Types and Variables

- Local variables don’t have to be declared, but do it anyway!
- Global variables must be declared.
- An variable that has *not* been declared will automatically be treated as a local variable.
- Icon is dynamically typed. This means that you don’t need to declare the types of variables.
- A variable may contain different types of data at runtime.

```plaintext
local X
X := "hello"  # String
X := 5  # Integer
X := 6.7  # Real
```

Types and Variables...

- You won’t get type errors at compile-time, but you will get them at runtime:

```plaintext
procedure main(args)
  t := "hello" + 4.5
end
```

```
Run-time error 102
File t.icn; Line 6
numeric expected
offending value: "hello"
Trace back:
  main()
  {"hello" + 4.5} from line 2
```

- `type(v)` will return the name (a string) of the type of `v`:

```plaintext
record complex(a,b)
t := "hello"
x := type(t)  # x="string"
t := [5,6,7]
x := type(t)  # x="list"
t := complex(4,5)
x := type(t)  # x="complex"
```

- Some data types are automatically converted to the required type. For example, a string (consisting entirely of digits) can be converted into a number, explicitly or implicitly:

```plaintext
write(5 + "6")  # implicit
write(5+integer("6"))  # explicit
```
**Icon Statements**

while e1 do e2  Evaluate e2 until e1 fails.
until e1 do e2  Same as while not e1 do e2.
repeat e  Evaluate the expression e repeatedly.
break  Jump out of the most closely nested loop.
next  Jump to the beginning of the most closely nested loop.

{}  Compound statement.
a := b  Assignment. Repeated assignments (a := b := c) are also OK.

**Expressions**

There are fundamental differences in the way Pascal & Icon statements are executed:
1. Icon statements are expressions that return values.
2. Icon expression either succeed or fail.
3. Failure doesn’t necessarily mean that something has gone wrong, rather, it means that there is no value to return. numeric("pi") fails because "pi" cannot be converted to number.

**Success & Failure**

i + j  Succeeds and returns the value i + j.
i < j  Succeeds if i < j, in which case j is returned. Fails otherwise.

if e1 then e2 else e3  If-expression. If e1 succeeds then evaluate and return the value of e2, otherwise evaluate and return e3. The else-part is optional.

case e of {
e1 : s1
e2 : s2
...default : s3
}

Similar to repeated if-expression: if e1 then s1 else if e2 then s2 else... else s3. The default-part is optional. e1, e2,... can be arbitrary expressions of arbitrary type, not just scalar constants as in Pascal.
Expressions...

Success & Failure

\textbf{stop(s)} Write \textit{s} and terminate.

\textit{expr1} \ | \ \textit{expr2} Generate the values from \textit{expr1}, then from \textit{expr2}.

\textit{x} \ | \ \textit{y} Generate the \textit{variables} \textit{x} and \textit{y}.

\textit{0} \ | \ \textit{1} Generate the \textit{values} \textit{0} and \textit{1}.

\textbf{if} \textit{i} = (0 \ | \ 1) \textbf{then} write("ok") Write "ok" if \textit{i} is 0 or 1.

\textbf{every} \textit{i} := (0 \ | \ 1) \textbf{do} write (\textit{i}) First write 0 then 1.

Expressions...

Success & Failure...

\textbf{every} (\textit{x} \ | \ \textit{y}) := 0 \ \textit{x} := 0; \ \textit{y} := 0

\textit{x} := \textit{p()} \ | \ \textbf{stop("error")} If \textit{p()} fails, then stop and write "error".

\textit{j} := \textit{i} + 10 This fails if \textit{i} = null.

/\textit{x} Succeeds (and produces null) if \textit{x} = null. Fails otherwise.

/\textit{x} := 0 Assign 0 to \textit{x} if \textit{x} = null.

\textbackslash \textit{x} Succeeds and produces \textit{x} if \textit{x} \neq null. Fails otherwise.

\textbackslash \textit{x} := 0 if \textit{x} \neq null then \textit{x} := 0.