5. The Construction of a Parser

Emil Sekerinski, **McMaster University, February 2022**

This and following chapters are based on the accompanying P0 compiler. The P0 language is influenced by Pascal, a language designed for ease of compilation. The following example illustrates variable and procedure declarations, value and result parameters, while-loops, and input-output in P0:

```
procedure quotrem(x, y: integer) → (q, r: integer)
    q, r := 0, x
    while r ≥ y do // q × y + r = x ∧ r ≥ y
        r, q := r - y, q + 1

program arithmetic
    var x, y, q, r: integer
      x ← read(); y ← read()
      q, r ← quotrem(x, y)
      write(q); write(r)
```

Procedures can also be called recursively:

```
procedure fact(n: integer) → (f: integer)
    if n = 0 then f := 1
    else
        f ← fact(n - 1); f := f × n

program factorial
    var y, z: integer
        y ← read(); z ← fact(y); write(z)
```

The full compiler is documented in a separate set of notebooks. This chapter covers the design decisions of the parser and the implications on the modularization:

- scanning and parsing
- coping with syntactic errors
- context dependencies
- data representation at run-time

Scanning and Parsing

Suppose identifiers were defined by:

```
identifier → letter { letter }
letter → 'a' | … | 'z'
```

This makes `if` and `then` identifiers that can be used as variables, which would make programs unreadable and difficult to parse. As as remedy, we define

```
keyword → 'if' | 'then' | ...
symbol → keyword | identifier | ...
```

but when scanning for a symbol, preference is given to `keyword` over `identifier`. That is, the ambiguity of `|` is resolved in favour of the first alternative. Preference of the first alternative can be made explicit by using an asymmetic choice operator, E_1 `/` E_2, as in:

```
symbol → keyword / identifier / ...
```

Alternatively, a subtraction operator E_1 `-` E_2 can be introduced for EBNF, allowing the definition:

```
identifier → (letter { letter }) – keyword
```

Since ambiguities rarely occur, we will treat these informally, noting that a formal treatment is possible. Above, `'if'` is a shorthand for `'i' 'f'`. In general, $'a_1a_2\ldots'$ is a shorthand for $'a_1'$ $'a_2'$ ….

A nuisance, as evident form previous grammars, is the need to explicitly allow white space. The approach taken since Algol 60 is to use

two grammars, one for symbols in terms of characters and one for programs in terms of symbols. For P0, the grammar of programs in terms or characters is:

```
program → {symbol}
symbol → { ' ' | comment} ( { '\n' {' ' | comment} } | identKW | number | '×' | '+' | '-' |
                    '=' | '≠' | '<' | '≤' | '>' | '≥' | ';' | ',' | ':' | ':=' | '.' | '..' |
'¬' | '(' |  ')' | '[' | ']' |
                    '{' | '}' | '←' | '→' | '#' | 'C' | '∪' | '∩' | '∈' | '⊆' | '⊇')
identKW → identifier / keyword
keyword → 'div' | 'mod' | 'and' | 'or' | 'if' | 'then' | 'else' | 'while' | 'do' | 'const' |
'type' | 'var' |
                     'set' | 'procedure' | 'program'
identifier → (letter { letter | digit }) − keyword
number → digit {digit}
comment ::= '//' {character − '\n'}
letter → 'a' | … | 'z' | 'A' | … | 'Z'
digit → '0' | … | '9'
```

We allow `character` to be any Unicode character; `\n` is the end-of-line character.

The task of the *scanner* is to recognize a sequence of characters according to the grammar of symbols and to produce a sequence of symbols. Conceptually, the synthesized attribute of `program` is a sequence of elements of type:

```
Symbol = IDENT(string) | NUMBER(integer) | TIMES | DIV | MOD | PLUS | MINUS | AND | OR |
    EQ | NE | LT | LE | GT | GE | SEMICOLON | COMMA | COLON | BECOMES | PERIOD | DOTDOT |
    NOT | LPAREN | RPAREN | LBRAK | RBRAK | LBRACE | RBRACE | LARROW | RARROW | CARD |
    COMPLEMENT | UNION | INTERSECTION | ELEMENT | SUBSET | SUPERSET | IF | THEN | ELSE |
    WHILE | DO | CONST | TYPE | VAR | PROCEDURE | PROGRAM | INDENT | DEDENT | EOF
```

White space is recognized and swallowed by the scanner. The symbol `NUMBER` has an associated integer and the symbol `IDENT` has an associated string. To simplify further processing, at the end of the input the scanner generates the symbol `EOF`. For example, the character sequence

```
while r ≥ 1 do
    r := r - 1
```

starts with the following sequence of symbols:

```
WHILE IDENT('r') GE NUMBER('1') DO INDENT IDENT(r) BECOMES
```

The above grammar for symbols is still ambiguous: keywords are sequences of characters, as are identifiers, so the input `iffy := 5` can be recognized as starting with the four identifiers, `i`, `f`, `f`, `y`, or as starting with the keyword `if` followed by identifier `fy` or some other combination. This ambiguity is resolved according to the *principle of the longest match:* identifier `iffy` is the longest matching symbol here. The same principle applies to `:=`, which is recognized as `BECOMES` and not as `COLON` followed by `EQ`.

The P0 grammar can now be defined in terms of symbols, for example:

```
statement → IF expression THEN statement [ELSE statement]
```

For readability it is common to use the character sequences of the symbols, now written in double quotes, rather than the symbols themselves, so above production would be written as:

```
statement → "if" expression "then" statement ["else" statement]
```

The P0 grammar is then:

```
selector → { "[" expression "]" | "." ident}
factor → ident selector | integer | "(" expression ")" | "{" [expression {"," expression}] "}"
| ("¬" | "#" | "C") factor
term → factor {("×" | "div" | "mod" | "∩" | "and") factor}
simpleExpression → ["+" | "-"] term {("+" | "-" | "∪" | "or") term}
expression → simpleExpression
    {("=" | "≠" | "<" | "≤" | ">" | "≥" | "∈" | "⊆" | "⊇") simpleExpression}
statementList → statement {";" statement}
statementBlock → statementList {statementList}
statementSuite → statementList | INDENT statementBlock DEDENT
statement →
    ident selector ":=" expression |
    ident {"," ident} (":=" expression {"," expression} |
```

```
            "←" ident "(" [expression {"," expression}] ")") |
        "if" expression "then" statementSuite ["else" statementSuite] |
        "while" expression "do" statementSuite
    type →
        ident |
        "[" expression ".." expression "]" "→" type |
        "(" typedIds ")" |
        "set" "[" expression ".." expression "]"
    typedIds → ident ":" type {"," ident ":" type}
    declarations →
        {"const" ident "=" expression}
        {"type" ident "=" type}
        {"var" typedIds}
        {"procedure" ident "(" [typedIds] ")" [ "→" "(" typedIds ")" ] body}
    body → INDENT declarations (statementBlock | INDENT statementBlock DEDENT) DEDENT
    program → declarations "program" ident body
```

The grammar of symbols in terms of characters is regular but the grammar of programs in terms of symbols is context-free.

process	input element	algorithm	grammar
lexical analysis	character	scanner	regular
syntax analysis	symbol	parser	context-free

For implementing the scanner, the techniques for recognizing regular expressions can be used. However, the simplicity of the grammar of P0 symbols allows to use the principles of recursive descent for lexical analysis as well.

In the P0 parser, the procedure for reading the next symbol is called `getSym()`; it assigns to global variable `sym` and in case `sym` is `NUMBER` or `IDENT`, additionally to `val`. Procedure `getSym()` in turn calls the scanner procedure `getChar()`, which assigns to global variable `ch` the next character of the input. The scanner does not explicitly construct the sequence of symbols as a data structure:

```
IDENT = 1; NUMBER = 2; TIMES = 3; DIV = 4; MOD = 5; PLUS = 6 ...

def getChar():
    if index == len(source): ch = chr(0)
    else: ch, index = source[index], index + 1

KEYWORDS = \
    {'div': DIV, 'mod': MOD, 'and': AND, 'or': OR, 'if': IF, 'then': THEN,
     'else': ELSE, 'while': WHILE, 'do': DO, 'const': CONST, 'type': TYPE,
     'var': VAR, 'set': SET, 'procedure': PROCEDURE, 'program': PROGRAM}

def identKW():
    global sym, val
    start = index - 1
    while ('A' <= ch <= 'Z') or ('a' <= ch <= 'z') or \
          ('0' <= ch <= '9'): getChar()
    val = source[start:index-1]
    sym = KEYWORDS[val] if val in KEYWORDS else IDENT

def getSym():
    global sym
    while  ch <= ' ': getChar()
    if ('A' <= ch <= 'Z') or ('a' <= ch <= 'z'): identKW()
    elif '0' <= ch <= '9': number()
    elif ch == '×': getChar(); sym := TIMES
    elif ch == '+': getChar(); sym := PLUS
    ...
    elif ch == chr(0): sym = EOF
    else: mark('illegal character')
```

Procedure `getChar()` assigns the next character to `ch`, or assigns `chr(0)` at the end of the source. Procedure `getSym()` assigns the next symbol to `sym`, or assigns `EOF` at the end of the source. This way, `ch` and `sym` always have a valid value.

Coping with Errors

A lexical error occurs when detecting illegal character, a malformed sequence of characters (e.g. a malformed floating point number), and constants that cannot be represented (e.g. a too larger number)

A syntactic error occurs when expecting

- terminal `'x'` and the next input symbol is not `'x'` or
- alternative E_1 | E_2 | … and the next input symbol is not in the first set of any E_i.

A type error occurs when type-checking detects a mismatch of types (e.g. comparing values of distinct types).

Code generation may also produce errors if limits of the target language are exceeded.

On detecting an error, the compiler can abort parsing: this can be achieved in a recursive descent parser by assigning a special error symbol to `sym` that causes all subsequent tests to fail and gets the parser out of the recursion. Alternatively, an exception can be raised. Either way, there is no overhead for parsing correct inputs. Aborting compilation is reasonable for short inputs, e.g. formulae in a spreadsheet, search queries, cells in notebooks.

In the P0 compiler, this is implemented by the procedure `mark`. To report the position of an error more accurately, procedure `getChar` keeps track of the current line and position within the line. Variables `line`, `pos` are updated with the current location in the source and `lastline`, `lastpos` are updated with the location of the previously read character:

```
def getChar():
    global line, pos, ch, index
    if index == len(source): ch = chr(0)
    else:
        ch, index = source[index], index + 1
        if ch == '\n':
            pos, line = 0, line + 1
        else:
            pos = pos + 1

def mark(msg: str):
    raise Exception('line ' + str(line) + ' pos ' + str(pos) + ' ' + msg)
```

The procedure parsing

```
   factor ::= ident selector | integer | "(" expression ")" | "¬" factor
```

illustrates the two cases: after parsing `(` and `expression`, the next symbol must be `)` symbol, otherwise `mark(") expected")` is called in that case. When parsing `factor`, the next symbol must be in the first set of one of the alternatives, otherwise `mark("expression expected")` is called:

```
def factor():
    if sym == IDENT: getSym(); selector()
    elif sym == NUMBER: getSym()
    elif sym == LPAREN:
        getSym(); expression()
        if sym == RPAREN: getSym()
        else: mark(") expected")
    elif sym == NOT:
        getSym(); factor()
    else: mark("expression expected")
```

Error reporting would still not be satisfactory as the following example shows:

```
program p
  var x: integer
    x := (

   line 4 pos 0 expression expected
```

Variable `sym` is the next symbol and variable `ch` is the next character after the next symbol, which after parsing `'('` would be the character for the end of the input, causing a position too far ahead to be reported. The position of the current symbol rather than the next symbol should be reported:

```
def getChar():
    global line, lastline, pos, lastpos, ch, index
    if index == len(source): ch = chr(0); pos = 1
    else:
        lastpos = pos
        if ch == '\n':
            pos, line = 1, line + 1
        else:
            lastline, pos = line, pos + 1
        ch, index = source[index], index + 1

def mark(msg):
    raise Exception('line ' + str(lastline) + ' pos ' + str(lastpos) + ' ' + msg)
```

Alternatively to aborting the compilation right away, on detecting an error, the parser can also continue parsing under a suitable *hypothesis* that a symbol is missing, is misspelled, or can be ignored. This gives the programmer more feedback at once. The hypotheses depend on the language but some examples include:

- `;` is likely to be missing and can be inserted automatically,

- `+` is unlikely to be missing.

However, both are identical to the parser. We give a simple strategy for adding error recovery to P0.

Missing symbol: symbols like a closing `)` in an expression and `;` in statement sequence or at end of declaration are called *weak symbols*. Parsing can be resumed after issuing an error message, for example:

```
   factor ::= ident selector | integer | "(" expression ")" | "not" factor.
```

```
def mark(msg: str):
    print('line ' + str(lastline) + ' pos ' + str(lastpos) + ' ' + msg)

def factor():
    if sym == IDENT:
        ...
    elif sym == NUMBER:
        ...
    elif sym == LPAREN:
        getSym(); expression()
        if sym == RPAREN: getSym()
        else: mark(") expected")
```

Wrong, unexpected symbol: The input is skipped until a *synchronization symbol* is read and parsing can be resumed.

Suppose we do error recovery at the beginning of the procedure parsing `A`. Good synchronization symbols are `follow(A)`. However, if `follow(A) = {";"}`, as common for statements, and `;` is missing, then a large part of the input would be skipped.

We therefore extend the synchronization set by *strong symbols* that are unlikely to be forgotten. The strong symbols of P0 are:

- `const`, `type`, `var`, `procedure`: start a new declaration
- `if`, `while`: start new statement

We arrive at following scheme:

```
procedure A()
        if sym ∉ first(A) then
            mark("first(A) expected")
            while sym ∉ first(A) ∪ follow(A) ∪ strongsymbols do
                getSym()
        …
```

Suitable synchronization points in the P0 compiler are:

- `factor`, `statement`, `declaration`

Constants for the synchronization sets are defined, for example for `factor`:

```
FIRSTFACTOR = {IDENT, NUMBER, LPAREN, NOT}
FOLLOWFACTOR = {TIMES, DIV, MOD, AND, OR, PLUS, MINUS, EQ, NE, LT, LE, GT, GE,
               COMMA, SEMICOLON, THEN, ELSE, RPAREN, RBRAK, DO, PERIOD}
STRONGSYMS = {CONST, TYPE, VAR, PROCEDURE, WHILE, IF, EOF}

def factor():
    if sym not in FIRSTFACTOR:
        mark("expression expected")
        while SC.sym not in FIRSTFACTOR | FOLLOWFACTOR | STRONGSYMS: getSym()
    if sym == IDENT:
        …
    elif sym == NUMBER:
        …
    elif sym == LPAREN:
        getSym(); expression()
        if sym == RPAREN: getSym()
        else: mark(") expected")
    …
```

Error reporting would still not be satisfactory:

```
program p
  var x: integer
    x := (

   line 4 pos 0 expression expected
   line 4 pos 0 ) expected
```

Spurious errors are reported in the form of multiple error messages at one position. A simple solution is to report only one error at one

position:

```
def mark(msg: str):
    if lastline > errline or lastpos > errpos:
        print('error: line', lastline, 'pos', lastpos, msg)
    errline, errpos = lastline, lastpos
```

Context Dependencies

Statically typed programming languages, by their very definition, create context-dependencies: each variable can only be used in the scope of its declaration and according to its declared type. In principle, context-dependencies can be specified with attribute grammars. For efficiency, context-dependencies are handled by a global *symbol table*, that is updated during parsing to maintain the context at the point of analysis. A symbol table has two main operations:

- on the declaration of an identifier, it is stored together with its properties in the symbol table;
- on each occurrence of an identifier, its properties are looked up in the symbol table.

Conceptually, the symbol table is a mapping (dictionary) from identifiers to its properties. As the order of declaration in the source is used for code generation, the declarations are kept as *list of entries* instead. The scoping rules of programming languages allow declarations to be nested. Thus there is one list of entries for each *level* and the symbol table is a list of lists.

For example, the P0 fragment with arrays and parameters

```
const N = 10
type T =  [1 .. N] → integer
var x: T
var y: boolean
var z: (f: integer, g: boolean)
procedure q(v: boolean) → (r: integer)
  var z: boolean
```

results in a list of two lists, the first being the declarations at level `0`, the outermost declarations, and the second list being the declarations at level `1` in `#!pascal: procedure q`:

```
Const(name = N, tp = Int, val = 10)
Type(name = T, val = Array(lower = 1, length = 10, base = Int))
Var(name = x, lev = 0, tp = Array(lower = 1, length = 10, base = Int))
Var(name = y, lev = 0, tp = Bool)
Var(name = z, lev = 0, tp = Record(fields = [Var(name = f, tp = Int), Var(name = g, tp =
Int)]))
Proc(name = q, lev = 0, par = [Var(tp = Bool)], res = [Var(tp = Int)])

Var(name = v, lev = 1, tp = Bool)
Var(name = r, lev = 1, tp = Int)
Var(name = z, lev = 1, tp = Bool)
```

The symbol table is initially populated with *standard identifiers*. In P0, these are `boolean`, `integer`, `true`, `false`, `read`, `write`, and `writeln`:

```
Type(name = boolean, val = Bool)
Type(name = integer, val = Int)
Const(name = true, tp = Bool, val = 1)
Const(name = false, tp = Bool, val = 0)
StdProc(name = read, lev = 0, par = [], res = [Var(tp = Int)])
StdProc(name = write, lev = 0, par = [Var(tp = Int)], res = [])
StdProc(name = writeln, lev = 0, par = [], res = [])
```

Data Representation at Run-time

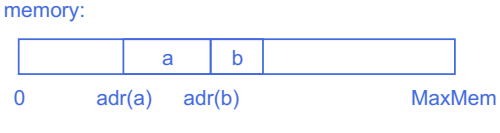
A compiler has to map the data types of the source language to the linear memory that processors can address. Commonly, the memory is a sequence of bytes addressed starting with `0`:

```
var memory: [0 .. MaxMem] → byte
```

With *sequential allocation* consecutively declared variables are allocated with monotonically increasing or decreasing addresses.

- Let `size(T)` be the size of variables of type `T` in bytes.
- Let `adr(a)` be the address of variable `a`.

For declarations `#!pascal: var a: T` and `#!pascal: var b: U`, monotonically increasing sequential allocation leads to the layout the right.



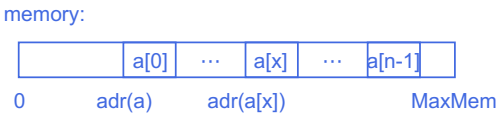
Consider a one-dimensional array `a` with lower bound zero and length given by constant `n`,

```
type T = [0 .. n - 1] → T₀
    var a: T
```

then:

$$size(T) = n \times size(T_0)$$
$$adr(a[x]) = adr(a) + x \times size(T_0)$$

Sequential allocation of `a` is shown to the right. The computation of `adr(a[x])` requires one multiplication with a constant, `size(T₀)`, and one addition.



Consider an `r`-dimensional array with all lower bounds zero,

```
type T = [0 .. nᵣ₋₁ - 1] × … × [0 .. n₁ - 1] × [0 .. n₀ - 1] → T₀
               = [0 .. nᵣ₋₁ - 1] → … → [0 .. n₁ - 1] → [0 .. n₀ - 1] → T₀
    var a: T
```

then:

$$size(T) = n_{r-1} \times \ldots \times n_1 \times n_0 \times size(T_0)$$
$$adr(a[x_{r-1}, \ldots, x_1, x_0]) = adr(a[x_{r-1}]\ldots[x_1][x_0])$$
$$= adr(a) + x_{r-1} \times n_{r-2} \times \ldots \times n_1 \times n_0 \times size(T_0) + \ldots + x_1 \times n_0 \times size(T0) + x_0 \times size(T_0)$$

That is, a multi-dimensional array is an array of arrays and the computation of `adr(a[xᵣ₋₁, …, x₁, x₀])` requires consequently `r` multiplications with constants and `r` additions.

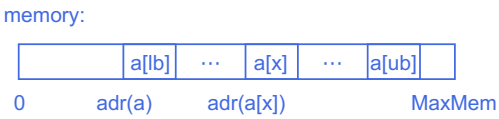
Consider now a one-dimensional array with an arbitrary lower bound, as in P0,

```
type T = [lb .. ub] → T₀
var a: T
```

then:

$$len(T) = ub - lb + 1$$
$$size(T) = len(T) \times size(T_0)$$
$$adr(a[x]) = adr(a) + (x - lb) \times size(T_0)$$
$$= adr(a) - lb \times size(T_0) + x \times size(T_0)$$

Sequential allocation of `a` is shown to the right. Note that `lb × size(T₀)` is a compile-time constant:



- Suppose `adr(a)` is know at compile-time, as for global variables. Then `adr(a) - lb × size(T₀)` can be computed at compile-time and no additional overhead is involved compared to arrays with a lower bound of zero.
- Similarly, if `adr(a)` is relative to a base address, as for local variables allocated on the stack, the base address can be adjusted and no additional overhead is involved compared to arrays with a lower bound of zero.
- Suppose `adr(a)` is not known at compile-time, as for heap allocated variables. Then `lb × size(T₀)` must be subtracted at run time, thus requiring one additional operation.

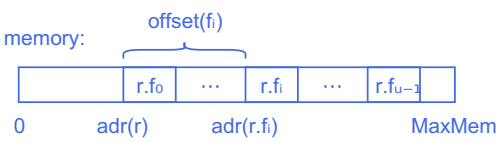
A range check always requires in any case one additional operation.

While the index of an array element is a computed value in general, the field of a record is named and does not need to be computed, simplifying the address calculation. In general, if

```
type T = (f₀: T₀, f₁: T₁, … , fᵤ₋₁: Tᵤ₋₁)
    var r: T
```

then:

$$size(T) = size(T_0) + \cdots + size(T_{u-1})$$
$$adr(r.f_i) = adr(r) + offset(f_i)$$
$$offset(f_i) = size(T_0) + \cdots + size(T_{i-1})$$

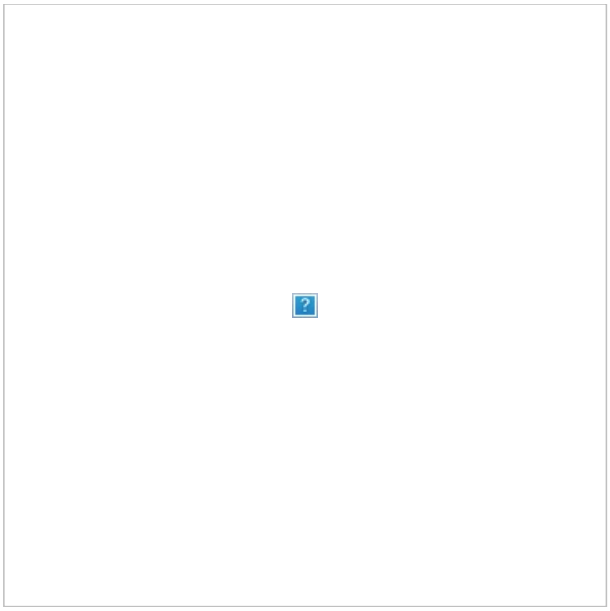


Sequential allocation of `r` is shown to the right.

Strict sequential allocation can cause variable of word size or less to cross word boundaries. This either causes two memory fetch cycles instead of one or may be illegal on some processors. Suppose that a word consists of four bytes, `size(boolean) = 1`, and `size(integer) = 4`:

```
var a: boolean
```

```
var b, c: integer
var d: boolean
```



If processors allow to address bytes on byte addresses, rather than word boundaries, one solution is to *reorder* the declarations. The other is to simply *align* all variables to word boundaries by defining `size(boolean) = 4` (one word). This automatically makes all records and arrays aligned as well.

Historic Notes and Further Reading

As for alignment on ARM processors, load and store instructions must be aligned on older 32-bit processors:

- `LDRB` / `STRB` (load / store byte): address must be byte aligned
- `LDRH` / `STRH` (load / store half word): address must be 2-byte aligned
- `LDR` / `STR` (load / store word): address must be 4-byte aligned

Unaligned access had to be implemented by software by combining a series of accesses. Since the introduction of the ARMv6 architecture 2002, unaligned addresses are allowed through hardware support for above instructions, but not for `LDM` / `STM` (load / store multiple) instructions. Still, the ARM documentation states that unaligned access may be slower as it may require two memory accesses (on currently common 64 or 128-bit buses).

On Intel processors (Sec. 4.1.1) unaligned addresses for words (2 bytes), double words (4 bytes), and quadwords (8 bytes) are allowed, but again with the caveat that it may require two memory accesses.

On RISC-V processors (Sec. 2.2) and MIPS processors addresses of 4-byte words have to be 4-byte aligned.

Some languages like C and Pascal don't define the size of numeric data types but rather allow the compiler to choose the most efficient one for the target processor. The original C report only requires that `size(char) = 1` based on the underlying ASCII character. As leaving the size of numeric data types open impacts portability of programs, other languages are specific. For example, the Java language reference gives the possible ranges of values, which imply:

```
size(byte) = 1, size(short) = 2, size(int) = 4, size(long) = 8
size(float) = 4, size(double) = 8, size(char) = 2
```

Nothing can be implied about the size of `bool`. The C# language reference states explicitly:

```
size(byte) = 1, size(short) = 2, size(int) = 4, size(long) = 8
size(float) = 4, size(double) = 8, size(decimal) = 16, size(char) = 2, size(bool) = 1
```

The Go language reference states about numeric types

```
size(int8) = 1, size(int16) = 2, size(int32) = 4, size(int64) = 8
size(float32) = 4, size(float64)
size(complex64) = 8, size(complex128) = 16
```

and allows `int` to be synonymous with either `int32` or `int64`.