, _)
increment(a, 1)
```

Default argument values for passed-by-reference arguments are useful to make the input argument optional. For example, if the function `divmod` is designed to return both the quotient and the remainder of a division operation through its arguments, default values make these arguments optional:

```
divmod(a, b, &quotient=0, &remainder=0)
    {
    quotient = a / b
    remainder = a % b
    }
```

With the preceding definition of function `divmod`, the following function calls are now all valid:

```
new p, q

divmod(10, 3, p, q)
divmod(10, 3, p, _)
divmod(10, 3, _, q)
divmod(10, 3, p)
```

Default arguments for array arguments are often convenient to set a default string or prompt to a function that receives a string argument. For example:

```
print_error(const message[], const title[] = "Error: ")
    {
    print(title)
    print(message)
    print("^n")
    }
```

The next example adds the fields of one array to another array, and by default increments the first three elements of the destination array by one:

```
addvector(a[], const b[] = {1, 1, 1}, size = 3)
    {
    for (new i = 0; i < size; i++)
        a[i] += b[i]
    }
```

• Arguments with tag names

A tag, see page 21, optionally precedes a function argument. Using tags improves the compile-time error checking of the script and it serves as "implicit documentation" of the function. For example, a function that computes the square root of an input value in fixed point precision may require that the input parameter is a fixed point value and that the result is fixed point as well. The function below uses the fixed point extension module, see page 62, and an approximation

algorithm known as "bisection" to calculate the square root. Note the use of tag overrides on numeric literals.

```
fixed:sqroot(fixed:value)
    {
    new fixed:low = fixed:0
    new fixed:high = value

    while (high - low > fixed:1)
        {
        new fixed:mid = (low + high) / fixed:2
        if (fmul(mid, mid) < value)
            low = mid
        else
            high = mid
        }

    return low
    }
```

The bisection algorithm is related to binary search, in the sense that it continuously halves the interval in which the result must lie. A "successive substitution" algorithm like Newton-Raphson, that takes the slope of the function's curve into account, achieves precise results more quickly, but at the cost that a stopping criterion is more difficult to state. State of the art algorithms for computing square roots combine bisection and Newton-Raphson algorithms.

In the case of an array, the array indices can be tagged as well. For example, a function that creates the intersection of two rectangles may be written as:

For the "rectangle" tag, see page 22

```
intersection(dest[rectangle], const first[rectangle], const second[rectangle])
    {
    if (first[right] > second[left] && first[left] < second[right]
        && first[bottom] > second[top] && first[top] < second[bottom])
        {
        /* there is an intersection, calculate it using the "min" and
         * "max" functions from the "core" library, see page 58.
         */
        dest[left] = max(first[left], second[left])
        dest[right] = min(first[right], second[right])
        dest[top] = max(first[top], second[top])
        dest[bottom] = min(first[bottom], second[bottom])
        return true
        }
    else
        {
        /* "first" and "second" do not intersect */
        dest = { 0, 0, 0, 0 }
        return false
        }
    }
```

• Variable arguments

A function that takes a variable number of arguments, uses the "ellipsis" operator ("...") in the function header to denote the position of the first variable argument. The function can access the arguments with the predefined functions `numargs`, `getarg` and `setarg` (see page 58).

Function `sum` returns the summation of all of its parameters. It uses a variable length parameter list.

```
sum(...)
    {
    new result = 0
    for (new i = 0; i < numargs(); ++i)
        result += getarg(i)
    return result
    }
```

This function could be used in:

```
new v = sum(1, 2, 3, 4, 5)
```

Tags: 21

A tag may precede the ellipsis to enforce that all subsequent parameters have the same tag, but otherwise there is no error checking with a variable argument list and this feature should therefore be used with caution.

• Coercion rules

If the function argument, as per the function definition (or its declaration), is a "value parameter", the caller can pass as a parameter to the function:

◇ a value, which is passed by value;
◇ a reference, whose dereferenced value is passed;
◇ an (indexed) array element, which is a value.

If the function argument is a reference, the caller can pass to the function:

◇ a value, whose address is passed;
◇ a reference, which is passed by value because it has the type that the function expects;
◇ an (indexed) array element, which is a value.

If the function argument is an array, the caller can pass to the function:

◇ an array with the same dimensions, whose starting address is passed;
◇ an (indexed) array element, in which case the address of the element is passed.

• Recursion

A `faculty` example function earlier in this chapter used a simple loop. An example function that calculated a number from the Fibonacci series also used a loop and an extra variable to do the trick. These two functions are the most popular routines to illustrate recursive functions, so by implementing these as iterative procedures, you might be inclined to think that Small does not support recursion.

"faculty": 25
"fibonacci": 8

Well, Small *does* support recursion, but the calculation of faculties and of Fibonacci numbers happen to be good examples of when *not* to use recursion. Faculty is easier to understand with a loop than it is with recursion. Solving Fibonacci numbers by recursion indeed simplifies the problem, but at the cost of being extremely inefficient: the recursive Fibonacci calculates the same values over and over again.

The program below is an implementation of the famous "Towers of Hanoi" game in a recursive function:

There exists an intriguing iterative solution to the Towers of Hanoi.

```
/* The Towers of Hanoi, a game solved through recursion */
#include <console>

main()
    {
    print("How many disks: ")
    new disks = getvalue()
    move(1, 3, 2, disks)
    }

move(from, to, spare, numdisks)
    {
    if (numdisks > 1)
        move(from, spare, to, numdisks-1)
    printf("Move disk from pillar %d to pillar %d^n", from, to)
    if (numdisks > 1)
        move(spare, to, from, numdisks-1)
    }
```

• Forward declarations

The current "reference implementation" of the Small compiler does not require functions to be declared before their first use. This section documents the requirements of early implementations of the Small compiler and of other implementations of the Small language (if they exist).

A Small parser may require that functions are defined before they can be used. That is, the implementation of the function must precede the first call to that function in the source file. In the cases that this is inconvenient, or impossible

(as in the case of indirect recursion), you can make a "forward declaration" of the function. Forward declarations are similar, in syntax and in purpose, to declarations of native functions.

Native function interface: pages 33 and 86

To create a forward declaration, precede the function name and its parameter list with the keyword `forward`. For compatibility with early versions of Small, and for similarity with C/C++, an alternative way to forwardly declare a function is by typing the function header and terminating it with a semicolon (which follows the closing parenthesis of the parameter list).

The full definition of the function, with a non-empty body, is implemented elsewhere in the source file.

• Public functions, function main

A stand-alone program must have the function `main`. This function is the starting point of the program. The function `main` may not have arguments.

A function library need not to have a `main` function, but it must have it either a `main` function, *or* at least one public function. Function `main` is the primary entry point into the compiled program; the public functions are alternative entry points to the program. The virtual machine can start execution with one of the public functions. A function library may have a `main` function to perform one-time initialization at startup.

To make a function public, prefix the function name with the keyword `public`. For example, a text editor may call the public function "`onkey`" for every key that the user typed in, so that the user can change (or reject) keystrokes. The `onkey` function below would replace every "`~`" character (code 126 in the ISO Latin-1 character set) by the "hard space" code in the ANSI character table:

```
public onkey(keycode)
    {
    if (key=='~')
        return 160      /* replace ~ by hard space (code 160 in Latin-1) */
    else
        return key      /* leave other keys unaltered */
    }
```

Functions whose name starts with the "@" symbol are also public. So an alternative way to write the public function `onkey` function is:

```
@onkey(keycode)
    return key=='~' ? 160 : key
```

The "@" character, when used, becomes part of the function name; that is, in the last example, the function is called "`@onkey`".

• Stock functions

A "stock" function is a function that the Small parser must "plug into" the program when it is used, and that it may simply "remove" from the program (without warning) when it is not used. Stock functions allow a compiler or interpreter to optimize the memory footprint and the file size of a (compiled) Small program: any stock function that is not referred to, is completely skipped —as if it were lacking from the source file.

A typical use of stock functions, hence, is in the creation of a set of "library" functions. A collection of general purpose functions, all marked as "stock" may be put in a separate include file, which is then included in any Small script. Only the library functions that are actually used get "linked" in.

To declare a stock function, prefix the function name with the keyword `stock`. Public functions and native functions cannot be declared "stock".

• Native functions

A Small program can call application-specific functions through a "native function". The native function must be declared in the Small program by means of a function prototype. The function name must be preceded by the keyword `native`.

Examples:

```
native getparam(a[], b[], size)

native multiply_matrix(a[], b[], size)

native openfile(const name[])
```

The names "`getparam`", "`multiply_matrix`" and "`openfile`" are the *internal* names of the native functions; these are the names by which the functions are known in the Small program. Optionally, you may also set an *external* name for the native function, which is the name of the function as the "host application" knows it. To do so, affix an equal sign to the function prototype followed by the external name. For example:

```
native getparam(a[], b[], size) = host_getparam

native multiply_matrix(a[], b[], size) = mtx_mul
```

Unless specified explicitly, the external name is equal to the internal name of a native function. One typical use for explicit external names is to overcome the maximum name length of a native function: the external name may not exceed 19 characters due to the specification of the executable file format. The internal name may be longer, but only if the external "alias" is given explicitly.

See page 86 for implementing native functions in C/C++ (on the "host application" side).

• User defined operators

Tags: 21

The only data type of Small is a "cell", typically a 32-bit number or bit pattern. The meaning of a value in a cell depends on the particular application —it need not always be a signed integer value. Small allows to attach a "meaning" to a cell with its "tag" mechanism.

Based on tags, Small also allows you to redefine operators for cells with a specific purpose. The example below defines a tag "`ones`" and an operator to add two "ones" values together (the example also implements operators for subtraction and negation). The example was inspired by the checksum algorithm of several protocols in the TCP/IP protocol suite: it simulates one's complement arithmetic by adding the carry bit of an arithmetic overflow back to the least significant bit of the value.

```
#include <console>

main()
    {
    new ones: chksum = 0xffffffff
    print("Input values in hexadecimal, zero to exit^n")

    new ones: value
    do
        {
        printf(">> ")
        value = ones: getvalue(.base=16)
        chksum = chksum * value
        printf("Checksum = %x^n", chksum)
        }
    while (value)
    }

ones: operator+(ones: a, ones: b)
    {
    const ones:mask =  0xffff   /* word mask */
    const ones:shift = 16       /* word shift */

    /* add low words and high words separately */
    new ones: r1 = (a & mask) + (b & mask)
```

```
    new ones: r2 = (a >>> shift) + (b >>> shift)

    new ones: carry
    restart:            /* code label (goto target) */

    /* add carry of the new low word to the high word, then
     * strip it from the low word
     */
    carry = (r1 >>> shift)
    r2 += carry
    r1 &= mask

    /* add the carry from the new high word back to the low
     * word, then strip it from the high word
     */
    carry = (r2 >>> shift)
    r1 += carry
    r2 &= mask

    /* a carry from the high word injected back into the low
     * word may cause the new low to overflow, so restart in
     * that case
     */
    if (carry)
        goto restart

    return (r2 << shift) | r1
    }

ones: operator-(ones: a)
    return (a == ones: 0xffffffff) ? a : ~a

ones: operator-(ones: a, ones: b)
    return a + -b
```

The notable line in the example is the line "`chksum = chksum + value`" in the loop in function `main`. Since both the variables `chksum` and `value` have the tag `ones`, the `+` operator refers to the user defined operator (instead of the default `+` operator). User defined operators are merely a notational convenience. The same effect is achieved by calling functions explicitly.

The definition of an operator is similar to the definition of a function, with the difference that the name of the operator is composed by the keyword "`operator`" and the character of the operator itself. In the above example, both the unary `-` and the binary `-` operators are redefined. An operator function for a binary operator must have two arguments, one for an unary operator must have one argument. Note that the binary `-` operator adds the two values together after inverting the sign of the second operand. The subtraction operator thereby refers to both the user defined "negation" (unary `-`) and addition operators.

A redefined operator must adhere to the following restrictions:

- ◇ Only the following operators may be redefined: `+`, `-`, `*`, `/`, `%`, `++`, `--`, `==`, `!=`, `<`, `>`, `<=`, `>=`, and `!`. That is, the sets of arithmetic and relational operators can be overloaded, but the bitwise operators, the logical operators and the assignment operator cannot. The `!` operator is a special case.
- ◇ You cannot invent new operators; you cannot define operator `#` for example.
- ◇ The precedence level and associativity of the operators, as well as their "arity" remain as defined. You cannot make an unary `+` operator, for example.
- ◇ The return tag of the relational operators and of the `!` operator must be `bool`.
- ◇ The return tag of the arithmetic operators is at your choosing, but you cannot redefine an operator that is identical to another operator except for its return tag. For example, you cannot make both
  `alpha: operator+(alpha: a, alpha: b)`
  and
  `beta: operator+(alpha: a, alpha: b)`.
- ◇ Small already defines operators to work on untagged cells, you cannot redefine the operators with only arguments without tags.
- ◇ The arguments of the operator function must be a non-array passed by value. You cannot make an operator work on arrays.

In the example given above, both arguments of the binary operators have the same tag. This is not required; you may, for example, define a binary + operator that adds an integer value to a "`ones`" number.

Basically, the operation of the Small parser is to check the tag(s) of the operand(s) that the operator works on and to look up whether a user defined operator exists for the combination of the operator and the tag(s). However, the parser recognizes special situations and provides the following features:

- ◇ The parser recognizes operators like `+=` as a sequence of `+` and `=` and it will call a user defined operator `+` if available. In the example program, the line "`chksum = chksum + value`" might have been abbreviated to "`chksum += value`".
- ◇ The parser recognizes commutative operators (`+`, `*`, `==`, and `!=`) and it will swap the operands of a commutative operator if that produces a fit with a user defined operator. For example, there is usually no need to implement both
  "`ones:operator+(ones:a, b)`"
  and "`ones:operator+(a, ones:b)`".
- ◇ Prefix and postfix operators are handled automatically. You only need to define one user operator for the `++` and `--` operators for a tag.
- ◇ The parser calls the `!` operator implictly in case of a test without explicit comparison. For example, in the statement "`if (var) ...`" when "`var`" has tag "`ones`", the user defined operator `!` will be called for `var`. The `!` operator

thus doubles as a "test for zero" operator. (In one's complement arithmetic, both the "all-ones" and the "all-zeros" bit patterns represent zero.)

◇ If you wish to forbid an operation, you can "forward declare" the operator without ever defining it (see page 31). This will flag an error when the user defined operator is invoked. For example, to forbid the `%` operator (remainder after division) on floating point values, you can add the line:
`forward float: operator%(float: a, float: b)`

User defined operators can be declared "`stock`" and "`native`". In the case of a native operator function, the definition should include an external name. For example (when, on the host's side, the native function is called `float_add`):

Native functions: 33

```
native float: operator+(float: val, float: val) = float_add
```

• Floating point and fixed point arithmetic

Small only has intrinsic support for integer arithmetic (integer numbers are numbers without a fractional part). Support for floating point arithmetic or fixed point arithmetic must be implemented through (native) functions. User operators, then, allow a more natural notation of expressions with fixed or floating point numbers.

The Small parser has support for literal values with a fractional part, which it calls "rational numbers". Support for rational literals must be enabled explicitly with a `#pragma`. The `#pragma` indicates how the rational numbers must be stored —floating point or fixed point. For fixed point rational values, the `#pragma` alse specifies the precision in decimals. Two examples for the `#pragma` are:

Rational literals: 40
#pragma rational: 57

```
#pragma rational float      /* floating point format */

#pragma rational fixed(3)   /* fixed point, with 3 decimals */
```

Since a fixed point value must still fit in a cell, the number of decimals has a direct influence of the range of a fixed point value. For a fixed point value with 3 decimals, the range would be $-2,147,482\ldots +2,147,482$.

The format for a rational number may only be specified once for the entire Small program. In an implementation one typically chooses either floating point support or fixed point support. As stated above, for the actual implementation of the floating point or fixed point arithmetic, Small requires the help of (native) functions and user defined operators. A good place to put the `#pragma` for rational number support would be in the include file that also defines the functions and operators.

The include file for fixed point arithmetic contains definitions like:

```
native fixed: operator*(fixed: val1, fixed: val2) = fmul
native fixed: operator/(fixed: val1, fixed: val2) = fdiv
```

For adding two fixed point values together, the default + operator is sufficient, and the same goes for subtraction. Adding a normal (integer) number to a fixed point number is different: the normal value must be scaled before adding it. Hence, the include file implements operators for that purpose too:

```
stock fixed: operator+(fixed: val1, val2)
    return val1 + fixed(val2)

stock fixed: operator-(fixed: val1, val2)
    return val1 - fixed(val2)

stock fixed: operator+(val1, fixed: val2)
    return fixed(val1) - val2
```

The + operator is commutative, so one implementation handles both cases. For the - operator, both cases must be implemented separately.

Finally, the include file forbids the use of the modulus operator (%) on fixed point values: the modulus is only applicable to integer values:

```
forward fixed: operator%(fixed: val1, fixed: val2)
forward fixed: operator%(fixed: val1, val2)
forward fixed: operator%(val1, fixed: val2)
```

By declaring the operator, the Small parser will attempt to use the user defined operator rather than the default % operator. By not implementing the operator, the parser will issue an error if a program attempted to use the user defined operator.

General syntax

Format

Identifiers, numbers and tokens are separated by spaces, tabs, carriage returns and "form feeds". Series of one or more of these separators are called white space.

Optional semicolons

Semicolons (to end a statement) are optional if they occur at the end of a line. Semicolons are required to separate multiple statements on a single line. An expression may still wrap over multiple lines, however postfix operators (`++`, `--` and `char`) *must* appear on the same line as their operand.

Optional semicolons: pages 57 and 69

Comments

Text between the tokens `/*` and `*/` (both tokens may be at the same line or at different lines) and text behind `//` (up to the end of the line) is a programming comment. The parser treats a comment as white space. Comments may not be nested.

Identifiers

Names of variables, functions and constants. Identifiers consist of the characters `a...z`, `A...Z`, `0...9`, `_` or `@`; the first character may not be a digit. The characters `@` and `_` by themselves are not valid identifiers, i.e. "`_Up`" is a valid identifier, but "`_`" is not.

Small is case sensitive.

A parser may truncate an identifier after a maximum length. The number of significant characters is implementation defined, but should be at least 16 characters.

Reserved words (keywords)

Statements	Operators	Directives	Other
assert	char	#assert	const
break	defined	#else	enum
case	sizeof	#emit	forward
continue		#endif	native
default		#endinput	new
do		#endscript	operator
else		#if	public
exit		#include	static

for	stock
goto	
if	
return	
sleep	
switch	
while	

Predefined constants: 43

Next to reserved words, Small also has several predefined constants, you cannot use the symbol names of the predefined constants for variable or function names.

Constants (literals)

Integer numeric constants

binary
: `0b` followed by a series of the digits 0 and 1.

decimal
: a series of digits between 0 and 9.

hexadecimal
: `0x` followed by a series of digits between 0 and 9 and the letters `a` to `f`.

In all number radices, and underscore may be used to separate groups of (hexa-)decimal digits. Underscore characters between the digits are ignored.

Rational number constants

Rational numbers are also called "real numbers" or "floating point numbers"

A rational number is a number with a fractional part. A rational number starts with one or more digits, contains a decimal point and has at least one digit following the decimal point. For example, "12.0" and "0.75" are valid rational numbers. Optionally, an exponent may be appended to the rational number; the exponent notation is the letter `e` (lower case) followed by a signed integer numeric constant. For example, "3.12e4" is a valid rational number with an exponent.

#pragma rational: 57

Support for rational numbers must be enabled with `#pragma rational` directive. Depending on the options set with this directive, the rational number represents a floating point or a fixed point number.

Character constants

A single ASCII character surrounded by single quotes is a character constant (for example: `'a'`, `'7'`, `'$'`). Character constants are assumed to be numeric constants.

Control characters

`'^a'`		Audible alarm (beep)
`'^b'`		Backspace
`'^e'`		Escape
`'^f'`		Formfeed
`'^n'`		Newline
`'^r'`		Carriage Return
`'^t'`		Horizontal tab
`'^v'`		Vertical tab
`'^^'`	`^`	the caret itself
`'^''`	`'`	single quote
`'^"'`	`"`	double quote
`'^`*ddd*`;'`		character code with *decimal* code "*ddd*"

The semicolon after the `^`*ddd*`;` code is optional. Its purpose is to give the control character sequence an explicit termination symbol when it is used in a string constant.

The caret ("`^`") is the default control character. You can set a different control character with the `#pragma ctrlchar` directive (page 56).

String constants

String constants are assumed to be arrays with a size that is sufficient to hold all characters plus a terminating 0. Each string is stored at a unique position in memory; there is no elimination of duplicate strings.

An *unpacked* string is a series of zero or more ASCII characters surrounded by double quotes. Each array element contains a single character.

unpacked string constant:

```
"the quick brown fox..."
```

A *packed* string literal follows the syntax for an unpacked string, but a "`!`" precedes the first double quote.

packed string constant:

```
!"...packed and sacked the lazy dog"
```

The syntax for packed literal strings and unpacked literal strings can be swapped with the `#pragma pack` directive, see page 57.

In the case of a packed string, the parser packs as many characters in a cell as will fit. A character is not addressable as a single unit, instead each element of the array contains multiple characters. The first character in a "pack" occupies the highest bits of the array element. In environments that store memory words with the high byte at the lower address (Big Endian, or Motorola format), the individual characters are stored in the memory cells in the same order as they are in the string. A packed string ends with a zero character and the string is padded (with zero bytes) to a multiple of cells.

Control characters may be used within strings.

Array constants

A series of numeric constants between braces is an array constant. Array constants can be used to initialize array variables with (see page 20) and they can be passed as function arguments (see page 25).

Symbolic constants

A source file declares symbolic constants with the `const` and the `enum` instructions. The `const` keyword declares a single constant and the `enum` defines a list of —usually— sequential constants sharing the same tag name.

const *identifier* = *constant expression*

Creates a symbolic constant with the value of the constant expression on the right hand of the assignment operator. The constant can be used at any place where a literal number is valid (for example: in expressions, in array declarations and in directives like "`#if`" and "`#assert`").

enum *name* { *constant list* }

The `enum` instruction creates a series of constants with incrementing values. The *constant list* is a series of identifiers separated by commas. Unless overruled, the first constant of an `enum` list has

Identifiers: 39

the value 0 and every subsequent constant has the value of its predecessor plus 1.

Both the value of a constant and the increment value can be set by appending the value to the constant's identifier. To set a value, use *name* `=` *value*
in the constant list. To set the increment, use:
name `:` *increment*
The increment value is reset to 1 after every constant symbol declaration in the constant list.

If both an increment and a value should be set for a constant, the increment ("`:`" notation) should precede the value ("`=`" notation.

The *name* token that follows the `enum` keyword is optional. If it is included, this name is used as the tag name for every symbol in the constant list. In addition, the `enum` command creates an extra constant with *name* for the constant name and the tag name. The value of the last constant is the value of the last symbol in the constant list plus the increment value of that last constant.

See page 21 for examples of the "enum" constant declarations

The symbols in the constant list may not be tagged.

A symbolic constant that is defined locally, is valid throughout the block. A local symbolic constant may not have the same name as a variable (local or global), a function, or another constant (local or global).

Predefined constants

`false`	0 (this constant is tagged as `bool:`)
`true`	1 (this constant is tagged as `bool:`)
`cellbits`	The size of a cell in bits; usually 32.
`cellmax`	The largest valid positive value that a cell can hold; usually 2147483647.
`cellmin`	The largest valid negative value that a cell can hold; usually -2147483648.
`charbits`	The size of a character in bits; 8 when using the ASCII or ISO Latin-1 characters sets and 16 when using the Unicode character set.
`charmax`	The largest valid character value; 255 for 8-bit characters and 65535 for 16-bit characters.
`charmin`	The smallest valid character value, currently set at zero (0).

`debug`	One (1) if the compiler generates code for assertions and run-time bounds checking, zero (0) otherwise.
`_Small`	The version number of the Small compiler scaled by 100 (that is, for version 1.5 the constant is "150").

Tag names

Identifiers: 39

A tag consists of an identifier followed by a colon. There may be no white space between the identifier and the colon.

Predefined tag names

`bool`	For "true/false" flags. The predefined constants `true` and `false` have this tag.
`fixed`	Rational numbers typically have this tag when fixed point support is enabled (page 57).
`float`	Rational numbers typically have this tag when floating point support is enabled (page 57).

Operators and expressions

• Notational conventions

The operation of some operators depends on the specific kinds of operands. Therefore, operands are notated thus:

e any expression;
v any expression to which a value can be assigned ("lvalue" expressions);
a an array;
f a function.

• Expressions

An expression consists of one or more operands with an operator. The operand can be a variable, a constant or another expression. An expression followed by a semicolon is a statement.

Examples of expressions:

```
v++
f(a1, a2)
v = (ia1 * ia2) / ia3
```

• Arithmetic

`+` `e1 + e2`
Results in the addition of e1 and e2.

`-` `e1 - e2`
Results in the subtraction of e1 and e2.

`-e`
Results in the arithmetic negation of a (two's complement).

`*` `e1 * e2`
Results in the multiplication of e1 and e2.

`/` `e1 / e2`
Results in the division of e1 by e2. The result is truncated to the nearest integral value that is less than or equal to the quotient. Both negative and positive values are rounded towards $-\infty$.

`%` `e1 % e2`
Results in the modulus (remainder of the division) of e1 by e2. The modulus is always a positive value.

`++` `v++`
increments v by 1; results in the value of v before it is incremented.
`++v`
increments v by 1; results in the value of v after it is incremented.

`--` `v--`
decrements v by 1; results in the value of v before it is decremented.
`--v`
decrements v by 1; results in the value of v after it is decremented.

Notes: The unary + is not defined in Small.
The operators `++` and `--` modify the operand. The operand must be an *lvalue*.

• Bit manipulation

`~` `~e`
results in the one's complement of e.

`>>` `e1 >> e2`
results in the *arithmetic* shift to the right of e1 by e2 bits. The shift operation is signed: the leftmost bit of e1 is copied to vacant bits in the result.

`>>>` `e1 >>> e2`
results in the *logical* shift to the right of e1 by e2 bits. The shift operation is unsigned: the vacant bits of the result are filled with zeros.

`<<` `e1 << e2`
results in the value of e1 shifted to the left by e2 bits; the rightmost bits are set to zero. There is no distinction between an arithmetic and a logical left shift

`&` `e1 & e2`
results in the bitwise logical "and" of e1 and e2.

`|` `e1 | e2`

results in the bitwise logical "or" of e1 and e2.

`^` `e1 ^ e2`
results in the bitwise "exclusive or" of e1 and e2.

• Assignment

The result of an assignment expression is the value of the left operand after the assignment. The left operand may not be tagged.

Tag names: 21

`=` `v = e`
assigns the value of e to variable v.

If "`v`" is an array, it must have an explicit size and "`e`" must be an array of the same size; "`e`" may be a string or a literal array.

Note: the following operators combine an assignment with an arithmetic or a bitwise operation; the result of the expression is the value of the left operand after the arithmetic or bitwise operation.

`+=` `v += e`
increments v with a.

`-=` `v -= e`
decrements v with e

`*=` `v *= e`
multiplies v with e

`/=` `v /= e`
divides v by e.

`%=` `v %= e`
assigns the remainder of the division of v by e to v.

`>>=` `v >>= e`
shifts v arithmetically to the right by e bits.

`>>>=` `v >>>= e`
shifts v logically to the right by e bits.

`<<=` `v <<= e`
shifts v to the left by e bits.

`&=` `v &= e`
applies a bitwise "and" to v and e and assigns the result to v.

`|=` `v |= e`
applies a bitwise "or" to v and e and assigns the result to v.

`^=` `v ^= e`
applies a bitwise "exclusive or" to v and e and assigns the result to v.

• Relational

A logical "false" is represented by an integer value of 0; a logical "true" is represented by any value other than 0. Value results of relational expressions are either 0 or 1, and their tag is set to "bool".

`==` `e1 == e2`
results in a logical "true" if e1 is equal to e2.

`!=` `e1 != e2`
results in a logical "true" if e1 differs from e2.

Note: the following operators may be "chained", as in the expression "`e1 <= e2 <= e3`", with the semantics that the result is "1" if *all* individual comparisons hold and "0" otherwise.

`<` `e1 < e2`
results in a logical "true" if e1 is smaller than e2.

`<=` `e1 <= e2`
results in a logical "true" if e1 is smaller than or equal to e2.

`>` `e1 > e2`
results in a logical "true" if e1 is greater than e2.

`>=` `e1 >= e2`
results in a logical "true" if e1 is greater than or equal to e2.

• Boolean

A logical "false" is represented by an integer value of 0; a logical "true" is represented by any value other than 0. Value results of Boolean expressions are either 0 or 1, and their tag is set to "bool".

`!` `!e`
results to a logical "true" if e was logically "false".

`||` `e1 || e2`
results to a logical "true" if either e1 or e2 (or both) are logically "true". The expression e2 is only evaluated if e1 is logically "false".

`&&` `e1 && e2`
results to a logical "true" if both e1 and e2 are logically "true". The expression e2 is only evaluated if e1 is logically "true".

• **Miscellaneous**

`[]` `a[e]`
array index: results to *cell* e from array a.

`{}` `a{e}`
array index: results to *character* e from "packed" array a.

`()` `f(e1,e2,...eN)`
results to the value returned by the function f. The function is called with the arguments e1, e2, ...eN. The order of evaluation of the arguments is undefined (an implementation may choose to evaluate function arguments in reversed order).

`? :` `e1 ? e2 : e3`
results in either e2 or e3, depending on the value of e1. The conditional expression is a compound expression with a two part operator, `?` and `:`. Expression e2 is evaluated if e1 is logically "true", e3 is evaluated if e1 is logically "false".

`:` *tagname*`: e`
tag override; the value of the expression e does not change, but its tag changes. See page 21 for more information.

`,` `e1, e2`
results in e2, e1 is evaluated before e2. If used in function argument lists or a conditional expression, the comma expression must be surrounded by parentheses.

`defined`
returns the value 1 if the symbol is defined. The symbol may be a constant (page 40), or a global or local variable.

`sizeof`
: returns the size in cells of the specified variable.

`char e char`
: returns the number of cells that are needed to hold a packed array of e characters.

• Operator precedence

The table beneath groups operators with equal precedence, starting with the operator group with the highest precedence at the top of the table.

If the expression evaluation order is not explicitly established by parentheses, it is determined by the association rules. For example: `a*b/c` is equivalent with `(a*b)/c` because of the left-to-right association, and `a=b=c` is equivalent with `a=(b=c)`.

`()`	function call	left-to-right
`[]`	array index (cell)	
`{}`	array index (character)	
`!`	logical not	right-to-left
`~`	one's complement	
`-`	two's complement (unary minus)	
`++`	increment	
`--`	decrement	
`:`	tag override	
`char`	convert number of packed characters to cells	
`defined`	symbol definition status	
`sizeof`	symbol size in cells	
`*`	multiplication	left-to-right
`/`	division	
`%`	modulus	
`+`	addition	left-to-right
`-`	subtraction	
`>>`	arithmetic shift right	left-to-right
`>>>`	logical shift right	
`<<`	shift left	
`&`	bitwise and	left-to-right
`^`	bitwise exclusive or	left-to-right
`|`	bitwise or	left-to-right
`<`	smaller than	left-to-right
`<=`	smaller than or equal to	
`>`	greater than	
`>=`	greater than or equal to	
`==`	equality	left-to-right
`!=`	inequality	
`&&`	logical and	left-to-right
`||`	logical or	left-to-right
`? :`	conditional	right-to-left
`=`	assignment	right-to-left
`*= /= %= += -= >>= >>>= <<= &= ^= |=`		
`,`	comma	left-to-right

Statements

A statement may take one or more lines, whereas one line may contain two or more statements.

Control flow statements (`if`, `if`–`else`, `for`, `while`, `do`–`while` and `switch`) may be nested.

Statement label

Identifiers: 39

A label consists of an identifier followed by a colon (`:`). A label is a "jump target" of the `goto` statement.

Each statement may be preceded by a label. There must be a statement after the label; an empty statement is allowed.

The scope of a label is the function in which it is declared (a `goto` statement cannot therefore jump out off the current function to another function).

Compound statement

A compound statement is a series of zero or more statements surrounded by braces (`{` and `}`). The final brace (`}`) should not be followed by a semicolon. Any statement may be replaced by a compound statement. A compound statement is also called a block. A compound statement with zero statements is a special case, and it is called an "empty statement".

Expression statement

Any expression becomes a statement when a semicolon (`;`) is appended to it. An expression also becomes a statement when only white space follows it on the line and the expression cannot be extended over the next line.

Empty statement

An empty statement performs no operation and consists of a compound block with zero statements; that is, it consists of the tokens "`{ }`". Empty statements are used in control flow statements if there is no action (e.g. `while (!iskey()) {}`) or when defining a label just before the closing brace of a compound statement. An empty statement does not end with a semicolon.

assert *expression*

Example: page 8

Aborts the program with a run-time error if the expression evaluates to logically "false".

break

Terminates and exits the smallest enclosing `do`, `for` or `while` statement from any point within the loop other than the logical end. The `break` statement moves program control to the next statement outside the loop.

Example: page 13

continue

Terminates the current iteration of the smallest enclosing `do`, `for` or `while` statement and moves program control to the condition part of the loop. If the looping statement is a `for` statement, control moves to the third expression in the `for` statement (and thereafter to the second expression).

do *statement* **while (** *expression* **)**

Executes a statement before the condition part (the `while` clause) is evaluated. The statement is repeated while the condition is logically "true". The statement is at least executed once.

exit *expression*

Abort the program. The expression is optional. If included, the `exit` instruction returns the expression value (plus the expression tag) to the host application. The significance and purpose of exit codes is implementation defined.

for (*expression 1* **;** *expression 2* **;** *expression 3* **)** *statement*

All three expressions are optional.

Example: page 6

Variable declarations: 18

expression 1 Evaluated only once, and before entering the loop. This expression may be used to initialize a variable. This expression may also hold a variable declaration, using the `new` syntax. A variable declared in this expression exists only in the `for` loop.

expression 2 Evaluated before each iteration of the loop and ends the loop if the expression results to logically "false". If omitted, the result of expression 2 is assumed to be logically "true".

expression 3 Evaluated after each execution of the statement. Program control moves from expression 3 to expression 2 for the next (conditional) iteration of the loop.

The statement `for( ; ; )` is equivalent with `while (true)`.

goto *label*

Moves program control (unconditionally) to the statement that follows

the specified label. The label must be within the same function as the goto statement (a goto statement cannot jump out of a function).

if (*expression*) *statement 1* **else** *statement 2*

Example: page 5

Executes statement 1 if the expression results to logically "true". The `else` clause of the `if` statement is optional. If the expression results to logically "false" and an `else` clause exists, the statement associated with the `else` clause (statement 2) executes.

When `if` statements are nested and `else` clauses are present, a given `else` is associated with the closest preceding `if` statement in the same block.

return *expression*

Example: page 8

Terminates the current function and moves program control to the statement following the calling statement.

The expression is optional, but if it is included the value of the expression is returned as the function result.

sleep *expression*

Abort the program, but leave it in a re-startable state. The expression is optional. If included, the `sleep` instruction returns the expression value (plus the expression tag) to the host application. The significance and purpose of exit codes/tags is implementation defined; typically, an application uses the `sleep` instruction to allow for light-weight multitasking of several concurrent Small programs, or to implement "latent" functions.

switch (*expression*) { *case list* }

Transfers control to different statements within the switch body depending on the value of the switch expression. The body of the `switch` statement is a compound statement, which contains a series of "case clauses".

Each "case clause" starts with the keyword `case` followed by a constant list and *one* statement. The constant list is a series of expressions, separated by comma's, that each evaluates to a constant value. The constant list ends with a colon. To specify a "range" in the constant list, separate the lower and upper bounds of the range with a double period ("`..`"). An example of a range is: "`case 1..9:`".

The `switch` statement moves control to a "case clause" if the value of one of the expressions in the constant list is equal to the `switch` expression result.

The "default clause" consists of the keyword `default` and a colon. The default clause is optional, but if it is included, it must be the last clause in the switch body. The `switch` statement moves control to the "default clause" is executed if none of the case clauses match the expression result.

Example:

```
switch (weekday(12,31,1999))
    {
    case 1, 7:          /* 1 == Sunday, 7 == Saturday */
        print("weekend")
    case 2:
        print("Monday")
    case 3:
        print("Tuesday")
    case 4:
        print("Wednesday")
    case 5:
        print("Thursday")
    case 6:
        print("Friday")
    default:
        print("invalid week day")
    }
```

while (*expression*) *statement*

Evaluates the expression and executes the statement if the expression result yields logically "true". After the statement has executed, program control returns to the expression again. The statement is thus executed while the expression is true.

Example: page 5

Directives

All directives must appear first on a line (they may be preceded by white space, but not by any other characters). All directives start with the character `#` and the complete instruction may not span more than one line.

#assert *constant expression*

Issues a compile time error if the supplied constant expression evaluates to zero. The `#assert` directive is most useful to guard against implementation defined constructs on which a program may depend, such as the cell size in bits, or the number of packed characters per cell.

See also "Predefined constants" on page 43

#emit *opcode, parameters*

The `#emit` directive serves as an inline assembler. It is currently used only for testing the abstract machine.

#endinput

Closes the current file and thereby ignores all the text below the `#endinput` directive.

#include "*filename*" **or** <*filename*>

Inserts the contents of the specified file at the current position within the current file. A filename between double quotes refers to a local file, and filename between angle brackets refers to a system file. A Small parser (compiler or interpreter) may treat system files in a special way; for example, a system file need not be a physical file for a Small parser that "magically" knows the contents of each system file.

The proposed default extension of include files is "`.INC`".

#if *constant expression*, **#else, #endif**

Portions of a program may be parsed or be ignored depending on certain conditions. The Small parser (compiler or interpreter) generates code only for those portions for which the condition is true.

The directive `#if` must be followed by a constant expression. To check whether a variable or constant is defined, use the `defined` operator.

The `#else` directive reverses the parsing state. If the parser ignored lines up to the directive, it starts parsing and if it parsed lines, it stops parsing. There should only be one `#else` associated with each `#if`, but a Small parser need not impose this restriction.

The `#endif` directive terminates a program portion that is parsed conditionally. Conditional directives can be nested and each `#if` directive must be ended by an `#endif` directive.

#pragma *extra information*

A pragma is a hook for a parser to specify additional settings, such as warning levels or extra capabilities. Common pragmas are:

`#pragma ctrlchar` *character*

Control character: 41

Defines the character to use to indicate a "control character". By default, the control character is "`^`".

For example

```
#pragma ctrlchar '$'
```

`#pragma dynamic` *value*

Sets the size, in cells, of the memory block for dynamic data (the

stack and the heap) to the value specified by the expression. The default size of the dynamic data block is implementation defined. An implementation may also choose to grow the block on an as-needed basis. See appendix C for details.

`#pragma library` *name*
: Sets the name of the (dynamically linked) library or module that contains required native functions.

`#pragma pack` *value*
: If `value` is zero, packed literal strings start with "`!"`" and unpacked literal strings with only a double quote ("`"`"), as described in this manual at page 41. If `value` is non-zero, the syntax for packed and unpacked literal strings is swapped: literal strings that start with a double quote are packed and literal strings that start with "`!"`" are unpacked.

`#pragma rational` *tagname*`(`*value*`)`
: Enables support for rational numbers (see page 40). The `tagname` is the name pf the tag that rational numbers will have; typically one chooses the names "float" or "fixed". The value in parantheses behind `tagname` is optional: if it is omitted, a rational number is stored as a "floating point" value according to the IEEE 754 norm; if it is present, a rational number is a fixed precision number ("scaled integer") with the specified number of decimals.

`#pragma semicolon` *value*
: If `value` is zero, no semicolon is required to end a statement if that statement is last on a line. Semicolons are still needed to separate multiple statements on the same line.

  When semicolons are optional (the default), a postfix operator (one of "`++`", "`--`" and "`char`") may not be the first token on a line.

`#pragma tabsize` *value*
: The number of characters between two consecutive TAB positions. The default value is 8. You may need to set the TAB size to avoid warning 217 (loose indentation) if the source code is indented alternately with spaces and with TAB characters. Alternatively, by setting the "`tabsize`" pragma to zero, the parser will no longer issue warning 217.

Proposed function library

Since Small is targeted as an application extension language, most of the functions that are accessible to Small programs will be specific to the host application. Nevertheless, a small set of functions may prove useful to many environments.

• Core functions

The "core" module consists of a set of functions that support the language itself. Several of the functions are needed to pull arguments out of a variable argument list (see page 30).

Since there are only few functions, I have opted to arrange them per category, rather than alphabetically.

heapspace()
: Return the free space on the heap. The stack and the heap occupy a shared memory area.

funcidx(const name[])
: Returns the index of the named public function. A host application runs a public function from the script by passing the public function's index to `amx_Exec()`. With this function, the script can query the index of a public function, and thereby return the "next function to call" to the application.

  If no public function with the given name exists, `funcidx` returns −1.

amx_Exec: 90

❖

numargs()
: Return the number of arguments passed to a function; `numargs()` is useful inside functions with a variable argument list.

getarg(arg, index=0)
: Retrieve an argument from a variable argument list. When the argument is an array, the `index` parameter specifies the index into the array. The return value is the retrieved argument.

setarg(arg, index=0, value)
: Set the value of an argument from a variable argument list. When the argument is an array, the `index` parameter specifies the index into the array. The return value is `false` if the argument or the index are invalid, and `true` on success.

❖

strlen(const string[])
Returns the length of a string, either packed or unpacked, as the number of characters (not the number of cells).

strpack(dest[], const source[])
Copy a string from `source` to `dest` where the destination string will be in packed format. The source string may either be a packed or an unpacked string.

strunpack(dest[], const source[])
Copy a string from `source` to `dest` where the destination string will be in unpacked format. The source string may either be a packed or an unpacked string.

tolower(c)
Returns the character code of the lower case letter of "`c`" if there is one, or the character code of "`c`" if the letter "`c`" has no lower case equivalent.

toupper(c)
Returns the character code of the upper case letter of "`c`" if there is one, or the character code of "`c`" if "`c`" has no upper case equivalent.

❖

swapchars(c)
Returns the value of `c` where all bytes in the cell are swapped (the lowest byte becomes the highest byte).

random(max)
Returns a pseudo-random number in the range `0` – `max-1`.

max(value1, value2)
Returns the higher value of `value1` and `value2`.

min(value1, value2)
Returns the lower value of `value1` and `value2`.

clamp(value, min=cellmin, max=cellmax)
Returns `value` if it is in the range `min` – `max`; returns `min` if `value` is lower than `min`; returns `max` if `value` is higher than `max`.

❖

Properties are general purpose names or values. The property list routines maintain a list of these name/value pairs that is shared among all abstract machines. The property list is therefore a way for concurrent abstract machines to exchange information.

All "property maintenance" functions have an optional "id" parameter. You can use this parameter to indicate which abstract machine the property belongs to. (A host application that supports concurrent abstract machines will usually provide each abstract machine with a unique id.) When querying (or deleting) a property, the id value that you pass in is matched to the id values of the list.

A property is identified with its "abstract machine id" plus *either* a name *or* a value. The name-based interface allows you to attach a value (e.g. the handle of an object) to a name of your choosing. The value-based interface allows you to attach a string to a number. The difference between the two is basically the search key versus the output parameter.

All property maintenance functions have a "name" and a "value" parameter. Only one of this pair must be filled in. When you give the value, the `getproperty` function stores the result in the `string` argument and the `setproperty` function reads the string to store from the `string` argument.

The number of properties that you can add is limited only by available memory.

getproperty(id=0, const name[]="", value=cellmin, string[]="")
Returns the value of a property when the `name` is passed in; fills in the `string` argument when the `value` is passed in. The `name` string may either be a packed or an unpacked string. If the property does not exist, this function returns zero.

setproperty(id=0, const name[]="", value=cellmin, const string[]="")
Add a new property or change an existing property.

deleteproperty(id=0, const name[]="", value=cellmin)
Returns the value of the property and subsequently removes it. If the property does not exist, the function returns zero.

existproperty(id=0, const name[]="", value=cellmin)
Returns `true` if the property exists and `false` otherwise.

• Console functions

For testing purposes, the console functions that read user input and that output strings in a scrollable window or on a standard terminal display are often convenient.

getchar(echo=true)
: Read one character from the keyboard and return it. The function can optionally echo the character on the console window.

getstring(string[], maxlength, bool
: pack=false): Read a string from the keyboard. Function `getstring` stops reading when either the enter key is typed, or the maximum length is reached. The maximum length is in characters (not cells). The function can read both packed and unpacked strings. The return value is the number of characters read.

getvalue(base=10, end='^r', ...)
: Read a value (a signed number) from the keyboard. The `getvalue` function allows you to read in a numeric radix from 2 to 36 (the `base` parameter) with decimal radix by default.

  By default the input ends when the user types the enter key, but one or more different keys may be selected (the `end` parameter and subsequent). In the list of terminating keys, a positive number (like `'^r'`) displays the key and terminates input, and a negative number terminates input without displaying the terminating key.

print(const str[], foreground=-1, background=-1)
: Prints a simple string on the console. The foreground and background colours may be optionally set. See `CONSOLE.INC` for a list of colours.

printf(const format[], ...)
: Prints a string with embedded codes:
  %c print a character at this position
  %d print a decimal number at this position
  %f print a floating point number at this position
  %r print a fixed point number at this position
  %s print a character string at this position

  The `printf` function works similarly to the `printf` function of the C language.

• Fixed point arithmetic

Small does not direcly support fixed point arithmetic. Support for fixed point artihmetic is built on a set of native functions and user defined operators. A fixed point number in Small is a 32-bit number with 3 decimals and a range of $-2,147,482$ to $+2,147,482$.

To convert from integers to cells, use one of the functions `fixed` or `fixedstr`. The function `fixed` creates a fixed point number with the same integral value as the input value and a fractional part of zero. Function `fixedstr` makes a fixed point number from a string, which can include a fractional part.

To convert back from fixed point numbers to plain cells, use the functions `fround` and `ffract`. Function `fround` is able to round upwards, to round downwards (truncation) and to round to the nearest integer. Function `ffract` gives the fractional part of a fixed point number, but still stores this as a fixed point number.

Adding and subtracting operations on fixed point values can use the conventional + and − operators. For multiplication and division, one must use the `fmul` and `fdiv` functions.

fixed:fixed(value)
: Create a fixed point number with the same (integral) value as the parameter `value`.

fixed:fixedstr(const string[])
: Create a fixed point number from a string. The string may specify a fractional part, e.g., "`123.45`".

fixed:fmul(fixed:oper1, fixed:oper2)
: Multiply two fixed point numbers.

fixed:fdiv(fixed:dividend, fixed:divisor)
: Fixed point division.

fixed:ffract(fixed:value)
: Returns the fractional part if `value`.

fround(fixed:value, fround_method:method=fround_round)
: Round a fixed point number and return the value as a cell. The rounding method may be one of:

  `fround_round` round to the nearest integer (default)

  `fround_floor` round downwards (truncate)

`fround_ceil` round upwards

When rounding negative values upwards or downwards, note that -2 is considered smaller than -1.

Pitfalls: differences from C

◇ Small lacks the typing mechanism of C. Small is an "integer-only" variety of C; there are no structures or unions, and floating point support must be done user defined operators and the help of native functions.

◇ Small does not provide "pointers". For the purpose of passing function arguments by reference, Small provides a "reference" argument, (page 25). The "placeholder" argument replaces some uses of the NULL pointer (page 27).

◇ The *final* dimension of a multi-dimensional array may have an unspecified length. In C, the *first* dimension may have an unspecified length. See page 20.

◇ Escape sequences ("`\n`", "`\t`", etc.) are replaced by control characters. The main difference is that the caret ("`^`") replaces the backslash ("\"). See "Character constants" on page 41; see also `#pragma ctrlchar` on page 56.

◇ Cases in a `switch` statement are *not* "fall through". Only a single instruction may (and must) follow each `case` label. To execute multiple instructions, you must use a compound statement. The `default` clause of a `switch` statement must be the last clause of the `switch` statement. More on page 54.

◇ A `break` statement breaks out of loops only. In C/C++, the `break` statement also ends a `case` in a `switch` statement. Switch statements are implemented differently in Small (see page 54).

◇ Numbers can have hexadecimal, decimal or binary radix. Octal radix is not supported. See "Constants" on page 40. Hexadecimal numbers must start with "`0x`" (a lower case "x"), the prefix "`0X`" is invalid.

◇ The accepted syntax for rational numbers is quite a bit stricter than that of floating point values in C. Values like ".5" and "6." are acceptable in C, but in Small one must write "0.5" and "6.0" respectively. In C, the decimal period is optional if an exponent is included, so one can write "2E8"; Small does not accept the upper case "E" (use a lower case "e") and it requires the decimal point: e.g. "2.0e8". See page 40 for more information.

◇ The "extern" keyword does not exist in Small; the current implementation of the compiler has no "linking phase".

◇ The compiler directives differ from C's preprocessor commands. Notably, the `#define` directive can only add numeric constants, and `#ifdef` and `#ifndef` are replaced by the more general `#if` directive (see "Directives" on page 55). To create numeric constants, see also page 42.

◇ `char` is an operator, not a type. See page 50 and the tips on page 65.

◇ The empty instruction is an empty compound block, not a semicolon (page 52). This modification avoids a frequent error.

◇ `defined` is an operator, not a preprocessor directive. The `defined` operator in Small operates on constants (with `const` and `enum`), global variables, local variables and functions.

◇ The `sizeof` operator returns the size of a variable in "cells" (not in "bytes").

◇ The direction for truncation for the operator `/` is always towards the smaller value, where -2 is smaller than -1. The `%` operator always gives a positive result, regardless of the signs of the operands. See page 45.

◇ There is no unary `+` operator, which is a "no-operation" operator anyway.

◇ Three of the bitwise operators have different precedence than in C. The precedence levels of the `&`, `^` and `|` operators is higher than the relational operators (Dennis Ritchie explained that these operators got their low precedence levels in C because early C compilers did not yet have the logical `&&` and `||` operators, so the bitwise `&` and `|` were used instead).

◇ Small supports "array assignment", with the restriction that both arrays must have the same size. For example, if "`a`" and "`b`" are both arrays with 6 cells, the expression "`a = b`" is valid. Next to literal strings, Small also supports literal arrays, allowing the expression "`a = {0,1,2,3,4,5}`" (where "`a`" is an array variable with 6 elements).

◇ Forward declarations (prototypes) —if provided— must match *exactly* with the function definition, parameter names may not be omitted from the prototype or differ from the function definition.

Assorted tips

• Working with characters and strings

Strings can be in packed or in unpacked format. In the packed format, each cell will typically hold four characters (in the current implementations, a cell is 32-bit and a character is often 8 bit). In this configuration, the first character in a "pack" of four is the highest byte of a cell and the fourth character is in the lowest byte of each cell.

A string must be stored in an array. For an unpacked string, the array must be large enough to hold all characters in the string plus a terminating zero cell. That is, in the example below, the variable `ustring` is defined as having five cells, which is just enough to contain the string with which it is initialized:

```
new ustring[5] = "test"
```

In a packed string, each cell contains several characters and the string ends with a zero character. The `char` operator helps with declaring the array size to contain the required number of *characters*. The example below will allocate enough cells to hold five packed characters. In a typical implementation, there will be two cells in the array.

```
new pstring[5 char] = !"test"
```

In other words, the `char` operators divides its left operand by the number of bytes that fit in a cell and rounds upwards. Again, in a typical implementation, this means dividing by four and rounding upwards.

You can design routines that work on strings in both packed and unpacked formats. To find out whether a string is packed or unpacked, look at the first cell of a string. If its value is higher than the maximum possible value of a character (higher than 255 for 8 bit characters), the string is a packed string. Otherwise it is an unpacked string.

See also page 58 for proposed core functions that operate on both packed and unpacked strings

The code snippet below returns `true` if the input string is packed and `false` otherwise (also note the use of tags):

```
bool: ispacked(string[])
    return bool: (string[0] > charmax)
```

An unpacked string ends with a full zero cell. The end of a packed string is marked with only a zero character. Since there may be up to four characters in a cell, this zero character may occur at any of the four positions in the "pack". The `{ }` operator extracts a character from a cell in an array. Basically, one uses the cell index operator ("`[ ]`") for unpacked strings and the character index operator ("`{ }`") to work on packed strings.

For example, a routine that returns the length in characters of any string (packed or unpacked) is:

```
my_strlen(string[])
    {
    new len = 0
    if (ispacked(string))
        while (string{len} != '^0')     /* get character from pack */
            ++len
```

```
    else
        while (string[len] != '^0')     /* get cell */
            ++len
    return len
    }
```

If you make functions to work exclusively on either packed or unpacked strings, it is a good idea to add an assertion to enforce this condition:

```
strupper(string[])
    {
    assert ispacked(string)

    for (new i=0; string{i} != '^0'; ++i)
        string{i} = toupper(string{i})
    }
```

Although, in preceding paragraphs we have assumed that a cell is 32 bits wide and a character is 8 bits, this cannot be relied upon. The size of a cell is implementation defined; the maximum and minimum values are in the predefined constants `cellmax` and `cellmin`. There are similar predefined constants for characters. One may safely assume, however, that both the size of a character in bytes and the size of a cell in bytes are powers of two.

Predefined constants: 43

The `char` operator allows you to determine how many packed characters fit in a cell. For example:

```
#if     4 char == 1
        /* code that assumes 4 packed characters per cell */
#else
  #if   4 char == 2
        /* code that assumes 2 packed characters per cell */
  #else
    #if 4 char == 4
        /* code that assumes 1 packed character per cell */
    #else
        #assert 0 /* unsupported cell/character size */
    #endif
  #endif
#endif
```

• Concatenating lines

Small is a free format language, but the parser directives must be on a single line. Strings may not run over several lines either. When this is inconvenient, you can use a backslash character ("\") at the end of a line to "glue" that line with the next line.

Directives: 55

For example:

```
#define max_path        max_drivename + max_directorystring + \
                        max_filename + max_extension
```

You also use the concatenation character to cut long literal strings over multiple lines. Note that the "\" eats up all trailing white space that comes after it and leading white space on the next line. The example below prints "Hello world" with one space between the two words (because there is a space between "Hello" and the backslash):

```
print("Hello \
      world")
```

Small: the compiler

The Small compiler is currently the only translator (or parser) that implements the Small language. The Small compiler translates a text file with source code to a binary file for an abstract machine. The output file format is in appendix C.

• Usage

Assuming that the Small compiler is called "`sc`" or "`sc.exe`", the command line syntax is:

```
sc <options> [filename]
```

The input file name is any legal filename. If no extension is given, "`.SMA`" is assumed. The compiler creates an output file with, by default, the same name as the input file and the extension "`.AMX`".

After switching to the directory with the sample programs, the command:

```
sc hello
```

should compile the very first "hello world" example (on page 4). *Should*, because the command implies that:

- ◇ the operating system can locate the "`sc`" program —you may need to add it to the search path;
- ◇ the Small compiler is able to determine its own location in the file system so that it can locate the include files —a few operating systems do not support this and require that you use the `-i` option (see below).

• Options

The options are:

`-a` "assembler", generate a text file with the pseudo-assembler code for the Small abstract machine, instead of binary code;

`-C` "compact encoding" of the binary file, which reduces the size a the output file typically to less than half the original size;

`-c`*size* set the character size, *size* must be 8 (for ASCII & ISO Latin-1) or 16 for Unicode;

`-D`*path* the "active" directory, where the compiler should search for its input files and store its output files (this option is not supported on every platform);

`-d`*level* debug level: 0 = none, 1 = bounds checking and assertions only, 2 = full symbolic information, 3 = full symbolic information and optimizations disabled;

`-e`*name* set the name of the error file (when set, there is no output to the screen);

`-i`*name* set the path to the include files;

`-o`*name* set the output filename and path;

Packed directive: 57

`-P` set "packed strings" by default; use the `!"..."` syntax for unpacked strings;

`-p`*name* the filename of the "prefix file", this is a file that is parsed before the input file (as a kind of implicit "include file");

`-S`*value* the size of the stack and the heap in cells;

`-s`*value* the number of lines to skip in the input file before starting to compile;

`-t`*value* the "size" of a TAB character, when set to zero (i.e. option `-t0`) the compiler will no longer issue warning 217 (loose indentation);

`-\` control characters start with "\" instead of "`^`" (for the sake of similarity with C);

`-;` semicolons are optional to end a statement if the statement is the last on the line (this is the default);

`-;+` semicolons are required, every statement must be terminated with a semicolon;

sym`=`*val* define constant "*sym*" with the given (numeric) *value*, the *value* is optional;

`@`*filename* read (more) options from the specified "response file".

All options should be separated by at least one space.

Error and warning messages

When the compiler finds an error in a file, it outputs a message giving, in this order:

◇ the name of the file

◇ the line number were the compiler detected the error between parentheses, directly behind the filename

◇ the error class ("Error", "Fatal" or "Warning")

◇ an error number between square brackets

◇ a descriptive error message

For example:

```
demo.c(3): Error [001]: expected token: ";", but found "{"
```

If the "verbose" option is active, the erroneous line is displayed too.

Note: the line number given by the compiler may specify a position behind the actual error, since the compiler cannot always establish an error before having analyzed the complete expression.

After termination, the return code of the compiler is:

0 no errors
1 errors found
2 warnings found
3 aborted by user

These return codes may be checked within batch processors (such as the "make" utility).

• Error categories

Errors are separated into three classes:

Errors	Describe situations where the compiler is unable to generate appropriate code. Errors messages are numbered from 1 to 99.
Fatal errors	Fatal errors describe errors from which the compiler cannot recover. Parsing is aborted. Fatal error messages are numbered from 100 to 199.
Warnings	Warnings are displayed for unintended compiler assumptions and common mistakes. Warning messages are numbered from 200 to 299.

• Errors

001 **expected token:** *token*, **but found** *token*
A required token is omitted.

002 **only a single statement (or expression) can follow each "case"**
Every case in a switch statement can hold exactly one statement. To put multiple statements in a case, enclose these statements between braces (which creates a combound statement).

Pitfalls: 64
Compound statement: 52

003 *reserved*
Reserved (unused) error message.

Functions: 24

004 **function** *name* **not defined**
Functions must be defined or prototyped before the first statement.

005 **function may not have arguments**
The function `main()` is the program entry point. It may not have arguments.

006 **must be assigned to an array**
String literals or arrays must be assigned to an array. This error message may also indicate a missing index (or indices) at the array on the right side of the "=" sign.

#assert directive: 55

007 **assertion failed**
Compile-time assertion failed.

008 **must be a constant expression; assumed zero**
The size of arrays and the parameters of most directives must be constant values.

009 **invalid array size (negative or zero)**
The number of elements of an array must always be 1 or more.

010 **illegal function or declaration**
The compiler expects a declaration of a global variable or of a function at the current location, but it cannot interpret it as such.

011 **invalid outside functions**
The instruction or statement is invalid at a global level. Local labels and (compound) statements are only valid if used within functions.

012 **invalid function call, not a valid address**
The symbol is not a function.

013 **no entry point (no public functions)**
The file does not contain a `main` function or any public function. The compiled file thereby does not have a starting point for the execution.

014 **invalid statement; not in switch**
The statements `case` and `default` are only valid inside a `switch` statement.

015 **"default" must be the last clause in switch statement**
Small requires the `default` clause to be the last clause in a `switch` statement.

016 **multiple defaults in "switch"**
Each `switch` statement may only have one `default` clause.

017 **undefined symbol** *symbol*
The symbol (variable, constant or function) is not declared.

018 **initialization data exceeds declared size**
An array with a specified size is initialized, but the number of initiallers exceeds the number of elements specified (e.g. "`arr[3]={1,2,3,4};`" the array is specified to have three elements, but there are four initiallers).

Initialization: 20

019 **not a label:** *name*
A `goto` statement branches to a symbol that is not a label.

020 **invalid symbol name**
A symbol may start with a letter, an underscore or an "at" sign ("@") and may be followed by a series of letters, digits, underscore characters and "@" characters.

Symbol name syntax: 39

021 **symbol already defined:** *identifier*
The symbol was already defined at the current level.

022 **must be lvalue (non-constant)**
The symbol that is altered (incremented, decremented, assigned a value, etc.) must be a variable that can be modified (this kind of variable is called an lvalue). Functions, string literals, arrays and constants are no lvalues. Variables declared with the "`const`" attribute are no lvalues either.

023 **array assignment must be simple assignment**
When assigning one array to another, you cannot combine an arithmetic operation with the assignment (e.g., you cannot use the "+=" operator).

024 **"break" or "continue" is out of context**
The statements `break` and `continue` are only valid inside the context of a loop (a `do`, `for` or `while` statement). Unlike the languages C/C++ and Java, `break` does not jump out of a `switch` statement.

025 **function heading differs from prototype**
The number of arguments given at a previous declaration of the function does not match the number of arguments given at the current declaration.

026 **no matching "#if..."**
The directive `#else` or `#endif` was encountered, but no matching `#if` directive was found.

Control characters: 41

027 **invalid character constant**
Probably caused by an unknown control character, like "`^x`".

028 **cannot subscript, not an array**
The subscript operators "[" and "]" are only valid with arrays.

029 **invalid expression, assumed zero**
The compiler could not interpret the expression.

030 **compound statement not closed at the end of file**
An unexpected end of file occurred. One or more compound statements are still unfinished (i.e. the closing brace "}" has not been found).

031 **unknown directive**
The character "#" appears first at a line, but no valid directive was specified.

032 **array index out of bounds**
The array index is larger than the highest valid entry of the array.

033 **array must be indexed (variable** *name***)**
An array as a whole cannot be used in a expression; you must indicate an element of the array between square brackets.

034 **argument does not have a default value (argument** *index***)**
You can only use the argument placeholder when the function definition specifies a default value for the argument.

035 **argument type mismatch (argument** *index***)**
The argument that you pass is different from the argument that the function expects, and the compiler cannot convert the passed-in argument to the required type. For example, you cannot pass the literal value "1" as an argument when the function expects an array or a reference.

036 **empty statement**
The line contains a semicolon that is not preceded by an expression. Small does not support a semicolon as an empty statement, use an empty compound block instead.

Empty compound block: 52

037 **invalid string (possibly non-terminated string)**
A string was not well-formed; for example, the final quote that ends a string is missing, or the filename for the `#include` directive was not enclosed in double quotes or angle brackets.

038 **extra characters on line**
There were trailing characters on a line that contained a directive (a directive starts with a `#` symbol, see page 55).

039 **constant symbol has no size**
A variable has a size (measured in a number of cells), a constant has no size. That is, you cannot use a (symbolic) constant with the `sizeof` operator, for example.

040 **duplicate "case" label (value *value*)**
A preceding "case label" in the list of the `switch` statement evaluates to the same value.

041 **invalid ellipsis, array size is not known**
You used a syntax like "`arr[] = { 1, ... };`", which is invalid, because the compiler cannot deduce the size of the array from the declaration.

042 **invalid combination of class specifiers**
A function is denoted as both "public" and "native", which is unsupported.

043 **character constant exceeds range for packed string**
Usually an attempt to store a Unicode character in a packed string where a packed character is 8-bits.

044 **mixing named and positional parameters**
You must either use named parameters or positional parameters for all parameters of the function.

045 **too many function arguments**
The maximum number of function arguments is currently limited to 64.

046 **unknown array size (variable *name*)**
For array assignment, the size of both arrays must be explicitly defined, also if they are passed as function arguments.

047 **array sizes must match**
For array assignment, the arrays on the left and the right size of the assignment operator must have the same size.

048 **array dimensions must match**
For an array assignment, the dimensions of the arrays on both sides of the "=" sign must match; when passing arrays to a function argument, the arrays passed to the function (in the function call) must match with the definition of the function arguments.

Single line comment: 39

049 **invalid line continuation**
A line continuation character (a backslash at the end of a line) is at an invalid position, for example at the end of a file or in a single line comment.

050 **invalid range**
A numeric range with the syntax "*n1* `..` *n2*", where *n1* and *n2* are numeric constants, is invalid. Either one of the values in not a valid number, or *n1* is not smaller than *n2*.

051 **invalid subscript, use "[ ]" operators on major dimensions**
You can use the "array character index" operator (braces: "`{ }`" only for the last dimension. For other dimensions, you must use the cell index operator (square brackets: "`[ ]`").

052 **only the last dimension may be variable length**
Except the last dimension, all array dimensions must have an explicit size.

053 **exceeding maximum number of dimensions**
The current implementation of the Small compiler only supports arrays with one or two dimensions.

054 **unmatched closing brace**
A closing brace ("`}`") was found without matching opening brace ("`{`").

055 **start of function body without function header**
An opening brace ("`{`") was found outside the scope of a function. This may be caused by a semicolon at the end of a preceding function header.

056 **local variables and function arguments cannot be public**
A local variable or a function argument starts with the character "@", which is invalid.

057 **Unfinished expression before compiler directive**
Compiler directives may only occur *between* statements, not *inside* a statement. This error typically occurs when an expression statement is split over multiple lines and a compiler directive appears between the start and the end of the expression. This is not supported.

058 **duplicate argument; same argument is passed twice**
In the function call, the same argument appears twice, possibly through a mixture of named and positional parameters.

Named versus positional parameters: 26

059 **function argument may not have a default value (variable** *name***)**
All arguments of **public functions** must be passed explicitly. Public functions are typically called from the host application, who has no knowledge of the default parameter values. Arguments of **user defined operators** are implied from the expression and cannot be inferred from the default value of an argument.

060 **multiple "#else" directives between "#if ... #endif**
Two or more `#else` directives appear in the body between the matching `#if` and `#endif`.

061 **operator cannot be redefined**
Only a select set of operators may be redefined, this operator is not one of them.

062 **number of operands does not fit the operator**
When redefining an operator, the number of operands that the operator has (1 for unary operators and 2 for binary operators) must be equal to the number of arguments of the operator function.

063 **operator requires that the function result has a "bool" tag**
Logical and relational operators are defined as having a result that is either `true` (1) or `false` (0) and having a "bool" tag. A user defined operator should adhere to this definition.

064 **cannot change predefined operators**
One cannot define operators to work on untagged values, for example, because Small already defines this operation.

065 **function argument may only have a single tag (argument** *number*)
In a user defined operator, a function argument may not have multiple tags.

066 **function argument may not be a reference argument or an array (argument** *number*)
In a user defined operator, all arguments must be cells (non-arrays) that are passed "by value".

067 **variable cannot be both a reference and an array (variable** *name*)
A function argument may be denoted as a "reference" or as an array, but not as both.

068 **invalid rational number precision in #pragma**
The precision was negative or too high. For floating point rational numbers, the precision specification should be omitted.

069 **rational number format already defined**
This `#pragma` conflicts with an earlier `#pragma` that specified a different format.

#pragma rational: 57

070 **rational number support was not enabled**
A rational literal number was encountered, but the format for rational numbers was not specified.

• Fatal Errors

100 **cannot read from file:** *filename*
The compiler cannot find the specified file or does not have access to it.

101 **cannot write to file:** *filename*
The compiler cannot write to the specified output file, probably caused by insufficient disk space or restricted access rights (the file could be read-only, for example).

102 **table overflow:** *table name*
This is an internal error of the compiler, caused by the limited size of its internal tables. The "table name" is one of the following:

"staging buffer": the staging buffer holds the code generated for an expression before it is passed to the peephole optimizer. The staging

buffer grows dynamically, so an overflow of the staging buffer basically is an "out of memory" error.

"loop table": the loop table is a stack used with nested `do`, `for`, and `while` statements. The table allows nesting of these statements up to 24 levels.

"literal table": this table keeps the literal constants (numbers, strings) that are used in expressions and as initiallers for arrays. The literal table grows dynamically, so an overflow of the literal table basically is an "out of memory" error.

"compiler stack": the compiler uses a stack to store temporary information it needs while parsing. An overflow of this stack is probably caused by deeply nested (or recursive) file inclusion or complex expression involving function calls with many arguments.

"option table": in case that there are more options on the command line or in the response file than the compiler can cope with.

103 **insufficient memory**
General "out of memory" error.

104 **invalid assembler instruction** *symbol*
An invalid opcode in an `#emit` directive.

105 **numeric overflow, exceeding capacity**
A numeric constant, notably a dimension of an array, is too large for the compiler to handle. For example, when compiled as a 16-bit application, the compiler cannot handle arrays with more than 32767 elements.

• Warnings

200 **symbol is truncated to 19 characters**
The symbol is longer than sixteen characters. Truncation may cause different symbol names to become equal, which may cause error 021 or warning 219.

201 **redefinition of constant (symbol** *name***)**
The symbol was previously defined to a different value.

202 **number of arguments does not match definition**
At a function call, the number of arguments passed to the function (actual arguments) differs from the number of formal arguments declared in the function heading. To declare functions with variable argument lists, use an ellipsis (`...`) behind the last known argument in the function heading; for example: `print(formatstring,...);` (see page 30).

203 **symbol is never used:** *identifier*
A symbol is defined but never used. Public functions are excluded from the symbol usage check (since these may be called from the outside).

204 **symbol is assigned a value that is never used:** *identifier*
A value is assigned to a symbol, but the contents of the symbol are never accessed.

205 **redundant code: constant expression is zero**
Where a conditional expression was expected, a constant expression with the value zero was found, e.g. "`while (0)`" or "`if (0)`". The the conditional code below the test is *never* executed, and it is therefore redundant.

206 **redundant test: constant expression is non-zero**
Where a conditional expression was expected, a constant expression with a non-zero value was found, e.g. `if (1)`. The test is redundant, because the conditional code is *always* executed.

207 **unknown "#pragma"**
The compiler ignores the pragma. The `#pragma` directives may change between compilers of different vendors and between different versions of a compiler of the same version.

208 **function uses both "return;" and "return** *value***;"**
The function returns both *with* and *without* a return value. The function should be consistent in always returning with a function result, or in never returning a function result.

209 **function should return a value**
The function does not have a `return` statement, or it does not have an expression behind the `return` statement, but the function's result is used in a expression.

210 **possible use of symbol before initialization:** *identifier*
A local (uninitialized) variable appears to be read before a value is as-

signed to it. The compiler cannot determine the actual order of reading from and storing into variables and bases its assumption of the execution order on the physical appearance order of statements an expressions in the source file.

211 **possibly unintended assignment**
Where a conditional expression was expected, the assignment operator (`=`) was found instead of the equality operator (`==`). As this is a frequent mistake, the compiler issues a warning. To avoid this message, put parentheses around the expression, e.g. `if ( (a=2) )`.

212 **possibly unintended bitwise operation**
Where a conditional expression was expected, a bitwise operator (`&` or `|`) was found instead of a Boolean operator (`&&` or `||`). As this is a frequent mistake, the compiler issues a warning. To avoid this message, put parentheses around the expression, e.g. `if ( (a&2) )`.

213 **tag mismatch**
A tag mismatch occurs when:
- ◇ assigning to a tagged variable a value that is untagged or that has a different tag
- ◇ the expressions on either side of a binary operator have different tags
- ◇ in a function call, passing an argument that is untagged or that has a different tag than what the function argument was defined with
- ◇ indexing an array which requires a tagged index with no tag or a wrong tag name

Tags are discussed on page 21

214 **possibly a "const" array argument was intended:** *identifier*
Arrays are always passed by reference. If a function does not modify the array argument, however, the compiler can sometimes generate more compact and quicker code if the array argument is specifically marked as "`const`".

215 **expression has no effect**
The result of the expression is apparently not stored in a variable or used in a test. The expression or expression statement is therefore redundant.

216 **nested comment**
Small does not support nested comments.

217 **loose indentation**
Statements at the same logical level do not start in the same column;

that is, the indents of the statements are different. Although Small is a free format language, loose indentation frequently hides a logical error in the control flow.
The compiler can also incorrectly assume loose indentation if the TAB size with which you indented the source code differs from the assumed size, see `#pragma tabsize` on page 57 and the compiler option `-t` at page 69.

218 **old style prototypes used with optional semicolon**
When using "optional semicolons", it is preferred to explicitly declare forward functions with the `forward` keyword than using terminating semicolon.

219 **local variable** *identifier* **shadows a symbol at a preceding level**
A local variable has the same name as a global variable, a function, a function argument, or a local variable at a lower precedence level. This is called "shadowing", as the new local variable makes the previously defined function or variable inaccessible.

220 **exported or native symbol** *identifier* **is truncated to** *value* **characters**
Symbol names for exported or native functions have a more restrictive length, due to restrictions in the file format, than names of internal functions. Although the symbol name can be used as is internally, it will be inserted in the native or exported table in its truncated form.

221 **label name** *identifier* **shadows tag name**
A code label (for the `goto` instruction) has the same name as a previously defined tag. This may indicate a faultily applied tag override; a typical case is an attempt to apply a tag override on the variable on the left of the `=` operator in an assignment statement.

222 **number of digits exceeds rational number precision**
A literal rational number has more decimals in its fractional part than the precision of a rational number supports. The remaining decimals are ignored.

• Run time errors

Run-time errors: 100

The function library that forms the abstract machine returns error codes. These error codes encompass both errors for loading and initializing a binary file and run-time errors due to programmer errors.

The abstract machine

The abstract machine is a C function library. There are several version: one that is written in ANSI C, and optimized versions that use GNU C extensions or assembler subroutines.

Using the abstract machine

To use the abstract machine:

◇ create an abstract machine for a compiled program with `amx_Init`,

◇ register all modules that the program uses with `amx_Register`,

◇ run the program with `amx_Exec`,

The example (in C) below illustrates these steps:

```
int main(int argc,char *argv[])
{
extern AMX_NATIVE_INFO core_Natives[];
extern AMX_NATIVE_INFO console_Natives[];

  AMX amx;
  cell ret;
  int err;
  void *program;

  if (argc != 2 || (program = loadprogram(&amx,argv[1])) == NULL) {
    printf("Usage: SRUN <filename>\n\n"
           "The filename must include the extension\n");
    return 1;
  } /* if */

  amx_Register(&amx, core_Natives, -1);
  err = amx_Register(&amx, console_Natives, -1);
  if (err == AMX_ERR_NONE)
    err = amx_Exec(&amx, &ret, AMX_EXEC_MAIN, 0);

  if (err != AMX_ERR_NONE)
    printf("Run time error %d on line %ld\n", err, amx.curline);
  else if (ret != 0)
    printf("%s returns %ld\n", argv[1], (long)ret);

  free(program);
  return 0;
}
```

The `cell` data type is defined in `AMX.H`, it currently is a 32-bit integer. The future may bring 16-bit or 64-bit versions of the abstract machine.

"assert" statement: 52

The preceding example checks for run time errors that may occur while executing the Small program. Such errors are usually flagged by native functions or by `assert` instructions in the source code of the Small program.

The abstract machine API has no functions that read a program from file into memory. The kernel routines of the abstract machine API do not use dynamic memory allocation. Routines to allocate memory for the bytecode compiled program and to load it from disk must be provided by you. The snippet below is a typical example that does this:

```
void *loadprogram(AMX *amx,char *filename)
{
  FILE *fp;
  AMX_HEADER hdr;
  void *program = NULL;

  if ((fp = fopen(filename,"rb")) != NULL) {
    fread(&hdr, sizeof hdr, 1, fp);
    if ((program = malloc((int)hdr.stp)) != NULL) {
      rewind(fp);
      fread(program, 1, (int)hdr.size, fp);
      fclose(fp);
      memset(amx,0,sizeof *amx);
      if (amx_Init(amx,program) == AMX_ERR_NONE)
        return program;
      free(program);
    } /* if */
  } /* if */
  return NULL;
}
```

• Controlling program execution

The two snippets presented above are enough to form an interpreter for Small programs. A drawback, however, is that the Small program runs uncontrolled once it is launched with `amx_Exec`. If the Small program enters an infinit loop, for example, the only way to break out of it is to kill the complete interpreter —or at least the thread that the interpreter runs in. Especially during development, it is convenient to be able to abort a Small program that is running awry.

The abstract machine has a mechanism to monitor the execution of the pseudo-code that goes under the name of a "debug hook". The abstract machine calls the debug hook, a function that the host application provides, at specific events,

such as the creation and destruction of variables and executing a new statement. Obviously, the debug hook has an impact on the execution speed of the abstract machine. To minimize the performance loss, the abstract machine first checks queries the debug hook whether it want to receive further events. The debug hook *must* return an acknowledging value on this initial call.

To install a debug hook, call `amx_SetDebugHook` before calling `amx_Init`. For example, in the `loadprogram` function presented earlier, add the line:

```
amx_SetDebugHook(amx, amx_AbortProc);
```

between the calls to `memset` and `amx_Init`. The function `amx_AbortProc` becomes the "debug hook" function that is attached to the specified abstract machine. A minimal implementation of this function is below:

```
int AMXAPI amx_AbortProc(AMX *amx)
{
  switch (amx->dbgcode) {
  case DBG_INIT:
    return AMX_ERR_NONE;
  case DBG_LINE:
    /* check whether an "abort" was requested */
    return abortflagged ? AMX_ERR_EXIT : AMX_ERR_NONE;
  default:
    return AMX_ERR_DEBUG;
  } /* switch */
}
```

The debug hook *must* return `AMX_ERR_NONE` on the `DBG_INIT` event, otherwise it will receive no further events. The only other event captured by this particular debug hook function is `DBG_LINE`, which notifies the start of a statement on a new source code line. If the debug hook returns an error code other than `AMX_ERR_NONE` on the `DBG_LINE` event, the abstract machine aborts execution and returns that error code.

Exactly *how* the host program decides whether to continue running or to abort the abstract machine is implementation dependent. This example uses a global variable, `abortflagged`, that is set to a non-zero value —by some magical procedure— if the abstract machine(s) must be aborted.

There exists a more or less portable way to achieve the "magic" referred to in the previous paragraph. If you set up a `signal` function to set the `abortflagged` variable to 1 on a `SIGINT` signal, you have an ANSI C-approved way to abort an abstract machine. The snippet for the signal function appears below:

```
void sigabort(int sig)
{
```

```
  abortflagged=1;
  signal(sig,sigabort); /* re-install the signal handler */
}
```

And somewhere, before calling `amx_Exec`, you add the line:

```
signal(SIGINT,sigabort);
```

Debug hook functions allow you to monitor stack usage, profile execution speed at the source line level and, well... write a debugger. Detailed information on the debug hook is currently found in a separate document (see also the appendices of this manual).

Extension modules

An extension module provides a Small program with application-specific ("native") functions. Creating an extension module is a three-step process:

1 writing the native functions (in C);
2 making the functions known to the abstract machine;
3 writing an include file that declares the native functions for the Small programs.

• 1. Writing the native functions

Every native function must have the following prototype:

```
cell AMX_NATIVE_CALL func(AMX *amx, cell *params);
```

The identifier "`func`" is a placeholder for a name of your choice. The `AMX` type is a structure that holds all information on the current state of the abstract machine (registers, stack, etc.); it is defined in the include file `AMX.H`. The symbol `AMX_NATIVE_CALL` holds the calling convention for the function. The file `AMX.H` defines it as an empty macro (so the default calling convention is used), but some operating systems or environments require a different calling convention. You can change the calling convention either by editing `AMX.H` or by defining the `AMX_NATIVE_CALL` macro before including `AMX.H`. Common calling conventions are `_cdecl`, `_far _pascal` and `_stdcall`.

The `params` argument points to an array that holds the parameter list of the function. The value of `params[0]` is the number of *bytes* passed to the function (divide by the size of a `cell` to get the number of parameters passed to the function); `params[1]` is the first argument, and so forth.

For arguments that are passed by reference, function `amx_GetAddr` converts the "abstract machine" address from the "`params`" array to a physical address. The pointer that `amx_GetAddr` returns lets you access variables inside the abstract machine directly. Function `amx_GetAddr` also verifies whether the input address is a valid address.

Strings, like other arrays, are always passed by reference.However, neither packed strings nor unpacked strings are universally compatible with C strings (on Big Endian computers, packed strings are compatible with C strings). Therefore, the abstract machine API provides two functions to convert C strings to and from Small strings: `amx_GetString` and `amx_SetString`.

See page 124 for details on the memory layout of multi-dimensional arrays

A native function may abort a program by calling `amx_RaiseError` with a non-zero code. The non-zero code is what `amx_Exec()` returns.

• 2. Linking the functions to the abstract machine

An application uses `amx_Register` to make any native functions known to the abstract machine. Function `amx_Register` expects a list of `AMX_NATIVE_INFO` structures. Each structure holds a pointer to the name of the native function and a function pointer.

Below is a full example of a file that implements two simple native functions: raising a value to a power and calculating the square root of a value. The list of `AMX_NATIVE_INFO` structures is at the bottom of the example.

```
#include "amx.h"

static cell power(AMX *amx, cell *params)
{
  /* power(value, exponent);
   *   params[1] = value
   *   params[2] = exponent
   */
  cell result = 1;
  while (params[2]-- > 0)
    result *= params[1];
  return result;
}

static cell sqroot(AMX *amx, cell *params)
{
  /* sqroot(value);
   *   params[1] = value
   * This routine uses a simple successive approximation algorithm.
   */
  cell div = params[1];
```

```
  cell result = 1;
  while (div > result) {        /* end when div == result, or just below */
    div = (div + result) / 2;   /* take mean value as new divisor */
    result = params[1] / div;
  } /* while */
  return div;
}

AMX_NATIVE_INFO power_Natives[] = {
  { "power",  power },
  { "sqroot", sqroot },
  { 0, 0 }        /* terminator */
};
```

In you application, you must also add a call to `amx_Register` with the list of native functions, as shown below:

```
extern AMX_NATIVE_INFO power_Natives[];

err = amx_Register(&amx, power_Natives, -1);
```

• 3. writing an include file for the native functions

The first step implements the native functions and the second step makes the functions known to the abstract machine. Now the third step is to make the native functions known to the Small compiler. To that end, one writes an include file that contains the prototypes of the native functions and all constants that may be useful in relation to the native functions.

```
native power(value, exponent);
native sqroot(value);
```

Function reference

With one exception, all functions return an error code if the function fails (the exception is `amx_NativeInfo`). A return code of zero means "no error".

See page 100 for the defined error codes.

---

amx_Allot — Reserve stack space in the abstract machine

Syntax:
```
amx_Allot(AMX *amx,int cells,cell *amx_addr,
          cell **phys_addr)
```

`amx` — The abstract machine.

`cells` — The number of cells to reserve.

`amx_addr` — The address of the allocated cell as the Small program (that runs in the abstract machine) can access it.

`phys_addr` — The address of the cell for C programs to access.

Notes: You can fill the allocated stack space by writing to the address in `phys_addr`. Pass `amx_addr` to the Small function when you call the function with `amx_Exec`.

Remove the stack space with `amx_Release`.

See also: `amx_Exec`, `amx_Release`

---

amx_Callback — The default callback

Syntax:
```
int amx_Callback(AMX *amx, cell index, cell *result,
                 cell *params)
```

`amx` — The abstract machine.

`index` — Index into the native function table; it points to the requested native function.

`result` — The function result (of the native function) should be returned through this parameter.

`params` — The parameters for the native function, passed as a list of long integers. The first number of the list is the number of bytes passed to the native functions (from which the number of arguments can be computed).

See page 100 for the defined error codes.

Returns: The callback should return an error code, or zero for no error. When the callback returns a non-zero code, `amx_Exec` aborts execution.

Notes: The abstract machine has a default callback function, which works in conjunction with `amx_Register`. You can override the default operation by setting a different callback function using function `amx_SetCallback`.

If you override the default callback function, you may also need to provide an alternative function for `amx_Registers`.

See also: `amx_Exec`, `amx_RaiseError`, `amx_SetCallback`

---

amx_Debug — The default debug hook

Syntax: `int amx_Debug(AMX *amx)`

`amx` The abstract machine.

Returns: The debug hook should return an error code, or zero for no error.

Notes: The default debug function is a stub that immediately returns. Programs can replace the default debug hook function to monitor symbols and to trace through code step by step. The debugger interface is described in a separate document and through an example program in the distribution: `SDBG.C`.

See also: `amx_SetDebugHook`

---

amx_Exec — Run code

Syntax:
```
int amx_Exec(AMX *amx, long *retval, int index, int
             numparams, ...)
```

`amx` The abstract machine from which to call a function.

`retval` Will hold the return value of the called function upon return.

`index` An index into the "public function table"; it indicates the function to execute. See `amx_FindPublic` for more information. Use `AMX_EXEC_MAIN` to start executing at the `main` function.

`numparams`	The number of function parameters that follow.
`...`	Optional parameters for the function. All these parameters must be cast to the type `cell`, which is usually a 32-bit integer.

Notes: This function calls the callback function for any native function call that the code in the AMX makes. `amx_Exec` assumes that all native functions are correctly initialized with `amx_Register`.

See also: `amx_FindPublic`, `amx_Register`

---

amx_FindPublic Return the index of a public function

Syntax:
```
int amx_FindPublic(AMX *amx, char *funcname, int *index)
```

`amx`	The abstract machine.
`funcname`	The name of the public function to find.
`index`	Upon return, this parameter holds the index of the requested public function.

See also: `amx_Exec`, `amx_FindPubVar`, `amx_GetPublic`, `amx_NumPublics`

---

amx_FindPubVar Return the address of a public variable

Syntax:
```
int amx_FindPubVar(AMX *amx, char *varname,
                   cell *amx_addr)
```

`amx`	The abstract machine.
`varname`	The name of the public variable to find.
`amx_addr`	Upon return, this parameter holds the variable address relative to the abstract machine.

Notes: The returned address is the address relative to the "data segment" in the abstract machine. Use `amx_GetAddr` to acquire a pointer to its "physical" address.

See also: `amx_FindPublic`, `amx_GetAddr`, `amx_GetPubVar`, `amx_NumPubVars`

amx_Flags Return various flags

Syntax: `int amx_Flags(AMX *amx,unsigned short *flags)`

`amx` The abstract machine.

`flags` A set of bit flags is stored in this parameter. It is a set of the following flags:

`AMX_FLAG_CHAR16` if a character is 16-bits rather than the default of 8 bits

`AMX_FLAG_DEBUG` if the program carries symbolic information

Notes: A typical use for this function is to check whether the compiled program contains symbolic (debug) information. There is no use in installing a debugger callback if the program has no symbolic information.

amx_GetAddr Resolve an AMX address

Syntax: `int amx_GetAddr(AMX *amx,cell amx_addr,cell **phys_addr)`

`amx` The abstract machine.

`amx_addr` The address relative to the abstract machine.

`phys_addr` A pointer to the variable that will hold the memory address of the indicated cell.

Notes: This function returns the memory address of an address in the abstract machine. One typically uses this function in an extension module, because it allows you to access variables inside the abstract machine.

amx_GetPublic Return a public function name

Syntax: `int amx_GetPublic(AMX *amx, int index, char *funcname)`

`amx` The abstract machine.

`index` The index of the requested function. Use zero to retrieve the name of the first public function.

`funcname` The string that will hold the name of the public function.

Notes: The string should be large enough to hold longest function name plus the terminating zero byte. Use `amx_NameLength` to inquire this length.

See also: `amx_FindPublic`, `amx_GetPubVar`, `amx_NameLength`, `amx_NumPublics`

---

amx_GetPubVar Return a public variable name and address

Syntax:
```
int amx_GetPubVar(AMX *amx, int index, char *varname,
                  cell *amx_addr)
```

`amx` The abstract machine.

`index` The index of the requested variable. Use zero to retrieve the name and address of the first public variable.

`varname` The string that will hold the name of the public variable.

`amx_addr` Upon return, this parameter holds the variable address relative to the abstract machine.

Notes: The string should be large enough to hold longest variable name plus the terminating zero byte. Use `amx_NameLength` to inquire this length.

The returned address is the address relative to the "data segment" in the abstract machine. Use `amx_GetAddr` to acquire a pointer to its "physical" address.

See also: `amx_FindPubVar`, `amx_GetAddr`, `amx_GetPublic`, `amx_NameLength`, `amx_NumPubVars`

---

amx_GetString Retrieve a string from the abstract machine

Syntax:
```
int amx_GetString(char *dest, cell *source)
```

`dest` A pointer to a character array of sufficient size to hold the converted source string.

`source` A pointer to the source string. Use `amx_GetAddr` to convert a string address in the AMX to the physical address.

Notes: This function converts both packed strings and unpacked strings from the "Small" format to the "C" format.

See also: `amx_SetString`

---

amx_GetUserData Return general purpose user data

Syntax: `int amx_GetUserData(AMX *amx, long tag, void **ptr)`

`amx` The abstract machine.

`tag` The "tag" of the user data.

`ptr` Will hold a pointer to the requested user data upon return.

Notes: The AMX stores multiple "user data" fields. Each field must have a unique tag. The tag may be any value (as long as it is unique), but it is usually formed by a four-letter mnemonic through the macro `AMX_USERTAG`.

The AMX does not use "user data" in any way. The storage can be used for any purpose.

See also: `amx_SetUserData`

---

amx_Init Create an abstract machine, load the binary file

Syntax: `int amx_Init(AMX *amx, void *program)`

`amx` This variable is initialized with the specific settings of the abstract machine.

`program` A pointer to the bytecode stream of the program.

Notes: `amx_Init` initializes the abstract machine with the settings from the binary file. Before calling this function, you should set the `amx` structure variable to all zeros.

See also: `amx_Release`

---

amx_NameLength Return the maximum name length

Syntax: `int amx_NameLength(AMX *amx, int *length)`

`amx` The abstract machine.

`length` Will hold the maximum name length upon return. The returned value includes the space needed for the terminating zero byte.

See also: `amx_GetPublic`, `amx_GetPubVar`

---

amx_NativeInfo Return a structure for amx_Register

Syntax: `AMX_NATIVE_INFO *amx_NativeInfo(char *name, AMX_NATIVE func)`

`name` The name of the function (as known to the Small program)

`func` A pointer to the native function.

Notes: This function creates a list with a single record for `amx_Register`. To register a single function, use the code snippet (where `my_solve` is a native function):

```
err = amx_Register(amx, amx_NativeInfo("solve", my_solve), 1);
```

This function returns a pointer to a static record.

See also: `amx_Register`

---

amx_NumPublics Return the number of public functions

Syntax: `int amx_NumPublics(AMX *amx, int *number)`

`amx` The abstract machine.

`number` Will hold the number of public functions upon return.

Notes: The function returns number of entries in the file's "public functions" table. To retrieve the function names, use `amx_GetPublic`.

See also: `amx_GetPublic`, `amx_NumPubVars`

---

amx_NumPubVars Return the number of public variables

Syntax: `int amx_NumPubVars(AMX *amx, int *number)`

`amx` The abstract machine.

`number` Will hold the number of public variables upon return.

Notes: The function returns number of entries in the file's "public variables" table. To retrieve the variable names, use `amx_GetPubVar`.

See also: `amx_GetPubVar`, `amx_NumPublics`

---

amx_RaiseError Flag an error

Syntax: `int amx_RaiseError(AMX *amx, int error)`

`amx` The abstract machine.

`error` The error code. This is the code that `amx_Exec()` returns.

Notes: This function should be called from a native function. It lets the default callback routine return an error code.

---

amx_Register Make native functions known

Syntax: `int amx_Register(AMX *amx, AMX_NATIVE_INFO *list, int number)`

`amx` The abstract machine.

`list` An array with structures where each structure holds a pointer to the name of a native function and a function pointer. The list is optionally terminated with a structure holding two `NULL` pointers.

`number` The number of structures in the `list` array, or -1 if the list ends with a structure holding two `NULL` pointers.

Notes: If this function returns the error code `AMX_ERR_NOTFOUND`, one or more native functions that are used by the Small program are not found in the provided list. You can call `amx_Register` again to register additional function lists.

See also: `amx_NativeInfo`

---

amx_Release Free stack space in the abstract machine

Syntax: `int amx_Release(AMX *amx,cell amx_addr)`

`amx` The abstract machine.

`amx_addr` The address of the allocated cell as the Small program (that runs in the abstract machine) sees it. This value is returned by `amx_Allot`.

Notes: `amx_Allot` allocates memory in descending stack order (the stack grows downwards). The `amx_addr` value passed to `amx_Release` frees all memory *below* that address. That is, a single call to `amx_Release` can free multiple calls to `amx_Allot` if you pass the `amx_addr` value of the first allocation.

See also: `amx_Exec`, `amx_Release`

---

amx_SetCallback Install a callback routine

Syntax: `int amx_SetCallback(AMX *amx, AMX_CALLBACK callback)`

`amx` The abstract machine.

`callback` The address for a callback function. See function `amx_Callback` for the prototype and calling convention of a callback routine.

Notes: If you change the callback function, you should not use functions `amx_Register` or `amx_RaiseError`. These functions work in conjunction with the default callback function. To set the default callback, set parameter `callback` to the function `amx_Callback`.

You may set the callback before or after calling `amx_Init`.

amx_SetDebugHook Install a debug routine

Syntax: `int amx_SetDebugHook(AMX *amx, AMX_DEBUG debug)`

`amx` The abstract machine.

`debug` The address for a callback function for the debugger. See `amx_Debug` for the prototype and calling convention of a debug hook routine.

Notes: If you use a non-default debug hook routine, you should set it before calling `amx_Init`.

To set the default debug routine, set parameter `debug` to the function `amx_Debug`.

amx_SetString Store a string in the abstract machine

Syntax: `int amx_SetString(cell *dest, char *source, int pack)`

`dest` A pointer to a character array in the AMX where the converted string is stored. Use `amx_GetAddr` to convert a string address in the AMX to the physical address.

`source` A pointer to the source string.

`pack` Non-zero to convert the source string to a packed string in the abstract machine, zero to convert the source string to a cell string.

See also: `amx_GetString`

amx_SetUserData Set general purpose user data

Syntax: `int amx_SetUserData(AMX *amx, long tag, void *ptr)`

`amx` The abstract machine.

`tag` The "tag" of the user data, which uniquely identifies the user data. This value should not be zero.

`ptr` A pointer to the user data.

Notes: The AMX stores multiple "user data" fields. Each field must have a unique tag. The tag may be any value (as long as it is unique) except zero, but it is usually formed by four characters through the macro `AMX_USERTAG`.

```
r = amx_SetUserData(amx, AMX_USERTAG('U','S','E','R'), "Fire");
```

The AMX does not use "user data" in any way. The storage can be used for any purpose.

See also: `amx_GetUserData`

---

amx_StrLen Get the string length in characters

Syntax: `int amx_StrLen(cell *cstring, int *length)`

`cstring` The string in the abstract machine.

`length` This parameter will hold the string length upon return.

Notes: This function determines the length in *characters* of the string, not including the zero-terminating character (or cell). A packed string occupies less cells than its number if characters.

Error codes

`AMX_ERR_NONE`
: No error.

`AMX_ERR_EXIT`
: Program aborted execution. This is usually not an error.

`AMX_ERR_ASSERT`
: A run-time assertion failed.

`AMX_ERR_STACKERR`
: Stack or heap overflow; the stack collides with the heap.

`AMX_ERR_BOUNDS`
: Array index is out of bounds.

`AMX_ERR_MEMACCESS`
: Accessing memory that is not allocated for the program.

`AMX_ERR_INVINSTR`
: Invalid instruction.

`AMX_ERR_STACKLOW`
: Stack underflow; more items are popped off the stack than were pushed onto it.

`AMX_ERR_HEAPLOW`
: Heap underflow; more items are removed from the heap than were inserted into it.

`AMX_ERR_CALLBACK`
: There is no callback function, and the program called a native function.

`AMX_ERR_NATIVE`
: Native function requested the abortion of the abstract machine.

`AMX_ERR_DIVIDE`
: Division by zero.

`AMX_ERR_MEMORY`
: General purpose out-of-memory error.

`AMX_ERR_FORMAT`
: Invalid format of the memory image for the abstract machine.

`AMX_ERR_VERSION`
: This program requires a newer version of the abstract machine.

`AMX_ERR_NOTFOUND`
: The requested native functions are not found.

`AMX_ERR_INDEX`
: Invalid index (invalid parameter to a function).

Rationale

The first issue in the presentation of a new computer language should be: *why a new language at all?*

Indeed, I *did* look at several existing languages before I designed my own. Not surprisingly these days, I specifically considered using Java. It turned out quickly, though, that Java's design goals were not my design goals. For example, where Java promotes distributed computing where "packages" reside on diverse machines, Small is designed so that the compiled applets can be easily stored in a compound file together with other data; and where Java is designed to be architecture neutral and application independent, Small is designed to be tightly coupled with an application; native functions are a taboo to some extent in Java (at least, it is considered "impure"), whereas native functions are "the reason to be" for Small. From the viewpoint of Small, the intended use of Java is upside down: native functions are seen as an auxiliary library that the application –in Java– uses; in Small, native functions are part of "the application" and the Small program itself is a set of auxiliary functions that the application uses.

A language for scripting applications: Small is targeted as an *extension language*, meant to write application-specific macros or subprograms with. Small is not the appropriate language for implementing business applications or operating systems in. Small is designed to be easily integrated with, and embedded in, other systems/applications.

As an extension language, Small programs typically manipulate objects of the host application. In an animation system, Small scripts deal with sprites, events and time intervals; in a communication application, Small scripts handle packets and connections. I assumed that the host application will make (a subset of) its resources and functionality available via functions, handles, magic cookies... in a similar way that a contemporary operating system provides an interface to processes written in C/C++ —e.g., the Win32 API or Linux' "`glibc`". To that end, Small has a simple and efficient interface to the "native" functions of the host application.

The first and foremost criterions for the Small language were execution speed and reliability. Reliability in the sense that a Small program should not be able to crash the application or tool in which it is embedded —at least, not easily. Although this limits the capabilities of the language significantly, the advantages are twofold:

- ◇ the application vendor can rest assured that its application will not crash due to user additions or macros,
- ◇ the user is free to experiment with the language with no (or little) risk of damaging the application files.

Speed is essential: Small programs would probably run in an abstract machine (I do not foresee native code Small compilers), and abstract machines are notoriously slow. I had to make a language that has low overhead and a language for which a fast abstract machine can be written. Speed should also be reliable, in the sense that a Small script should not slow down over time or have an occaisional performance hiccup. Consequently, Small excludes any required "background process", such as garbage collection, and the core of the abstract machine does not implicitly allocate any system or application resources while it runs. That is, Small does not allocate memory or open files, not without the help of a native function that the script calls *explicitly*.

As Dennis Ritchie said, by intent the C language confines itself to facilities that can be mapped relatively efficiently and directly to machine instructions. The same is true for Small, and this is also a partial explication why Small looks so much like C.

A brief analysis showed that the instruction decoding logic for an abstract machine would quickly become the bottleneck in the performance of the abstract machine. To keep the decoding simple, each opcode should have the same size (excluding operands), and the opcode should fully specify the instruction (including the addressing methods, size of the operands, etc.). That meant that for each operation on a variable, the abstract machine needed a separate opcode for every combination of variable type, storage class and access method (direct, or dereferenced). For even three types (`int`, `char` and `unsigned int`), two storage classes (global and local) and three access methods (direct, indirect or indexed), a total of 18 opcodes (3*2*3) are needed to simply fetch the value of a variable.

At the same time, to keep the abstract machine small and manageable, I set a maximum of approximately 100 instructions.[3] With 18 opcodes to load a variable in a register, 18 more to store a register into a variable, another 18 to get the address of a variable, etc... I was quickly approaching (and exceeding) my limit

[3] 133 Opcodes are defined at this writing. To exploit performance gains by forcing proper alignment of memory words, the current abstract machine uses 32-bit opcodes. There is no technical limit on the number of opcodes, but in the interest of a small footprint, the number of opcodes should be restricted.

of a hundred opcodes.

The languages BOB and REXX inspired me to design a typeless language. This saved me a lot of opcodes. At the same time, the language could no longer be called a "subset of C". I was changing the language. Why, then, not go a foot further in changing the language? This is where a few more design guidelines came into play:

◇ give the programmer a general purpose tool, not a special purpose solution
◇ avoid error prone language constructs; promote error checking
◇ be pragmatic

A general purpose tool: Small is targeted as an extension language, without specifying exactly what it will extent. Typically, the application or the tool that uses Small for its extension language will provide many, optimized routines or commands to operate on its native objects, be it text, database records or animated sprites. The extension language exists to permit the user to do what the application developer forgot, or decided not to include. Rather than providing a comprehensive library of functions to sort data, match regular expressions, or draw cubic Bézier splines, Small should supply a (general purpose) means to use, extend and combine the specific ("native") functions that an application provides.

Small lacks a comprehensive standard library. By intent, Small also lacks features like pointers, dynamic memory allocation, direct access to the operating system or to the hardware, that are needed to remain competitive in the field of general purpose application or system programming. You cannot build linked lists or dynamic tree data structures in Small, and neither can you access any memory beyond the boundaries of the abstract machine. That is not to say that a Small program can never use dynamic, sorted symbol tables, or change a parameter in the operating system; it *can* do that, but it needs to do so by calling a "native" function that an application provides to the abstract machine.

In other words, if an application chooses to implement the well known `peek` and `poke` functions (from BASIC) in the abstract machine, a Small program can access any byte in memory, insofar the operating system permits this. Likewise, an application can provide native functions that insert, delete or search symbols in a table and allows several operations on them. The proposed core functions `getproperty` and `setproperty` are an example of native functions that build a linked list in the background.

Promote error checking: As you may have noticed, one of the foremost design criterions of the C language, "trust the programmer", is absent from my list of design criterions. Users of script languages are not always full time programmers;

and even if they are, Small will probably not be their *primary* language. Most Small programmers will keep learning the language as they go, and will even after years not have become experts. Enough reason, hence, to replace error prone elements from the C language (pointers) with saver, albeit less general, constructs (references).[4] References are copied from C++. They are nothing else than pointers in disguise, but they are restricted in various, mostly useful, ways. Turn to a C++ book to find more justification for references.

I find it disturbing that many, even modern, programming languages have so little built-in, or easy to use, support for confirming that programs do as the programmer intended. I am not referring to theoretical correctness (which is too costly to achieve for anything bigger than toy programs), but practical, easy to use, verification mechanisms as a help to the programmer. Small provides both compile time and execution time assertions to use for preconditions, postconditions and invariants.

The typing mechanism that most programming languages use also an automatic "catcher" of a whole class of bugs. By virtue of being a typeless language, Small lacked these error checking abilities. This was clearly a weakness, and I invented the "tag" mechanism to re-introduce the ability to verify function parameter passing, array indexing and other operations.

Be pragmatic: The object-oriented programming paradigm has not entirely lived up to its promise, in my opinion. On the one hand, OOP solves many tasks in an easier or cleaner way, due to the added abstraction layer. On the other hand, contemporary object-oriented languages leave you struggling with the language as much as with the task at hand. Jean-Paul Tremblay and Paul Sorenson criticize the C language's large operator set with the argument that studies have shown that people have difficulty with memorizing and understanding deep hierarchies.[5] The same argument also applies to the class hierarchies in object-oriented programming libraries. Object-oriented programming is not a solution for a non-expert programmer with little patience for artificial complexity. The criterion "be pragmatic" is a reminder to seek solutions, not elegancy. Sarcastically, perhaps, I have attempted to make Small a *subject oriented* language.

---

[4] You should see this remark in the context of my earlier assertion that many "Small" programmers will be novice programmers. In my (teaching) experience, novice programmers make many pointer errors, as opposed to experienced C/C++ programmers.

[5] "The Theory and Practice of Compiler Writing", McGraw-Hill, 1985, pp. 92.

• Practical design criterions

The fact that Small looks so much like C cannot be a coincidence, and it isn't. Small started as a C dialect and stayed that way, because C has a proven track record. The changes from C were mostly born out of necessity after rubbing out the features of C that I did not want in a scripting language: no preprocessor, no pointers, no variable types.

Small, being a typeless language, needed a different means to declare variables. In the course of modifying this, I also dropped the C requirement that all variables should be declared at the top of a compound statement. Small is a little more like C++ in this respect.

C language functions can pass "output values" via pointer arguments. The standard function `scanf`, for example, stores the values or strings that it reads from the console into its arguments. You can design a function in C so that it optionally returns a value through a pointer argument; if the caller of the function does not care for the return value, it passes `NULL` as the pointer value. The standard function `strtol` is an example of a function that does this. This technique frequently saves you from declaring and passing dummy variables. Small replaces pointers with references, but references cannot be `NULL`. Thus, Small needed a different technique to "drop" the values that a function returns via references. Its solution is the use of an "argument placeholder" that is written as an underscore character ("_"); Prolog programmers will recognize it as a similar feature in that language. The argument placeholder reserves a temporary anonymous data object (called a "cell" in Small) that is automatically destroyed after the function call.

The temporary cell for the argument placeholder should still have a value. Therefore, a function must specify for each passed-by-reference argument what value it will have upon entry when the caller passes the placeholder instead of an actual argument. By extension, I also added default values for arguments that are "passed-by-value". The feature to optionally remove all arguments with default values from the right was copied from C++.

When speaking of BCPL and B, Dennis Ritchie said that C was invented in part to provide a plausible way of dealing with character strings when one begins with a word-oriented language. Small provides two options for working with strings, packed and unpacked strings. In an unpacked string, every character fits in a cell. The overhead for a typical 32-bit implementation is large: one character would take four bytes. Packed strings store up to four characters in one cell, at the cost of being significantly more difficult to handle if you could only access full cells. Modern BCPL implementations provide two array indexing methods: one to get

a word from an array and one to get a character from an array. Small copies this concept, although the syntax differs from that of BCPL. The packed string feature also led to the new operator `char`.

Unicode applications often have to deal with two characters sets: 8-bit for legacy file formats and standardized transfer formats (like many of the Internet protocols) and the 16-bit Unicode character set. Although the Small compiler has an option that makes characters 16-bit (so only two characters fit in a 32-bit cell), a more convenient approach may be to store 8-bit character strings in packed strings and 16-bit (Unicode) strings in unpacked strings. This turns a weakness in Small — the need to distinguish packed strings from unpacked strings— into a strength: Small can make that distinction quite easily.

Notwithstanding the above mentioned changes, plus those in the chapter "Pitfalls: differences from C" (page 64), I have tried to keep close to C.

Design of the abstract machine APPENDIX B

The first issue is: why an abstract machine at all? By compiling into the native machine language of the processor of your choice, the performance will be so much better.

There is only one real reason to use an abstract machine: cross-platform compatibility of the compiled binary code. At the time that Small was designed, both 16-bit and 32-bit platforms on the 80x86 processor series were important for me. By the time I can forget about 16-bit operating systems, alternate microprocessors (like PowerPC and DEC Alpha) may have become essential.

Other reasons (while not essential) are:

◇ It is far easier to keep a program running in an abstract machine inside its "sandbox". For example, an unbounded recursion in an abstract machine crashes the abstract machine itself, but not much else. If you run native machine code, the recursive routine may damage the system stack and crash the application. Although modern operating systems support multithreading, with a separate stack per thread, the default action for an overrun of any stack is still to shut down the entire application.

◇ It is easier to design a language where a data object (an array) can contain bytecode which is later executed. Modern operating systems separate code and data sections: you cannot write into a code section and you cannot execute data; that is, not without serious effort.

  The current Small language does not have the ability to execute bytecode from an array, but the abstract machine is not too tightly coupled to the language. That is, future versions of the Small language may provide a means to execute a code stream from a variable without requiring me to redesign the abstract machine.

My first stab at designing an abstract machine was to look at current implementations. It appears that it is some kind of a tradition to implement abstract machines as stack machines, even though the design for microprocessors has moved towards register based implementations. All the abstract machines I encountered are stack based. These include:

◇ Microsoft C/C++ 7.0 (P-code option)
◇ Lua
◇ BOB
◇ Java VM (JVM)
◇ the B language (predecessor of C)
◇ the Amsterdam Compiler Kit

Stack machines are surely compact, flexible and simple to implement, but they are also more difficult to optimize for speed. To see why, let's analyze a specific example.

```
a = b + 2;      /* where "a" and "b" are simple variables */
```

Native code

In 32-bit assembler, this would be:

```
mov     eax, [b]
add     eax, 2
mov     [a], eax
```

Stack based abstract machine

Forth is the archetype for a stack machine, I will therefore use it as an example. The same routine in Forth would be:

```
b @ 2 + a !
```

where each letter is an instruction (the "@" stands for "fetch" and "!" for store; note that stack machines run code in "reverse polish notation"). So these are six instructions in bytecode, but the code expands to:

```
b       push    offset b
@       pop     eax
        push    [eax]
2       push    2
+       pop     edx
        pop     eax
        add     eax, edx
        push    eax
a       push    offset a
!       pop     edx
        pop     eax
        mov     [edx], eax
```

Two observations: **1.** the stack machine makes heavy use of memory (bad for performance) and **2.** the expanded code is quite large when compared to the native code (12 instructions versus 3).

The expanded code is what a "just-in-time" compiler (JIT) might make from it (though one may expect an optimizing JIT to reduce the redundant "pushes" and "pops" somewhat). When running the code in an abstract machine, the abstract machine must also expand the code, but in addition, it has overhead for fetching and decoding instructions. This overhead is at least two native instructions per bytecode instruction (more on this later). For six bytecode instructions, one should add another 12 native instructions to the 12 native instructions of the expanded code. And still, the example is greatly simplified, because the code

runs on the systems stack and uses the systems address space.

In other words, a stack-based abstract machine runs a native 3-instruction code snippet in 6 bytecode instructions, which turn out to take 24 native instructions, and more if you want to run the abstract machine on its own stack and in its own (protected) data space.

Register-based abstract machine
Microprocessors have use registers since their theoretical inception by Von Neumann. Extending this architecture to an abstract machine is only natural. There are two advantages: the abstract machine instructions map better to the native instructions (you may actually use the processor's registers to implement the abstract machine's registers) and the number of virtual instructions that is needed to executed a simple expression can be reduced.

As an example, here is the code for the Small "AMX", a two-register abstract machine (AMX stands for "Abstract Machine eXecutive"):

```
load.pri    b   ; "pri" is the primary register, i.e. the accumulator
const.alt   2   ; "alt" is the alternate register
add             ; pri = pri + alt
stor.pri    a   ; store "pri" in variable "a"
```

In expanded code, this would be:

```
load.pri   b        mov   eax, [b]
const.alt  2        mov   edx, 2
add                 add   eax, edx
stor.pri   a        mov   [a], eax
```

The four bytecode instructions map nicely to native instructions. Here again, we will have to add the overhead for fetching and decoding the bytecode instructions (2 native instructions per bytecode instruction). When compared to a stack-based abstract machine, the register-based abstract machine runs twice as fast; in 12 native instructions, versus 24 native instructions for a stack-based abstract machine.

There is more: in my experience, stack-based abstract machines are easier to optimize for size and register-based abstract machines are easier to optimize for speed. So a register-based abstract machine can indeed be twice as fast as a stack-based abstract machine.

To elaborate a little further on optimizing: I have intentionally chosen to add "2" to a variable. Incrementing or decrementing a value by one or two is such a common case that Forth has a special operator for them: the word "2+" adds 2 to a value. Assuming that a good (stack-based) abstract machine also has special

opcodes for common operations, using this "2+" word instead of the general words "2" and "+" removes one bytecode instruction and 3 native instructions. This would brings the native instruction count down to 21. However, the same optimization trick applies to the register-based abstract machine. The Small abstract machine has an "add.c" opcode that adds a constant value to the primary register. The optimized sequence would be:

```
load.pri  b         mov   eax, [b]
add.c     2         add   eax, 2
stor.pri  a         mov   [a], eax
```

which results to 3 native instructions plus 6 instructions of overhead for fetching and decoding the bytecode instructions. The register-based abstract machine (which needs 9 native instructions) is still approximately twice as fast as the stack-based abstract machine (at 21 native instructions).

• Threading

In an indirect threaded interpreter, each opcode is an index in a table that contains a "jump address" for every instruction. In a direct threaded interpreter, the opcode *is* the jump address itself. Direct threading often requires that all opcodes are "relocated" to jump addresses upon compilation or upon loading a pre-compiled file. The file format of the Small abstract machine is designed such that both indirect and direct threading are possible.

A threaded abstract machine is conventionally written in assembler, because most high level languages cannot store label addresses in an array. The GNU C compiler (GCC), however, extends the C language with an unary "`&&`" operator that returns the address of a label. This address can be stored in a "`void *`" variable type and it can be used later in a `goto` instruction. Basically, the following snippet does the same a "`goto home`":

```
void *ptr = &&home;
goto *ptr;
```

The ANSI C version of the abstract machine uses a large `switch` statement to choose the correct instructions for every opcode. Due to direct threading, the GNU C version of the abstract machine runs approximately twice as fast as the ANSI C version. Fortunately, GNU C runs on quite a few platforms. This means that the fast GNU C version is still fairly portable.

• Optimizing in assembler

The following discussion assumes an Intel 80386 or compatible processor. The same technique also applies to 16-bit processors and to processors of other brands, but the names (and number) of registers will be different.

It is beneficial to use the processor's registers to implement the registers of the abstract machine. The details of the abstract machine for the Small system are in appendix C. Further assumptions are:

- ◇ `PRI` is an alias for the processor's register `EAX` and `ALT` is `EDX`
- ◇ `ESI` is the code instruction pointer (`CIP`)
- ◇ `EDI` points to the start of the data segment, `ECX` is the stack pointer (`STK`), `EBX` is the frame pointer (`FRM`) and `EBP` is available as a general purpose intermediate register; the remaining registers in the AMX (`STP` and `HEA`, see appendix C) are local variables.

Every opcode has a set of machine instructions attached to it, plus a trailer that branches to the next instruction. The trailer is identical for every opcode. As an example, below is the implementation of the `ADD.C` opcode:

```
add     eax, [esi]      ; add constant
add     esi, 4          ; skip constant
; the code below is the same for every instruction
add     esi, 4          ; pre-adjust instruction pointer
jmp     [esi-4]         ; jump to address
```

Note that the "trailer" which chains to the next instruction via (direct) threading consists of two instructions; this trailer was the origin of the premise of a 2-instruction overhead for instruction fetching and decoding in the earlier analysis.

In the implementation of the abstract machine, one can hand-optimize the sequences further. In the above example, the two "`add esi, 4`" instructions can, of course, be folded into a single instruction that adds 8 to `ESI`.

Abstract machine reference

The abstract machine consists of a set of registers, a proposed (or imposed) memory layout and a set of instructions. Each is discussed in a separate section.

• Register layout

The abstract machine mimics a dual-register processor. In addition to the two "general purpose" registers, it has a few internal registers. Below is the list with the names and description of all registers:

`PRI` primary register (ALU, general purpose).

`ALT` alternate register (general purpose).

`FRM` stack frame pointer, stack-relative memory reads and writes are relative to the address in this register.

`CIP` code instruction pointer.

`DAT` offset to the start of the data.

`COD` offset to the start of the code.

`STP` stack top.

`STK` stack index, indicates the current position in the stack. The stack runs downwards from the `STP` register towards zero.

`HEA` heap pointer. Dynamically allocated memory comes from the heap and the `HEA` register indicates the top of the heap.

Notably missing from the register set is a "flags" register. The abstract machine keeps no separate set of flags; instead all conditional branches are taken depending on the contents of the `PRI` register.

• Memory image

The heap and the stack share a memory block. The stack grows downwards from `STP` towards zero; the heap grows upwards. An exception occurs when the `STK` and the `HEA` registers collide. (An exception means that the abstract machine aborts with an error message. There is currently no exception trapping mechanism.)

The figure below is a proposed memory image layout. Alternative layouts are possible. Specifically, an implementation may choose to keep the heap and the stack in a separate memory block next to the memory block for the code, the data and the prefix. The top of the figure represents the lowest address in memory.

The file format is a dump of the memory image. That is, the binary file starts with the prefix, and is followed by the code and data sections. The heap and

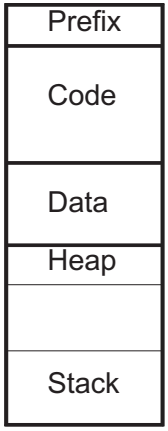

FIGURE 1: *Memory layout of the abstract machine*

stack sections are not stored in the binary file, the abstract machine can build them from information in the "prefix" section. The prefix also contains startup information, and the definitions of native and public functions.

All multiple byte values are stored with the low byte at the lower address (Little Endian). This is valid for the prefix and for the generated code and data.

`size`	4 bytes	size of the memory image, excluding the stack/heap
`magic`	2 bytes	must be `F1E0` (hexadecimal)
`version`	2 bytes	required minimal version of the abstract machine
`flags`	2 bytes	flags, see below
`defsize`	2 bytes	size of a structure in the "native functions" and the "public functions" tables
`cod`	4 bytes	start of the code section
`dat`	4 bytes	start of the data section
`hea`	4 bytes	initial value of the heap, end of the data section
`stp`	4 bytes	stack top value (the total memory requirements)
`cip`	4 bytes	starting address (`main()` function), `-1` if none
`num-pub`	2 bytes	number of public functions
`off-pub`	4 bytes	offset to the "public functions" table
`num-ntv`	2 bytes	number of native functions
`off-ntv`	4 bytes	offset to the "native functions" table
`num_lib`	2 bytes	number of external libraries (dynamically loaded)
`off-lib`	4 bytes	offset to the table of libraries

`num-pvar`	2 bytes	number public variables
`off-pvar`	4 bytes	offset to the "public variables" table
`public`	variable	public functions table (see below)
`native`	variable	native functions table (see below)
`library`	variable	library table (see below)
`pubvar`	variable	public variables table (see below)

Each bit in the `flags` field contains one setting. Currently, the defined settings are:

`0` if set, a character (in a packed string) is 16-bit
`1` if set, the file contains symbolic (debug) information

The fixed part of the prefix followed by a series of tables. Each table contains zero or more records. The size of these records is in the `defsize` field in the prefix. The records in the public functions table have the format:

`address`	4 bytes	the address (relative to `COD`) of the function
`name`	`defsize` - 4	the name of the public function

The format of the native functions table is very similar (see below). The order of the records in the table is important, because the parameter of the `SYSREQ.C` instruction is an index into the native functions table.

`address`	4 bytes	used internally, should be zero in the file
`name`	`defsize` - 4	the name of the native function

The library table has the same format as the native functions table. The "`address`" field is used internally and should be zero in the file. The "`name`" field holds the library name.

The "public variables" table, again, has a similar record lay out as the public functions table. The address field of a public variable contains the variable's address relative to the `DAT` section.

• Instruction reference

Every instruction consists of an opcode followed by zero or one parameters. Each opcode is one byte in size; an instruction parameter has the size of a cell (usually four bytes). A few "debugging" instructions (at the end of the list) form an exception to these rules: they have two or more parameters and those parameters are not always cell sized.

Many instructions have implied registers as operands. This reduces the number of operands that are needed to decode an instruction and, hence, it reduces the time needed to decode an instruction. In several cases, the implied register is part of the name of the opcode. For example, `PUSH.pri` is the name of the opcode that stores the `PRI` register on the stack. This instruction has no parameters: its parameter (`PRI`) is implied in the opcode name.

The instruction reference is ordered by opcode. The description of two opcodes is sometimes combined in one row in the table, because the opcodes differ only in a source or a destination register. In these cases, the opcodes and the variants of the registers are separated by a "/".

The "semantics" column gives a brief description of what the opcode does. It uses the C language syntax for operators, which are the same as those of the Small language. An item between square brackets indicates a memory access (relative to the `DAT` register, except for jump and call instructions). So, `PRI = [address]` means that the value read from memory at location `DAT + address` is stored in `PRI`.

opcode	mnemonic	parameters	semantics
1/2	LOAD.pri/alt	address	PRI/ALT = [address]
3/4	LOAD.S.pri/alt	offset	PRI/ALT = [FRM + offset]
5/6	LREF.pri/alt	address	PRI/ALT = [[address]]
7/8	LREF.S.pri/alt	offset	PRI/ALT = [[FRM + offset]]
9	LOAD.I		PRI = [PRI] (full cell)
10	LODB.I	number	PRI = "number" bytes from [PRI] (read 1/2/4 bytes)
11/12	CONST.pri/alt	value	PRI/ALT = value
13/14	ADDR.pri/alt	offset	PRI/ALT = FRM + offset
15/16	STOR.pri/alt	address	[address] = PRI/ALT
17/18	STOR.S.pri/alt	offset	[FRM + offset] = PRI/ALT
19/20	SREF.pri/alt	address	[[address]] = PRI/ALT
21/22	SREF.S.pri/alt	offset	[[FRM + offset]] = PRI/ALT
23	STOR.I		[ALT] = PRI (full cell)
24	STRB.I	number	"number" bytes at [ALT] = PRI (write 1/2/4 bytes)
25	LIDX		PRI = [ALT + (PRI × *cell size*)]
26	LIDX.B	shift	PRI = [ALT + (PRI << *shift*)]
27	IDXADDR		PRI = ALT + (PRI × *cell size*) (calculate indexed address)
28	IDXADDR.B	shift	PRI = ALT + (PRI << *shift*) (calculate indexed address)
29/30	ALIGN.pri/alt	number	Little Endian: PRI/ALT ^= *cell size*− number

31	LCTRL	index	PRI is set to the current value of any of the special registers. The index parameter must be: 0=COD, 1=DAT, 2=HEA, 3=STP, 4=STK, 5=FRM, 6=CIP (of the next instruction)
32	SCTRL	index	set the indexed special registers to the value in PRI. The index parameter must be: 2=HEA, 4=STK, 5=FRM, 6=CIP
33/34	MOVE.pri/alt		PRI=ALT / ALT=PRI
35	XCHG		Exchange PRI and ALT
36/37	PUSH.pri/alt		[STK] = PRI/ALT, STK = STK − *cell size*
38	PUSH.R	value	Repeat *value* ×: [STK] = PRI, STK = STK − *cell size*
39	PUSH.C	value	[STK] = value, STK = STK − *cell size*
40	PUSH	address	[STK] = [address], STK = STK − *cell size*
41	PUSH.S	offset	[STK] = [FRM + offset], STK = STK − *cell size*
42/43	POP.pri/alt		STK = STK + *cell size*, PRI/ALT = [STK]
44	STACK	value	ALT = STK, STK = STK + value
45	HEAP	value	ALT = HEA, HEA = HEA + value
46	PROC		[STK] = FRM, STK = STK − *cell size*, FRM = STK
47	RET		STK = STK + *cell size*, FRM = [STK], STK = STK + *cell size*, CIP = [STK], The RET instruction cleans up the stack frame and returns from the function to the instruction after the call.
48	RETN		STK = STK + *cell size*, FRM = [STK], STK = STK + *cell size*, CIP = [STK], STK = STK + [STK] The RETN instruction removes a specified number of bytes from the stack. The value to adjust STK with must be pushed prior to the call.
49	CALL	address	[STK] = CIP + 5, STK = STK − *cell size* CIP = address The CALL instruction jumps to an address after storing the address of the next sequential instruction on the stack.
50	CALL.pri		[STK] = CIP + 1, STK = STK − *cell size* CIP = PRI jumps to the address in PRI after storing the address of the next sequential instruction on the stack.
51	JUMP	address	CIP = address (jump to the address)
52	JREL	offset	CIP = CIP + offset (jump “offset” bytes from current position)

53	JZER	address	if PRI == 0 then CIP = [CIP + 1]
54	JNZ	address	if PRI != 0 then CIP = [CIP + 1]
55	JEQ	address	if PRI == ALT then CIP = [CIP + 1]
56	JNEQ	address	if PRI != ALT then CIP = [CIP + 1]
57	JLESS	address	if PRI < ALT then CIP = [CIP + 1] (unsigned)
58	JLEQ	address	if PRI <= ALT then CIP = [CIP + 1] (unsigned)
59	JGRTR	address	if PRI > ALT then CIP = [CIP + 1] (unsigned)
60	JGEQ	address	if PRI >= ALT then CIP = [CIP + 1] (unsigned)
61	JSLESS	address	if PRI < ALT then CIP = [CIP + 1] (signed)
62	JSLEQ	address	if PRI <= ALT then CIP = [CIP + 1] (signed)
63	JSGRTR	address	if PRI > ALT then CIP = [CIP + 1] (signed)
64	JSGEQ	address	if PRI >= ALT then CIP = [CIP + 1] (signed)
65	SHL		PRI = PRI << ALT
66	SHR		PRI = PRI >> ALT (without sign extension)
67	SSHR		PRI = PRI >> ALT with sign extension
68	SHL.C.pri	value	PRI = PRI << value
69	SHL.C.alt	value	ALT = ALT << value
70	SHR.C.pri	value	PRI = PRI >> value (without sign extension)
71	SHR.C.alt	value	ALT = ALT >> value (without sign extension)
72	SMUL		PRI = PRI * ALT (signed multiply)
73	SDIV		PRI = PRI / ALT (signed divide), ALT = PRI mod ALT
74	SDIV.alt		PRI = ALT / PRI (signed divide), ALT = ALT mod PRI
75	UMUL		PRI = PRI * ALT (unsigned multiply)
76	UDIV		PRI = PRI / ALT (unsigned divide), ALT = PRI mod ALT
77	UDIV.alt		PRI = ALT / PRI (unsigned divide), ALT = ALT mod PRI
78	ADD		PRI = PRI + ALT
79	SUB		PRI = PRI − ALT
80	SUB.alt		PRI = ALT − PRI
81	AND		PRI = PRI & ALT
82	OR		PRI = PRI | ALT
83	XOR		PRI = PRI ^ ALT
84	NOT		PRI = !PRI
85	NEG		PRI = −PRI
86	INVERT		PRI = ~PRI
87	ADD.C	value	PRI = PRI + value
88	SMUL.C	value	PRI = PRI * value
89/90	ZERO.pri/alt		PRI/ALT = 0

91	ZERO	address	[address] = 0
92	ZERO.S	offset	[FRM + offset] = 0
93/94	SIGN.pri/alt		sign extent the byte in PRI or ALT to a cell
95	EQ		PRI = PRI == ALT ? 1 : 0
96	NEQ		PRI = PRI != ALT ? 1 : 0
97	LESS		PRI = PRI < ALT ? 1 : 0 (unsigned)
98	LEQ		PRI = PRI <= ALT ? 1 : 0 (unsigned)
99	GRTR		PRI = PRI > ALT ? 1 : 0 (unsigned)
100	GEQ		PRI = PRI >= ALT ? 1 : 0 (unsigned)
101	SLESS		PRI = PRI < ALT ? 1 : 0 (signed)
102	SLEQ		PRI = PRI <= ALT ? 1 : 0 (signed)
103	SGRTR		PRI = PRI > ALT ? 1 : 0 (signed)
104	SGEQ		PRI = PRI >= ALT ? 1 : 0 (signed)
105	EQ.C.pri	value	PRI = PRI == value ? 1 : 0
106	EQ.C.alt	value	PRI = ALT == value ? 1 : 0
107/108	INC.pri/alt		PRI = PRI + 1 / ALT = ALT + 1
109	INC	address	[address] = [address] + 1
110	INC.S	offset	[FRM + offset] = [FRM + offset] + 1
111	INC.I		[PRI] = [PRI] + 1
112/113	DEC.pri/alt		PRI = PRI − 1 / ALT = ALT − 1
114	DEC	address	[address] = [address] − 1
115	DEC.S	offset	[FRM + offset] = [FRM + offset] − 1
116	DEC.I		[PRI] = [PRI] − 1
117	MOVS	number	Copy memory from [PRI] to [ALT]. The parameter specifies the number of bytes. The blocks should not overlap.
118	CMPS	number	Compare memory blocks at [PRI] and [ALT]. The parameter specifies the number of bytes. The blocks should not overlap.
119	FILL	number	Fill memory at [ALT] with value in [PRI]. The parameter specifies the number of bytes, which must be a multiple of the cell size.
120	HALT	0	Abort execution (exit value in PRI), parameters other than 0 have a special meaning.
121	BOUNDS	value	Abort execution if PRI > value or if PRI < 0
122	SYSREQ.pri		call system service, service number in PRI
123	SYSREQ.C	value	call system service
124	FILE	size ord name	source file information pair: name and ordinal (see below)

125	LINE	line ord	source line number and file ordinal (see below)
126	SYMBOL	sze off flg name	symbol information (see below)
127	SRANGE	lvl size	symbol range and dimensions (see below)
128	JUMP.pri		CIP = PRI (indirect jump)
129	SWITCH	address	Compare PRI to the values in the case table (whose address is passed) and jump to the associated address.
130	CASETBL	...	A variable number of case records follows this opcode, where each record takes two cells. See the notes below for details on the case table lay-out.
131/132	SWAP.pri/alt		[STK] = PRI/ALT and PRI/ALT = [STK]
133	PUSHADDR	offset	[STK] = FRM + offset, STK = STK − *cell size*

• Compact file format

The default file format that the compiler generates is a very simple format that the abstract machine can execute directly after loading (or mapping) the file into memory. That is, if the machine uses *Little Endian* byte ordering. On a Big Endian processor all cells must be swapped. The alternative, "compact binary files", not only have a reduced size, the file format is also universal for Big Endian and Little Endian computers.

The header of the module (see page 113) and all tables (public functions, native functions, libraries public variables) are not compressed. The data that follows these tables is encoded with variable length codes: every four-byte cell is encoded in one to five bytes.

The highest bit of each byte is a "continuation" bit. If it is set, another bytes with seven more significant bits follows. The most significant 7 bits are stored first (at the lower file offset/memory address). When a series of bytes have been decoded, bit 6 (the next to most signification bit) of the first byte is repeated to fill the complete 32-bits.

Decoding examples:

```
0x21            0x00000021
0x41            0xffffffc1
0x80 0x41       0x00000041
0x7f            0xffffffff
```

• Cross-platform support

There is some level of cross-platform support in the abstract machine. Both Big Endian and Little Endian memory addressing schemes are in common use today. Big Endian is the "network byte order", as it is used for various network protocols, notably the Internet protocol suite. The Intel 80x86 and Pentium CPU series use Little Endian addressing.

The abstract machine is optimized for manipulating "cells", 32-bit quantities. Bytes or 16-bit words can only be read or written indirectly, by first generating an address and then use the `LODB.I` or `STRB.I` instructions. The `ALIGN.pri` instruction helps in generating the address.

Packed strings: 41

The abstract machine assumes that when multiple characters are packed in a cell, the first character occupies the highest bits in the cell and the last character is in the lowest bits of the cell. This is how the Small language stores packed strings. On a Big Endian computer, the order of the characters is "natural" in the sense that the first character of a pack is at the lowest address and the last character is at the highest address. On a Little Endian computer, the order of the characters is reversed. When accessing the second character of a pack, you should read/write from a lower address then when accessing the first character of the pack.

The Small compiler could easily generate the required extra code to adjust the address for each character in the pack. The draw-back would be that a module written for a Big Endian computer would not run on a Little Endian computer and vice versa. So instead, the Small compiler generates a special `ALIGN` instruction, whose semantics depend on whether the abstract machine runs on a Big Endian or a Little Endian computer. More specifically, the `ALIGN` instruction does nothing on a Big Endian computer and performs a simple bitwise "exclusive or" operation on a Little Endian computer.

• The "switch" instruction and case table lay-out

The `SWITCH` instruction compares the value of `PRI` with the case value in every record in the associated case table and if it finds a match, it jumps to the address in the matching record. The `SWITCH` opcode has one parameter, which is the address of the case table in de code segment (i.e., the address is relative to `COD`). At this address, a `CASETBL` opcode should appear.

Every record in a case table, except the first, contains a case value and a jump address, in that order. The first record keeps the number of subsequent records in the case table in its first cell and the "none-matched" jump address in its

second cell. If none of the case values of the subsequent records matches `PRI`, the `SWITCH` instruction jumps to this "none-matched" address. Note again that the first record is excluded in the "number of records" field in the first record.

The records in the case table are sorted on their value. An abstract machine may take advantage of this lay-out to search through the table with a binary search.

• Debugger support

There is limited support for source level debuggers, built-in in the instruction set. These opcodes are not "regular" in the sense that they have more than one parameter.

The `size` parameter of the `FILE` and `SYMBOL` instructions gives the length of the instruction in *bytes*, excluding the bytes for the opcode and of the `size` field itself. The value of the `size` should always be a multiple of the size of a cell.

The `name` parameter of the `FILE` and `SYMBOL` instructions is a variable length, zero terminated string.

The `ord` parameter of the `FILE` and `LINE` instructions and the `off` parameter of the `SYMBOL` instruction are regular cell-sized parameters. The `line` parameter of the `LINE` instruction also has the size of a cell.

The `flg` parameter of the `SYMBOL` opcode holds the class and the type of the symbol. The type is in the lowest byte; it is one of the following values:

1 a variable
2 a "reference", a variable that contains an address to another variable (in other words, a pointer).
3 an array
4 a reference to an array (a pointer to an array)
9 a function
10 a reference to a function (a pointer to a function)

The class is in the second byte; its value is:

0 the symbol refers to a global variable or to a function
1 the symbol refers to a local variable with a stack relative address
2 the symbol refers to a "static" local variable; the address is not stack relative

The "`off`" parameter is relative to either:

`COD` if the symbol refers to a function

`DAT` if the symbol refers to a global variable or a static local variable
`FRM` if the symbol refers to a local variable

An instruction for symbolic information is stored near the place where the variable or function to which it refers is created or declared. For local symbols, the symbolic information precedes the instructions that allocate, and optionally fill, the stack space for the variable(s). There is no run-time allocation for global symbols; therefore a symbolic debugger must browse through the code section to parse the symbolic information instructions and to collect the global symbols. This strategy was chosen as a compromise that minimized the overall effort to add symbolic debugging support to the compiler and to create a debugger. Writing the compiler was much easier when the symbolic information could be written where the variable was declared in the source code. A debugger should have a disassembler anyway. Combining these two resulted in decent debugger support with a low cost in terms of complexity.

The `SRANGE` instruction extends a preceding `SYMBOL` instruction with information about the dimensions and the size of an array. The first parameter gives the dimension and the second parameter the length of that dimension.

- For single dimension arrays, a single `SRANGE` instruction follows the `SYMBOL` instruction. The first parameter of the `SRANGE` instruction is zero (0) and the second parameter gives the size of the array.
- For two-dimensional arrays, two `SRANGE` instructions complete the symbol definition. The first `SRANGE` instruction has its level (first parameter) set to one (1) and the second parameter set to the size of the major dimension. The second `SRANGE` instruction holds a level of zero and the size of the minor dimension.

When the "`size`" field of an `SRANGE` instruction is zero, the array size is indeterminate. When no `SRANGE` instruction follows a `SYMBOL` instruction that defines an array, the array should be assumed a single-dimensional array with an indeterminate size.

The Small compiler generates a `LINE` instruction before any other instruction for that line. The "`ord`" parameter is the file number to which the line relates. The Small compiler generates the `FILE` instruction at the point where the file is read. So a debugger would gather the filenames (and their ordinals) in the same way (and perhaps in the same phase) as the global symbols.

Code generation notes

APPENDIX D

The code generation of the Small compiler is fairly straightforward (also due to the simplicity of the abstract machine). A few points are worth mentioning:

◇ The abstract machine has instructions that the Small compiler currently does not generate. For example, the `LREF.pri` instruction works like the dereference operator ("*") in C/C++. Small does not support pointers directly, but references are just pointers in disguise. Small only supports references in function arguments, however, which means that the "pointer operations" in Small are always stack-relative. In other words, the Small compiler does *not* generate the `LREF.pri` instruction, although if *does* generate the `LREF.S.pri` instruction.

  The abstract machine is fairly independent from the Small language, even though they were developed for each other. The Small language can easily grow in the future, possibly with a "reference" variable type, thereby giving the `LREF.pri` instruction a reason of being. The abstract machine cannot easily grow, however, because new instructions immediately make the new abstract machine incompatible with previous versions. That is, programs compiled for the new abstract machine won't run on the earlier release.

◇ For a native function, the Small compiler generates a `SYSREQ.C` instruction instead of the normal function call. The parameter of the `SYSREQ.C` instruction is an index in the native function table. A function in Small cleans up its arguments that were pushed on the stack, because it returns with the `RETN` instruction. The `SYSREQ.C` instruction does not remove items from the stack, so the Small compiler does this explicitly with a `STACK` instruction behind the `SYSREQ.C` instruction.

  The arguments of a native function are pushed on the stack in the same manner as for a normal function.

  In the "Small" implementation of the abstract machine (see page 83), the "system request" instructions are linked to the user-installed callback function. Thus, a native function in a Small program issues a call to a user-defined callback function in the abstract machine.

◇ At a function call, a Small program pushes the function arguments onto the stack in reverse order (that is, from right to left). It ends the list of function arguments on the stack by pushing the number of bytes that it pushed to the stack. Since the Small compiler only passes cell-sized function arguments to

a function, the number of bytes is the number of arguments multiplied by the size of a cell.

A function in Small ends with a `RETN` instruction. This instruction removes the function arguments from the stack.

◇ When a function has a "reference" argument with a default value, the compiler allocates space for that default value on the heap.

For a function that has an array argument with a default value, the compiler allocates space for the default array value on the heap. However, if the array argument (with a default value) is also `const`, the Small compiler passes the default array directly (there is no need to make a copy on the heap here, as the function will not attempt to change the array argument and, thereby, overwrite the default value).

◇ The arguments of a function that has "variable arguments" (denoted with the `...` operator, see page 30) are always passed by reference. For constants and expressions that are not *lvalues*, the compiler copies the values to a cell that is allocated from the heap, and it passes the address of the cell to the function.

◇ For the "`switch`" instruction, the Small compiler generates a `SWITCH` opcode and a case table with the `CASETBL` opcode. The case table is generated in the `COD` segment; it is considered "read-only" data. The "none-matched" address in the case table jumps to the instruction of the `default` case, if any.

Case blocks in Small are not drop through. At the end of every instruction in a `case` list, the Small compiler generates a jump to an "exit" label just after the `switch` instruction. The Small compiler generates the case table between the code for the last `case` and the exit label. By doing this, every case, including the `default` case, jumps around the case table.

◇ Multi-dimensional arrays are implemented as vectors that hold the offsets to the sub-arrays. For example, a two-dimensional array with four "rows" and three "columns" consists of a single-dimensional array with four elements, where each element is the offset to a three-element single-dimensional array. The total memory footprint of array is $4 + 4 \times 3$ cells. Multi-dimensional arrays in Small are similar to pointer arrays in C/C++.

As stated above, the "major dimension" of multi-dimensional arrays holds the offsets to the sub-arrays. This offset is in bytes (not in cells) and it is relative to the address of the cell from which the offset was read. Returning to the example of a two-dimensional array with four rows and three columns (and

assuming a cell size of four bytes), the memory block that is allocated for the array starts with the four-cell array for the "rows", followed by four arrays with each three elements. The first "column" array starts at four cells behind the "rows" array and, therefore, the first element of the "rows" array holds the value 16 ($4 \times$ *cellsize*). The second column array starts at three cells behind the first column array, which is seven cells behind start of the rows array. The offset to the second column array is stored in the it second element of the rows array, and the offset of the second column relative to the second cell of the rows array is *six* cells. The second value in the rows array is therefore 24 ($6 \times$ *cellsize*).

Index

◇ Names of persons (not products) are in *italics*.
◇ Function names, constants and compiler reserved words are in `typewriter font`.