7. Further Data Types and Control Structures

Emil Sekerinski, McMaster University, March 2022

Floating-Point Numbers

The *floating point representation* of a real number `x` is an approximation by a triple with *sign* `s`, *exponent* `e`, and *mantissa* `m` such that

$x = (-1)^s \times B^{e-w} \times 1.m$ where $1 \le 1.m < B$

The *base* `B` and *bias* `w` are fixed. The IEEE standard 754 standard specifies:

Precision	`B`	`w`	Sign bits	Exponent bits	Mantissa bits	Total bits
Half	2	$15 = 2^4 - 2$	1	5	10	16
Single	2	$127 = 2^7 - 1$	1	8	23	32
Double	2	$1023 = 2^{10} - 1$	1	11	52	64
Quadruple	2	$16383 = 2^{14} - 1$	1	15	112	128
Octuple	2	$262143 = 2^{18} - 1$	1	19	236	256

Following table gives some decimal numbers and their floating-point representation with single precision:

Decimal	`s`	`e`	`1.m`	Binary
`1.0`	`0`	`127`	`1.0`	`0 01111111 00000000000000000000000`
`0.5`	`0`	`126`	`1.0`	`0 01111110 00000000000000000000000`
`2.0`	`0`	`128`	`1.0`	`0 10000000 00000000000000000000000`
`10.0`	`0`	`130`	`1.25`	`0 10000010 01000000000000000000000`
`0.1`	`0`	`123`	`1.6`	`0 01111011 10011001100110011001101`
`-1.5`	`1`	`127`	`1.5`	`0 01111111 10000000000000000000000`

The value `0` is a special case, represented with all bits being `0`. Floating-point numbers with all bits of `e` being `1` represent `∞` if `m = 0` or `NaN` (not a number) if `m ≠ 0`.

Programming languages offer signed and unsigned integers and integers of various lengths (short, normal, long). For example, assume `a` is a 4-byte integer, `b` an 8-byte integer, and `f` a single float, and each is supported by different machine instructions. Consider:

```
b := b + a         f := a + f         a := b         a := f
```

For the *compatibility* between numeric types several options exist:

- Operands of arithmetic operators and both sides of assignments must be of the same type. If needed, conversion functions must be explicitly written, as they affect efficiency.
- Operands of arithmetic operators must be of same type, but on assignments implicit conversions may take place.
- Operands of arithmetic operators may be of mixed integer type (short, normal, long) or of mixed floating-point type, but not mixed integer and floating-point type, since e.g. converting a short integer to a long integer involves only a sign extension.
- For convenience, integer and real expressions may be freely mixed.

For conversions, the options are:

- Implicit conversions take only place from “smaller” to “larger” types. Hence a notion of *type inclusion* in underlying:

  `2-byte integer ⊆ 4-byte integer ⊆ single float`

- Implicit conversions take place from any numeric type to any other numeric type, with a check whether the result fits in the in the destination.

For mixed expressions, the options are:

- The precision of an operator is the larger precision of the operands.
- The precision of an operator is the largest possible precision; if needed, the result is converted to smaller precision on assignment.
- The precision of an operator is the precision of the result (left hand side of an assignment).

Floating-Point Number in WebAssembly

WebAssembly supports single and double precision floating-point numbers with identical instructions. The grammar extensions for single precision are:

```
float ::= num '.' [num] [('E' | 'e') ['+' | '-'] num]
num_type ::= ... | "f32" | "f64"
instr ::= ... |
          "f32.const" float | "f64.const" float |
          "f32.add" | "f64.add" | "f32.sub" | "f64.sub" |
          "f32.mul" | "f64.mul" | "f32.div" | "f64.div" |
          "f32.sqrt" | "f64.sqrt" |
          "f32.min" | "f64.min" | "f32.max" | "f64.max" |
          "f32.ceil" | "f64.ceil" | "f32.floor" | "f64.floor" |
          "f32.abs" | "f64.abs" | "f32.neg" | "f64.neg" |
          "f32.eq" | "f64.eq" | "f32.ne" | "f64.ne" |
          "f32.lt" | "f64.lt" | "f32.le" | "f64.le" |
          "f32.gt" | "f64.gt" | "f32.ge" | "f64.ge" |
          "i32.trunc_f32_s" | "i64.trunc_f64_s" | "f32.convert_i32_s" | "f64.convert_i64_s" |
          "f32.load" "offset" "=" num | "f64.load" "offset" "=" num |
          "f32.store" "offset" "=" num | "f64.store" "offset" "=" num
```

The instructions `i32.trunc_f32_s` and `i64.trunc_f64_s` pop a floating-point number from the stack and push the integer obtained by truncation on the stack. The instructions `f32.convert_i64_s` and `f32.convert_i64_s` pop an integer from the stack and push the converted floating-point value on the stack.

Sets

Sets can be defined as boolean-value functions, `set(U) = U → boolean`. For `x: set(U)`, membership tests whether `x` maps to `true`, formally `e ∈ x = x(e)`. Small sets `set [l .. u]` of subranges can be implemented compactly by *bitsets*. For example, a 32-bit word can represent sets with elements `0` to `31`. Thus the set `{3, 6}` is represented by:

```
00000000 00000000 00000000 01001000
```

- For the union `s ∪ t`, intersection `s ∩ t`, and complement `∁s` of sets `s, t`, the bitwise or, bitwise and, and bitwise complement of their representations is taken.
- For the membership test `x ∈ s`, binary `1` is shifted `x` positions to the left followed by a bitwise and with `s`. The result is `0` if `x ∉ s`, otherwise not `0`.

If `w` is the word size, `T = set(U)` where `U = [l .. u]` is a subrange with `n = u − l + 1` elements is represented by `n` bits stored in `⌈n / w⌉` words.

Bit Operations and "tee" in WebAssembly

The additional WebAssembly instructions are:

```
instr ::= ... |
          "i32.popcnt" | "i64.popcnt" |
          "i32.and" | "i64.and" | "i32.or" | "i64.or" | "i32.xor" | "i64.xor" |
          "i32.shl" | "i64.shl" |
          "local.tee" name
```

On words, unary operator `#` stands for the number of `1` bits in a word; binary operators `&`, `|`, `^`, `~`, `<<`, stand for the bitwise and, bitwise of, bitwise exclusive or, bitwise complement, and shift left. Identical instructions exists for 32-bit and 64-bit words. The `tee` instructions duplicates the top of the stack to the specified local variable:

instruction	effect	trap condition
`i32/64.popcnt`	`s[sp - 1] := #s[sp - 1]`	
`i32/64.and`	`s[sp - 2], sp := s[sp - 2] & s[sp - 1], sp - 1`	
`i32/64.or`	`s[sp - 2], sp := s[sp - 2] | s[sp - 1], sp - 1`	
`i32/64.xor`	`s[sp - 2], sp := s[sp - 2] ^ s[sp - 1], sp - 1`	
`i32/64.shl`	`s[sp - 2], sp := s[sp - 2] << s[sp - 1], sp - 1`	
`local.tee x`	`s[loc(x)] := s[sp - 1]`	

Translation Scheme for Sets to WebAssembly

Sets `set(U)` with `U = [l .. u]` and `0 ≤ l ≤ u < 32` are stored in a single `i32` word:

```
(func $program
(local $s i32)
(local $i i32)
(local $b i32)
```

```
type S = set [1..10]
program p
    var s: S
    var i: integer
    var b: boolean
        s, i := {}, 3
        s := {i} ∪ s
        s := ∁s
        b := {i, 5} ⊆ s
        i := #s
```

```
(local $0 i32)
i32.const 0   ;; {}
i32.const 3   ;; 3
local.set $i
local.set $s
local.get $i
local.set $0
i32.const 1
local.get $0
i32.shl
local.get $s
i32.or
local.set $s
local.get $s
i32.const 0x7fe   ;; {1, 2, …, 10}
i32.xor
local.set $s
local.get $i
local.set $0
i32.const 1
local.get $0
i32.shl
i32.const 5   ;; 5
local.set $0
i32.const 1
local.get $0
i32.shl
i32.or
local.tee $0
local.get $0
local.get $s
i32.and
i32.eq
local.set $b
local.get $s
i32.popcnt
local.set $i
)
```

The translation scheme for set declarations is:

D	code(D)	
`var x: set(U)`	`(local $x i32)`	for local declaration
`var x: set(U)`	`(global $x (mut i32) i32.const 0)`	for global declaration

The translation scheme for procedure declarations is extended to declare an auxiliary local variable, `$0`:

D	code(D)	

`procedure p(v1: T1, … , vn: Tn) → (r1: U1, … , rm: Um)`
`D`
`S`
	`(func $p (param $v1 i32) … (param $vn i32)`
	`(result i32) … (result i32)`
	`(local $r1 i32)`
	`…`
	`(local $rm i32)`
	`(local $0 i32)`
	`code(D)`
	`code(S)`
	`local.get $r1`
	`…`
	`local.get $rm`
	`)`
		if all `Ti`, `Uj` are `integer`, `boolean`, `set`

Since WebAssembly does not have an instruction that implements subset and superset tests, these can be expressed through union and intersection:

```
s ⊆ t  ≡  s = s ∩ t
s ⊇ t  ≡  s = s ∪ t
```

The set `{E}` is constructed by shifting `1` to the left `E` times. The set `{E1, E2}` is constructed the same way as `{E1} ∪ {E2}` would be. The sequence `local.tee $0` `local.get $0` duplicates the element on the top of the stack:

E	code(E)	condition

`{E}`	`i32.const 1`
	`code(E)`
	`i32.shl`	

`{E1, … , En}`	`code({E1} ∪ … ∪ {En})`	

`# E`	`code(E)` `i32.popcnt`	

`C E`	`code(E)`
	`i32.const {l, … , u}`
	`i32.xor`
		`type(E) = set [l .. u]`

$E_1 \cap E_2$	code(E_1) code(E_2) `i32.and`	
$E_1 \cup E_2$	code(E_1) code(E_2) `i32.or`	

$E_1 \in E_2$	`i32.const 1`
	code(E_1)
	`i32.shl`
	code(E_2)
	`i32.and`	

$E_1 \subseteq E_2$	code(E_1)
	`local.tee $0`
	`local.get $0`
	code(E_2)
	`i32.and`
	`i32.eq`	

$E_1 \supseteq E_2$	code(E_1)
	`local.tee $0`
	`local.get $0`
	code(E_2)
	`i32.or`
	`i32.eq`	

Enumeration Types and Disjoint Unions

An *enumeration type* introduces a type and all its possible values. Enumeration types are sets, but in languages like Pascal, the values are ordered: `succ(e)` and `pred(e)` give the next and previous value and they can be compared by `<`. The order can be used to declare subranges of enumerations and for iteration, for example in Pascal:

```
type Day = (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday);
type Weekday = Monday .. Friday;
var hours: array [Weekday] of 0 .. 24;
var w: Weekday; h: 0 .. 120;

h := 0; for w := Monday to Friday do h := h + hours[w]
```

An enumeration is represented by a “small” unsigned integer; `succ` and `pred` are then addition and subtraction by `1` and `<` is integer comparison. This transformation can be done by the front-end in a compiler, so that no additional support for code generation is needed.

A *disjoint union* constructs a type by *tagging* the *variants:*

```
type Symbol = Plus | Num(integer) | Id(string)
var s: Symbol

s := Num(0)

case s of
    Plus: write('+')
    Num(n): write(n)
    Id(i): write(i)
```

Here, `s` is either `Plus`, `Num(n)`, or `Id(i)` for some integer `n` and some string `i`.

A disjoint union of the form

```
type T = t_1(T_1) | … | t_n(T_n)
```

has $\#T = \#T_1 + \cdots + \#T_n$ possible values. A disjoint union where each variant has no type associated (or rather has the unit type associated) degenerates to an enumeration type:

```
type T = t_1 | … | t_n
```

Disjoint union types can be represented by one “small” integer for the tag plus the representation of the corresponding variant.

Disjoint unions allow to express recursive types, when some variants “terminate the recursion”. For example, a list of integers can be defined as:

```
type List = nil | cons(hd: integer, tl: List)
```

The list with elements `3` and `5` is `cons(3, cons(5, nil))`. Such recursive types can be used to express any tree-like structures. While the presence of recursion does not affect the representation of a `cons` object, the memory needed for a variable of `List` type can no longer be determined statically, necessitating dynamic memory allocation.

Disjoint unions can be expressed using *variant records* in Pascal and by *union types* in C:

```
type Tag = (Plus, Num, Id);typedef enum {Plus, Num, Id} Tag;
type Symbol =              typedef struct {
  record                       int pos;
    pos: integer;              Tag t;
    case t: Tag of             union {
      Plus: ();                    int n;
      Num: (n: integer);           char *i;
      Id: (i: string)          };
  end                      } Symbol;
```

A record of type `Symbol` always has fields `pos` and `t`; depending on the value of `t`, a record may have field `n` or `i`.

In object-oriented languages, disjoint unions can be expressed by inheritance, for example in Java:

```
class Symbol {int pos;}
class Plus extends Symbol {};
class Num extends Symbol {int n;};
class Id extends Symbol{String i;};
```

A benefit of expressing a disjoint union by inheritance is that new variants can be added. The downside is that intention is not explicitly visible, that type tests on values are needed, which can break in presence of extensions, and that using inheritance comes with an overhead. While early object-oriented languages did neither support enumeration types nor disjoint unions (Oberon, Java), newer languages support them again (Rust, Java).

Compatibility of Structured Types

Consider following declarations with records:

```
type R = (f: integer, g: integer)
var x, y: R
var z: (f: integer, g: integer)
```

With *name compatibility*, the assignment `x := y` is allowed, but the assignment `x := z` is not allowed. With *structural compatibility*, `x := z` is allowed. The argument for name compatibility is that the name carries a meaning and any operations that were meant for a type should name that type. Some object-oriented languages use name compatibility for classes (Java). In practice, that may require programs to be “refactored” when being extended, which may be avoided with structural compatibility (Go).

Pascal relaxes name compatibility to allow *aliases*. Below, `T` does not introduce a new type and variable `i` can be used as an integer:

```
type T = integer
var i: T
```

The P0 compiler implements structural compatibility.

Comparing two record types for compatibility requires recursively comparing the fields for compatibility. In the case of recursive types, care has to be taken. Consider two definitions of lists by disjoint unions:

```
type L = nil | cons(hd: integer, tl: L)
type M = nil | cons(hd: integer, tl: M)
```

If one were to compare the types of all fields recursively, the recursion would never end. That can be corrected by initially assuming that the types `L`, `M` are compatible and when the recursive comparison needs to check that `L`, `M` are compatible, use that assumption to stop the recursion. With this algorithm, following three types are compatible:

```
type L = nil | cons(hd: integer, tl: L)
type M = nil | cons(hd: integer, tl: N)
type N = nil | cons(hd: integer, tl: M)
```

Consider following array declarations:

```
type A = [0 .. 9] → integer
type B = [1 .. 10] → integer
```

As the upper and lower bound of the domain are part of the type, `A` and `B` are distinct types. That can be relaxed by allowing two arrays to be compatible if they are of the same length. The P0 compiler implements that.

Having the length of arrays as part of the type requires all procedures with array parameters to specify the length of the array, making them less reusable. A remedy is to allow *open arrays* (*conformant arrays*) as parameters:

```
procedure sum(l: integer, u: integer, a: array [l .. u] of integer) → (s: integer)
    var i: integer
        s, i := l, 0
        while i ≤ u do s, i := s + a[i], i + 1
```

Here, `l` and `u` are passed as parameters as well. This is a case of *dependent types*, i.e. types that depend on values. In general, dependent types make type-checking undecidable, so some safe approximation for types being compatible needs to be made.

An alternative is to leave out the length of an array from the type. Since the size of an array is in that case no longer statically determined, An alternative is to leave out the length of an array from the type. Since the size of an array is in that case no longer statically determined, arrays cannot be allocated statically, necessitating dynamic memory management. Using the notation `T[N]` for creating an array with `N` elements, `[]T` for an array of type `T`, and `length(a)` for the length of `a:[]T`, we can write:

```
procedure sum(a: []integer) → (s: integer)
    var i: integer
        s, i := 0, 0
        while i < length(a) do s, i := s + a[i], i + 1

var x: []integer
    x := integer[5]; ... sum(x) ...
```

Arrays are then sequences that, once created, cannot change in length. Languages like C and Go have both arrays of fixed length and dynamically created arrays.

Procedure as Values

Being able to pass procedures as values allows higher-order procedures like `map`: with

```
procedure plus1(a: integer) → (b: integer)
    b := a + 1
```

the call `map(plus1, [3, 7, 9])` returns `[4, 8, 10]`. To make functions like `map` general, *type variables* are used:

```
map: (α → β) × seq α → seq β
map (plus1, [3, 7, 9])
```

In words, `map` takes as arguments a function of type `α → β` and a sequence with elements of type `α`, for any `α` and `β`. Languages like ML infer the types `α`, `β` implicitly on application, as above. Other languages require those to be specified, as in:

```
map(α, β): (α → β) × seq α → seq β
map(integer, integer)(plus1, [3, 7, 9])
```

While type-checking becomes significantly more complex, the implementation can be rather straighforward. For procedure values it is sufficient only to pass the address of the procedure in memory, assuming that procedures cannot be modified at runtime. For a generic procedure like `map`, the body can be duplicated for each instance of the type parameters.

Procedure variables can be problematic. Nested procedures, in particular anonymous ones (lambda abstractions), are useful for being passed around to functions like `map`. Writing `(a: integer) → (b: integer)` for the type of procedures that have one integer parameter and one integer result, consider:

```
var v: (a: integer) → (b: integer)
var g: integer
procedure p(f: (a: integer) → (b: integer), c: integer) → (d: integer)
    d ← f(c)
procedure q()
    var i: integer
    procedure r(a: integer) → (b: integer)
        b := a + i
    g ← p(r, g) // ① ok
    v := r  // ② dangerous
    g ← v(g) // ③ ok
procedure s()
    q(); g ← v(g) // ④ problematic
```

The calls ① and ② have the well-defined effect of updating global variable `g`. However, after the call to `q()`, global variable `v` points to the nested procedure `r`, which accesses intermediate-level variable `i`, which is no longer visible as `q()` has terminated. Thus the call ④ is problematic:

- A simple solution is to disallow nested procedures, which C and WebAssembly do. Intermediate-level variables then do not exist.
- Less restrictive is to allow nested procedures, but do not allow them to *escape* their scope. This would disallow assignment ②.
- If procedures can escape their scope, then all variables to which they refer have to be preserved. Above, after the termination of `q`, variable `i` would have to be preserved, thus enlarging the state. This means that on the termination of a call, the local variables cannot always be discarded and the stack discipline for allocation of local variables is no longer adequate.

Indirect Calls in WebAssembly

WebAssembly functions can be called indirectly by referring to their number, rather than their address. For this, functions have to be explicitly entered into a *table*, which is another type of WebAssembly store. In a WebAssembly module, the declaration

```
(table n funcref)
```

where `n` is a constant, allocates statically `n` function references. The elements of the table are populated by:

```
(elem i fn)
```

where `i` is an instruction with an integer result and `fn` the name of a function. The call

```
call_indirect ft
```

pops an index to the table from the stack and calls the function at that index, provided the function type `ft` matches the type of the function in the table. That check is done at runtime.

Following program calls either `plus1` or `plus2` depending on the interactive input:

In [ ]:
```
def runpywasm(wasmfile):
    def write(s, i): print(i)
    def read(s): return int(input())
    import pywasm
    vm = pywasm.load(wasmfile, {'lib': {'write': write, 'read': read}})
```

In [ ]:
```
%%writefile indirect.wat
(module
  (import "lib" "write" (func $write (param i32)))
  (import "lib" "read" (func $read (result i32)))
  (func $plus1 (param $x i32) (result i32)
    i32.const 1
    local.get $x
    i32.add)
  (func $plus2 (param $x i32) (result i32)
    i32.const 2
    local.get $x
    i32.add)
  (func $program
    call $read ;; push function parameter on stack
    call $read ;; push function index on stack
    call_indirect (param i32) (result i32)
    call $write)
  (table 2 funcref)
  (elem (i32.const 0) $plus1)
  (elem (i32.const 1) $plus2)
  (start $program)
)
```

In [ ]:
```
!wat2wasm indirect.wat
```

In [ ]:
```
runpywasm("indirect.wasm")
```

The grammar of WebAssembly programs is extended accordingly:

```
instr ::= ... | "call_indirect" func_type
module ::= "(" "module" {import} {global} {func} [table] {elem} [memory] [start] ")"
table ::= "(" "table" num "funcref" ")"
elem ::= "(" "elem" instr name ")"
```

Procedure as Values in WebAssembly

As procedures are referred to by table indices of type `i32`, these can be assigned and passed as parameters like other `i32` values. Here, procedure `p` in WebAssembly takes a single `i32` parameter:

```
procedure seven() → (r: integer)
    r := 7
procedure nine() → (r: integer)
    r := 9
procedure p(f: () → (r: integer))
    var a: integer
        a ← f(); write(a)
program q
    p(seven)
    p(nine)
```

```
(func $seven (result i32)
  i32.const 7)
(func $nine (result i32)
  i32.const 9)
(func $p (param $f i32)
  local.get $f
  call_indirect (result i32)
  call $write)
(func $program
  i32.const 0
  call $p
  i32.const 1
  call $p)
(table 2 funcref)
(elem (i32.const 0) $seven)
(elem (i32.const 1) $nine)
```

Case-Statements

Case-statements allow a case analysis for all values of an enumeration type to be expressed symmetrically, as the order of the individual cases does not matter.
Case-statements can be implemented by *computed jumps*, where the location of the code of a case is looked up in a *jump table*.

```
var c: (green, yellow, red)
...
case c of
    green: ...
    yellow: ...
    red: ...
```

The benefit compared to nested if-then-else statement is that independently of the number of cases, the selection takes constant time. This scheme is suitable for case statements with enumeration types, disjoint unions, and with subranges of integers. Historically, Pascal and C have case statements to allow efficient processing of ASCII characters in scanners, in regular expressions search, etc.

```
case ch of
    'a' .. 'z', 'A', .. 'Z': Identifier()
    '0' .. '9': Number()
    '(': sym := LPAREN; getCh()
    ')': sym := RPAREN; getCh()
```

Jump Tables in WebAssembly

The instruction `br_table l₀ ... lₙ₋₁ lₙ` pops an integer, say `i`, from the stack and jumps to label `lᵢ` if `0 ≤ i < n` and to `lₙ`, the default label, otherwise. Like with `br l` and `br_if l`, the labels have to refer to enclosing blocks.

```
instr ::= ... | "br_table" {name} name
```

The default label is used for the `else` part of a case-statement. If the cases are not contiguous, the missing cases also have labels to the block with the `else` part.

```
program p
  var i: integer
    i ← read()
    case i of
        1: write(1)
        3: write(3)
        4: write(4)
    else: write(0)
```

In [ ]:
```
%%writefile jumptable.wat
(module
  (import "lib" "write" (func $write (param i32)))
  (import "lib" "read" (func $read (result i32)))
  (func $program
    (local $i i32)
    call $read
    local.set $i
    block $done
      block $else
        block $4
          block $3
            block $1
              local.get $i
              i32.const 1
              i32.sub
              br_table $1 $else $3 $4 $else
            end
            i32.const 1
            call $write
            br $done
          end
          i32.const 3
          call $write
          br $done
        end
        i32.const 4
        call $write
        br $done
      end
      i32.const 0
      call $write
    end)
  (start $program)
)
```

In [ ]:
```
!wat2wasm jumptable.wat
```

In [ ]:
```
runpywasm("jumptable.wasm")
```

If the case labels do not appear in ascending order in the source, the compiler needs to sort them. If the lowest label is not `0`, an addition (subtraction) is inserted before indexing the jump table. The size of the jump table is the difference between the largest and smallest label. That difference can be too large for jump tables:

```
case x of
  0: …
  1000: …
  1000000: …
end
```

The compiler then has to resort to generating nested if-statements instead.

For-Loops

In C and related languages, the for-loop

```
for (expr₁; expr₂; expr₃) stat
```
is equivalent to:

```
expr₁; while (expr₂) {stat expr₃;}
```
Syntactically, `expr₁`, `expr₂`, `expr₃` are expressions, although C allows expressions to be have effects on variables and to be used as statements. Commonly, `expr₁` is an initialization of a loop variable, `expr₂` is a relational expression, and `expr₃` updates the loop variable. This transformation can be done on the abstract syntax tree, before code generation.

In Pascal, the for-loop

```
for v := exp₁ to exp₂ do stat
```
is equivalent to

```
t₁ := exp₁, t₂ := exp₂;
if t₁ <= t₂ then
    begin v := t₁; stat;
        while v <> t₂ do begin v := v + 1; stat end
    end
```
where `t₁`, `t₂` are auxiliary variables. The value of `v` is undefined at the end of the loop; this is meant to allow `v` to be kept in a register and that register not saved in memory.

One difference to C-style for-loops is that the expressions are evaluated only once. Thus, while in C the loop

```
n = 10; for (i = 0; i < n; i++) n++;
```
never terminates, in Pascal the equivalent loop terminates after `10` iterations:

```
n := 10; for i := 1 to n do n := n + 1
```
Guaranteeing the termination of for-loops is meant to contribute to safer programs.

The other difference to C-style for-loops is that the loop variable is not incremented past the final value. The range of `unsigned char` in C and `byte` in Pascal is `0 .. 255`. While in C the loop

```
unsigned char i;
unsigned char a[256];
for (i = 0; i <= 255; i++) a[i] = 0;
```
never terminates as `i` is incremented past `255` and becomes `0`, in Pascal the equivalent loop terminates as expected:

```
var i: byte;
var a: array[0..255] of byte;
begin
  for i := 0 to 255 do a[i] := 0
end
```