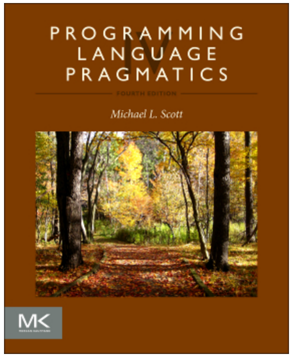
Intermediate Code Generation

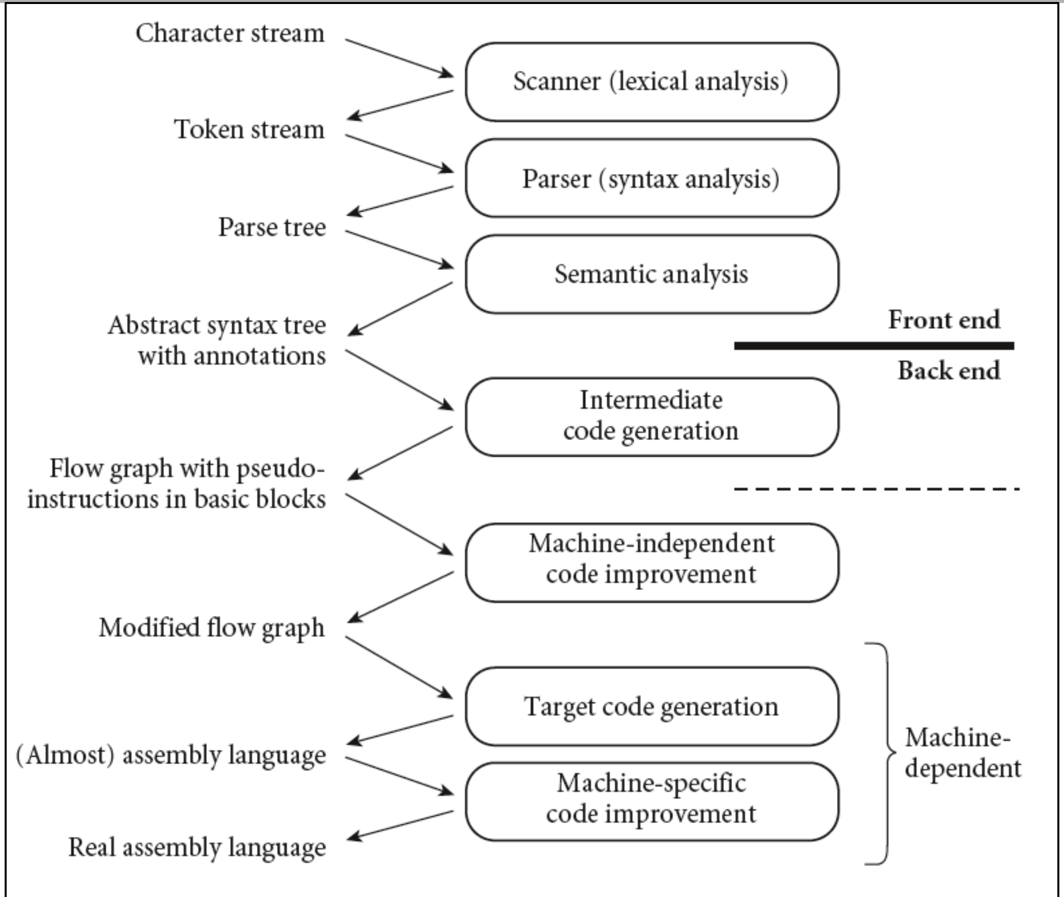
17-363/17-663: Programming Language Pragmatics



Reading: PLP chapter 15

Prof. Jonathan Aldrich

Review: Compiler Structure


Character stream
Scanner (lexical analysis)
Token stream
Parser (syntax analysis)
Parse tree
Semantic analysis
Abstract syntax tree with annotations
Front end
Back end
Intermediate code generation
Flow graph with pseudo-instructions in basic blocks
Machine-independent code improvement
Modified flow graph
Target code generation
(Almost) assembly language
Machine-specific code improvement
Real assembly language
Machine-dependent

language-dependent

machine- and language-independent

- Review: phases of compilation
  - Machine-independent phases form a “middle end” in addition to front end and back end

Intermediate Representations

- Many compilers have multiple intermediate representations
  - Different compilation steps work at different levels of abstraction

- High-level: Abstract Syntax Trees

- Mid-level: Control Flow Graphs
  - Nodes are **basic blocks**: instruction sequences with no jumps in or out
    - Idealized, machine-independent instructions are given in 3-address code:

      `r1 := r2` *`op`* `r3`
  - Edges are jumps

- Low Level: Instructions for an idealized machine
  - May be the same notation used inside basic blocks, above
  - Often with infinite "virtual registers" instead of finite physical ones

- Note: there are no hard boundaries between these levels

Stack-Based Bytecode

- Bytecode is an IR optimized for compactness, interpretability
  - Compactness is important for code sent over a network
    - The name comes from the typical “one instruction per byte”
  - Can build a fast interpreter by branching on the byte
    - Or more sophisticated techniques, like direct threaded code, jump tables, and computed gotos
    - In commercially important language, it usually gets compiled “just in time” for performance anyway
  - Still simple & portable, like other IRs

- Examples: Java bytecode, Pascal p-code, Microsoft CIL, WebAssembly

Stack-Based vs. Pseudo-Assembly

- Heron's formula: `A = sqrt [s(s-a)(s-b)(s-c)]`   `where s = (a+b+c)/2`

```
stack-based:       3-address pseudo-assembly:
push a             r2 := a
push b             r3 := b
push c             r4 := c
add                r1 := r2 + r3
add                r1 := r1 + r4
push 2             r1 := r1 / 2        -- s
divide
pop s
push s
push s             r2 := r1 - r2       -- s-a
push a
subtract
push s             r3 := r1 - r3       -- s-b
push b
subtract
push s             r4 := r1 - r4       -- s-c
push c
subtract
multiply           r3 := r3 * r4
multiply           r2 := r2 * r3
multiply           r1 := r1 * r2
push sqrt
call sqrt
call
```

Tradeoff: space vs. time

- Bytecode is more compact
  - 23 instructions in 25 bytes
  - Most instructions fit in a byte
    - including push small integers or first few variables
  - 2 extra bytes to specify `sqrt`
- 3-address easier to optimize
  - Reorder / replace instructions, considering registers or pipeline are all easier
  - But: most instructions 4 bytes
    - The call instruction is 8 bytes
  - 13 instructions in 56 bytes

WebAssembly

- Bytecode target we'll use for our compilers
- Goals
  - Portable
  - Browser platform (interop with JavaScript)
  - Machine-independent
  - Memory-safe (all errors → traps, read/write only within local state)
  - Compact (for sending over the network)
  - Fast (to interpret, to compile, and execute compiled code)
  - Typed (but low-level: almost everything is an i32 or float)

- Two formats
  - Portable binary format (.wasm)
  - Textual equivalent based on S-expressions (.wat)
    - S ::= int_const | str_const | symbol | id | (S*)

- Semantics specified with inference rules

Basic WebAssembly Bytecode

Instructions	Stack afterward
	(empty)
i32.const 1	1
i32.const 2	1 2
i32.add	3

Variables & Main in WebAssembly

TypeScript Source

```
let x:number = 1;
x = x + 1
```

WASM

```
(func (export "main") (local $x i32)
                          (empty)
    i32.const 1           i32
    local.set $x                (empty)
    local.get $x                i32
    i32.const 1           i32 i32
    i32.add               i32
    local.set $x                (empty)
))
```

If in WebAssembly

```
see if_false.ts / if_false.wat
```

Typechecking WebAssembly

- Stacks must match at control flow merges!

```
                         (empty)
local.get $condition     i32
if [i32]                 (empty)
i32.const 1              i32
else
                         (empty)
end                      ??? type error!
i32.const 2
i32.add                  // error if took else branch!
```

Typechecking WebAssembly

- Stacks must match at control flow merges! Fixed now.

```
                         (empty)
local.get $condition     i32
if [i32]                 (empty)
i32.const 1              i32
else
i32.const 2              i32
end                      i32
i32.const 2              i32 i32
i32.add                  i32
```

Practice!

- Translate the following pseudocode to WebAssembly:

```
if x > 0 then x else –x
```

- Some useful instructions: local.get, i32.const, i32.gt_s, i32.sub, if/else/end,

Loops, Functions, and Nonlocal Returns

```
see while_count
see return
```

Imports, Memories, Running from JavaScript

`see hello and run.js`

- Compiling and running

```
% wat2wasm wat/part1/hello.wat -o wasm/part1/hello.wasm

% node run.js wasm/part1/hello.wasm 1
% wizeng wasm/part1/hello.wasm 1
```

- argument `1` indicates 1 page of memory (64k)
  - see `run.js` code for how this is passed in
- Shortcut: `./single.sh part1/hello`

Global variables

```
(module
  (import "console" "log_int" (func $log_int (param
i32)))
  (global $tmp (mut i32) (i32.const 0))
  (func (export "main")
    global.get $tmp
    i32.const 1
    i32.add
    global.set $tmp
))
```

Memories

```
(module
  (import "console" "log_int" (func $log_int (param
i32)))
  (import "js" "mem" (memory 1))
  (func (export "main")
    i32.const 0
    i32.const 1
    i32.store
    i32.const 0
    i32.load
))
```

- See also `run.js`

Tables, Types, Indirect Calls

```
(module
  (import "console" "log_int" (func $log_int (param i32)))
  (import "js" "mem" (memory 1))
  (table 2 funcref)
  (elem (i32.const 0) $foo $bar)
  (type $fn1arg (func (param i32) (result i32)))
  (func (export "main")
    i32.const 10
    i32.const 0
    call_indirect (type $fn1arg)
    i32.const 1
    call_indirect (type $fn1arg)
  )
  (func $foo (param $x i32) (result i32) ...)
  (func $bar (param $x i32) (result i32) ...)
)
```

Generating code

- Generally a tree traversal
  - Producing instructions (or S-expressions, for WebAssembly)
- May need some information from typechecker
  - Or just (re-)compute, if it’s simple
- May need to collect information to return along with code
  - E.g. in Assignment 7, this includes a list of all local variables declared (must be declared at the top of a function in WebAssembly)
  - If not already computed during typechecking

Other handy instructions

```
drop
```

Intermediate Representations

- An *intermediate representation* (IR) provides the connection between the front end and the back end of the compiler, and continues to represent the program during the various back-end phases.
- IRs can be classified in terms of their *level*, or degree of machine dependence.
- High-level IRs
  - IRs are often based on trees or directed acyclic graphs (DAGs) that directly capture the structure of modern programming languages
  - facilitates certain kinds of machine-independent code improvement, incremental program updates, direct interpretation, and other operations based strongly on the structure of the source
  - Because the permissible structure of a tree can be described formally by a set of productions (cf., Section 4.6), manipulations of tree-based forms can be written as attribute grammars
  - *Stack-based* languages are another common type of high level IR

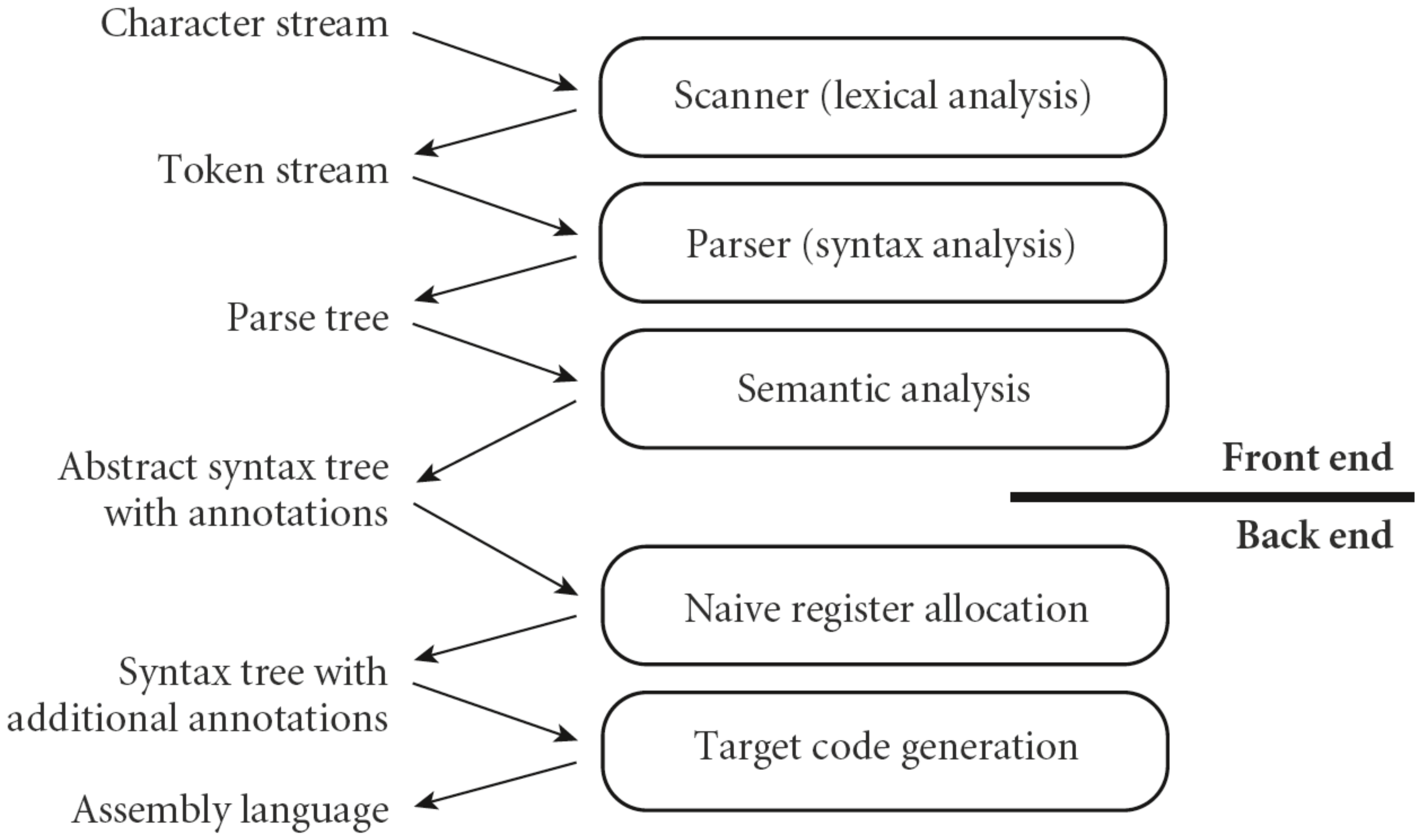
Intermediate Representation

- The most common medium-level IRs consist of three-address instructions for a simple idealized machine, typically one with an unlimited number of registers
  - Since the typical instruction specifies two operands, an operator, and a destination, three-address instructions are called *quadruples*
  - In older compilers, one may sometimes find an intermediate form consisting of *triples* or *indirect triples* in which destinations are specified implicitly
    - the index of a triple in the instruction stream is the name of the result
    - an operand is generally named by specifying the index of the triple that produced it.

Intermediate Representations

- Different compilers use different IRs
  - Many compilers use more than one IR internally, though in the common two-pass organization one of these is distinguished as “the” intermediate form
    - connection between the front end and the back end.
  - the syntax trees passed from semantic analysis to intermediate code generation constitute a high level IR
  - control flow graphs containing pseudo-assembly language (passed in and out of machine-independent code improvement) are a medium level IR
  - the assembly language of the target machine serves as a low level IR
- Compilers that have back ends for different target architectures do as much work as possible on a high or medium level IR
  - the machine-independent parts of the code improver can be shared by different back ends

Intermediate Representations



Character stream

Scanner (lexical analysis)

Token stream

Parser (syntax analysis)

Parse tree

Semantic analysis

Abstract syntax tree with annotations

Front end

Back end

Naive register allocation

Syntax tree with additional annotations

Target code generation

Assembly language

Figure 15.5 **A simpler, nonoptimizing compiler structure, assumed in Section 15.3.** The target code generation phase closely resembles the intermediate code generation phase of Figure 15.1.

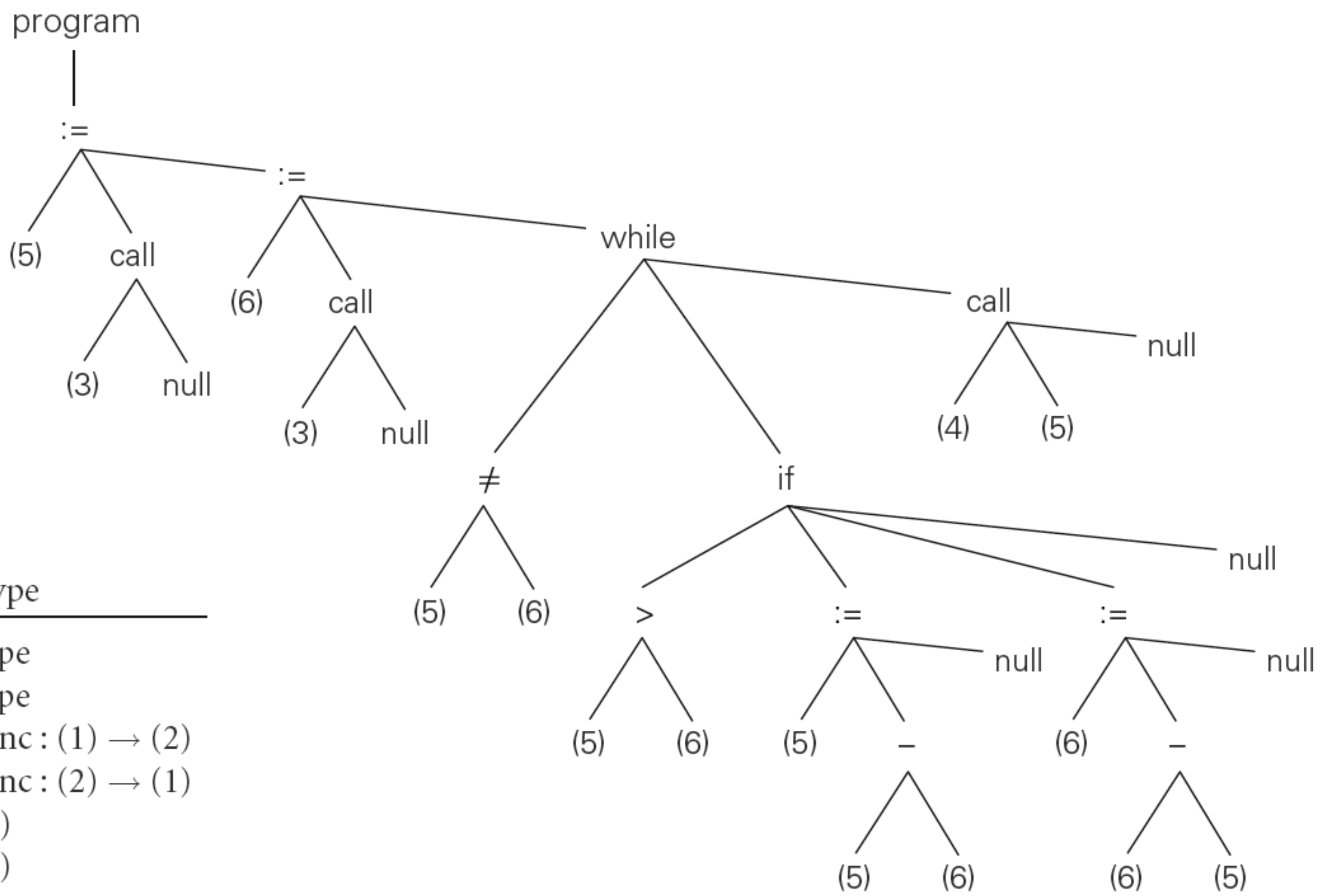
Back-End Compiler Structure

- Certain optimizations can be performed on syntax trees, but a less hierarchical representation of the program makes most analyses and optimizations easier
- Our example compiler therefore includes an explicit phase for intermediate code generation
  - The code generator groups the nodes into *basic blocks*
  - It then creates a *control flow graph* in which the nodes are basic blocks and the arcs represent interblock control flow
    - Within each basic block, operations are represented as instructions for an idealized machine with an unlimited number of registers - we will call these *virtual registers*
    - By allocating a new one for every computed value, the compiler can avoid creating artificial connections between otherwise independent computations too early in the compilation process

Back-End Compiler Structure

- The machine-independent code optimization phase performs transformations on the control flow graph.
  - *local code optimizations* - it modifies the instruction sequence within each basic block to eliminate redundant loads, stores, and arithmetic computations
  - *global code optimizations* - it also identifies and removes a variety of redundancies across the boundaries between basic blocks within a subroutine
  - an expression whose value is computed immediately before an if statement need not be recomputed after else
  - An expression that appears within the body of a loop need only be evaluated once if its value will not change in subsequent iterations
- Some global optimizations change the number of basic blocks and/or the arcs among them

Back-End Compiler Structure



Index	Symbol	Type
1	`void`	type
2	`int`	type
3	`getint`	func : (1) → (2)
4	`putint`	func : (2) → (1)
5	`i`	(2)
6	`j`	(2)

Figure 15.2 **Syntax tree and symbol table for the GCD program.** The only difference from Figure 1.6 is the addition of explicit null nodes to indicate empty argument lists and to terminate statement lists.

Back-End Compiler Structure

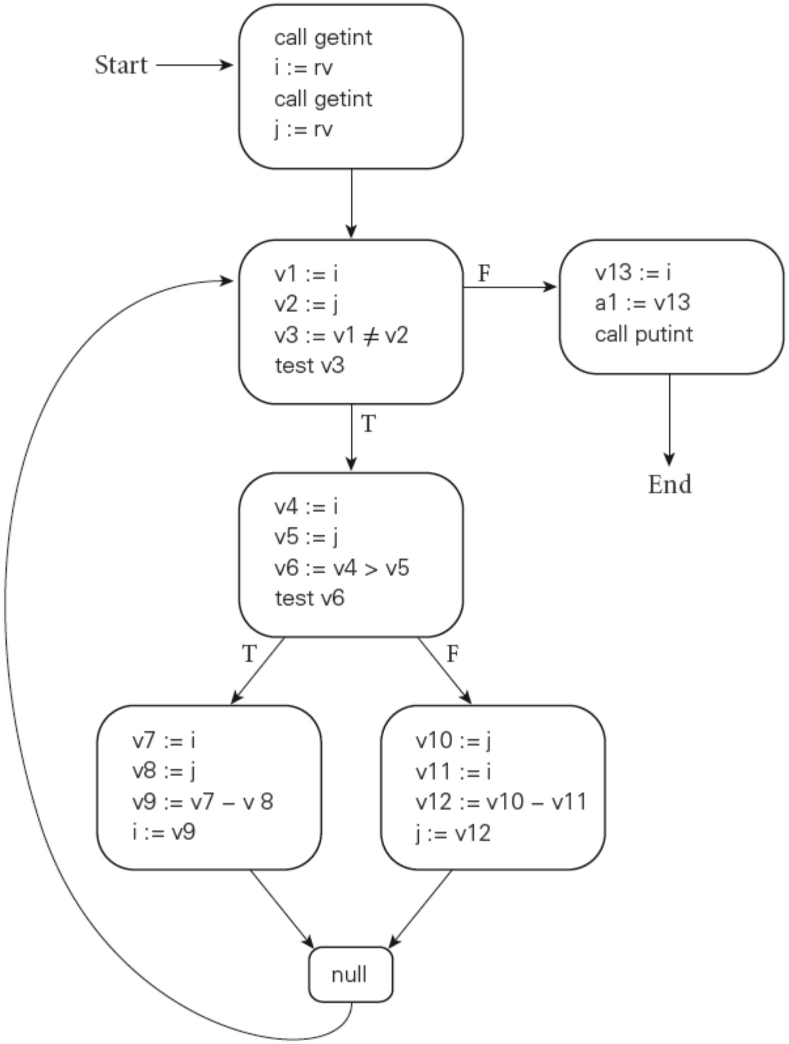


Figure 15.3 **Control flow graph for the GCD program.** Code within basic blocks is shown in the pseudo-assembly notation introduced in Sidebar 5.1, with a different virtual register (here named v1 . . . v13) for every computed value. Registers a1 and rv are used to pass values to and from subroutines.

Back-End Compiler Structure

- The next phase of compilation is *target code generation*
  - This phase strings the basic blocks together into a linear program, translating each block into the instruction set of the target machine and generating branch instructions (or "fall-throughs") that correspond to the arcs of the control flow graph.
- The output of this phase differs from real assembly language primarily in its continued reliance on virtual registers

Back-End Compiler Structure

- The final phase of our example compiler structure consists of register allocation and instruction scheduling - machine-specific code improvement
- Register allocation requires that we map the unlimited virtual registers onto the bounded set of registers available in the target machine
  - If there aren’t enough architectural registers to go around, we may need to generate additional loads and stores to multiplex a given architectural register among two or more virtual registers
  - As described in Section 5.5, instruction scheduling consists of reordering the instructions of each basic block to fill the pipeline(s) of the target machine

Back-End Compiler Structure

- Phases and Passes
  - A *pass* of compilation is a phase or sequence of phases that is serialized with respect to the rest of compilation
    - it does not start until previous phases have completed
    - it finishes before any subsequent phases start.
    - if desired, a pass may be written as a separate program, reading its input from a file and writing its output to a file.

  -- Two-pass compilers are particularly common

    they may be divided between the front end and the back end (between semantic analysis and intermediate code generation)

    or

    - they may be divided between intermediate code generation and global code improvement
    - In the latter case, the first pass is still commonly referred to as the front end and the second pass as the back end

Code Generation

- The back end of Figure 15.1 is too complex to present in any detail in a single chapter
  - To limit the scope of our discussion, we will content ourselves in this chapter with producing correct but naive code
  - This choice will allow us to consider a significantly simpler back end.
  - Starting with Figure 15.1, we drop the machine-independent code improver and then merge intermediate and target code generation into a single phase
    - generates linear assembly language - no code improvements for control flow, therefore, there is no need to represent that flow explicitly in a control flow graph

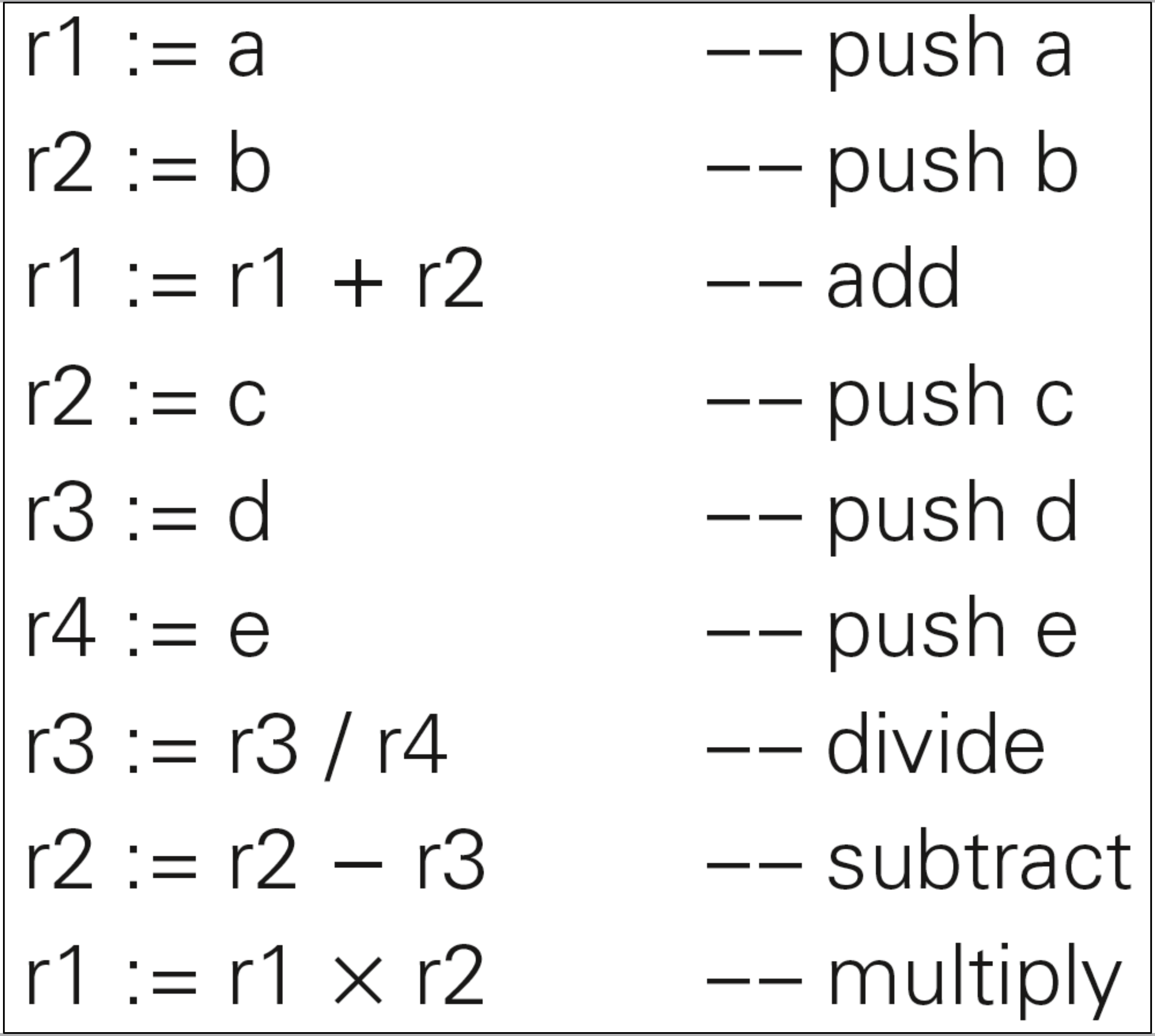
Code Generation

- We also adopt a much simpler register allocation algorithm
  - operates directly on the syntax tree prior to code generation - eliminates need for virtual registers and the subsequent mapping onto architectural registers
- Finally, we drop instruction scheduling. The resulting compiler structure appears in Figure 15.5.
  - Its code generation phase closely resembles the intermediate code generation of Figure 15.1.
- An Attribute Grammar for GCD Example is presented in Section 15.3.1

Code Generation

- Register Allocation
  - Evaluation of the rules of the attribute grammar itself consists of two main tasks
  - In each subtree we first determine the registers that will be used to hold various quantities at run time; then we generate code.
  - Our naive register allocation strategy uses the next_free_reg inherited attribute to manage registers $r_1 \ldots r_k$ as an expression evaluation stack
- To calculate the value of (a + b) × (c - (d / e)) for example, we would generate the following:

Code Generation



```
r1 := a              -- push a
r2 := b              -- push b
r1 := r1 + r2        -- add
r2 := c              -- push c
r3 := d              -- push d
r4 := e              -- push e
r3 := r3 / r4        -- divide
r2 := r2 − r3        -- subtract
r1 := r1 × r2        -- multiply
```

Code Generation

- In a particularly complicated fragment of code it is possible to run out of architectural registers.
  - In this case we must *spill* one or more registers to memory
- Our naive register allocator pushes a register onto the program's subroutine call stack
  - In effect, architectural registers hold the top *k* elements of an expression evaluation stack of effectively unlimited size
- It should be emphasized that our register allocation algorithm, makes very poor use of machine resources
- If we were generating medium level intermediate code, we would employ virtual registers, rather than architectural ones
  - Mapping of virtual registers to architectural registers would occur much later in the compilation process.
- Target code for the GCD program appears in Figure 14.7.

Address Space Organization

- Assemblers, linkers, and loaders typically operate on a pair of related file formats
  - *relocatable* object code
  - *executable* object code
- Relocatable object code is acceptable as input to a linker
  - multiple files in this format can be combined to create an executable program
- Executable object code is acceptable as input to a loader:
  - it can be brought into memory and run

Address Space Organization

- A relocatable object file includes the following descriptive information:
  - ***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 file 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
- Imported and exported names are known as *external symbols*

Address Space Organization

- Running program segments
  - *code*
  - *constants*
  - *initialized data*
  - *uninitialized data:* may be allocated at load time or on demand in response to page faults
    - Usually zero filled, both to provide repeatable symptoms for programs that erroneously read data they have not yet written
  - *stack:* may be allocated in some fixed amount at load time
    - more commonly, is given a small initial size, and then
    - extended automatically by the operating system in response to (faulting) accesses beyond the current segment end.

Address Space Organization

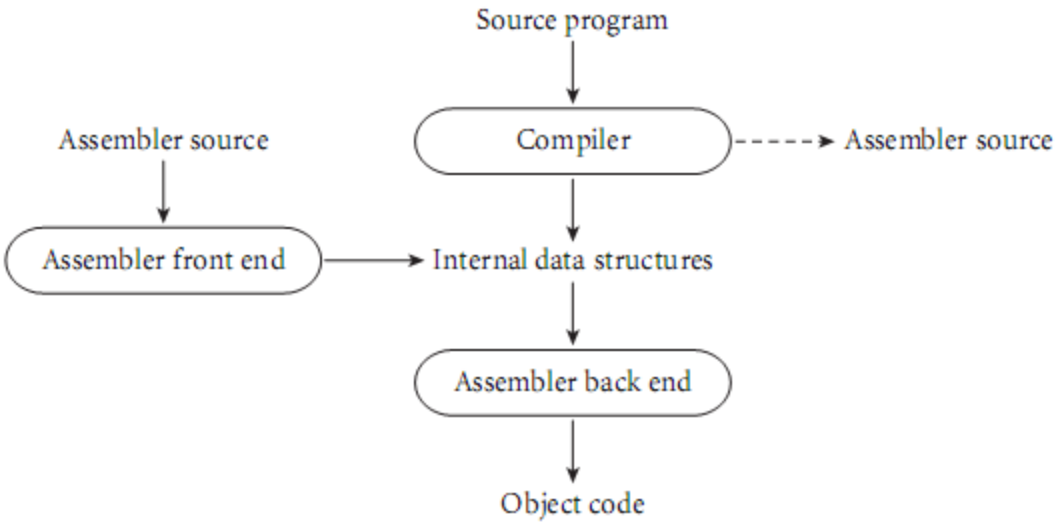
- Running program segments (2):
  - *heap:* may also be allocated in some fixed amount at load time.
    - more commonly, is given a small initial size, and is then
    - extended in response to explicit requests from heap-management library routines
  - *files:* In many systems, library routines allow a program to *map* a file into memory
    - The map 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
  - dynamic libraries: Modern operating systems typically arrange for most programs to share a single copy of the code for popular libraries

Assembly

- Some compilers translate source files directly into object files acceptable to the linker
- More commonly, they generate assembly language that must subsequently be processed by an assembler to create an object file
  - symbolic (textual) notation for code.
  - within a compiler it would still be symbolic, most likely consisting of records and linked lists
- To translate this symbolic representation into executable code, we must
  - replace opcodes and operands with their machine language encodings
  - replace uses of symbolic names with actual addresses

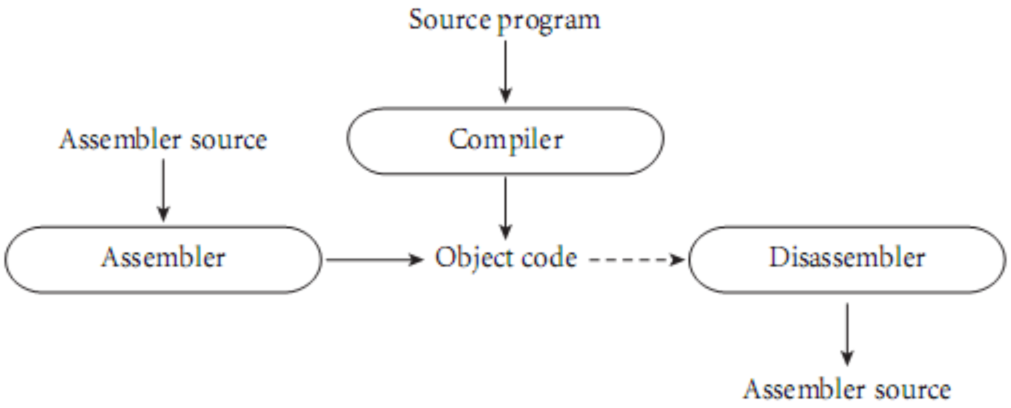
Assembly

- When passing assembly language from the compiler to the assembler, it makes sense to use some internal (records and linked lists) representation
- At the same time, we must provide a textual front end to accommodate the occasional need for human input:



Assembly

- An alternative organization has the compiler generate object code directly
  - This organization gives the compiler a bit more flexibility: operations normally performed by an assembler (e.g., assignment of addresses to variables) can be performed earlier if desired.
  - Because there is no separate assembly pass, the overall translation to object code may be slightly faster



Assembly

- Emitting Instructions
  - The most basic task of the assembler is to translate symbolic representations of instructions into binary form
  - In some assemblers this is easy
    - there is a one-one correspondence between mnemonic operations and instruction op-codes
  - Many assemblers extend the instruction set in minor ways to make the assembly language easier for human beings to read
  - Most MIPS assemblers, for example, provide a large number of *pseudoinstructions* that translate into different real instructions depending on their arguments, or that correspond to multi-instruction sequences

Assembly

- Assemblers respond to a variety of *directives* (MIPS):
  - *segment switching*
    - .text directive indicates that subsequent instructions and data should be placed in the code (text) segment.
    - .data directive indicates that subsequent instructions and data should be placed in the initialized data segment.
    - .space *n* directive indicates that *n* bytes of space should be reserved in the uninitialized data segment
    - .byte, .half, .word, .float, and .double directives each take a sequence of arguments
    - related .ascii directive takes a single character string as argument, which it places in consecutive bytes
  - symbol identification
    - .global name directive indicates that name should be entered into the table of exported symbols.
  - alignment
    - .align n directive causes the subsequent output to be aligned at an address evenly divisible by 2n

Assembly

- RISC assemblers implement a virtual machine - instruction set is “nicer” than that of the real hardware
  - In addition to pseudoinstructions, the virtual machine may have non-delayed branches
  - If desired, the compiler or assembly language programmer can ignore the existence of branch delays
  - The assembler will move nearby instructions to fill delay slots if possible, or generate nops if necessary.
  - To support systems programmers, the assembler must also make it possible to specify that delay slots have already been filled

Assembly

- Assemblers commonly work in several phases
  - if the input is textual, an initial phase scans and parses the input, and builds an internal representation
  - there are two additional phases.
    - first phase identifies all internal and external (imported) symbols, assigning locations to the internal ones
      - complicated by the length of some instructions (on a CISC machine)

        or

      - complicated by number of real instructions produced by a pseudo-instruction (on a RISC machine)
    - final phase produces object code

Assembly

- CISC assemblers distinguish between *absolute* and *relocatable* words in an object file
- Absolute words are known at assembly time; they need not be changed by the linker
  - constants and register-register instructions
- A relocatable word must be modified by adding to it the address within the final program of the code or data segment of the current object file
  - A CISC jump instruction might consist of a one-byte jmp opcode followed by a four-byte target address
  - For a local target, the address bytes in the object file would contain the symbol’s offset within the file
  - The linker finalizes the address by adding the offset of the file’s code segment within the final program

Linking

- Most language implementations - certainly all that are intended for the construction of large programs - support separate compilation
  - fragments of the program can be compiled and assembled more-or-less independently
- After compilation, these fragments (known as *compilation units*) are “glued together” by a *linker*
  - programmer explicitly divides the program into modules or files separately compiled
  - integrated environments may abandon the notion of a file in favor of a database of subroutines separately compiled
- Linker joins together compilation units

Linking

- A *static linker* does its work prior to program execution, producing an executable object file
- A *dynamic linker* does its work after the program has been brought into memory for execution
- Each of the compilation units of a program to be linked must be a relocatable object file
  - some files will have been produced by compiling fragments of the application being constructed
  - others will be general purpose library packages needed by the application
- Since most programs make use of libraries, even a “one-file” application typically needs to be linked

Linking

- Linking involves two subtasks: relocation and the resolution of external references
- Some authors refer to relocation as *loading*, and call the entire "joining together" process "link-loading."
- In this book we use "loading" to refer to the process of bringing an executable object file into memory for execution
  - on very simple machines loading entails relocation
  - the operating system uses virtual memory to giving the impression that it starts at some standard address (zero)
  - often loading also entails a certain amount of linking

Linking

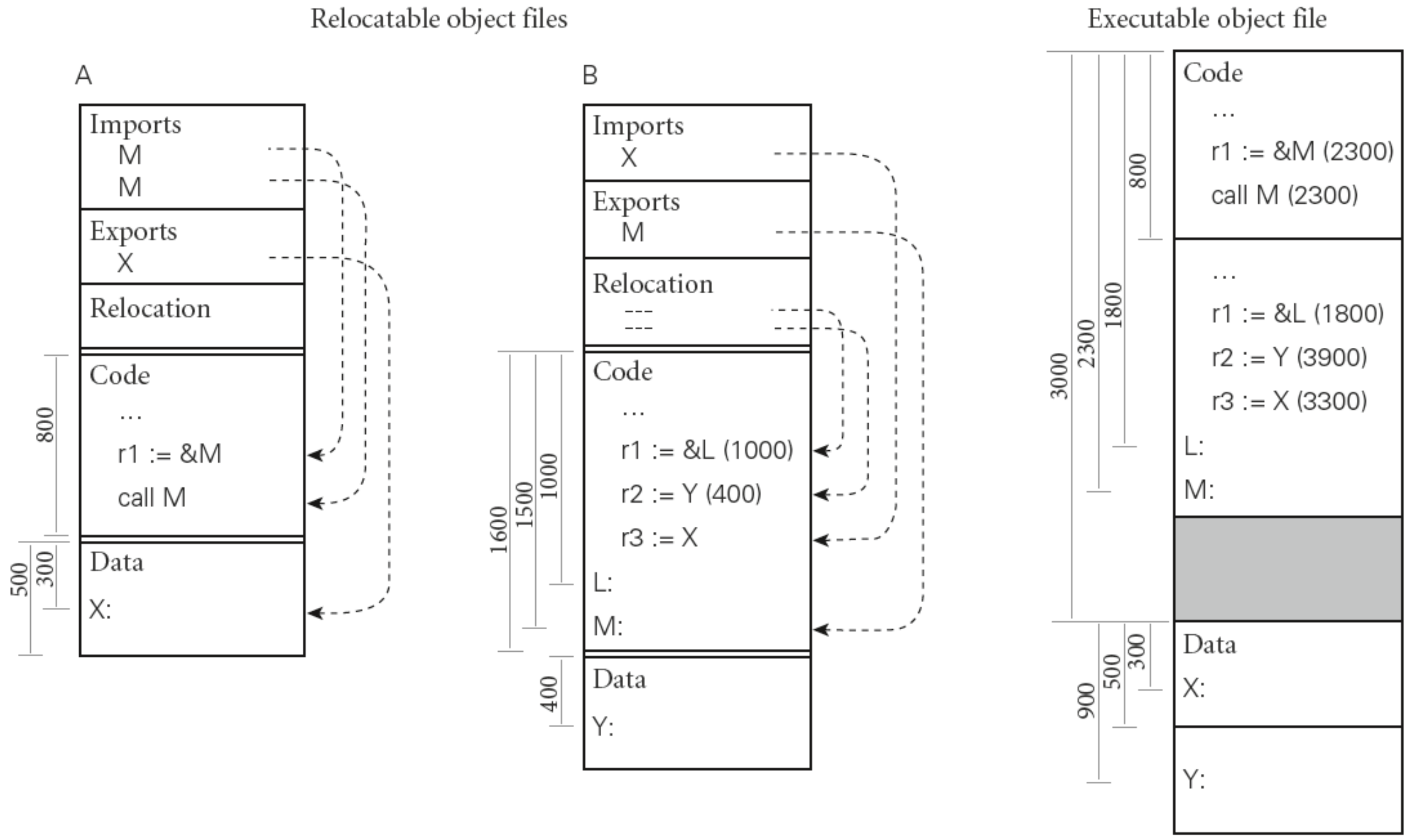


Figure 15.9 **Linking relocatable object files** A **and** B **to make an executable object file.** For simplicity of presentation, A's code section has been placed at offset 0, with B's code section immediately after, at offset 800 (addresses increase down the page). To allow the operating system to establish different protections for the code and data segments, A's data section has been placed at the next page boundary (offset 3000), with B's data section immediately after (offset 3500). External references to M and X have been set to use the appropriate addresses. Internal references to L and Y have been updated by adding in the starting addresses of B's code and data sections, respectively.

Dynamic Linking

- On a multi-user system, it is common for several instances of a program (an editor or web browser, for example) to be executing simultaneously
  - It would be highly wasteful to allocate space in memory for a separate, identical copy of the code of such a program for every running instance
- Many operating systems therefore keep track of the programs that are running, and set up memory mapping tables so that all instances of the same program share the same read-only copy of the program’s code segment
  - Each instance receives its own writable copy of the data segment
  - Code segment sharing can save enormous amounts of space
  - It does not work, however, for instances of programs that are similar but not identical