Programming Languages: Introduction to Ada

Jan Daciuk

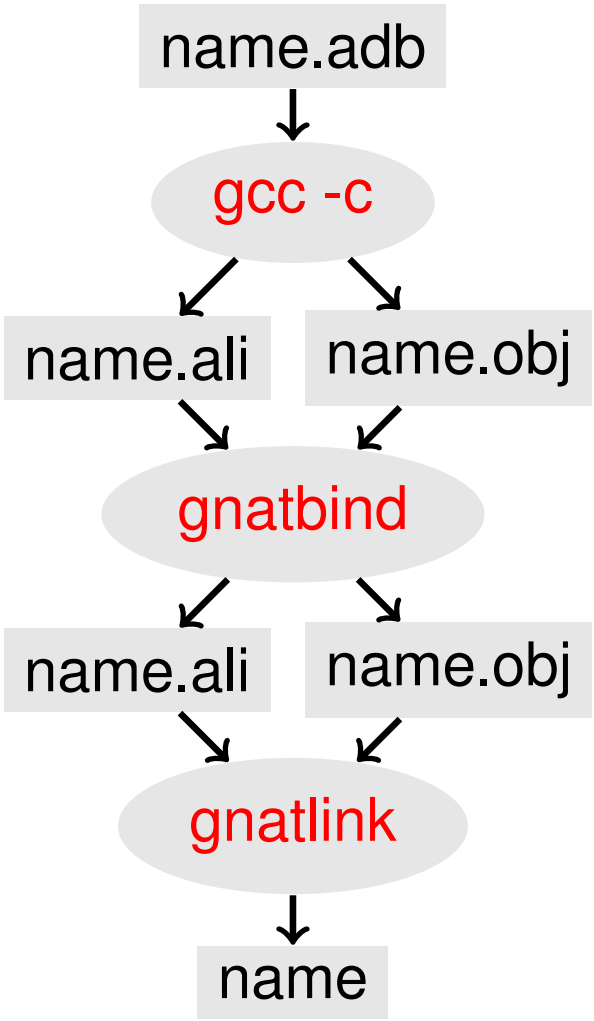
November 23, 2021

FACULTY OF ELECTRONICS, TELECOMMUNICATIONS AND INFORMATICS
POLITECHNIKA GDAŃSKA

Supplementary Reading

- Ada95 Reference Manual /usr/share/doc/ada-reference-manual/html/aarm95tc1/AA-TOC.html
- Ada Syntax Diagrams http://cui.unige.ch/isi/bnf/Ada95/BNFindex.html
- Ada Programming http://en.wikibooks.org/wiki/Ada_Programming
- GNAT User's Guide `/usr/share/doc/gnat-4.4-doc/gnat_ugn.html`
- Using the GNAT Programming Studio /usr/share/doc/gnat-gps/html/index.html

Compilation – Simple Program in One File



source file

compilation

Ada library information, obj

binding

Ada library information, obj

linking

executable program

Compilation – Simple Program in One File

Actually, so many commands are not really necessary. One commands invokes them all:

```
~$ gnatmake name.adb
gcc-4.4 -c name.adb
gnatbind -x name.ali
gnatlink name.ali
```

After that invocation, we get an executable program.

Compilation – Several Modules

producer.ads

```
package Producer is
  procedure Produce(...);
  procedure Send(...);
end Producer;
```

Consumer.ads

```
package Consumer is
  procedure Get(...);
  Procedure Consume(...);
end Consumer;
```

producer.adb

```
package body Producer is
...
end Producer;
```

consumer.adb

```
package body Consumer is
...
end Consumer;
```

gmain.adb

```
with...; use...;
procedure Gmain is
...
end Gmain;
```

Compilation – Several Modules

Modules are compiled in arbitrary order:

```
gcc -c gmain.adb
gcc -c producent.adb
gcc -c konsument.adb
```

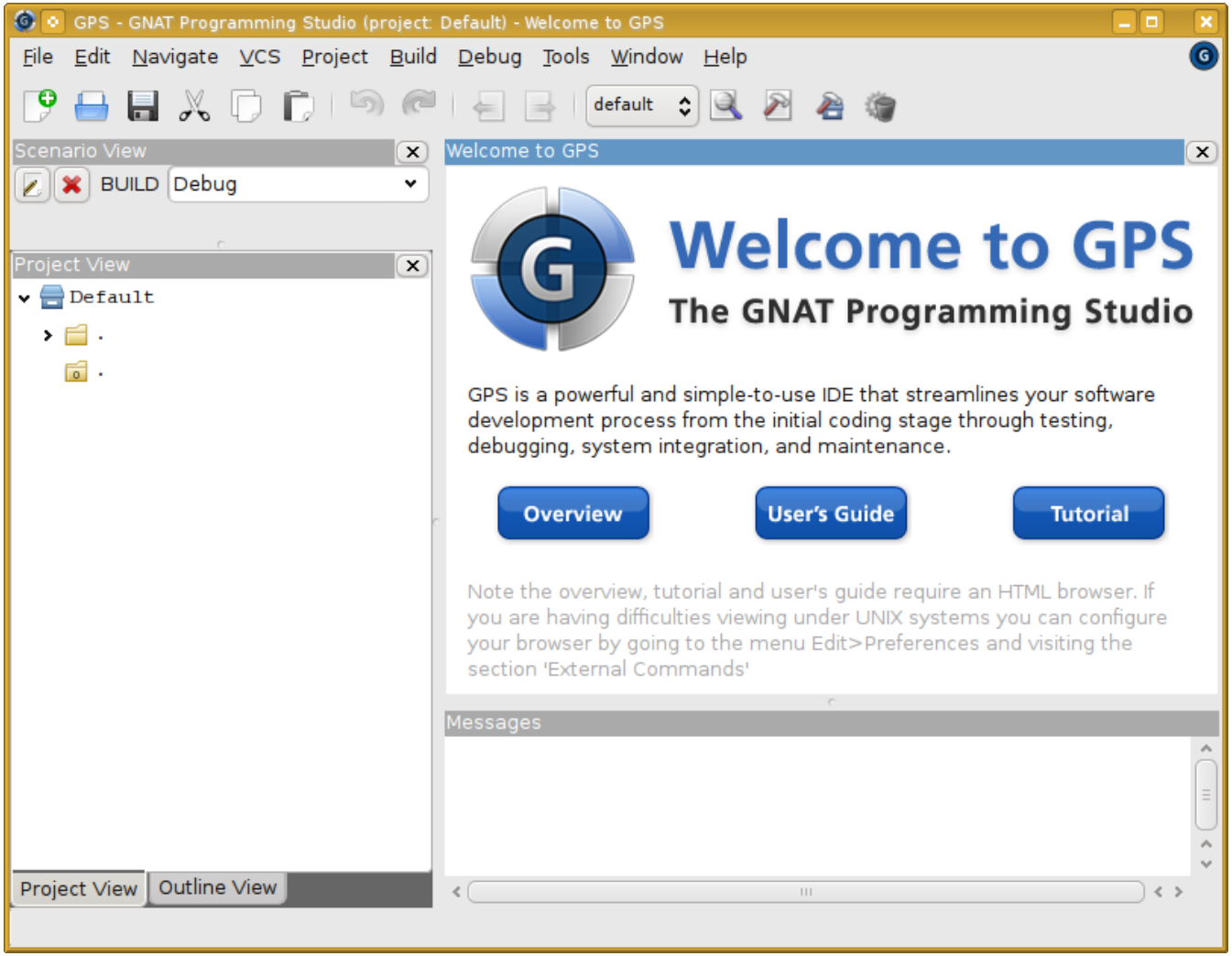
Next they are bound and linked:

```
gnatbind gmain
gnatlink gmain
```

One can economize much by simply typing:

```
gnatmake gmain
```

GNAT Programming Studio



Types

- enumeration **type** X **is** (A, B,...);
- logical BOOLEAN: FALSE i TRUE
- integer INTEGER, SHORT_INTEGER i LONG_INTEGER
- character CHARACTER
- real with precision
  - relative **type** X **is digits** *n* **range** *f1*..*f2*
  - absolute **type** X **is delta** *f* **range** *f1*..*f2*
- subtypes (restricted types) **subtype** X **is** Y **range**...
- records **type** X **is record**...**end record**
- arrays **type** X **is array** (*l*..*r*) **of**
- strings STRING (as a character array)
- pointers **type** X **is access** Y

Operators

- raising to power **
- absolute value **abs**
- multiplication and division *, /, **rem**, **mod**
- unary +, -, **not**
- additive +, -
- concatenation &
- relational /=, =, >, >=, <, <=
- membership **in**
- logical **and**, **or**, **xor**
- conditional logical **and then**, **or else**

Instructions

- assignment X := *expression*
- conditional **if** *cond1* **then**. . . **elsif** *cond2* **then**. . . **else**. . . **end if**
- choice **case** X **is when** Y|Z =>. . . **when others** =>. . . **end case**
- loop
  - **loop**. . . **end loop**
  - **while** *condition* **loop**. . . **end loop**
  - **for** X **in** *range* **loop**. . . **end loop**
- invocation of a subroutine: parameters identified by position or by name
  - f(0.5, 0.25)
  - f(Y => 0.25, X => 0.5)

Subroutine

procedure *name*(*parameters*) **is**
declarations
begin
. . .
end *name*;

function *name*(*parameters*) **return** *typ* **is**
declarations
begin
. . .
    return *expression*;
end *name*;

Parameters:

- **function** f(X, Y : FLOAT; N : INTEGER := 0) **return** FLOAT **is**. . .
- kinds:
  - input (also default) **in**
  - output **out** (only in procedures)
  - **in out**
  - using pointer **access**
- in case of no parameters, parentheses are omitted

Overloading

There may exist many functions or procedures with the same name, but with different parameters (as in C++).

Operators can also be overloaded, e.g.
function ”*”(X: MATRIX; Y: COL_VEC) **return** COL_VEC **is**
...
end ”*”;

Exceptions

```
except: exception;                          ← declaration
...
begin
    ...
    raise except;                           ← raising
    ...
exception
    when except =>
        ...                                 ← handling
end;
```

Exceptions can also be raised by standard libraries.

Packages

Package interface contains constants, variables, types, functions, procedures, etc. offered by the package to other entities in the program. Declarations are enclosed in a construction:

package *name* **is**
package interface (constants, procedures, etc.)
end *name*;

The package interface is put into a file with `.ads` extension.

Packages

To use a packages, one should signal the desire to use it:

with X, Y;

Then, one can use the constants, variables, types, procedures, functions, etc. implemented by the packages X and Y by prepending their identifiers with a package name and a dot (period), e.g. X.Variable.

If no name conflict occurs, one can use the names coming from the packages without package names and a dot, provided that a directive is given:

use X, Y;

That works similarly to **using namespace** in C++.

Packages

The body (contents, implementation) of the package is enclosed in a construction:

package body *name* **is**
package implementation
end *name*;

Declarations and definitions contained inside have local range unless they are put into the package interface.

Tasks

Tasks are first declared then defined:

task t;

task body t **is**
    task body
end t;

Declarations and definition take place in their parent units, e.g. in a **declare** block or a procedure.

Concurrency

```
task buffer is
   entry Write(C: CHARACTER);
   entry Read(C: out CHARACTER);
end buffer;
```

```
task body buffer is
   CH: CHARACTER;
begin
   loop
      accept Write(C: CHARACTER) do
         CH := C;
      end Write;
      accept Read(C: out CHARAC-
TER) do
         C := CH;
      end Read;
      exit when CH = ASCII.EOT;
   end loop;
end buffer;
```

Nondeterminism in Tasks

Selective accept (calling arbitrary entry point):

```
select
  accept ...
    ...
  end ...;
or
  accept ...
    ...
  end ...;
end select
```

There can be more **or** parts, and **accept** may be replaced with **delay** determining a timeout on the receiver side or with **terminate** meaning task termination (then the task can terminate at any moment).

Nondeterminism in Tasks

Timed entry call (the task must be called within the specified time, or else precautionary measures will be taken):

select
  task invocation
or delay *seconds*
  precautionary measure
end select;

There may be more **accept** branches.

Nondeterminism is Tasks

Conditional entry call (invocation of a task, or taking precautionary measures if the task cannot be executed immediately):

select
  task invocation
else
  precautionary measure
end select;

Nondeterminism in Tasks

Asynchronous select (if the main part is blocked, e.g. as a result of calling an entry point or **delay** instruction, the secondary part starts; termination of one part terminates the other one):

select
  main part
then abort
  secondary part
end select;

Nondeterminism in Tasks

Each branch of the **select** instruction can be preceded with a guard, which takes the form of:

when *condition* **=>**

If the condition is FALSE, the branch is skipped.