CSc 520

Principles of Programming Languages

49: Modularity

Christian Collberg

collberg@cs.arizona.edu

Department of Computer Science

University of Arizona

Separate Compilation

Separate Compilation

- From the very beginning of language design history, it was realized that monolithic languages (the entire program is stored in one file and compiled all at once) were no good.
- Monlithic languages made compilation slow and made it difficult for several programmers to work on the same problem.
- As early as 1958, FORTRAN II had separately compiled procedures!
- Eventually it was realized that a more formal approach had to be taken to the definition of separately compiled modules. A number of languages (Mesa, Modula-2, Ada, . . .) constructed module systems built on the ideas of David Parnas:

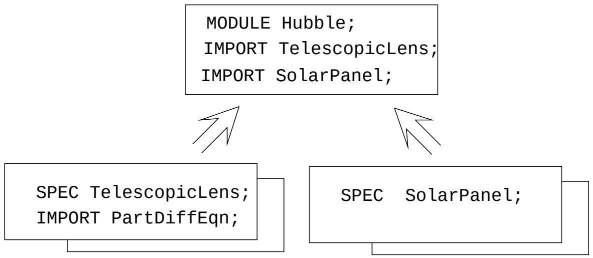
Separate Compilation. . .

The specification must provide

1. to the intended user all the information that he will need to use the program, and nothing more.
2. to the implementer all the information about the intended use that he needs to complete the program, and no additional information.

Separate Compilation...

- Each module has two parts, the specification and the implementation. Much like `.h` and `.c` files in C, only each part is separately compiled.



Module Concepts

- In one-to-one languages, there is one specification unit for every implementation unit. In many-to-many languages, each module can consist of several implementation and several specification units.

#specs		#impls	language
one	-to-	zero	Eiffel
one	-to-	one	Modula-2, Ada
many	-to-	many	Modula-3, Mesa

Module Concepts...

- The specification unit of a module contains the declarations of the types, constants, exceptions, procedures, etc, that the module exports.
- The implementation unit contains the implementation (procedure bodies, e.g.) of these objects.

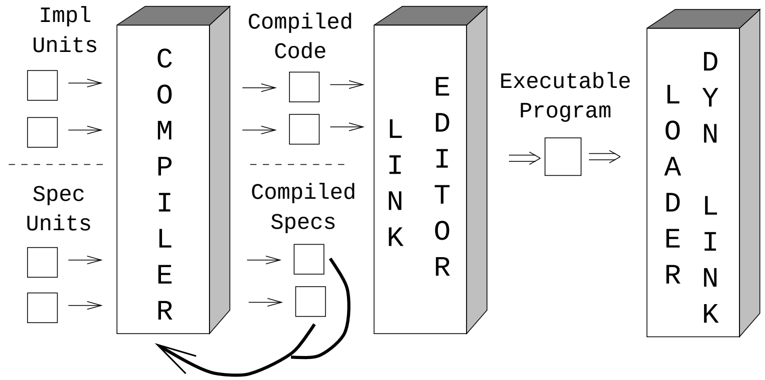
Separate Compilation — Problems

- How do we perform inter-module type checking? E.g., we must make sure that imported procedures are called with the right types of arguments.
- How are compiled modules joined together to form an executable program?
- How can we make sure that specification and implementation units are compiled in the correct order?
- How can we implement Parnas' information hiding; i.e. how can we make sure that only the neccessary information is given in the specification unit, and the rest deferred to the implementation unit?

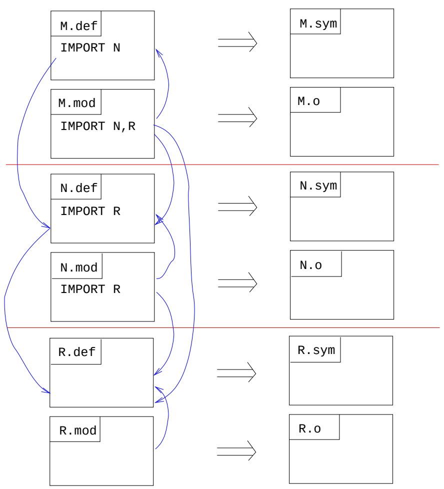
Separate Compilation…

- Let's assume that a module `M`'s specification part is kept in a file called `M.def`, and that the implementation part is in `M.mod`.
- Usually, `M.def` is compiled into a file `M.sym`, which contains a compiled version of `M.def`'s symbol table.
- `M.mod` is compiled into a `.o` object file.
- Assume that `M` imports module `N`. When `M.mod` is compiled, the compiler needs access to `N`'s symbol table, in order to be able to type-check `M`. The compiler therefore reads `N.sym`.
- If `M` imports `N` and `N` imports `R`, then `M` may (indirectly) be able to refer to `R`'s objects. Hence, when `M` is compiled, we need access to `R`'s object.

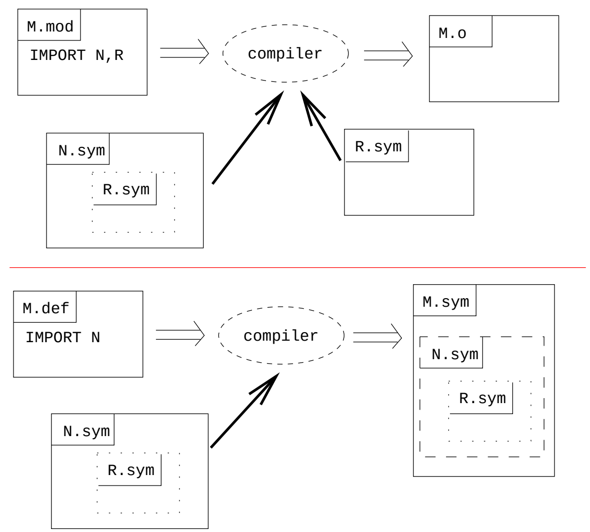
Separate Compilation…



Separate Compilation…



Separate Compilation…



Separate Compilation...

Time-stamps

- If, in the slide before last, `R.def` was edited, `R.def` will (naturally) have to be recompiled. Furthermore, `N.def` will have to be recompiled since it makes use of symbols from `R.def`, and now that `R.def` has changed we need to type-check `N.def` again. For the same reason, `M.def` must also be recompiled.
- How does the compiler detect these dependencies? Each compiled specification unit `M.sym`, contains (in addition to the compiled symbol table) a time-stamp, the time when the module was compiled. It also holds time-stamps for all imported modules. This is enough to detect compilation order violations.

Encoding the Symbol Table

- Each specification unit is compiled into a symbol file, an encoding of the symbol table of exported symbols.
- We can encode the symbol table as a sequence of tuples. Each tuple defines an identifier. It stores the module which defines the name; the kind (const,proc,type,etc), name, and type of the identifier; and extra information.

nr	kind	mod	name	type	extra
(1)	module		M		...
(2)	const	(1)	C	(3)	val=45
(3)	const	(0)	int		basic
(4)	...	...	...	...	...

Example

```
DEFINITION MODULE M;
   IMPORT N;
   TYPE T = RECORD a : INTEGER; b : N.T; END;
END M.

DEFINITION MODULE N;
   IMPORT R;
   TYPE T = ARRAY [1..R.C] OF R.T;
END N.

DEFINITION MODULE R;
   TYPE T = CHAR;
   CONST C = 45;
END R.
```


Example — M.sym

nr	kind	mod	name	type	extra
(1)	module		M		TS="10-06 23:11"
(2)	import		N		TS="10-05 09:24"
(3)	import		R		TS="10-06 14:46"
(4)	type_std		CHAR		
(5)	type_std		INT		
(6)	type_equiv	(3)	T	(4)	

Example — M.sym. . .

r	kind	mod	name	type	extra
7)	const	(3)	C	(5)	val=45
8)	type_range	(2)	T$1	(5)	range=[1, (7)]
9)	type_array	(2)	T	(6)	range=(8)
10)	type_rec	(1)	T		
11)	field	(1)	a	(5)	record=(10)
12)	field	(1)	b	(9)	record=(10)

Information Hiding

Information Hiding

- Let's look at how three languages (Mesa, Ada, Modula-2) have implemented Parnas' principles of information hiding.
- In all three languages modules come in two parts; i.e. they have one-to-one module systems.
- All three languages allow you to export types, procedures, and constants. In the specification part of the module you give procedure headers, constants declarations, and the names of opaque (hidden) types.
- Procedure bodies (and, for some of the languages, implementations of hidden types) are given in the implementation unit.

Modular Languages — Mesa

- Mesa was the first "real" modular language, developed at Xerox Parc in the early 70's.
- In Mesa the specification module's definition of the stack type (T), contains the size (in bytes) of the type.
- Like Modula-2, Mesa does not support garbage collection. But, in this case, the type T is statically allocated, so no dynamic allocation is necessary.
- "[202]" in the definition of T refers to T's size.
- "Stack.Init [@S]" passes the address of S to Init. This construction must be used since Mesa only has pass-by-value parameters.

Modular Languages — Mesa. . .

```
Stack :   DEFINITIONS  =
BEGIN
    T       :   TYPE [202];
    PT      :   TYPE = LONG POINTER TO T;

    Init    :   PROC [S : PT];

    Push    :   PROC [S : PT; E : INTEGER];

    Pop     :   PROC [S : PT] RETURNS INTEGER;
END.
```


Modular Languages — Mesa. . .

```
StackImpl :   PROGRAM EXPORTS Stack = BEGIN
    T : PUBLIC TYPE = RECORD [
                space :   ARRAY [1..100] OF INTEGER;
                index :   [0 ..  100]
        ];
    (* Impl of Init, Push, and Pop.  *)
    END.

Main :   PROGRAM IMPORTS Stack =
    BEGIN
        S : Stack.T;
        Stack.Init [@S];
        Stack.Push [@S, 314];
    END.
```


Modular Languages — Ada

```
generic
  type ITEM is private;
package GENERIC_STACK is
  type STACK (SIZE : POSITIVE) is limited private;
  procedure PUSH (S : in out STACK; E : in ITEM);
  procedure POP ( S : in out STACK; E : out ITEM);
  pragma INLINE (PUSH, POP);
private
  type VECTOR is array (POSITIVE range < >) of ITEM;
  type STACK (SIZE : POSITIVE) is record
      SPACE : VECTOR (1 ..  SIZE); INDEX : NATURAL := 0;
    end record;
end GENERIC_STACK;
```


Modular Languages — Ada. . .

```
package body GENERIC_STACK is
    -- Implementations of ...
    -- PUSH and POP ...
end GENERIC_STACK;

with GENERIC_STACK;
procedure MAIN is
    package STACK_INT is new GENERIC_STACK (INTEGER);

    S : STACK_INT.STACK (100);
begin
    STACK_INT.PUSH (S, 314);
end MAIN;
```

Modular Languages — Modula-2

```
INITION MODULE GenStack;
IMPORT SYSTEM;

TYPE Stack;

PROCEDURE Create () :  Stack;
PROCEDURE Destroy (VAR S : Stack);

PROCEDURE Push (S : Stack; E : SYSTEM.ADDRESS);
PROCEDURE Pop (S : Stack; VAR E:SYSTEM.ADDRESS)
 GenStack.
```


Modular Languages — Modula-2. . .

```
IMPLEMENTATION MODULE GenStack;
   IMPORT SYSTEM, Storage;
   TYPE Stack = POINTER TO RECORD
            space :  ARRAY [1..100] OF SYSTEM.ADDRESS;
            index :  CARDINAL;
         END;
   (* Implementations of Create,...  *)
END GenStack.
```


Modular Languages — Modula-2. . .

```
MODULE Main;
   IMPORT GenStack, Storage;
   VAR S : GenStack.Stack;
   E : POINTER TO INTEGER;
BEGIN
   S := GenStack.Create ();
   NEW (E); E^ := 314;
   GenStack.Push (S, E);
   GenStack.Destroy (S);
END Main.
```


Language Comparisons

- Notice the difference between an Ada package and a Modula-2 module:

Ada

- An Ada module specification has two parts, a public part and a private part.
- The private part contains the definitions of all those items that we don't want a user to know about. In the stack example, the private part reveals that the stack is implemented as an array.

Language Comparisons…

Modula-2

- The implementation of the stack type is not given in the specification part of the module. Rather, the information that the stack uses an array implementation is hidden within the module's implementation unit, which is available only to the module's implementer.
- Note that the Stack type is implemented as a pointer. This is in contrast to the Ada implementation which used a static representation.
- Note that – since Modula-2 does not support garbage collection – we need explicit procedures for memory allocation and deallocation.

Information Hiding – How?

A separately compiled modular program goes through several processing stages from source code to binary executable program:

Compilation Check the static semantic correctness of and generate code for each module.

Binding Combine the code generated for each module into one program. Resolve inter-modular references.

Loading Load the program generated during binding into the memory of the computer. If we have dynamic linking, then the relevant dynamic libraries must also be loaded.

Execution Execute any start-up code. Run the loaded program.

Information Hiding – How?…

- Specification units are often compiled as well, to an intermediate form containing all the information of the symbols the module makes available.
- This information is then loaded by the compiler when it compiles a client module, i.e. a module which makes use of the exported symbols.
- The thing to remember from previous slides is that the only information available to the compiler regarding imported modules, is what is given in the specification unit.
- We now have all the clues we need in order to understand why the languages we looked at earlier have such (seemingly arbitrary) rules regarding exported types:

Information Hiding – How?…

- When compiling a module the compiler needs access to the sizes of all imported types and the code of all imported procedures.
- Therefore, (since the compiler only has access to the information in the interface) this information needs to be given there.
- Different languages reveal the information in different ways:

  Ada, C++ Reveal stack type.

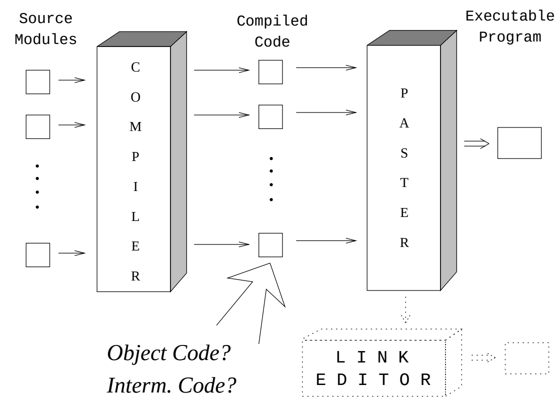
  Modula-2 Requires that the stack types is a pointer. Since all pointers are the same size this will allow the compiler to know the size.

  Mesa Reveal stack size.
- Language design is influenced by compiler requirements!

Binding Time

- In some systems the systems linker is replaced by a module binder which allows information (such as sizes of types) to be exchanged at binding time.
- Some of the work traditionally performed by the compiler is deferred till module binding time. This means that certain operations (such as inline expansion, optimization, and code generation) is done by the compiler (when there is enough information available for it to do so) or otherwise performed by the binder.
- In order to be able to perform these types of operations, the code produced by the compiler is sometimes intermediate code rather than machine code, as is usual.

Exchanging Information at Binding Time



Readings and References

- Read Scott: 122–129,539–542
- Christian Collberg, *Flexible Encapsulation*, PhD Thesis, Lund University.
- Mary Fernandez, *Simple and Effective Link-Time Optimization of Modula-3 Programs*, PLDI'95.
- Mary Fernandez, *A Retargetable, Optimizing Linker*, PhD Thesis, Princeton University.

Summary

- There is an increasing amount of research into link-time optimization. This is challenging work since linked programs are large, maybe 10 M lines of code.