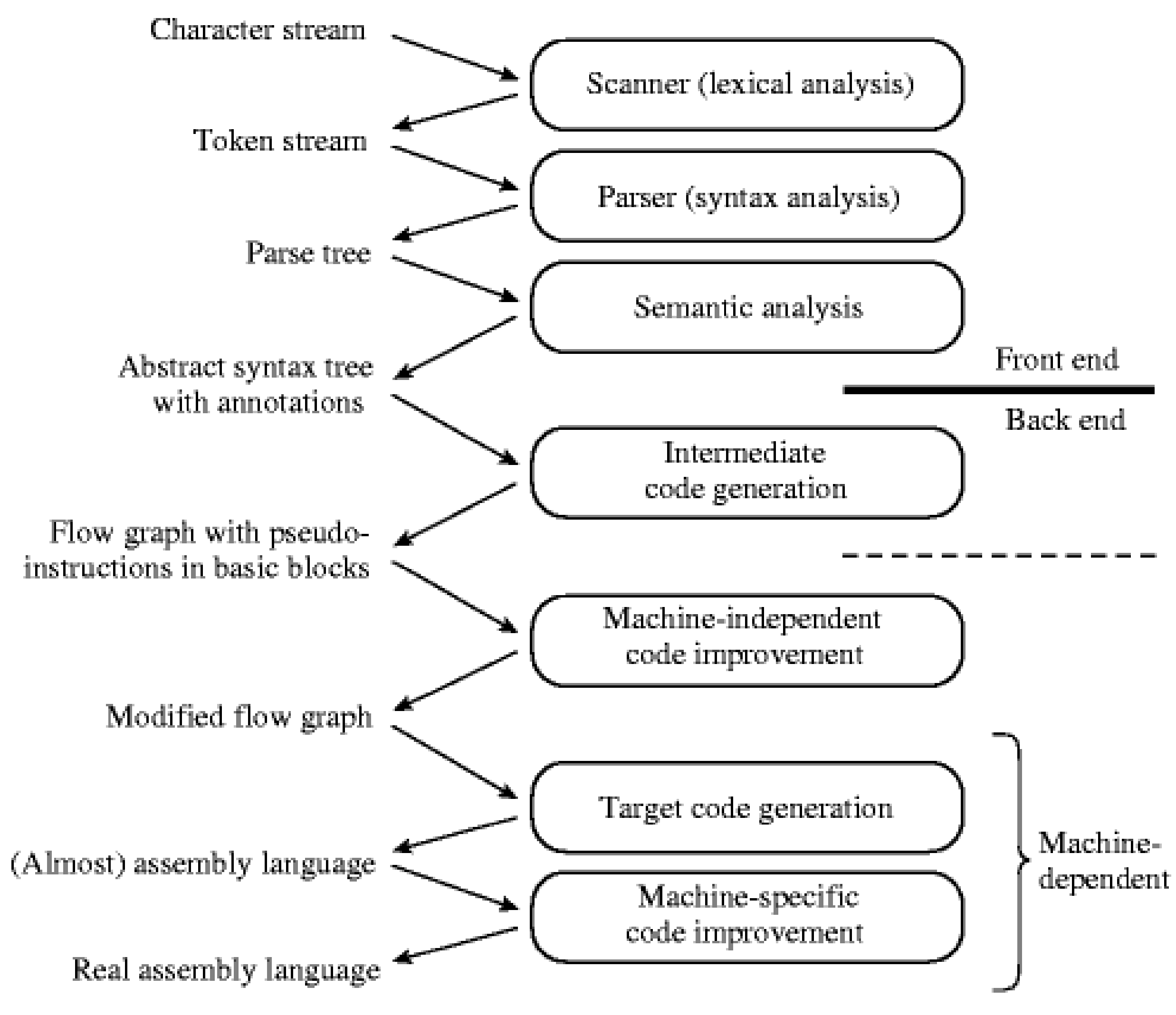
Building a Runnable Program

Corso di Programmazione Avanzata

Giuseppe Attardi
attardi@di.unipi.it

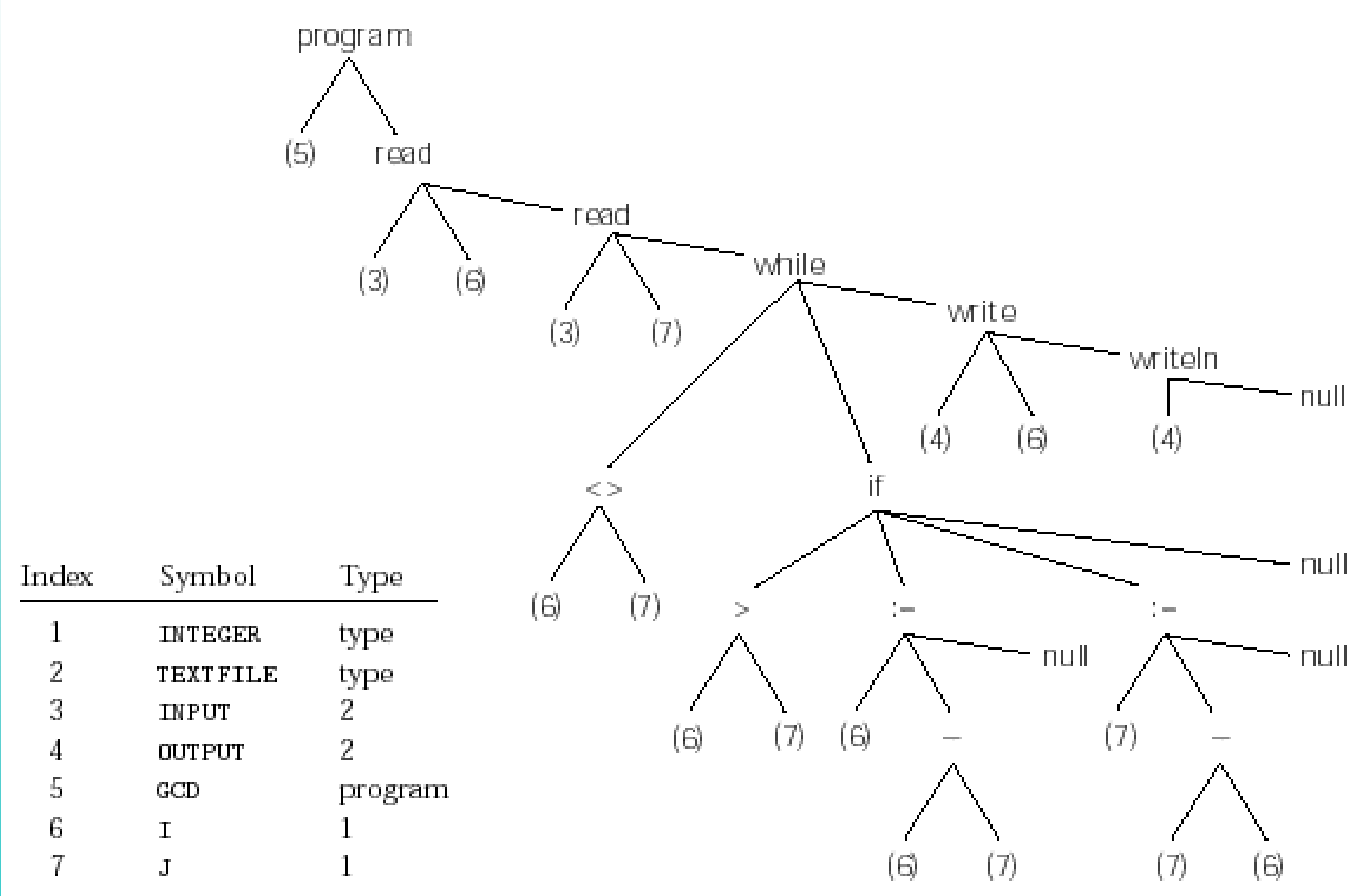
Compiler Architecture



Example: GCD

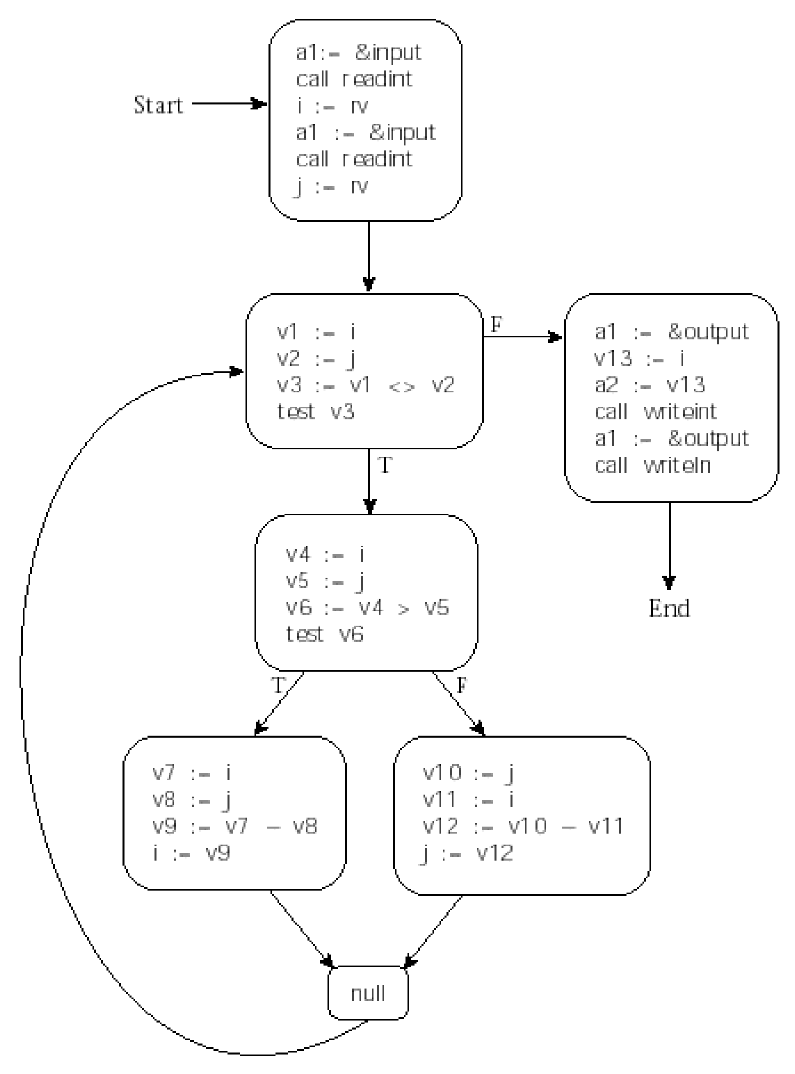
```
program gcd(input, output);
var i, j : integer;
begin
  read(i, j);
  while i <> j do
      if i > j then i := i – j
      else j := j – i;
  writeln(i)
end.
```

Syntax tree



Index	Symbol	Type
1	INTEGER	type
2	TEXTFILE	type
3	INPUT	2
4	OUTPUT	2
5	GCD	program
6	I	1
7	J	1

Control Flow Graph



Intermediate Form

- gcc back end for several processors
- GNU Register Transfer Language
- RTL expressions
  - Linked to each other
  - Instruction Codes:
    - insn
    - jump_insn
    - call_insn
    - code_label
    - barrier
    - note

Generating Code

```
void emit(enum code_ops operation, int arg)
   {
   code[code_offset].op = operation;
   code[code_offset++].arg = arg;
}
void back_patch(int addr,  enum code_ops
   operation, int arg) {
   code[addr].op  = operation;
   code[addr].arg = arg;
}
```

YACC

```
exp : NUMBER       { emit(LDI, $1 );  }
  | IDENTIFIER     { context_check(LD,  $1 ); }
  | exp '<' exp    { emit(LT, 0); }
  | exp '=' exp    { emit(EQ, 0); }
  | exp '>' exp    { emit(GT, 0); }
  | exp '+' exp    { emit(ADD, 0); }
  | exp '-' exp    { emit(SUB, 0); }
  | exp '*' exp    { emit(MULT, 0); }
  | exp '/' exp    { emit(DIV, 0); }
  | '(' exp ')'
```

Example

- **Generation for: (a + b) x (c – d/e)**

```
LD    0    -- a
LD    1    -- b
ADD        -- a + b
LD    2    -- c
LD    3    -- d
LD    4    -- e
DIV        -- d/e
SUB        -- c – d/e
MUL        -- (a + b) x (c – d/e)
```

GNU RTL

Representation for: d = (a + b) * c

```
(insn 8 6 10 (set (reg:SI 2)
                (mem:SI (symbol_ref:SI ("a")))))

(insn 10 8 12 (set (reg:SI 3)
                (mem:SI (symbol_ref:SI ("b")))))

(insn 12 10 14 (set (reg:SI 2) (plus:SI (reg:SI 2) (reg:SI 3))))

(insn 14 12 15 (set (reg:SI 3)
                (mem:SI (symbol_ref:SI ("c")))))
(insn 15 14 17 (set (reg:SI 2) (mult:SI (reg:SI 2) (reg:SI 3))))
(insn 17 15 19 (set (mem:SI (symbol_ref:SI ("d")))
                      (reg:SI 2)))
```

Object File Format

- *import table* Identifies instructions that refer to named locations whose addresses are unknown, but are presumed to lie in other files yet to be linked to this one
- *relocation table* Identifies instructions that refer to locations within the current file, but that must be modified at link time to reflect the offset of the current le within the final, executable program
- *export table* Lists the names and addresses of locations in the current file that may be referred to in other files

Program segments

- *uninitialized data* May be allocated at load time or on demand in response to page faults. Usually zero-ed, both to provide repeatable symptoms for programs that erroneously read data they have not yet written, and to enhance security on multi-user systems, by preventing a program from reading the contents of pages written by previous users.
- *stack* May be allocated in some fixed amount at load time. More commonly, is given a small initial size, and is then extended automatically by the operating system in response to (faulting) accesses beyond the current segment end.
- *heap* Like stack, may be allocated in some fixed amount at load time. More commonly, is given a small initial size, and is then extended in response to explicit requests (via system call) from heap-management library routines.
- *Files* In many systems, library routines allow a program to map a file into memory. The routine interacts with the operating system to create a new segment for the file, and returns the address of the beginning of the segment. The contents of the segment are usually fetched from disk on demand, in response to page faults.

COFF

Structure	Located	Purpose
File Header	Beginning of file	Overview of the file; controls layout of other sections
Optional Header	Follows file header	For executables, used to store the initial %eip
Section Header	Follow optional header; count determined by file header	Maintain location and size information about code and data sections
Section Data	pointer in section header	Contains code and data for the program
Relocation Directives	pointer in section header	Contain fixup information needed when relocating a section
Line Numbers	pointer in section header	Hold address of each line number in code/data sections
Symbol Table	pointer in file header	Contains one entry for each symbol this file defines or references
String Table	Follows symbol table	Stores symbol names; first four bytes are total length

COFF: File Header

```
typedef struct {
    unsigned short f_magic; /* magic number */
    unsigned short f_nscns; /* number of sections */
    unsigned long f_timdat; /* time & date stamp */
    unsigned long f_symptr; /* file pointer to symtab */
    unsigned long f_nsyms; /* number of symtab entries */
    unsigned short f_opthdr; /* sizeof(optional hdr) */
    unsigned short f_flags; /* flags */
} FILHDR;
```

COFF: Section Header

```
typedef struct {
    char s_name[8]; /* section name */
    unsigned long s_paddr; /* physical address */
    unsigned long s_vaddr; /* virtual address */
    unsigned long s_size; /* section size */
    unsigned long s_scnptr; /* file ptr to raw data for section */
    unsigned long s_relptr; /* file ptr to relocation */
    unsigned long s_lnnoptr; /* file ptr to line numbers */
    unsigned short s_nreloc; /* number of relocation entries */
    unsigned short s_nlnno; /* number of line number entries */
    unsigned long s_flags; /* flags */
} SCNHDR;
```

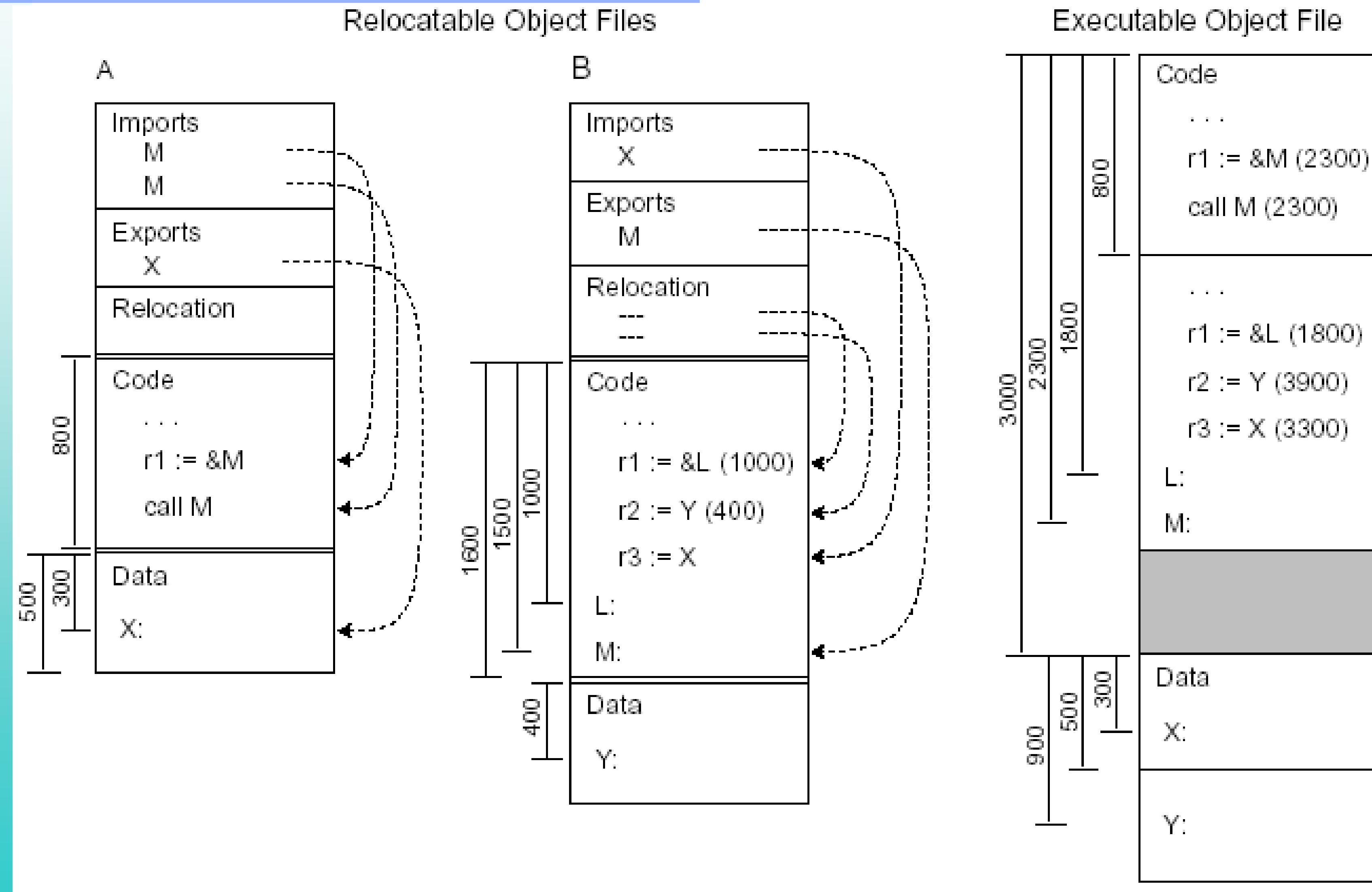
COFF: Typical Sections

- **TEXT: executable code**
- **DATA: initialized data**
- **BSS: non initialized data (no data stored in file)**

COFF: Relocation Directives

```
typedef struct {
  unsigned long r_vaddr; /* address of relocation */
  unsigned long r_symndx; /* symbol we're adjusting for */
  unsigned short r_type; /* type of relocation */
} RELOC;
```

Linking



Dynamic Linking

Dynamic Linking

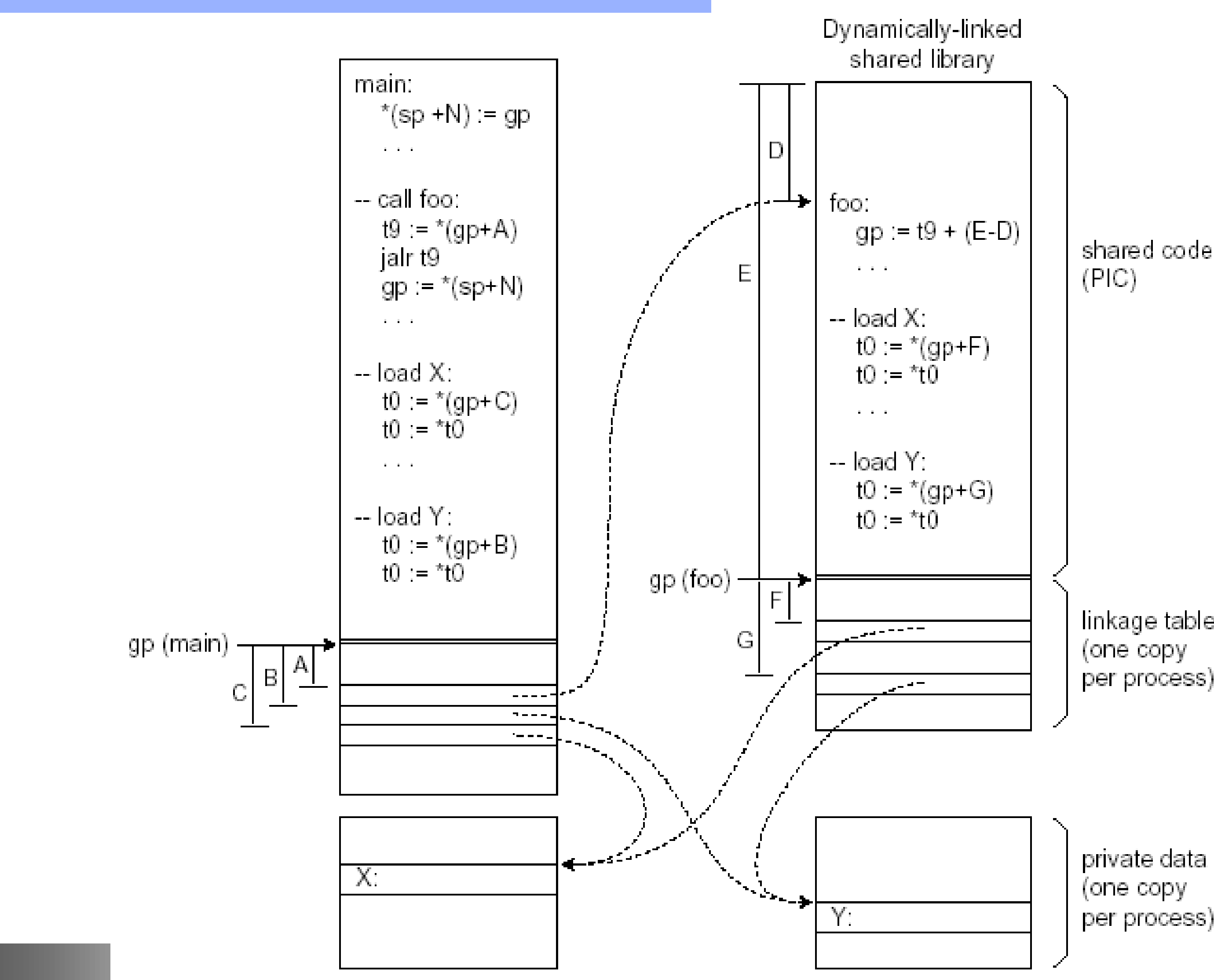
- Allows sharing a single copy of the library
- Each DLL has its own code and data segments
- Each program has private copy of data segment, shares code segment
- DL library must either:
  - be located at fixed address
  - have no relocatable words in code

Position Independent Code

- **Generating PIC requires:**

1. **use PC-relative addressing, rather than jumps to absolute addresses, for all internal branches.**
2. **similarly, avoid absolute references to statically allocated data, by using displacement addressing with respect to some standard base register. If the code and data segments are guaranteed to lie at a known offset from one another, then an entry point to a shared library can compute an appropriate base register value using the PC. Otherwise the caller must set the base register as part of the calling sequence.**
3. **use an extra level of indirection for every control transfer out of the PIC segment, and for every load or store of static memory outside the corresponding data segment. The indirection allows the (non-PIC) target address to be kept in the data segment, which is private to each program instance.**

Linking DLL (MIPS)



GOT (Global Offset Table)

- Linker creates a GOT with pointers to all global data
- GOT referred through register (EBX)
- The function prologue of every function needs to set up this register to the correct value
- Code to load EBX:

```
            call .L2 ;; push PC on stack
.L2:        popl %ebx; PC into register EBX
            addl $_GLOBAL_OFFSET_TABLE+[.-.L2],%ebx
   ;; adjust ebx to GOT address
```

Referencing global variables

```
static int a; /* static */
extern int b; /* global */

a = 1;
movl $1, a@GOT(%ebx)

b = 2;
movl b@GOT(%ebx), %eax
movl $2, (%eax)
```

PLT (Procedure Linkage Table)

- Indirection for functions similar to GOT for data
- Lazy procedure linkage

PLT structure (x86)

```
;; special first entry:
PLT0:   pushl GOT+4
        jmp *GOT+8
;; regular entry for proc1
PLT1:   jmp *GOT+m(%ebx)
        push #reloc_offset
        jmp PLT0
```

PLT

- Initially GOT+m(%ebx) contains address of PLT1+1
- Code there pushes relocation offset for the symbol proc1 and then
- Jumps to PLT0 to perform resolution and linkage for proc1

PLT operation

- **Dynamic linker places two values in the GOT:**

GOT+4	**code identifying library**
GOT+8	**address of linker symbol resolution routine**

Lazy procedure linkage

- The first time the program calls a PLT entry, the first jump in the PLT entry does nothing, since the GOT entry through which it jumps points back into the PLT entry
- Then the push instruction pushes the offset value which indirectly identifies both the symbol to resolve and the GOT entry into which to resolve it, and jumps to PLT0
- The instructions in PLT0 push another code that identifies the library and then jumps into stub code in the dynamic linker with the two identifying codes at the top of the stack
- It is a jump, rather than a call: above the two identifying words just pushed is the return address back to the routine that called into the PLT

Linker symbol resolution routine

- Uses the two parameters to find the library's symbol table and the routine's entry in that symbol table
- Looks up the symbol value using the concatenated runtime symbol table, and stores the routine's address into the GOT entry
- Then the stub code restores the registers, pops the two words that the PLT pushed, and jumps off to the routine
- The GOT entry having been updated, subsequent calls to that PLT entry jump directly to the routine itself without entering the dynamic linker

Exploit by viruses

- **Malicious virus code can exploit DL to get access to functions external to the host file**
- **http://downloads.securityfocus.com/library/subversiveld.pdf**

Runtime loading

- **Stub subroutine for foo:**

```
t9 := (gp + k) -- lazy linker entry point
t7 := ra
t8 := n            -- index of stub
call *t9           -- overwrites ra
```