syntax

- **block structure is marked more clearly/verbosely**
- **<<label>> goto label;**
- **; is a terminator not a separator**
- **69 reserved words**
- **types are more sophisticated**
- **arrays have ranges, slices, sliding**
- **type String is array (Positive range <>) of Character;**
- **type operators**
- **pointers are more controlled:**
- **expressions are "dynamic" or "static"**

block structure is marked more clearly/verbosely

```
if ... then ... elsif ... then ... else ... end if;
loop ... exit when ... end loop;
case expr is when value => action; ... end case;
procedure name ... is ... begin ... end name;
for name in reverse 1 .. N loop .. end loop;
while condition loop .. end loop;
null; -- needed *explicitly* in a number of contexts
<<label>>   goto label;
   -- for machine generated code
```

```
case expr is when value => action; ... end case;
    -- when others => default action;
    -- (others must be last, and is ok if none left)
    -- when range => action;
    -- when val1 | val2 => action;
    -- values must be static
    -- *all* possible values must be satisfied
```

```
<<label>>   goto label;
```

```
<<label>>   goto label;
        -- can't go into a if/case/loop or across arms of if/case
```

69 reserved words

```
begin ... exception when EX => action; end; -- like C++ catch
declare ... begin ... end;
varname: type; -- in declarations
cname: constant type := initvalue;
cname: constant := literal number;
  -- *without* nailing down a type...
```

begin ... exception when EX => action; end; -- like C++ catch

-- parallel syntax to Ada case...

-- exceptions are in namespace, if you can name it

-- you can raised it

declare ... begin ... end;

-- declarations ordered, like let*

-- safer name:type; syntax

cname: constant := literal number;

types are more sophisticated

type name is (enum, values);

type newname is new oldname; -- explicit convert only

constrained types

type name is record ... end record; -- just like struct

type name is tagged ...; -- inheritable, extendable

type newname is new name with ...; -- with null record

subtype days is Integer range 1..31;

aggregate construction

constrained types

```
type name is digits 7;
-- pick a predefined type matching constraint
type name is range -1_000 .. +1_000;
-- ranges can be null, can be dynamic
```

```
subtype days is Integer range 1..31;
    -- not a newtype, just a named constraint
    -- most places a range can be used explicitly
```

aggregate construction

```
-- positional (4, Jul, 1776);
-- by name (month=>jul, day=>4, year=>1776);
```

arrays have ranges, slices, sliding

```
type name is array (a..b) of type;
type String is array (Positive range <>) of Character;
name(0..2) := (3,2,1);
name(3..5) := "XYZ";
name(4..5) = ('X', 'Y', others => ' ');
array assignment between *same* type, not merely conforming
-- but if the name matches, it's enough that the *dimensions*
-- match, assignment will "slide".
```

```
type String is array (Positive range <>) of Character;
    -- <> is "box" giving an "unconstrained" array
    -- useful in arguments, the range supplied later
```

```
type String is array (Positive range <>) of Character;
    type Mat is array (Integer range <>, Integer range <>) of Float
    array of array AOA(i)(j)
    multi dimensional array MDA(i,j)
```

```
type Mat is array (Integer range <>, Integer range <>) of Float
    -- must nail *both* or *neither* in subtype
    -- can even "deduce" them in place:
    U2: constant Mat := ((1.0,0.0),(0.0,1.0));
    -- but this uses Integer’first as bound!
    -- so use Positive range <> instead...
```

type operators

all discrete types (including float) have

-- name’Length name’First name’Last name’Range

-- name’Succ(value) name’Pred(value)

non-reals also have

-- name’Pos(value)=int name’Val(int)=value

pointers are more controlled:

```
type name_ptr is access name;
var: name_ptr;
var := new name;
var.part  (or var.all.part) var.all
```

pointers are more controlled:

- **arguments are in, out, in out**
- **functions have a return value, procedures don’t**
- **-- so functions can be used in expressions**
- **strongly typed assignment var:=expression;**

strongly typed assignment var:=expression;

- **both sides must have same type**
- **constraints on var must be satisfied**
- **var1,var2 := expression -- evaluates expression twice!**

raw syntax:

’x’ is a character

"xyz" is a string of characters

-- "A""B" is (’A’,’"’,’B’)

-- must fit on a single line and not contain control characters

numbers

identifiers

comments

one dimensional array concatenation: "foo" & "bar" = "foobar"

numbers

- **35, 1_024, 1e6 (integer!), 3.14159_26536 (not 1. or .1)**
- **base notation: 2#111# 14#ABC# 2#11.1#e11**
- **-- (base and exp always decimal)**
- ***literal numbers are never negative***

- identifiers
  - lower case is same as upper except in character strings and literals
  - iso-latin-1 so accents in identifiers are ok
  - identifiers must begin with letters
  - max identifier len must be at least 200 characters, all significant

comments

- **in comments, *anything* is ok**
- **comments begin with -- *only*, like // comments**

one dimensional array concatenation: "foo" & "bar" = "foobar"

-- also takes an element as lhs or rhs (or both)

-- "line 1" & Ada.Characters.Latin_1.LF;

numeric operators

arithmetic: types of left and right must match, same type is produced

Integer only:

(exp) ** if lhs is Integer, rhs is non negative integer

boolean (True,False) return (types match): = /= < <= > >=

conversion from float to intager always rounds

-- halfway values round away from zero

precedence:

order of operations is *not* specified.

arithmetic: types of left and right must match, same type is produced

+ - unary or binary

binary

/ binary; integer / trucates towards 0

abs unary

binary

Integer only:

- **rem - remainder (sign of lhs)**
- **mod - modulo (sign of rhs) but see modular types**
- **also: (A+B) mod n = (A mod n)+(B mod n) mod n**

precedence:

lowest: comparators

binary + -

unary + -

/ mod rem

highest: ** abs

/ mod rem

/ mod rem

ab**c is forbidden (similarly ** with abs) without parens.**

note that -AB = -(A**B), not (-A)**B**

boolean operators: (bool x bool -> bool)

not (precedence of abs)

and or xor (equal, but lower than comparators)

-- so can’t be mixed without parens to disambiguate

ranges scalar in, not in range (precedence with comparators)

shortcuts

and then (like C &&), or else (like C ||)

-- these are *not* strictly operators -- they can’t be overloaded

overloading: any operator in "" can be a function name.

-- can’t change syntax

-- if = returns boolean, /= is created implicitly and can’t be replaced

type conversion:

Float(I) *converts* I

Integer’(I) qualifies I

double conversion

```
double conversion
    a: float;
    procedure x(r: in out integer);
    x(Integer(a)); -- a converted, passed, converted back...
```