Compilation

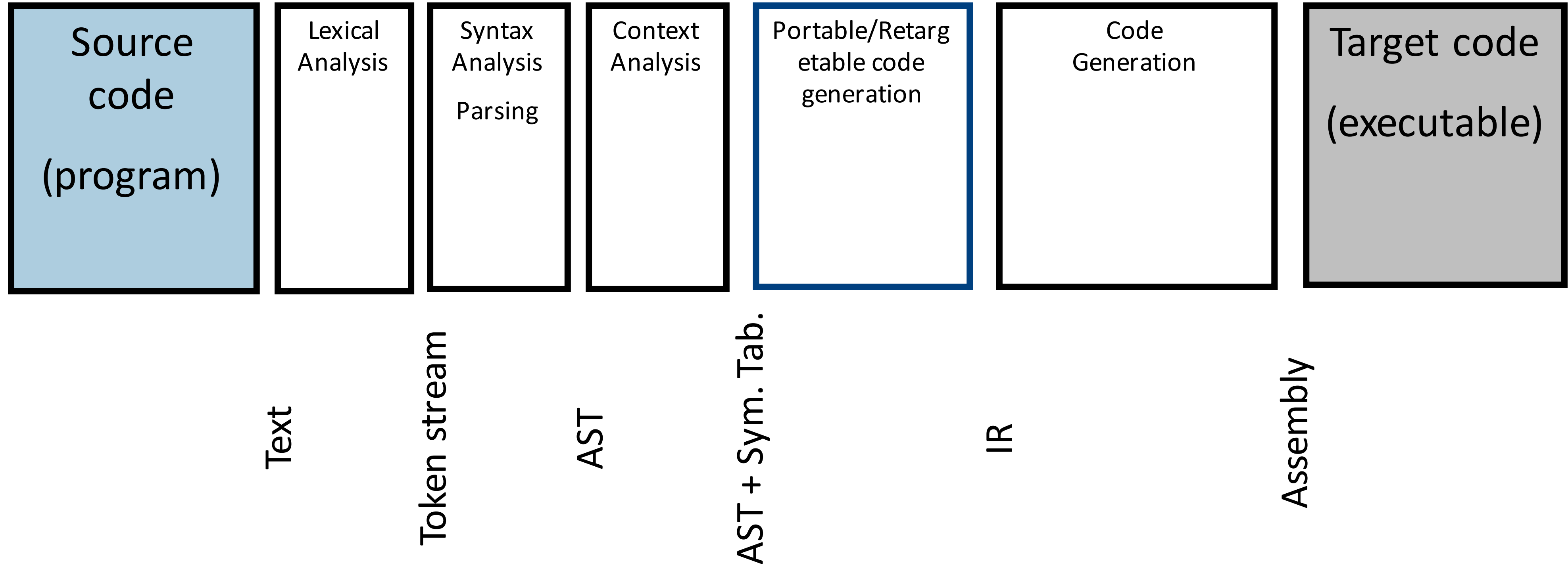
0368-3133 2015/16a

Lecture 11

Code generation, Assemblers, and linkers

Noam Rinetzky

Stages of compilation



Simple instructions

- Load_Mem a, R1 // R1 = a
- Store_Reg R1, a // a = R1
- Add R1, R2 // R2 = R2+1
- Load_Const cst, R1 // R1 = cst
- …

Not So Simple instructions

- Load_Mem a, R1 // R1 = a
- Store_Reg R1, a // a = R1
- Add R1, R2 // R2 = R2+1
- Load_Const cst, R1 // R1 = cst
- …

- Add_Scaled_Reg c, R1,R2 // R2= r2 + c * R1

Not So Simple instructions

- Load_Mem a, R1 // R1 = a
- Store_Reg R1, a // a = R1
- Add R1, R2 // R2 = R2+1
- Load_Const cst, R1 // R1 = cst
- …

- Add_Mem a, R1 // R1= a + R1
  - more efficient than:
    - Load_mem a, R2
    - Add R2,R1

Not So Simple instructions

- Load_Mem a, R1 // R1 = a
- Store_Reg R1, a // a = R1
- Add R1, R2 // R2 = R2+1
- Load_Const cst, R1 // R1 = cst
- …

- Add_Scaled_Reg c, R1,R2 // R2= R2 + c * R1
  - more efficient than: // Artificial instruction
    - Mult_Const c, R1
    - Add R1,R2

Goal

- Observation: Real machines have hundreds of instructions
  - + different addressing modes

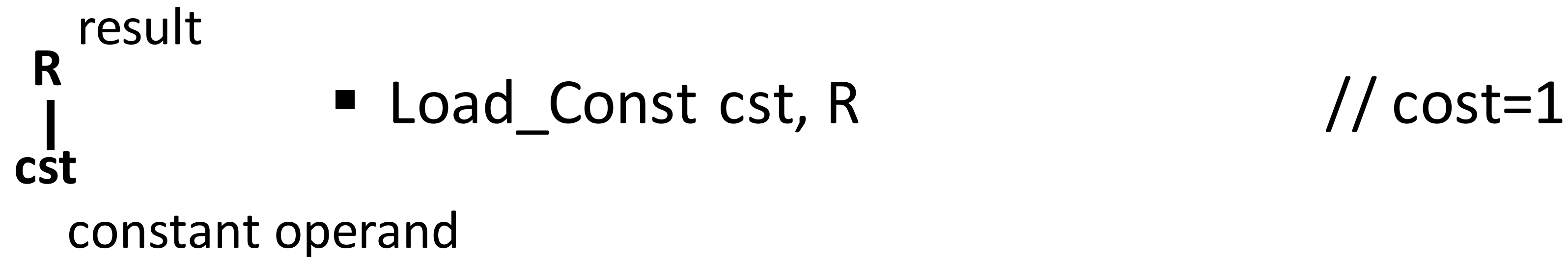
- Goal: Generate efficient code
  - Use available instructions

- Input: AST
  - Generated from the parse tree using simple techniques
- Output: A “more efficient” AST

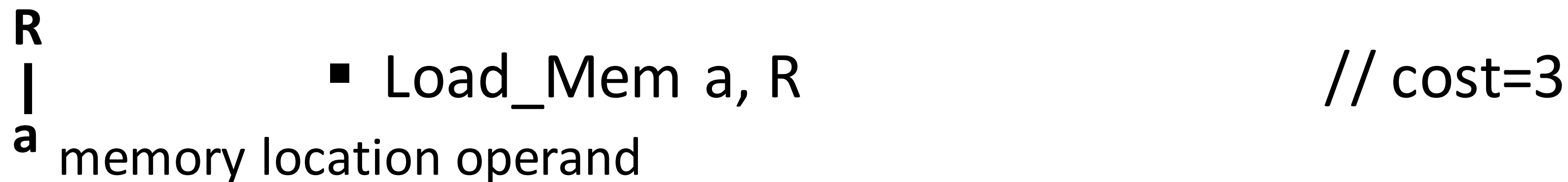
Code Generation Algorithms

- Top down (maximal munch)
- Bottom up (dynamic programming)

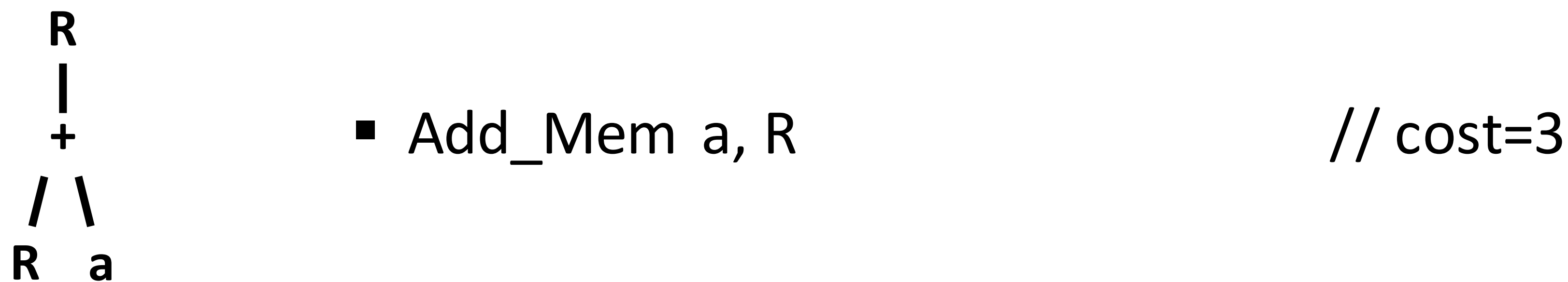
Instruction's AST: Pattern Tree



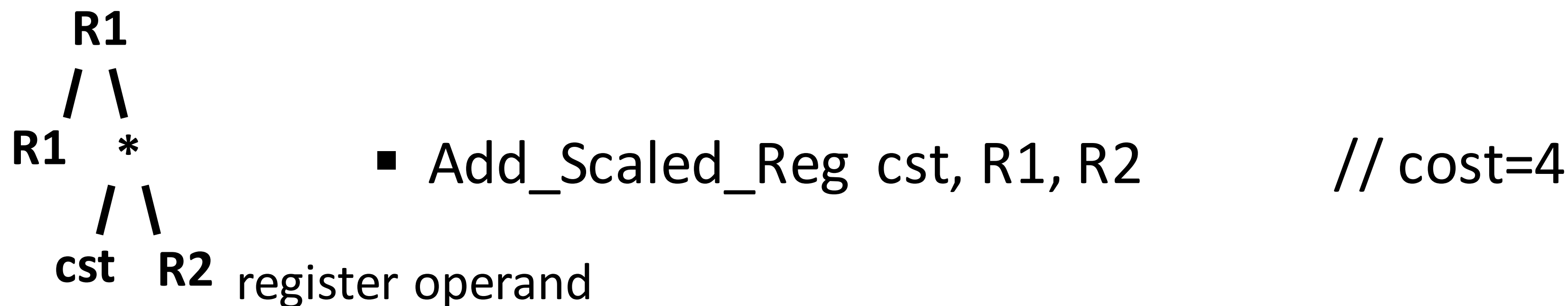
- Load_Const cst, R // cost=1



- Load_Mem a, R // cost=3



- Add_Mem a, R // cost=3



- Add_Scaled_Reg cst, R1, R2 // cost=4

Instruction’s AST: Pattern Tree

```
#1    R
      |
     cst
```

- Load_Const cst, R // cost=1

```
#2    R
      |
      a
```

- Load_Mem a, R // cost=3

```
#3    R
      |
      +
     / \
    R   a
```

- Add_Mem a, R // cost=3

```
       R1
        |
#7      +
       / \
     R1   *  #7.1
         / \
       cst  R2
```

- Add_Scaled_Reg cst, R1, R2 // cost=4

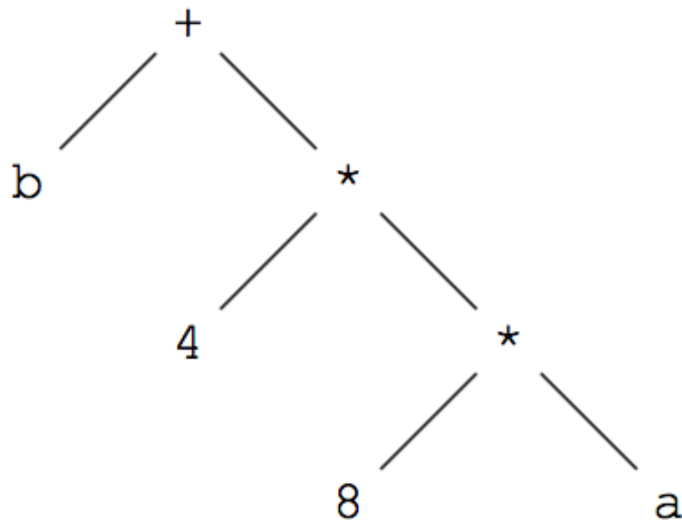
Instruction costs

- Measures time (cycle) / space (bytes)

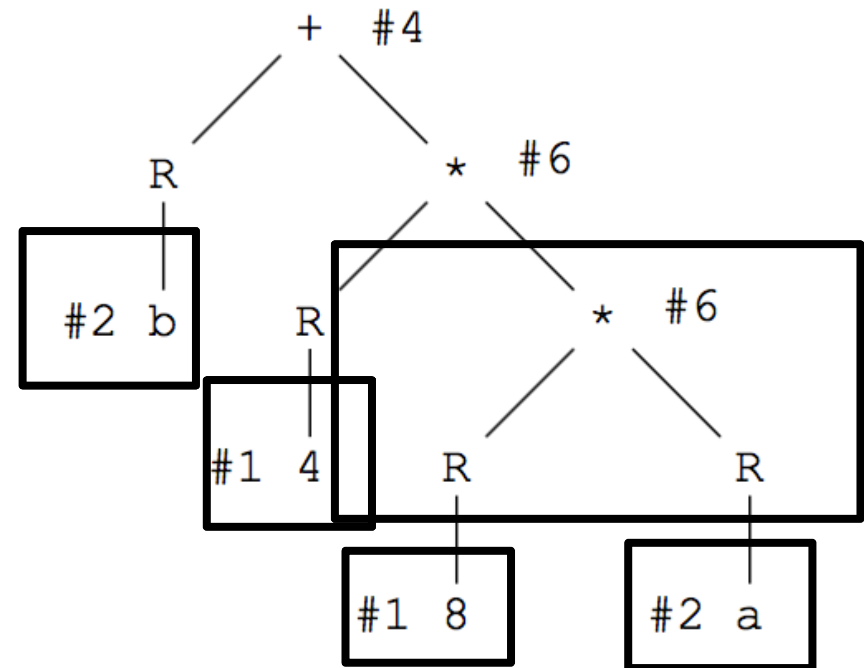
- Models:
  - Fixed,
    - e.g., Load_Constant costs 1 cycles
  - context-defendant
    - e.g., Add_Scaled_Reg c, R1,R2 // R2= R2 + c * R1

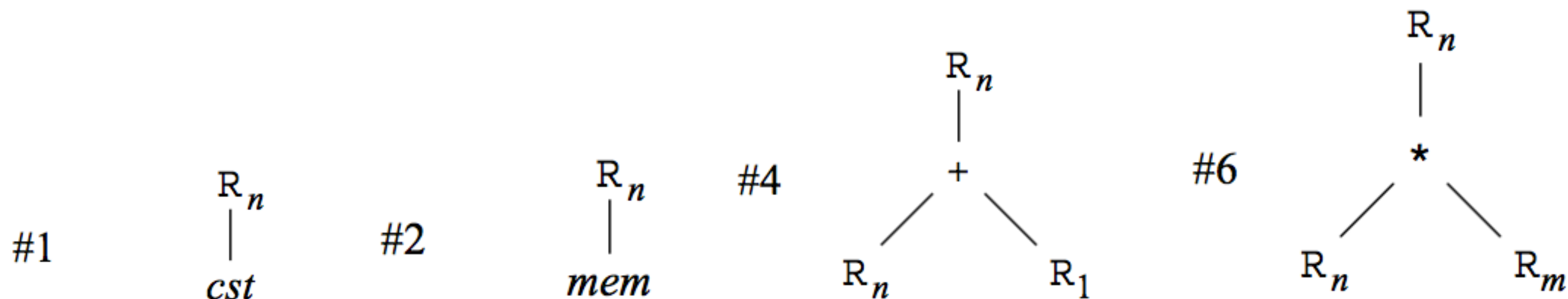
Example – Naïve rewrite

Input tree



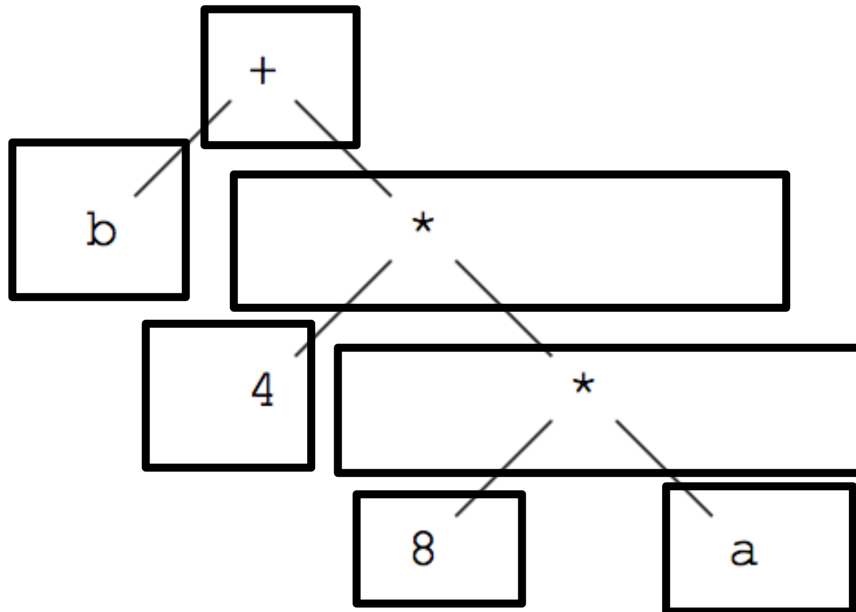
Naïve Rewrite



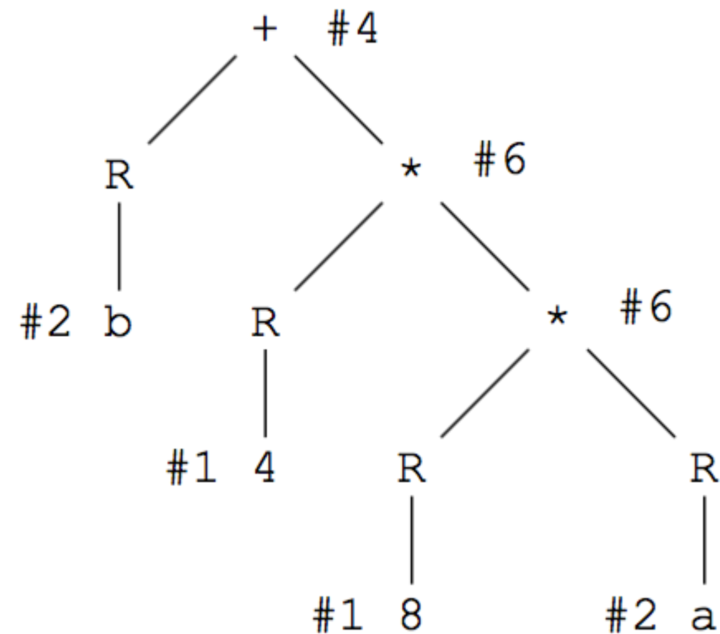


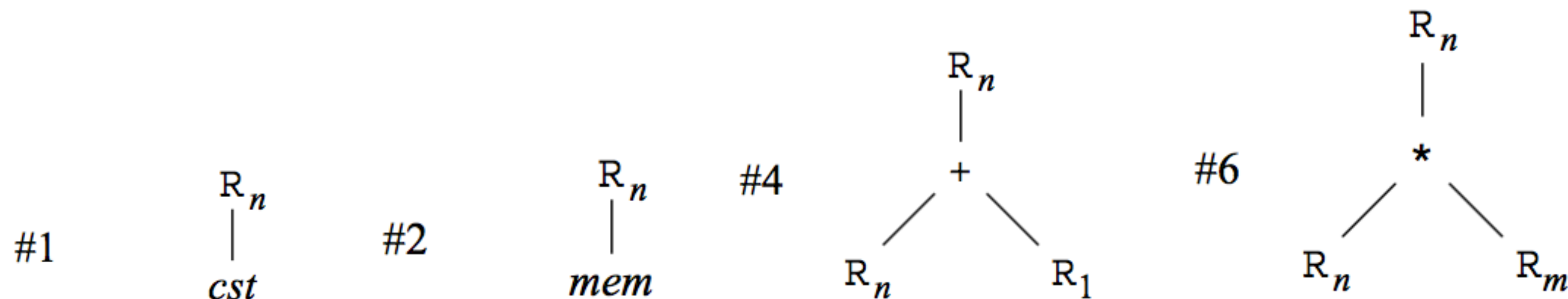
Example – Naïve rewrite

Input tree



Naïve Rewrite





Top-Down Rewrite Algorithm

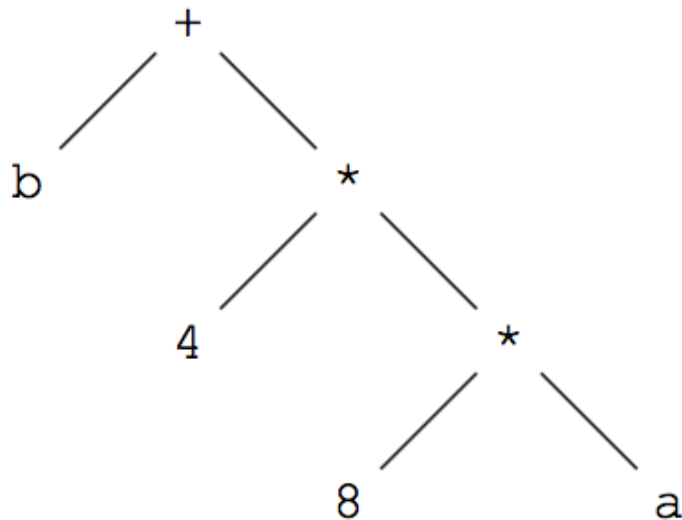
- aka **Maximal Munch**

- Based on **tiling**

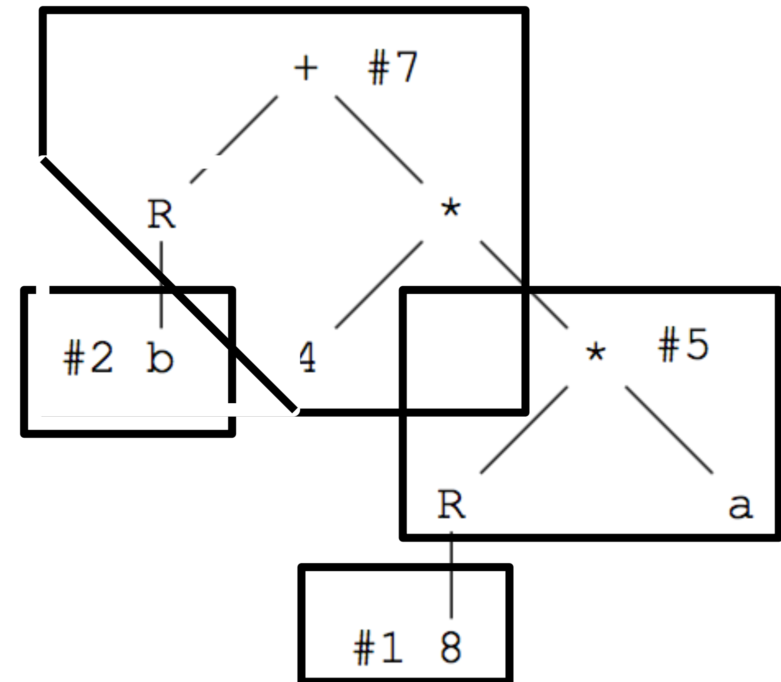
- Start from the root
- Choose largest tile
  - (covers largest number of nodes)
    - Break ties arbitrarily
- Continue recursively

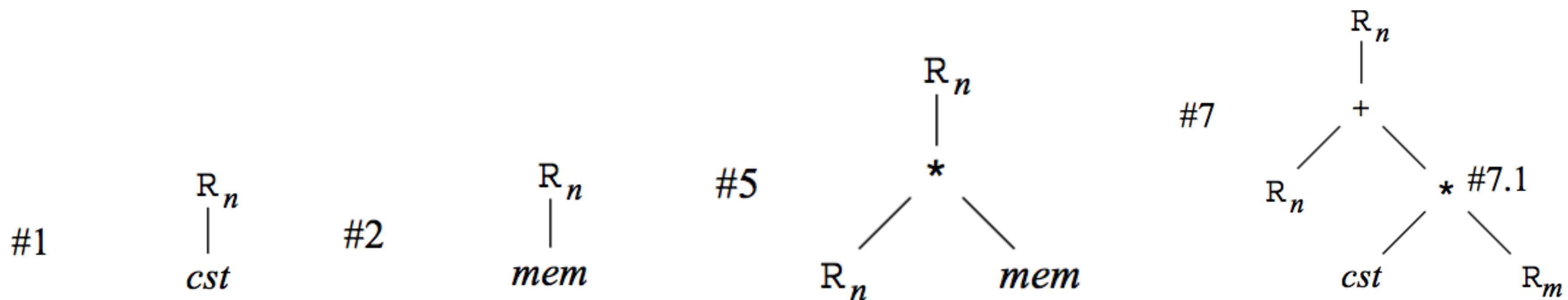
Top-down largest fit rewrite

Input tree



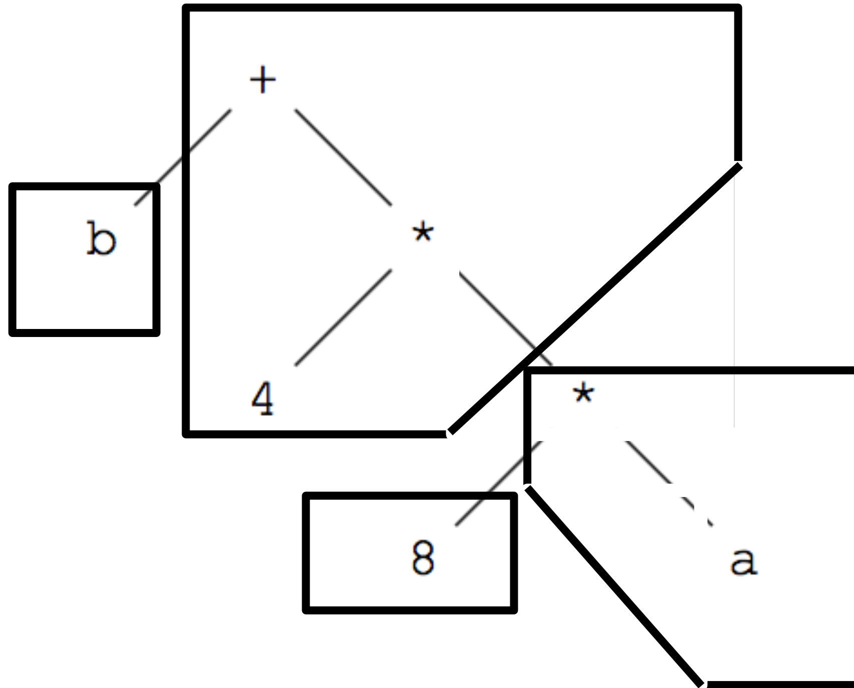
TDLF-Rewrite



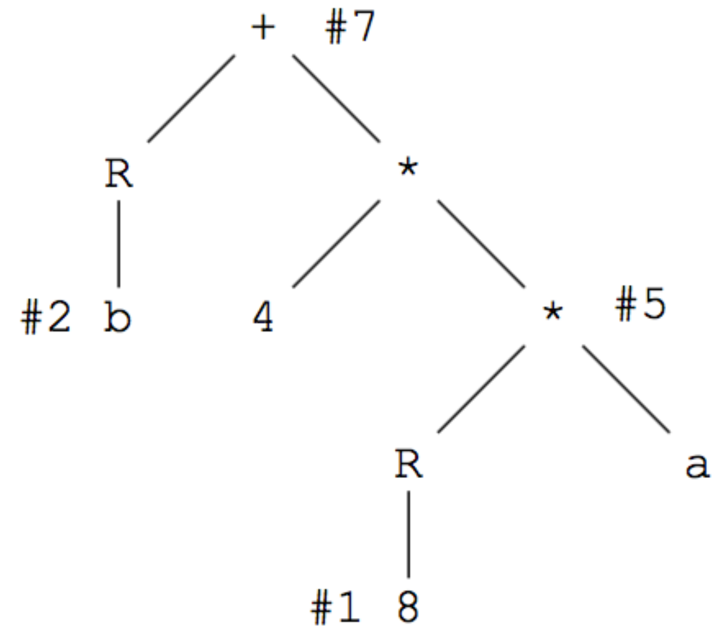


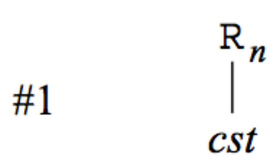
Top-down largest fit rewrite

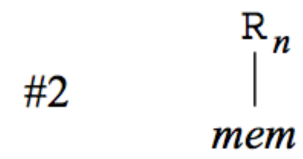
Input tree

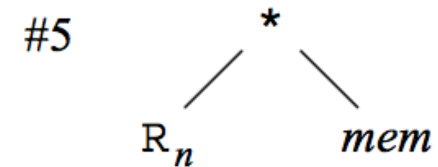


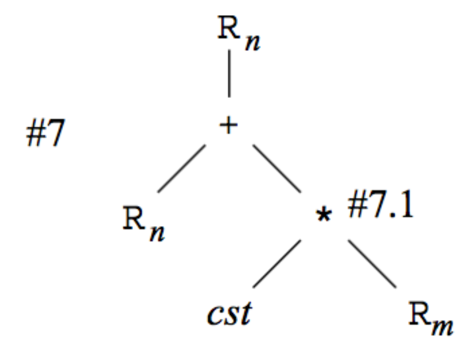
TDLF-Rewrite











Resulting code

Naïve Rewrite

```
Load_Const 8,R1    ; 1 unit
Load_Mem   a,R2    ; 3 units
Mult_Reg   R2,R1   ; 4 units
Load_Const 4,R2    ; 1 unit
Mult_Reg   R1,R2   ; 4 units
Load_Mem   b,R1    ; 3 units
Add_Reg    R2,R1   ; 1 unit
           Total   = 17 units
```

TDLF-Rewrite

```
Load_Const      8,R1     ; 1 unit
Mult_Mem        a,R1     ; 6 units
Load_Mem        b,R2     ; 3 units
Add_Scaled_Reg  4,R1,R2  ; 4 units
                Total    = 14 units
```

Top-Down Rewrite Algorithm

- **Maximal Munch**

- If for every node there exists a single node tile then the algorithm does not get stuck

Challenges

- How to find all possible rewrites?
- How do we find most efficient rewrite?
  - And do it efficiently?

BURS: Bottom Up Rewriting System

- How to find all possible rewrites?
- Solution: Bottom-Up pattern matching

- How do we find most efficient rewrite?
  - And do it efficiently?
- Solution: Dynamic Programming

BURS: Bottom Up Rewriting System

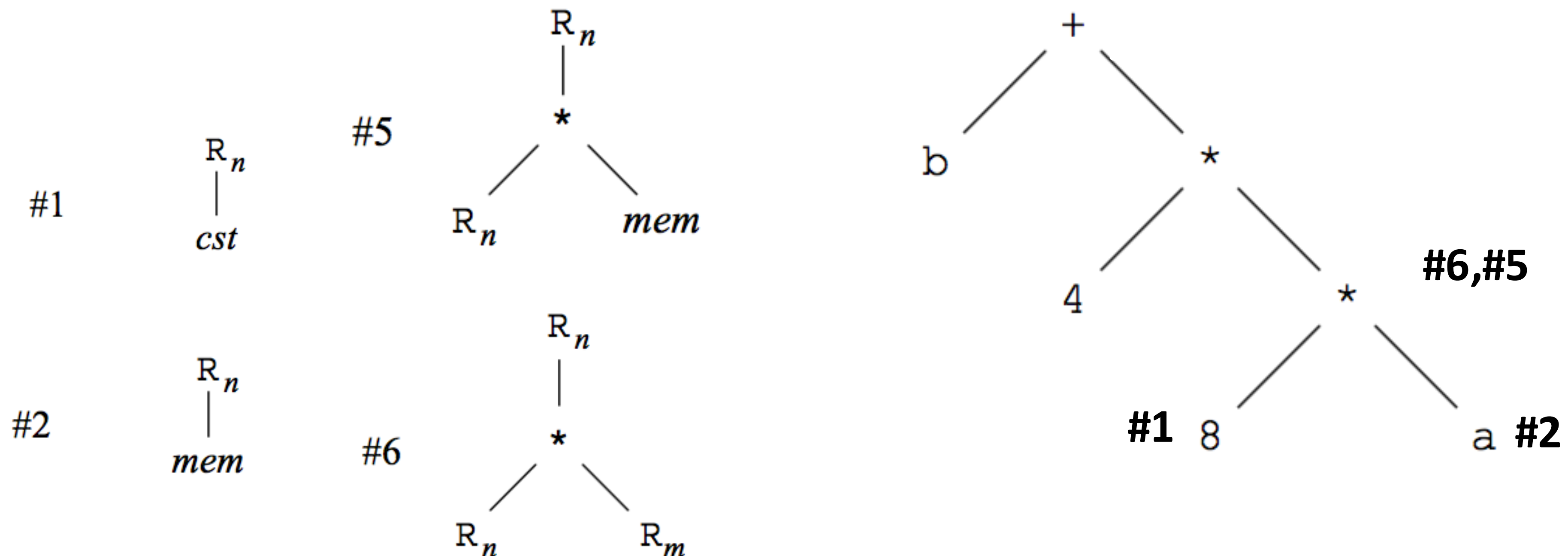
- Instruction-collecting Scan
  - Bottom up
  - Identify possible instructions for every node
  - Uses pattern-matching

- Instruction-selecting Scan
  - top-down
  - selects one (previously-identified) instruction

- Code-generation Scan
  - bottom-up
  - Generate code (sequence of instructions)

Bottom-up Pattern matching

- "Tree version" of lexical analysis
- Record at AST node N a label L of a pattern I if it is possible to identify (the part of pattern I rooted at) L at node N



Labels

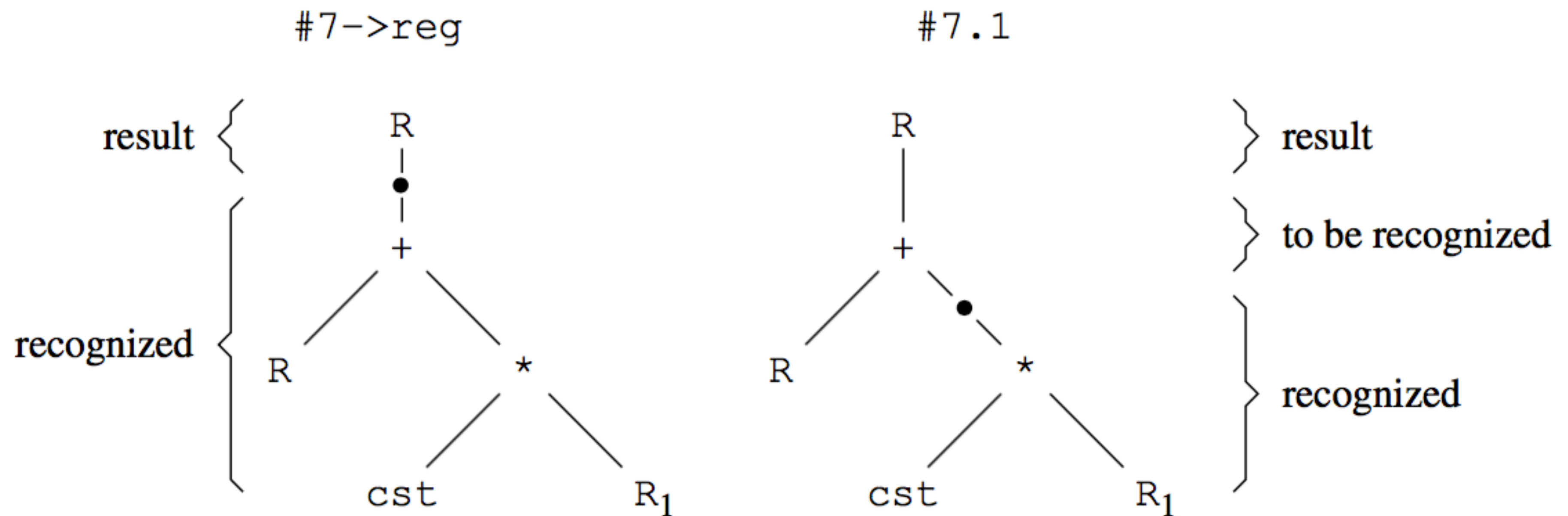
- Instruction leaves result of expression in a location
  - register
  - memory
  - constant

- The location determines which instructions can take the result as an operand

- Convention: use labels of the form L→location
  - e.g., #7→reg
  - Location is determined by Label
  - →cst means node is a constant

Dotted trees

- Labels can be seen as dotted trees



Bottom up pattern matching

type operator: "Load", '+', '*';
type location: "Constant", "Memory", "Register", a label;

type label: —— a node in a pattern tree
    field operator: operator;
    field firstOperand: location;
    field secondOperand: location;
    field result: location;

Bottom up pattern matching

```
procedure BottomUpPatternMatching (Node):
    if Node is an operation:
        BottomUpPatternMatching (Node.left);
        BottomUpPatternMatching (Node.right);
        Node.labelSet ← LabelSetFor (Node);
    else if Node is a constant:
        Node.labelSet ← LabelSetForConstant ();
    else —— Node is a variable:
        Node.labelSet ← LabelSetForVariable ();
```

Bottom up pattern matching

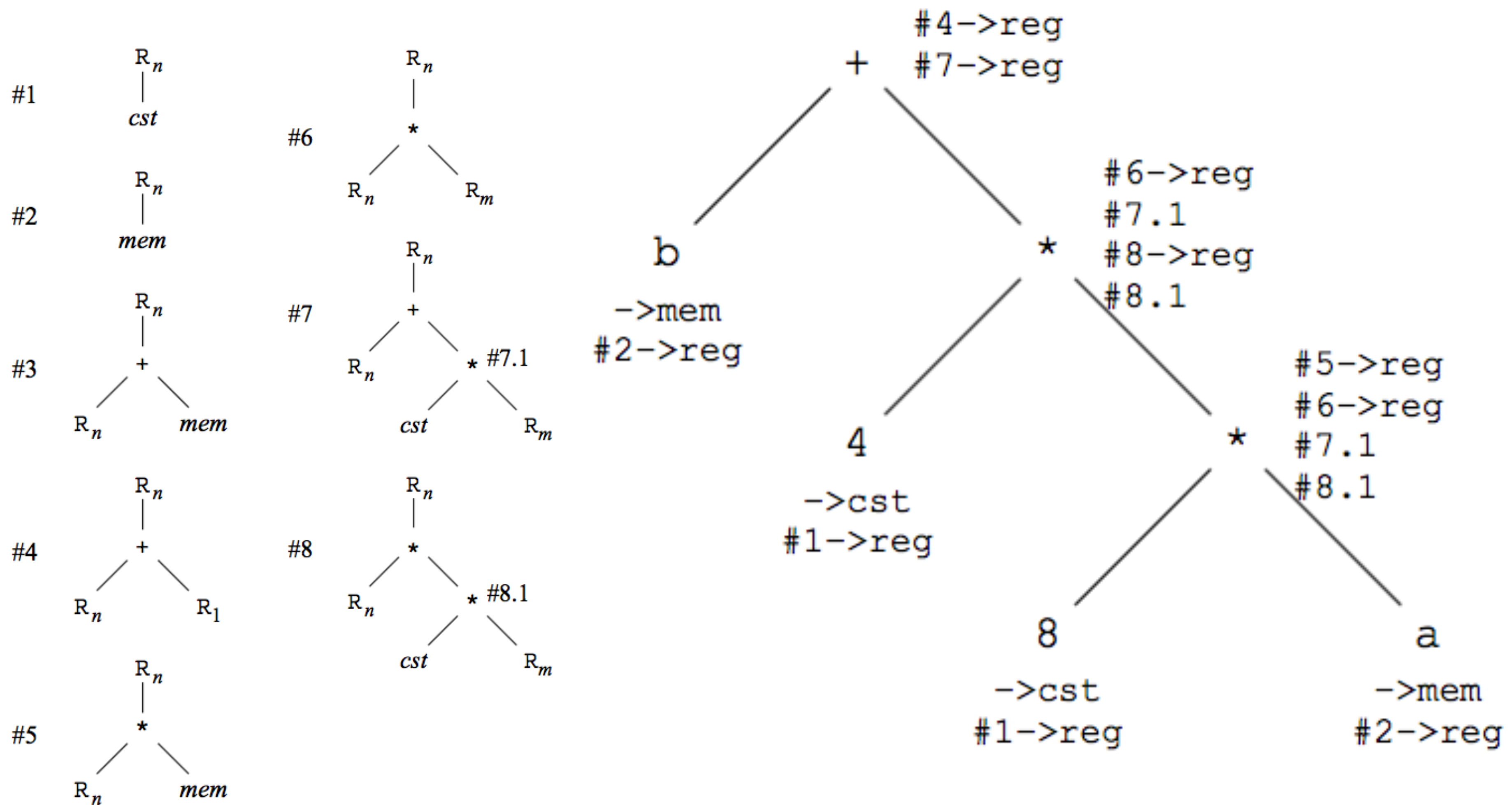
```
function LabelSetForConstant () returning a label set:
    LabelSet ← { (NoOperator, NoLocation, NoLocation, "Constant") };
    for each Label in the MachineLabelSet:
        if Label.operator = "Load" and Label.firstOperand = "Constant":
            Insert Label into LabelSet;
    return LabelSet;

function LabelSetForVariable () returning a label set:
    LabelSet ← { (NoOperator, NoLocation, NoLocation, "Memory") };
    for each Label in the MachineLabelSet:
        if Label.operator = "Load" and Label.firstOperand = "Memory":
            Insert Label into LabelSet;
    return LabelSet;
```

Bottom up pattern matching

```
function LabelSetFor (Node) returning a label set:
    LabelSet ← ∅;
    for each Label in MachineLabelSet:
        for each LeftLabel in Node.left.labelSet:
            for each RightLabel in Node.right.labelSet:
                if Label.operator = Node.operator
                        and Label.firstOperand = LeftLabel.result
                        and Label.secondOperand = RightLabel.result:
                    Insert Label into LabelSet;
    return LabelSet;
```

Example



Pattern matching using Tree Automaton

- Storing the set of patterns explicitly is redundant:
  - Number of patterns is fixed
  - Relation between pattern is know
- Idea:
  - Create a state machine over sets of patterns
    - Input: States of children + operator of root
    - Output: State of root

Pattern matching using Tree Automaton

```
procedure BottomUpPatternMatching (Node):
    if Node is an operation:
        BottomUpPatternMatching (Node.left);
        BottomUpPatternMatching (Node.right);
        Node.state ← NextState [Node.operator, Node.left.state, Node.right.state];
    else if Node is a constant:
        Node.state ← StateForConstant;
    else –– Node is a variable:
        Node.state ← StateForVariable;
```

Pattern matching using Tree Automaton

- In theory, may lead to large autoamton
  - Hundreds of operators
  - Thousands of states
  - Two dimensions
- In practice, most entries are empty

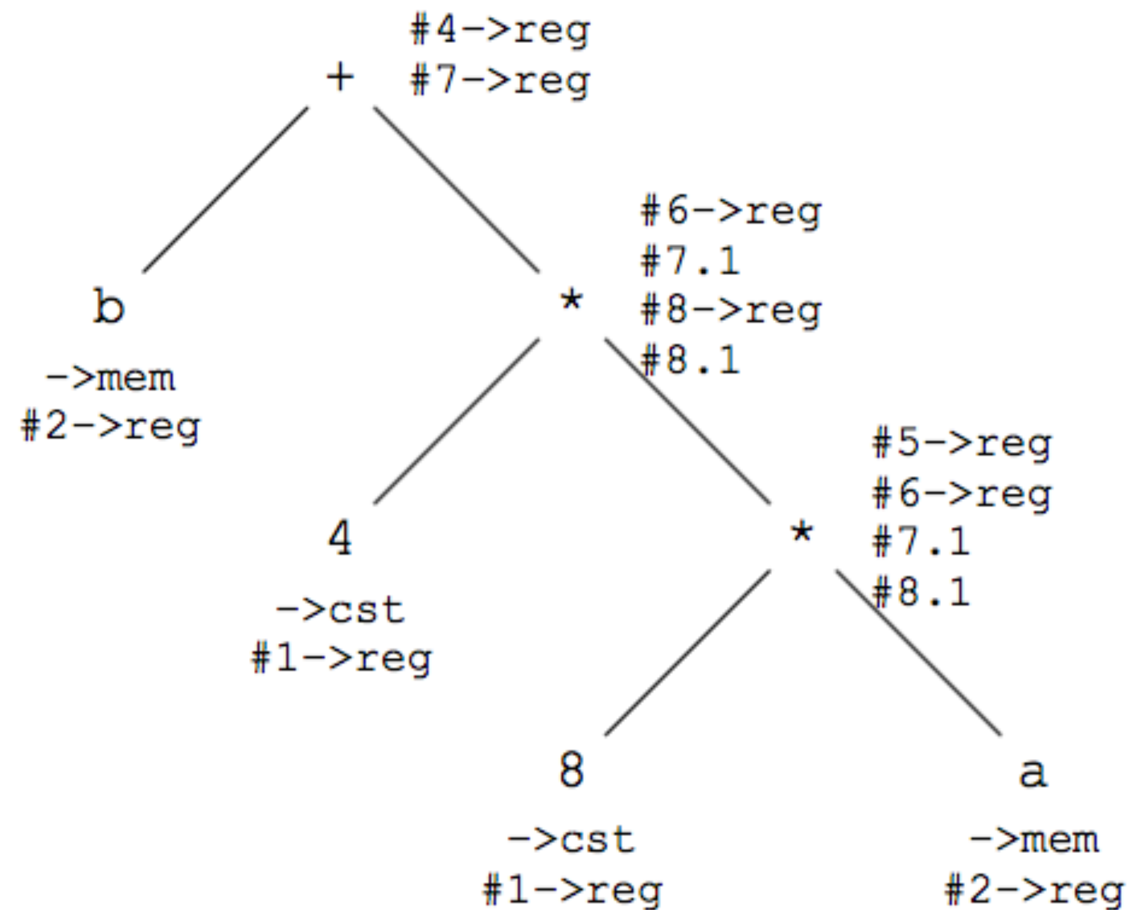
- Question:
  - How to handle operators with 1 operand?
  - with many operands?

Instruction Selection

We can build many trees!

Naïve approach:
 Try them all
 Select cheapest

Problem: Exponential blowup



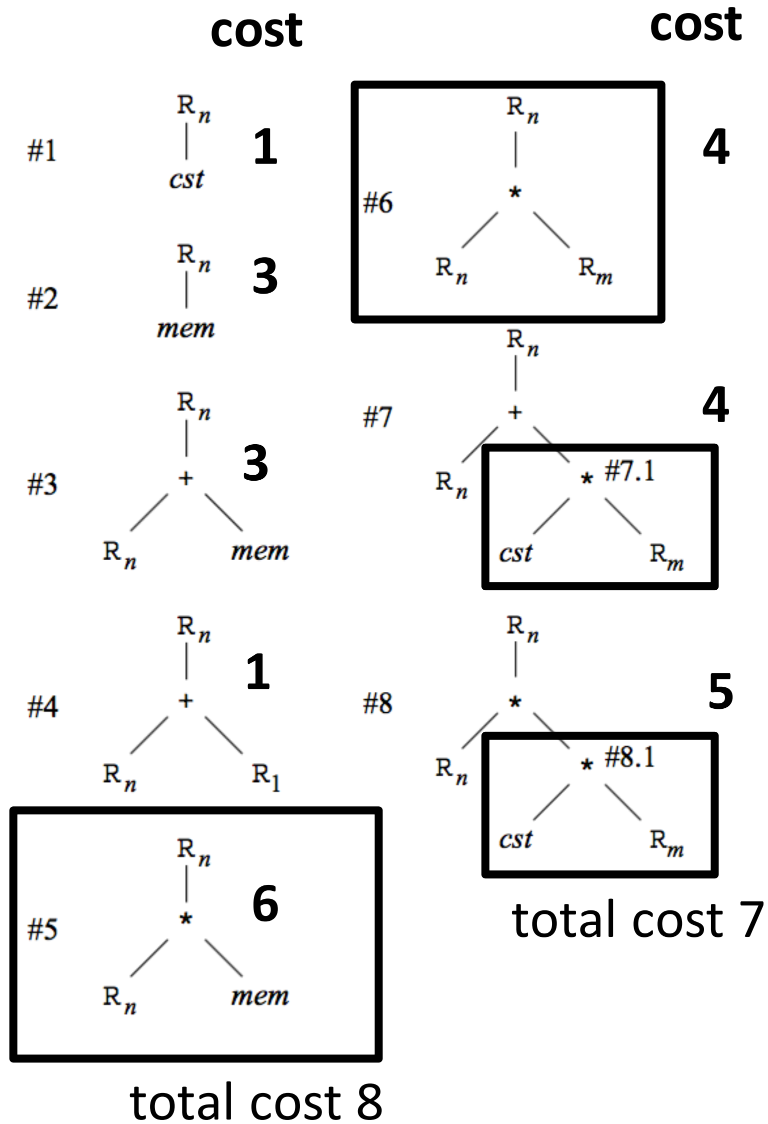
Instruction Selection with Dynamic Programming

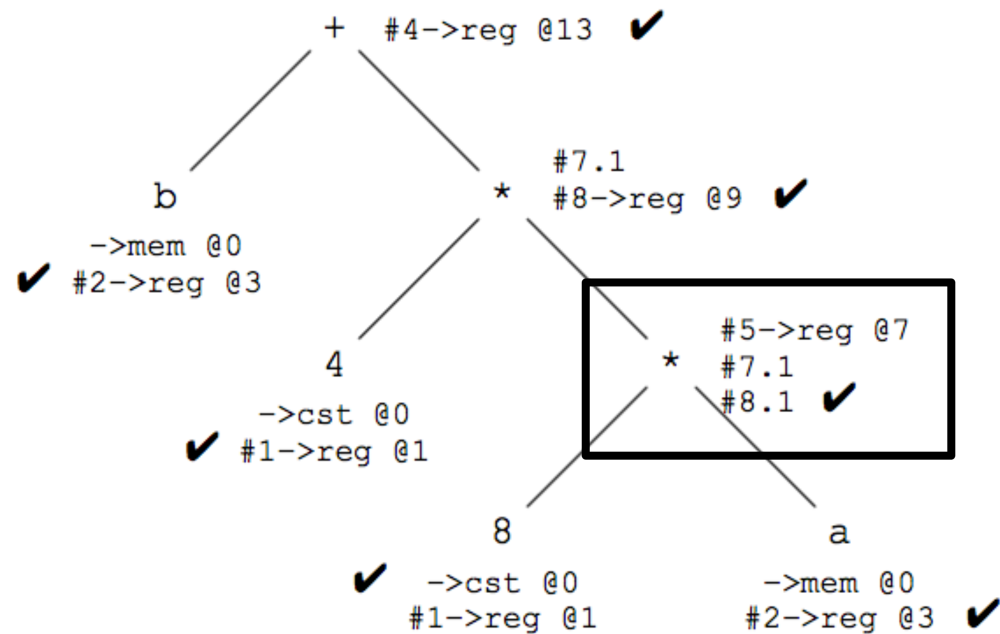
- Cost of sub-tree is sum of
  - The cost of the operator
  - The costs of the operands

- Idea: Compute the cost while detecting the patterns

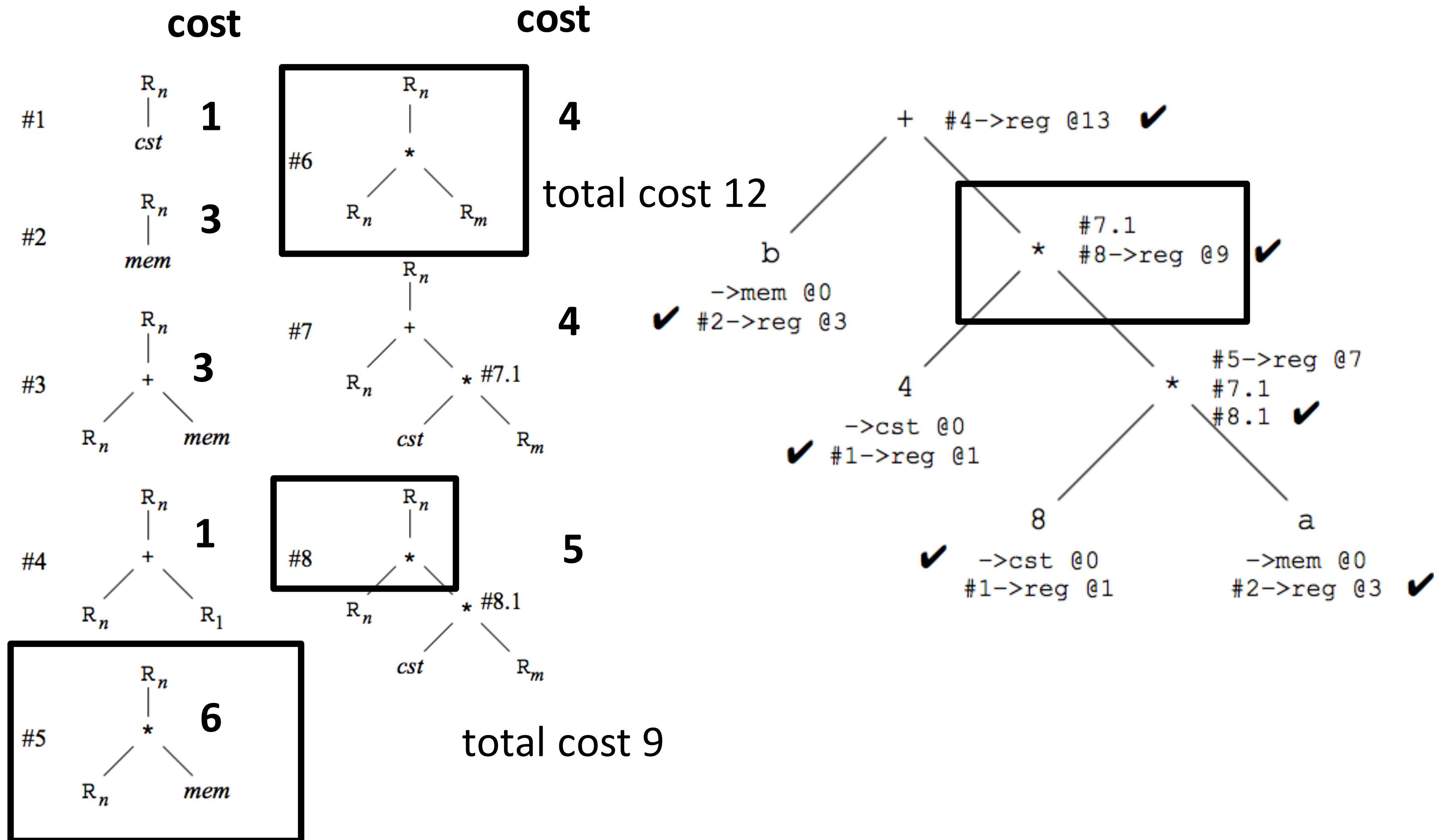
- Label: Label→ Location @ cost
  - E.g., #5→reg @ 3

Example

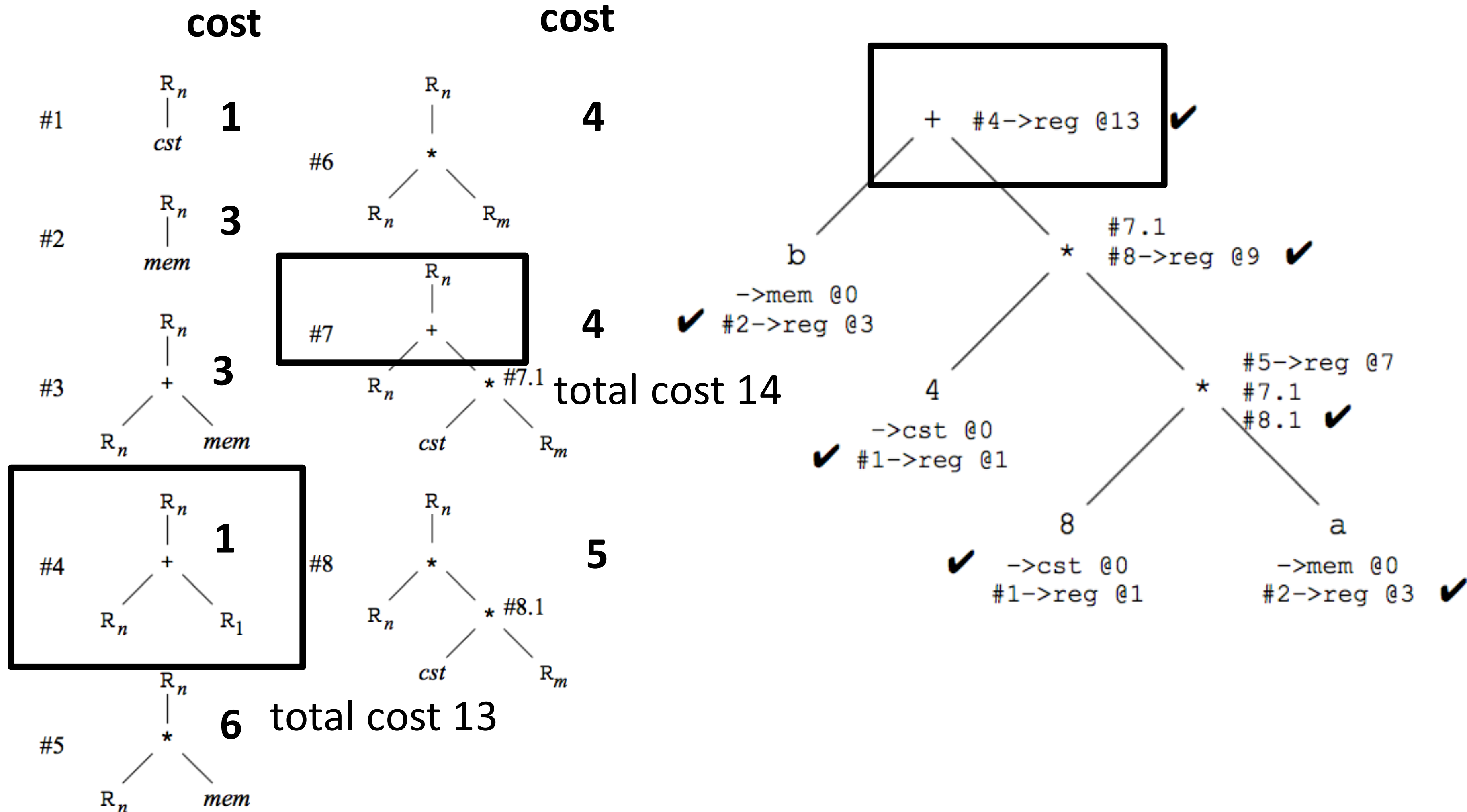




Example



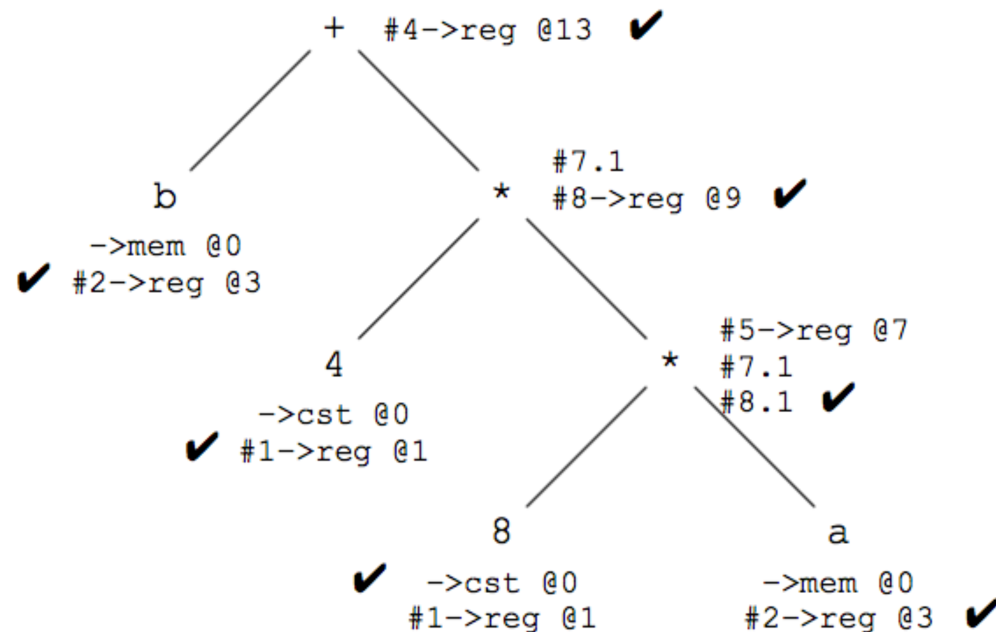
Example

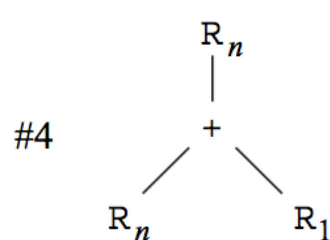


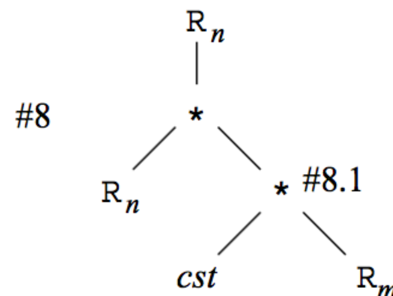
Instruction Selection

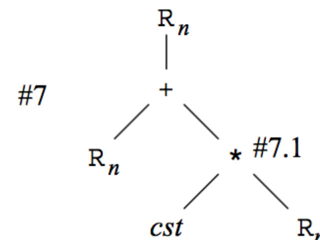
We can build the cheapest tree

- Select label at root based on location of result
- Select label at children based on expected location of operands









Linearize code

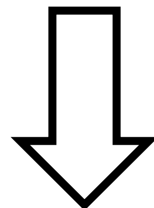
- Standard AST→Code procedure
  - E.g., create the register-heavy code first

```
Load_Mem          a,R1      ; 3 units
Load_Const        4,R2      ; 1 unit
Mult_Scaled_Reg   8,R1,R2   ; 5 units
Load_Mem          b,R1      ; 3 units
Add_Reg           R2,R1     ; 1 unit
                  Total     = 13 units
```

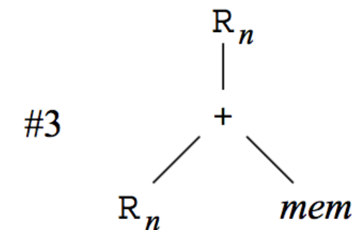
Commutativity

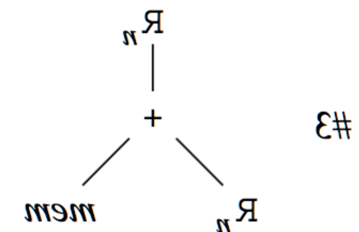
- We got the optimal tree wrt. given patterns

```
Load_Mem     b,R1    ; 3 units
Add_Reg      R2,R1   ; 1 unit
             Total   = 13 units
```



```
Add_Mem b,R2 ; 3 units
```





Solutions:
Add patterns
Mark commutative operations and handle them specially

Automatic code generators

- Combining pattern matching with instruction selection leads to great saving
- Possible if costs are constants

- Creates a complicated tree automaton

Compilation

0368-3133 2014/15a
Lecture 11

Assembler, Linker and Loader

Noam Rinetzky

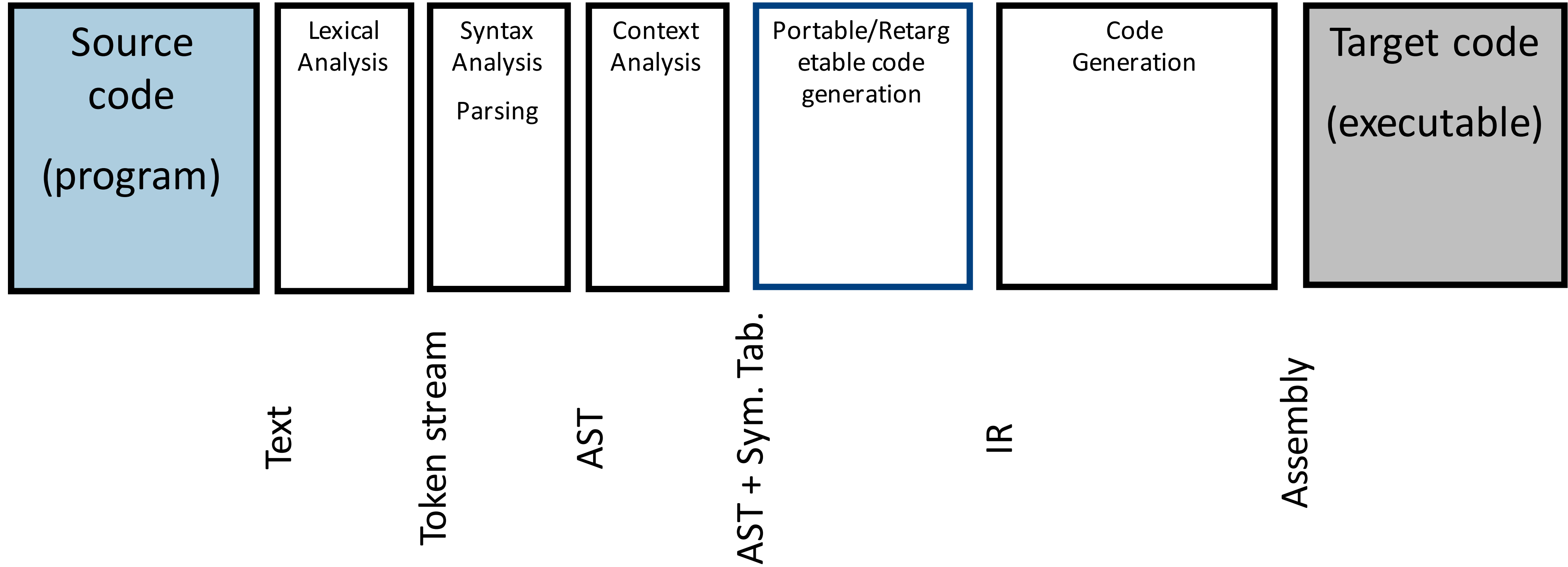
What is a compiler?

"A compiler is a computer program that transforms source code written in a programming language (source language) into another language (target language).

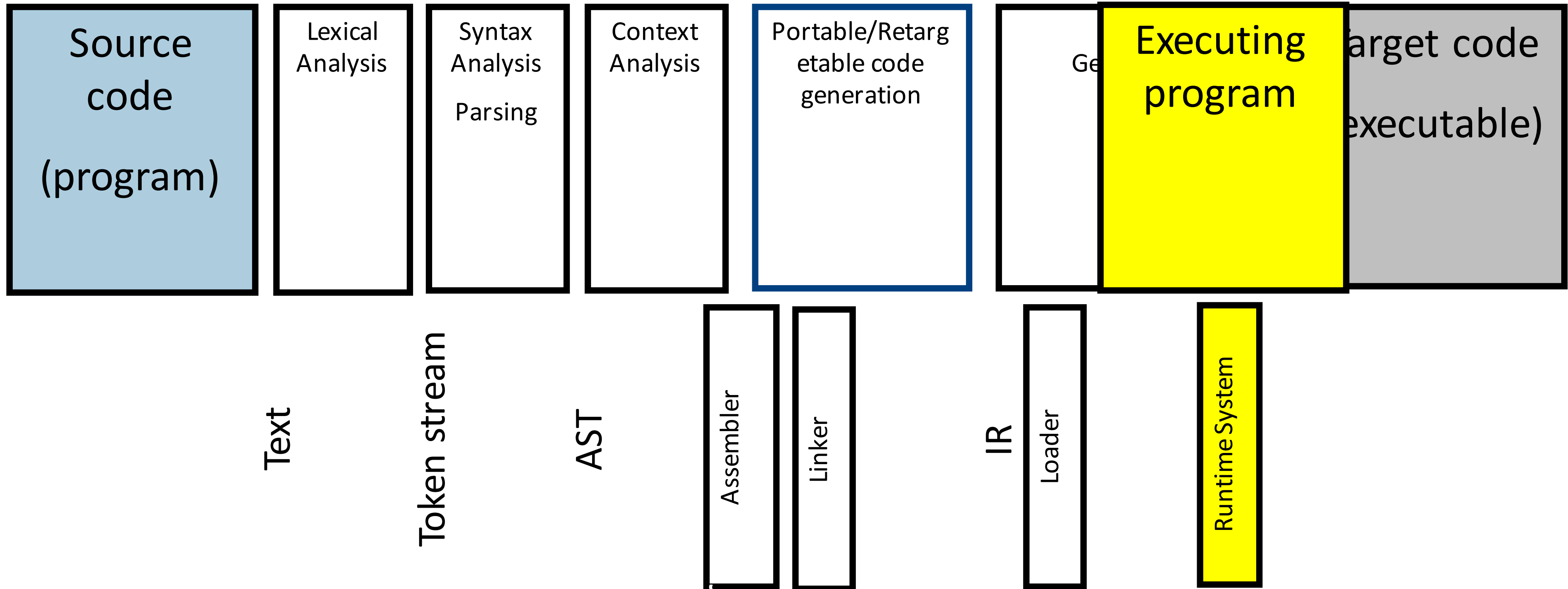
The most common reason for wanting to transform source code is to create an executable program."

--Wikipedia

Stages of compilation



Compilation → Execution



Program Runtime State

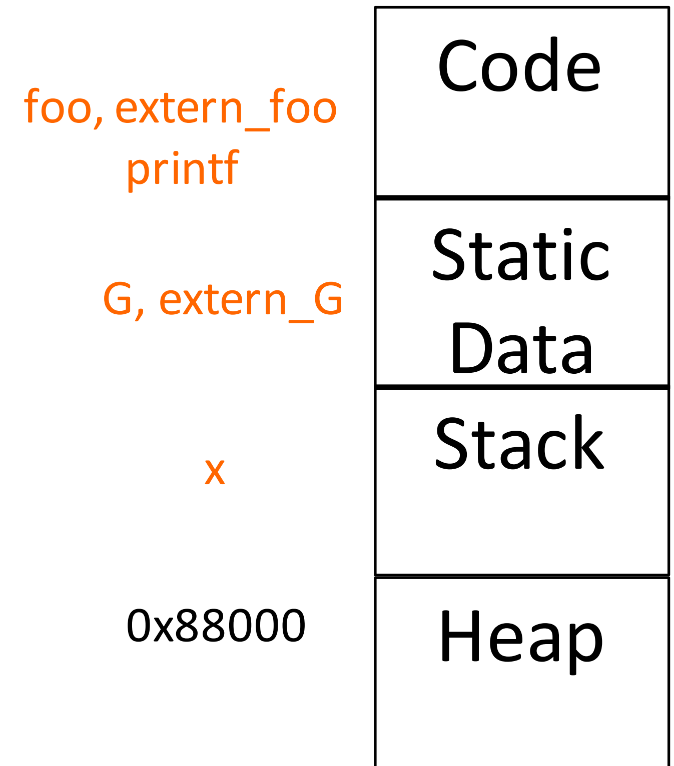
Registers

Address	Symbols	Segment
0x11000	foo, extern_foo printf	Code
0x22000	G, extern_G	Static Data
0x33000	x	Stack
0x88000		Heap
0x99000		

Challenges

- goto L2 ➔ JMP 0x110FF
- G:=3 ➔ MOV 0x2200F, 0..011
- foo() ➔ CALL 0x130FF
- extern_G := 1 ➔ MOV 0x2400F, 0..01
- extern_foo() ➔ CALL 0x140FF
- printf() ➔ CALL 0x150FF

- x:=2 ➔ MOV FP+32, 0…010
- goto L2 ➔ JMP [PC +] 0x000FF



Assembly → Image

Source program

Compiler

Assembly lang. program (.s)

Assembler

Machine lang. Module (.o): program (+library) modules

Linker

"compilation" time

Executable (".exe"):

"execution" time

Loader

Image (in memory):

Libraries (.o)
(dynamic loading)

Outline

- Assembly
- Linker / Link editor
- Loader

- Static linking
- Dynamic linking

Assembly → Image

Source file (*e.g., utils*) | Source file (*e.g., main*) | library
Compiler | **Compiler** | **Compiler**
Assembly (.s) | Assembly (.s) | Assembly (.s)
Assembler | **Assembler** | **Assembler**
Object (.o) | Object (.o) | Object (.o)

Linker

Executable (".elf")

Loader

Image (in memory):

Assembler

- Converts (symbolic) assembler to binary (object) code
    - Object files contain a combination of machine instructions, data, and information needed to place instructions properly in memory
  - Yet another(simple) compiler
    - One-to one translation
- Converts constants to machine repr. (3➔0…011)
- Resolve internal references
- Records info for code & data relocation

Object File Format

Header	Text Segment	Data Segment	Relocation Information	Symbol Table	Debugging Information

- Header: Admin info + “file map”
- Text seg.: machine instruction
- Data seg.: (Initialized) data in machine format
- Relocation info: instructions and data that depend on absolute addresses
- Symbol table: “exported” references + unresolved references

Handling Internal Addresses

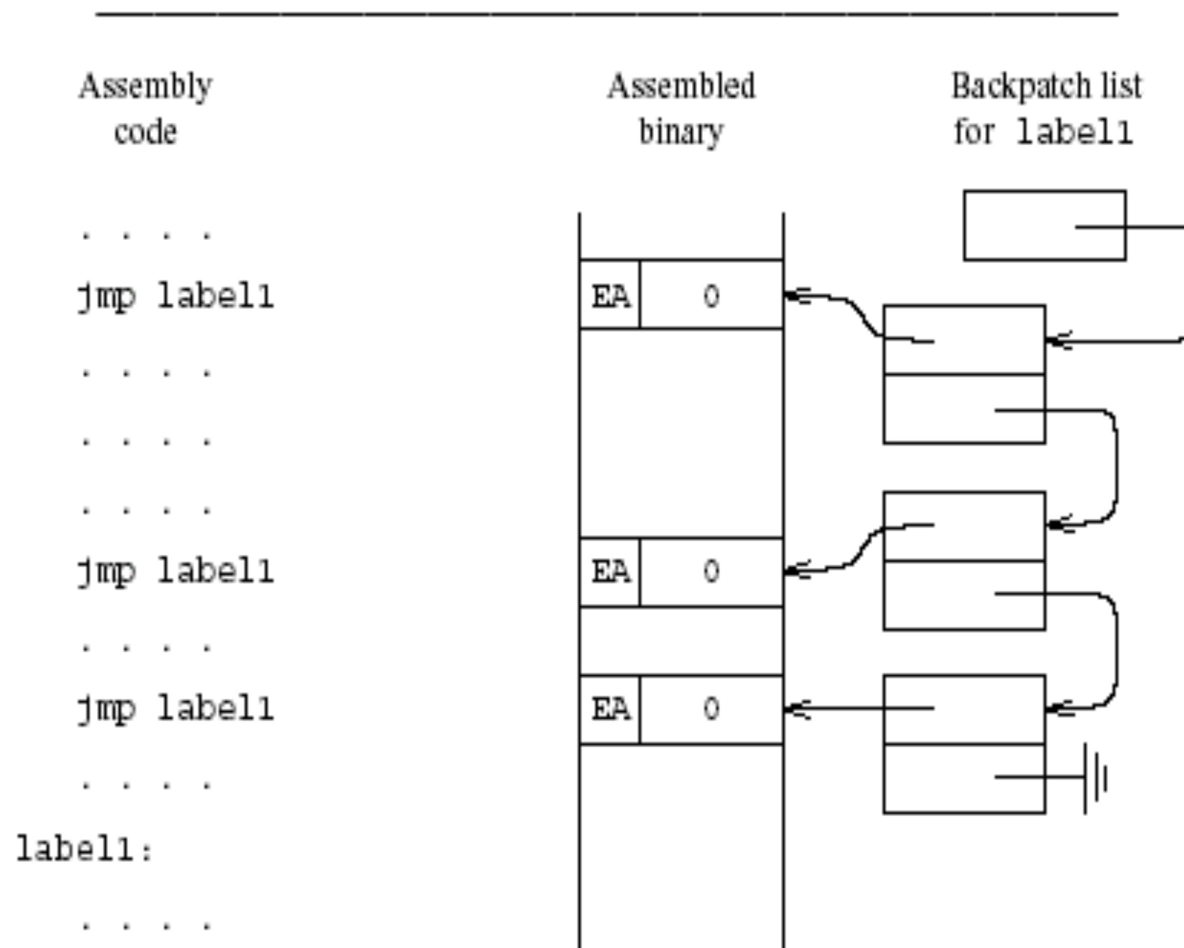
```
.data
        ...
        .align 8
var1:
        .long 666
        ...
.code
        ...
        addl var1,%eax
        ...
        jmp label1
        ...
label1:
        ...
        ...
```

Resolving Internal Addresses

- Two scans of the code
  - Construct a table label → address
  - Replace labels with values

- One scan of the code (Backpatching)
  - Simultaneously construct the table and resolve symbolic addresses
    - Maintains list of unresolved labels
  - Useful beyond assemblers

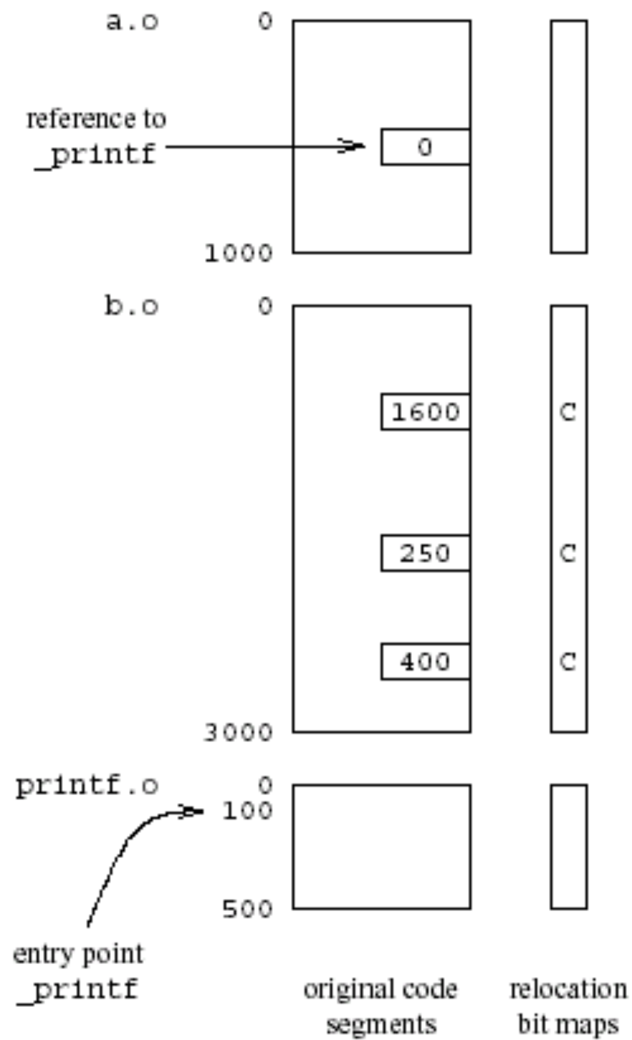
Backpatching



Handling External Addresses

- Record symbol table in “external” table
  - Exported (defined) symbols
    - G, foo()
  - Imported (required) symbols
    - Extern_G, extern_bar(), printf()
- Relocation bits
  - Mark instructions that depend on absolute (fixed) addresses
    - Instructions using globals,

Example



External references resolved by the Linker using the relocation info.

Example of External Symbol Table

External symbol	Type	Address
_options	entry point	50 data
__main	entry point	100 code
_printf	reference	500 code
_atoi	reference	600 code
_printf	reference	650 code
_exit	reference	700 code
_msg_list	entry point	300 data
_Out_Of_Memory	entry point	800 code
_fprintf	reference	900 code
_exit	reference	950 code
_file_list	reference	4 data

Assembler Summary

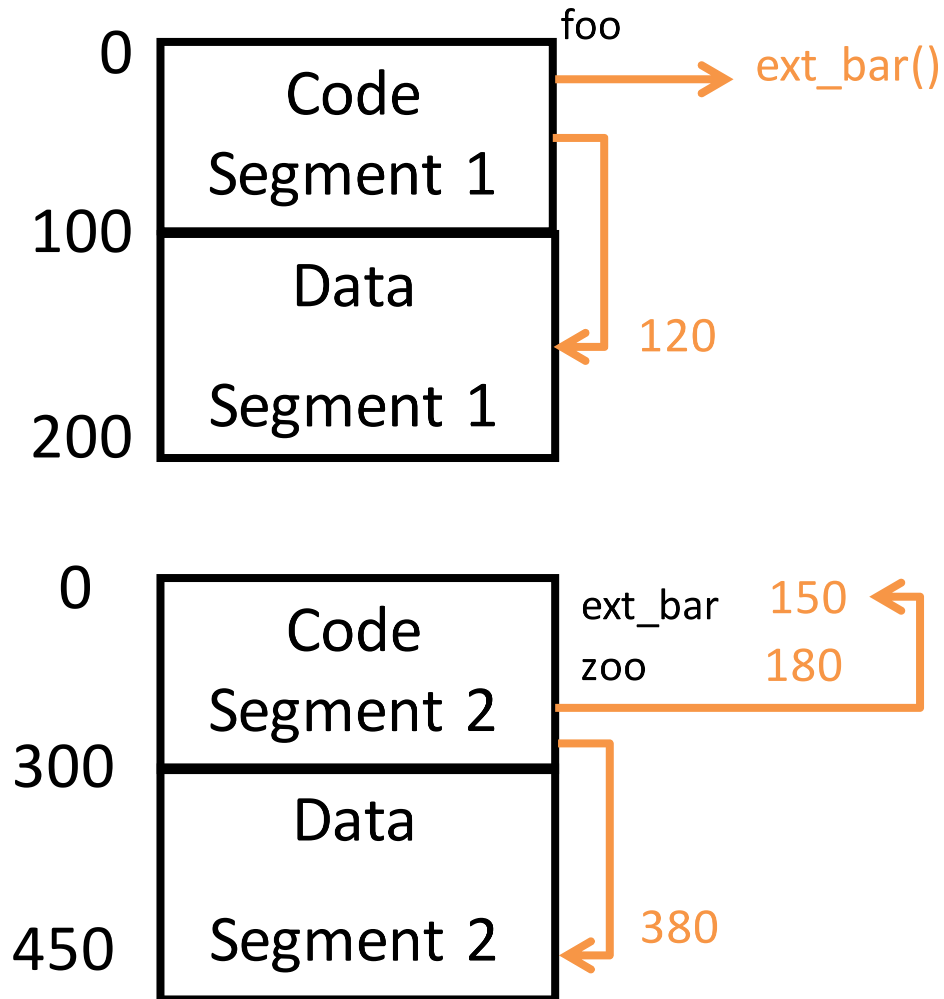
- Converts symbolic machine code to binary
  - addl %edx, %ecx ⇒ 000 0001 11 010 001 = 01 D1 (Hex)
- Format conversions
  - 3 → 0x0..011 or 0x000000110…0
- Resolves internal addresses

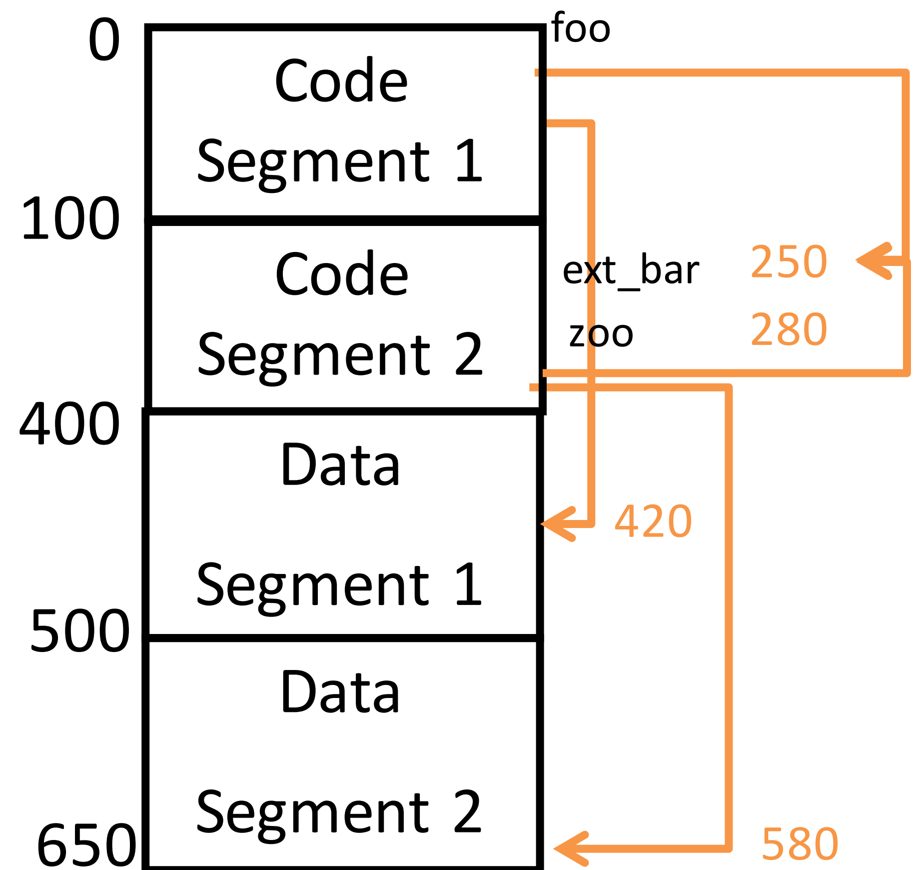
- Some assemblers support overloading
  - Different opcodes based on types

Linker

- Merges object files to an executable
  - Enables separate compilation
- Combine memory layouts of object modules
  - Links program calls to library routines
    - printf(), malloc()
  - Relocates instructions by adjusting absolute references
  - Resolves references among files

Linker





Relocation information

- Information needed to change addresses

- Positions in the code which contains addresses
  - Data
  - Code

- Two implementations
  - Bitmap
  - Linked-lists

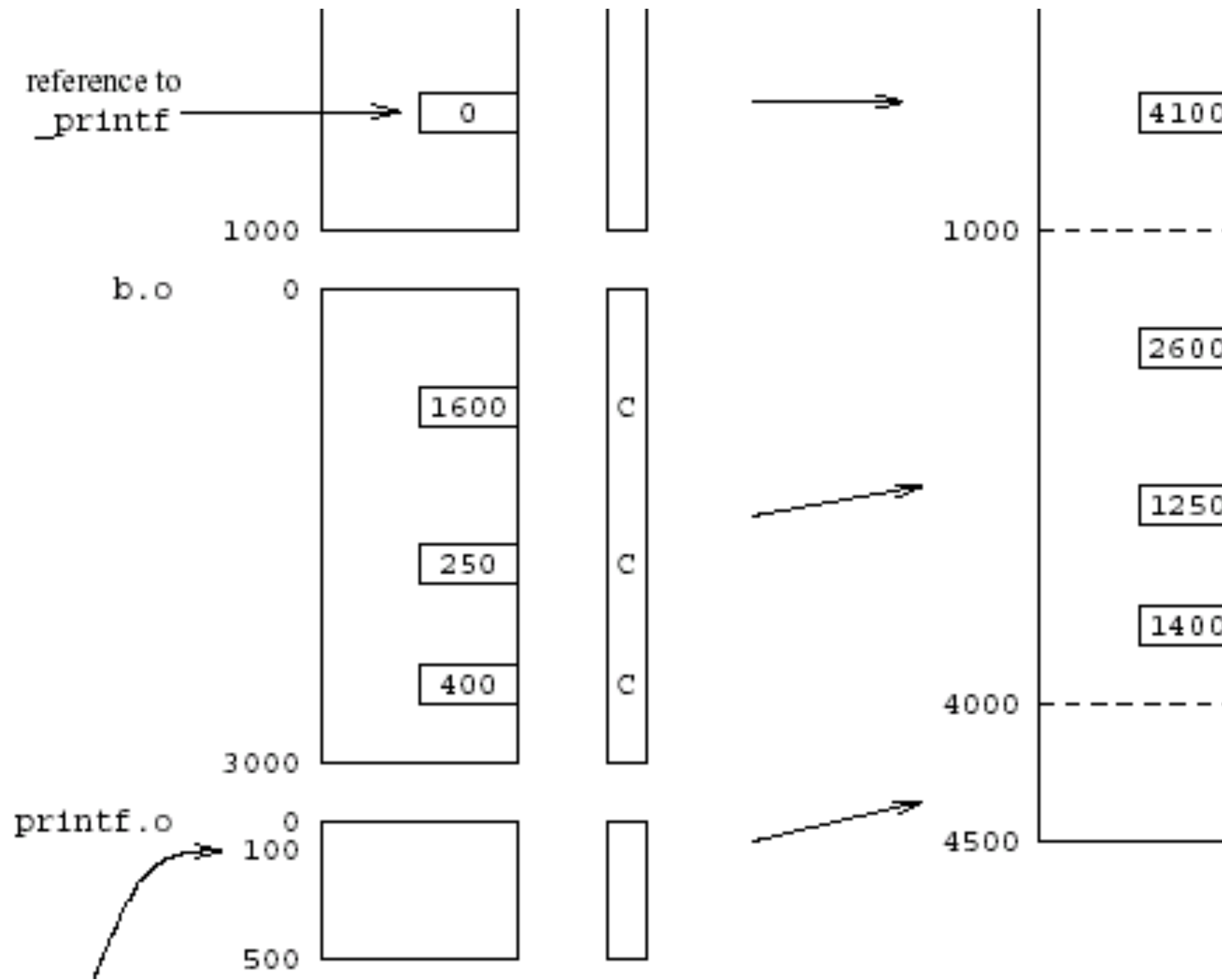
External References

- The code may include references to external names (identifiers)
  - Library calls
  - External data
- Stored in external symbol table

Example of External Symbol Table

External symbol	Type	Address
_options	entry point	50 data
__main	entry point	100 code
_printf	reference	500 code
_atoi	reference	600 code
_printf	reference	650 code
_exit	reference	700 code
_msg_list	entry point	300 data
_Out_Of_Memory	entry point	800 code
_fprintf	reference	900 code
_exit	reference	950 code
_file_list	reference	4 data

Example



Linker (Summary)

- Merge several executables
  - Resolve external references
  - Relocate addresses
- User mode
- Provided by the operating system
  - But can be specific for the compiler
    - More secure code
    - Better error diagnosis

Linker Design Issues

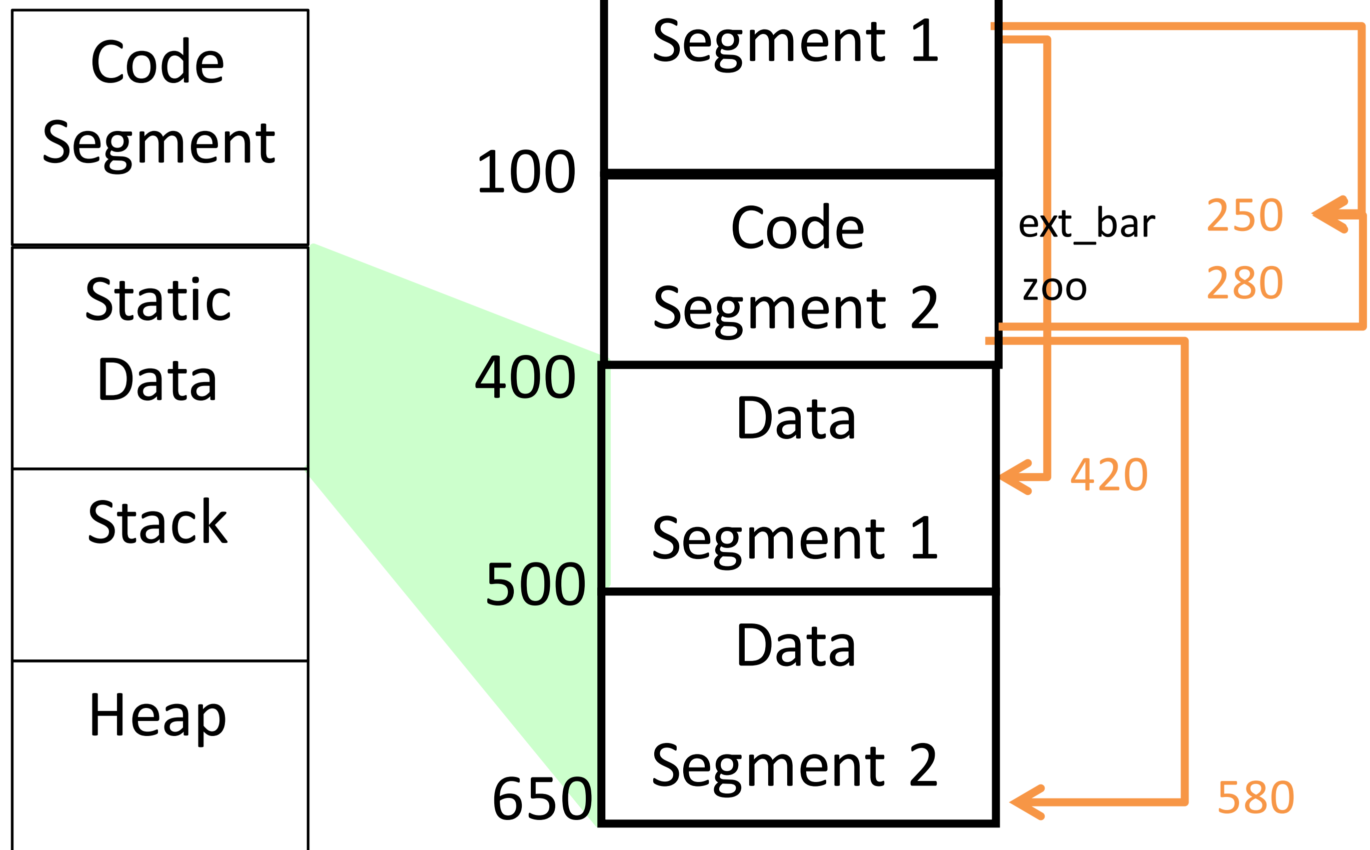
- Merges
  - Code segments
  - Data segments
  - Relocation bit maps
  - External symbol tables
- Retain information about static length
- Real life complications
  - Aggregate initializations
  - Object file formats
  - Large library
  - Efficient search procedures

Loader

- Brings an executable file from disk into memory and starts it running
  - Read executable file's header to determine the size of text and data segments
  - Create a new address space for the program
  - Copies instructions and data into memory
  - Copies arguments passed to the program on the stack
- Initializes the machine registers including the stack ptr
- Jumps to a startup routine that copies the program's arguments from the stack to registers and calls the program's main routine

Program Loading

Registers



Loader (Summary)

- Initializes the runtime state
- Part of the operating system
  - Privileged mode
- Does not depend on the programming language
- “Invisible activation record”

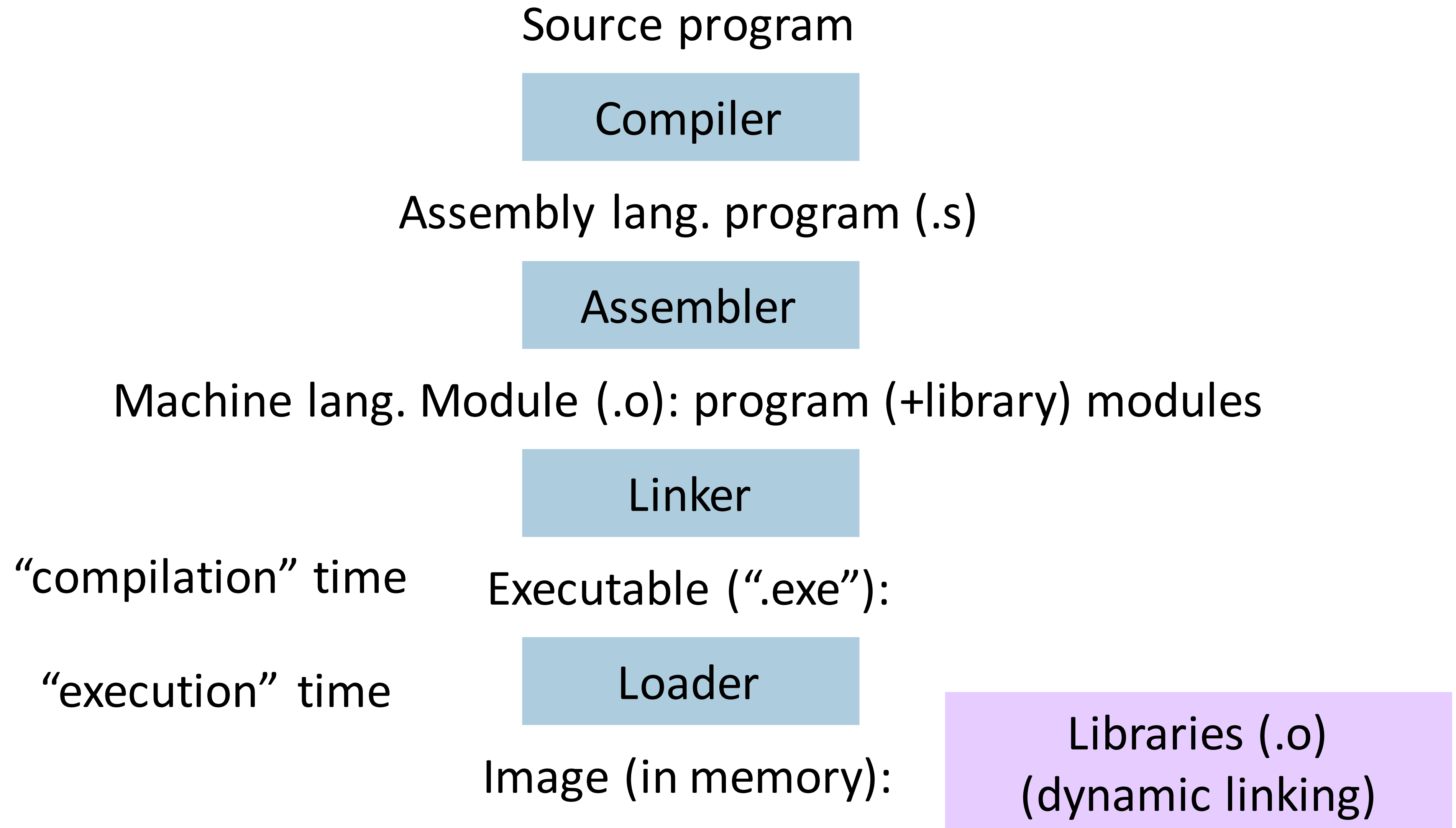
Static Linking (Recap)

- Assembler generates binary code
  - Unresolved addresses
  - Relocatable addresses
- Linker generates executable code
- Loader generates runtime states (images)

Dynamic Linking

- Why dynamic linking?
  - Shared libraries
    - Save space
    - Consistency
  - Dynamic loading
    - Load on demand

What's the challenge?



Position-Independent Code (PIC)

- Code which does not need to be changed regardless of the address in which it is loaded
  - Enable loading the same object file at different addresses
    - Thus, shared libraries and dynamic loading

- “Good” instructions for PIC: use relative addresses
  - relative jumps
  - reference to activation records

- “Bad” instructions for : use fixed addresses
  - Accessing global and static data
  - Procedure calls
    - Where are the library procedures located?

How?

“All problems in computer science can be solved by another level of indirection"

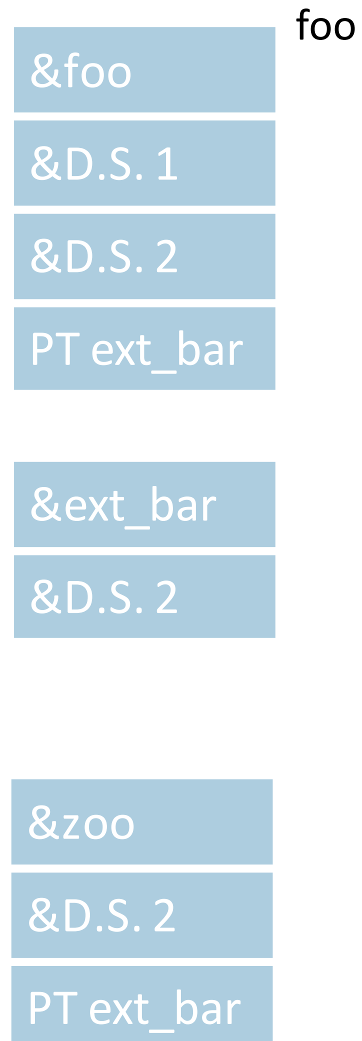
Butler Lampson

PIC: The Main Idea

- Keep the global data in a table
- Refer to all data relative to the designated register

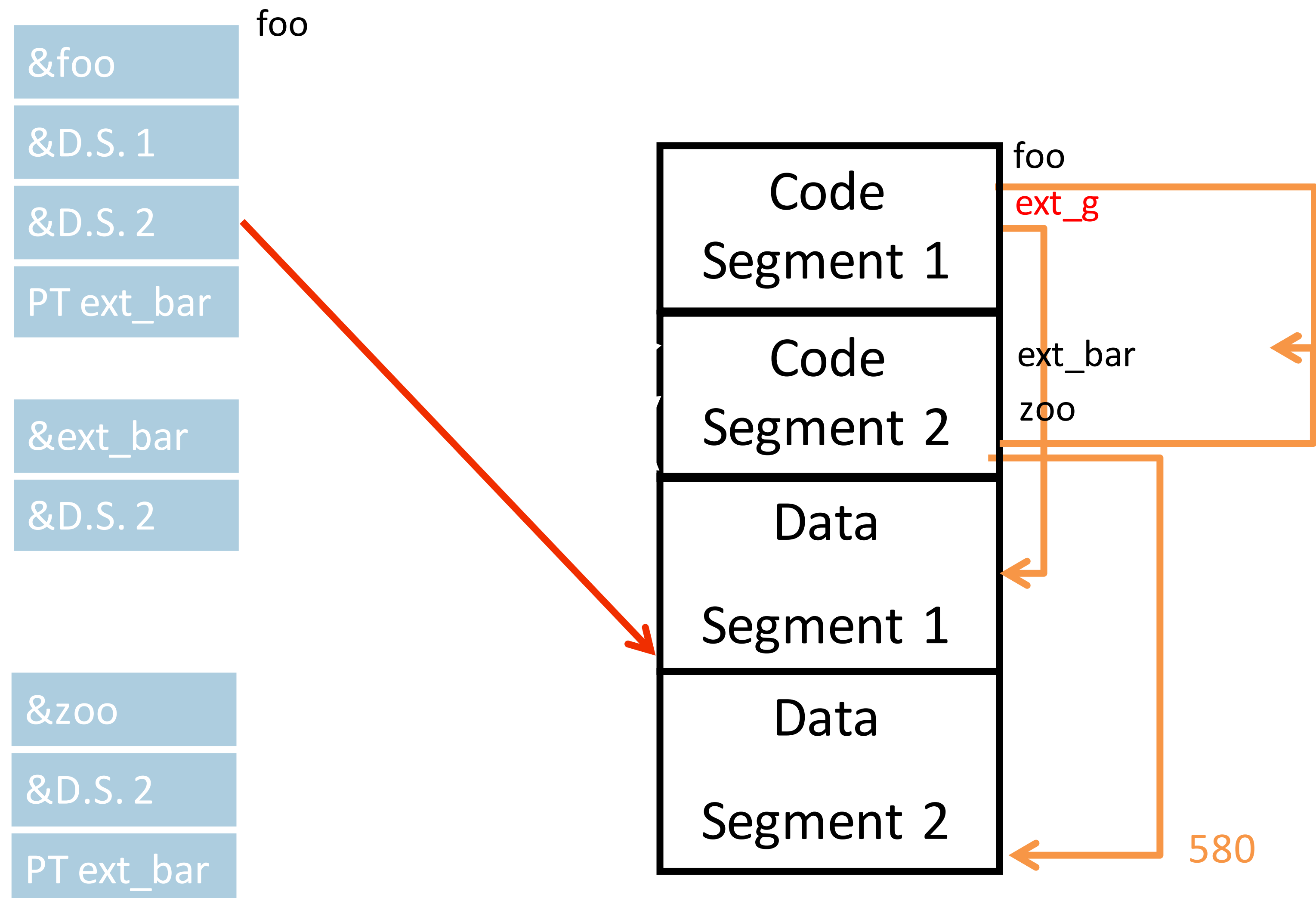
Per-Routine Pointer Table

- Record for every routine in a table



Per-Routine Pointer Table

- Record for every routine in a table

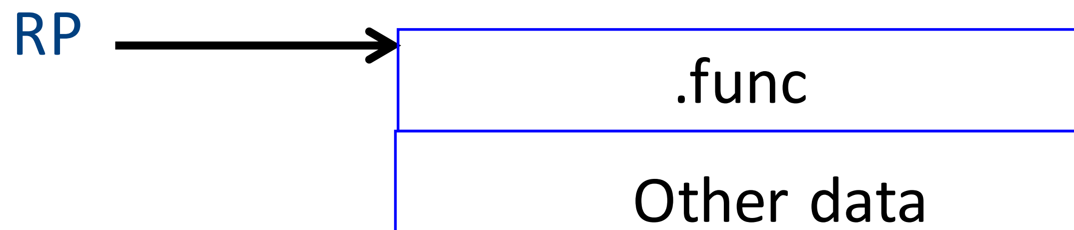


Per-Routine Pointer Table

- Record for every routine in a table
- Record used as a address to procedure

Caller:
1. Load Pointer table address into RP
2. Load Code address from 0(RP) into RC
3. Call via RC

Callee:
1. RP points to pointer table
2. Table has addresses of pointer table for sub-procedures



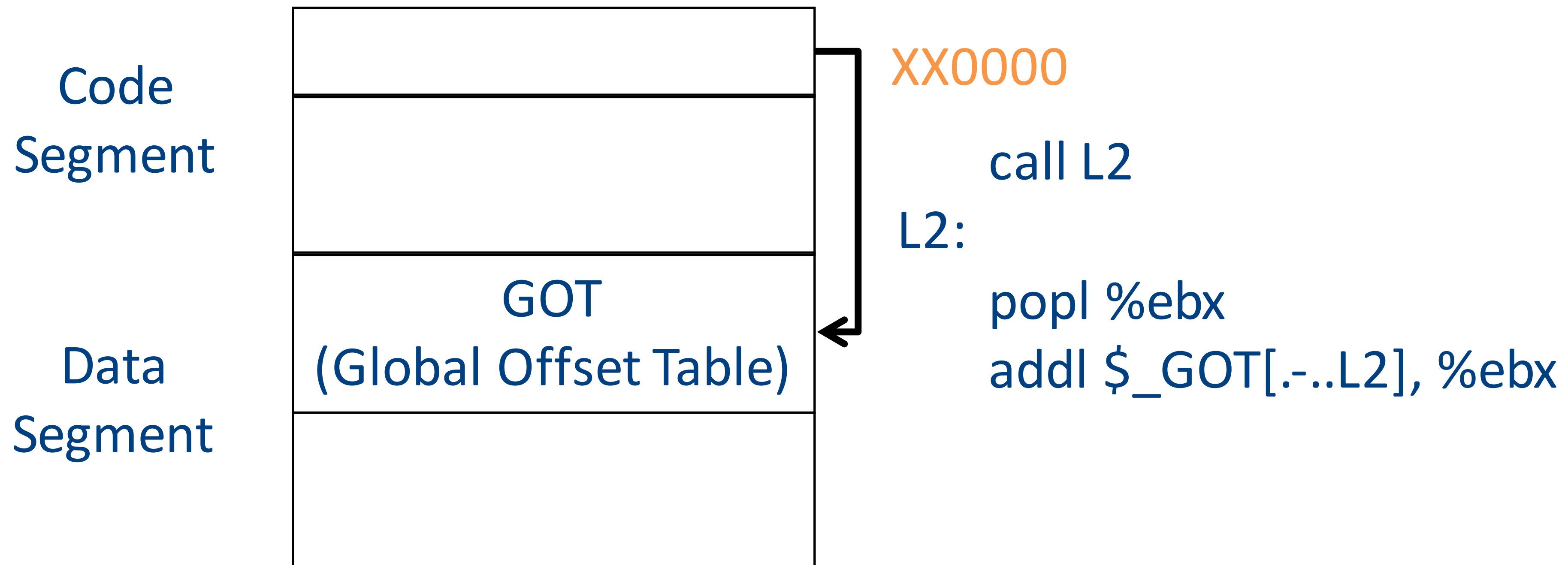
PIC: The Main Idea

- Keep the global data in a table
- Refer to all data relative to the designated register

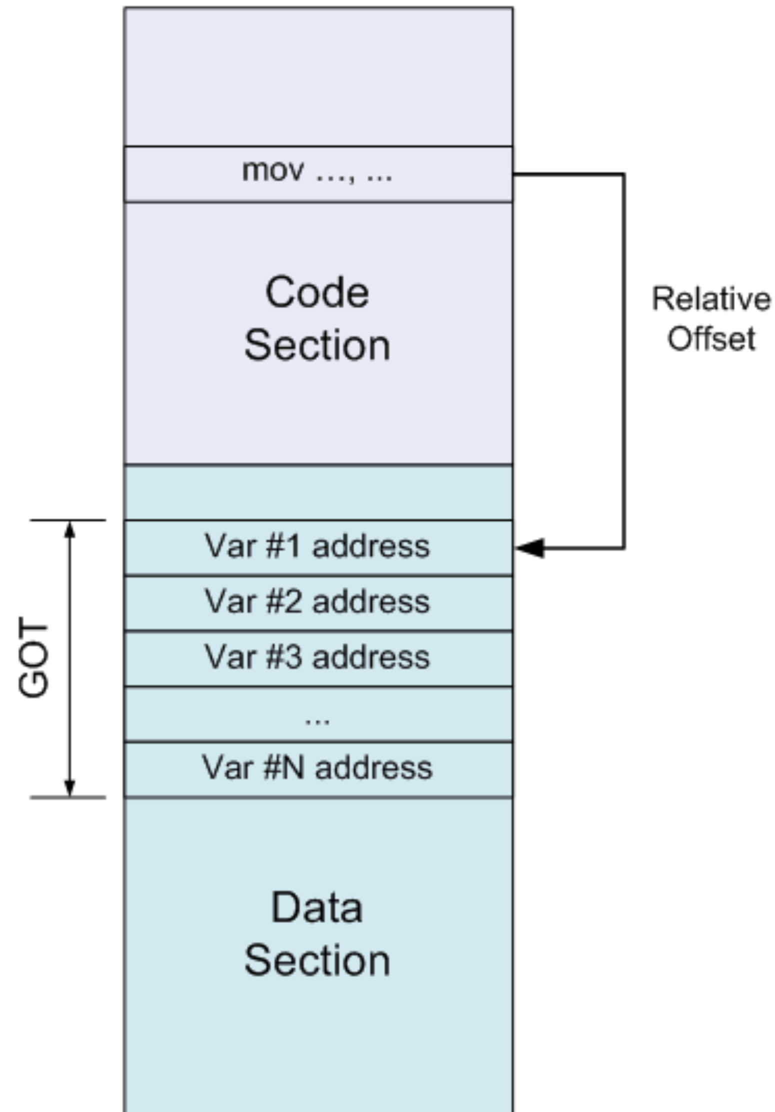
- Efficiency: use a register to point to the beginning of the table
  - Troublesome in CISC machines

ELF-Position Independent Code

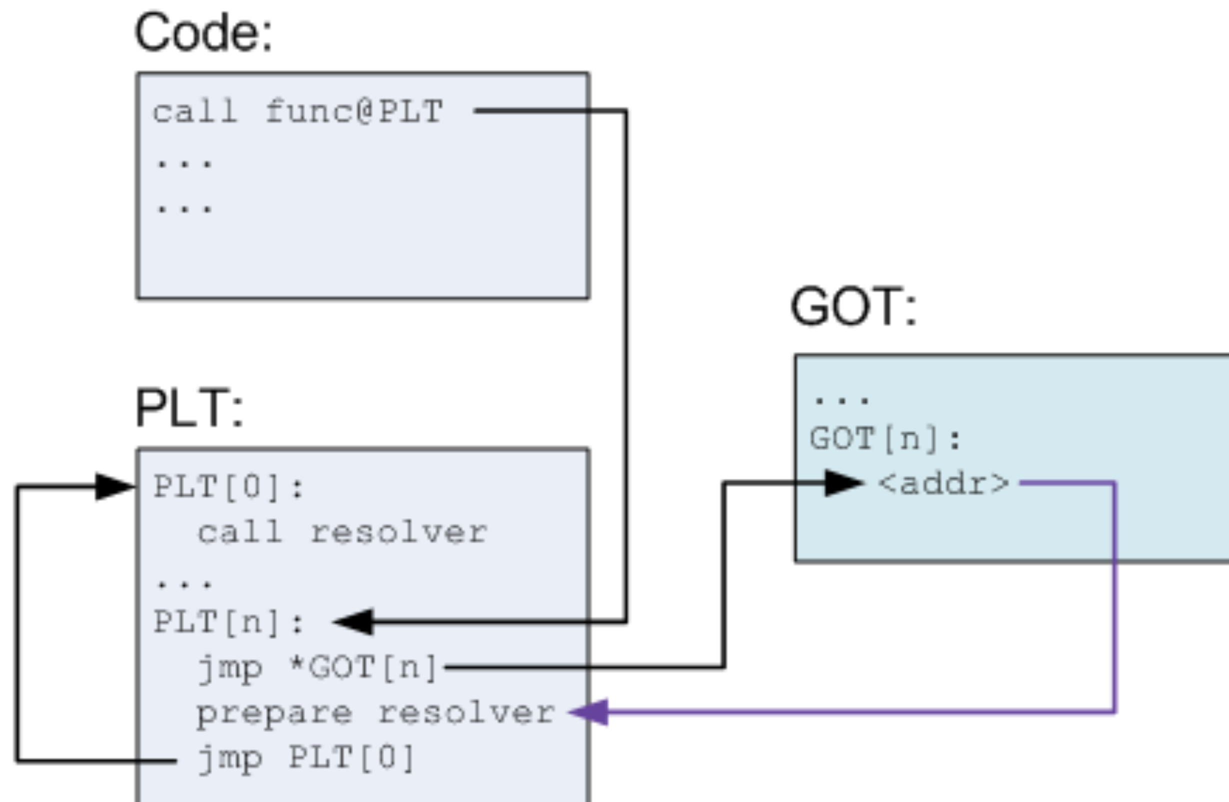
- Executable and Linkable code Format
  - Introduced in Unix System V
- Observation
  - Executable consists of code followed by data
  - The offset of the data from the beginning of the code is known at compile-time



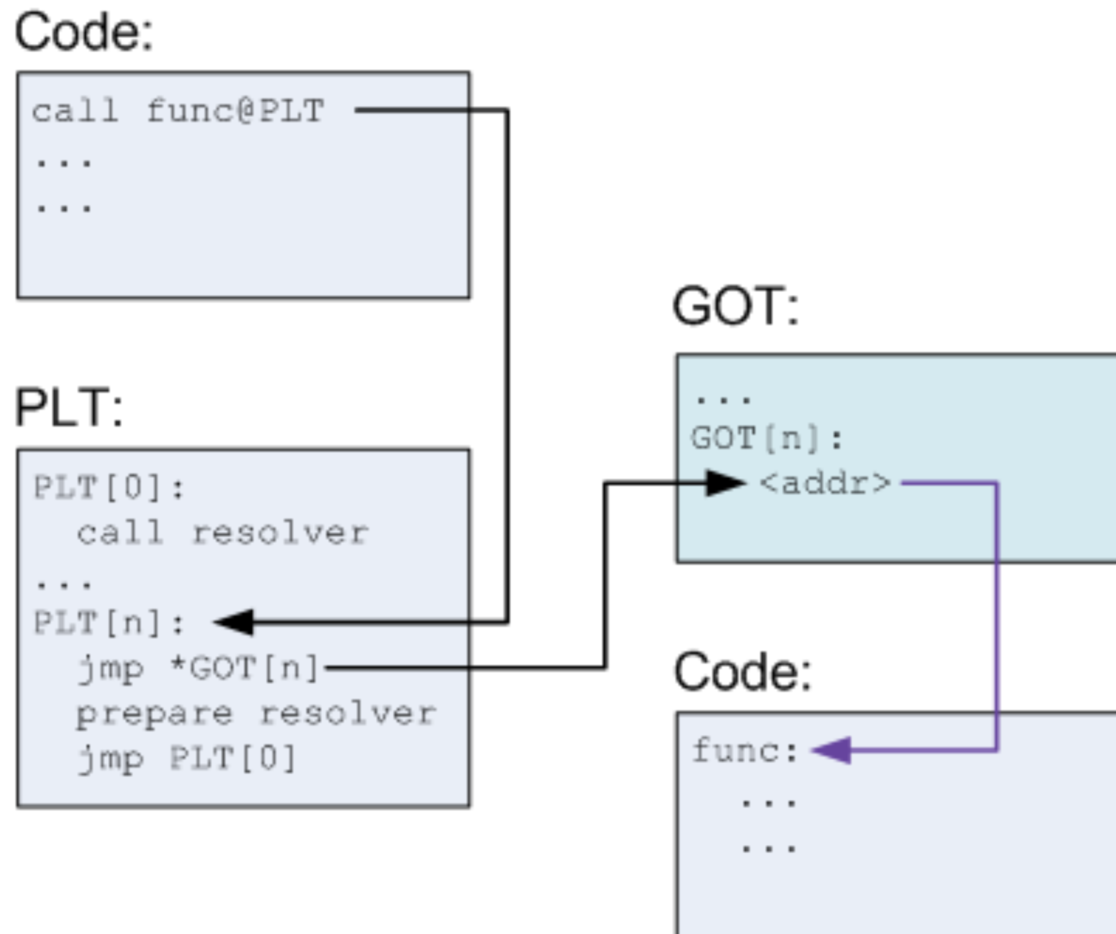
ELF: Accessing global data



ELF: Calling Procedures
(before 1st call)



ELF: Calling Procedures
(after 1st call)



PIC benefits and costs

- Enable loading w/o relocation
- Share memory locations among processes

- Data segment may need to be reloaded
- GOT can be large
- More runtime overhead
- More space overhead

Shared Libraries

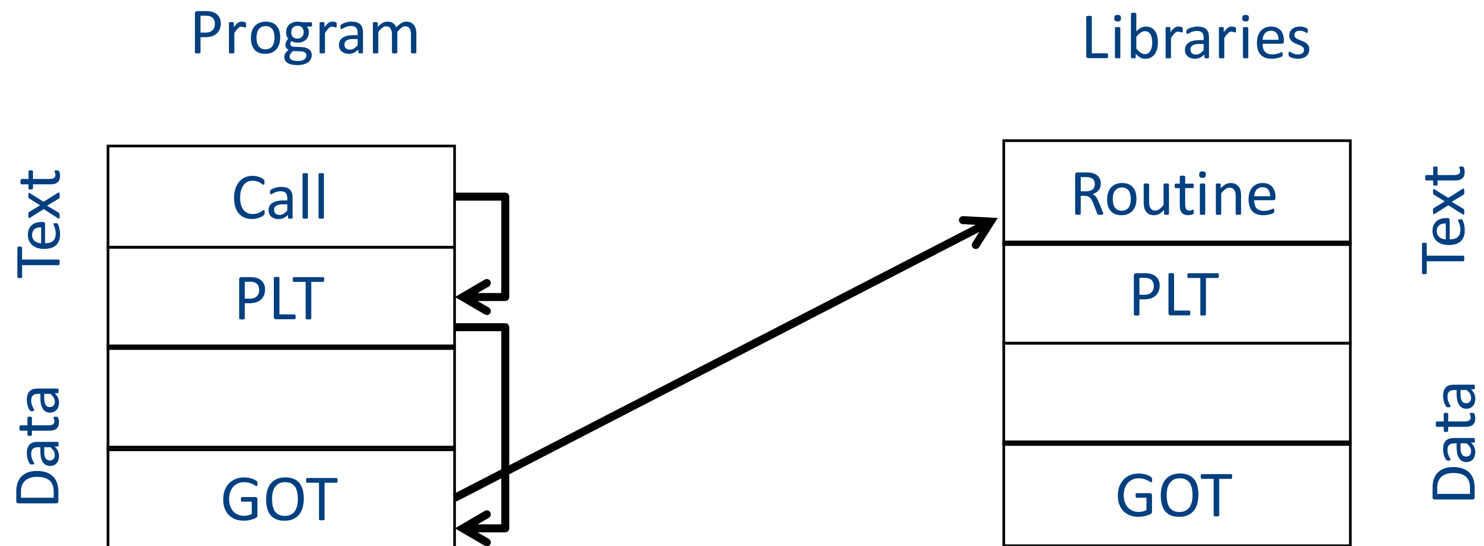
- Heavily used libraries
- Significant code space
  - 5-10 Mega for print
  - Significant disk space
  - Significant memory space
- Can be saved by sharing the same code
- Enforce consistency
- But introduces some overhead

- Can be implemented either with static or dynamic loading

Shared Libraries

- Heavily used libraries
- Significant code space
  - 5-10 Mega for print
  - Significant disk space
  - Significant memory space
- Can be saved by sharing the same code
- Enforce consistency
- But introduces some overhead

Content of ELF file



Consistency

- How to guarantee that the code/library used the “right” library version

Loading Dynamically Linked Programs

- Start the dynamic linker
- Find the libraries
- Initialization
  - Resolve symbols
  - GOT
    - Typically small
  - Library specific initialization
- Lazy procedure linkage

Microsoft Dynamic Libraries (DLL)

- Similar to ELF
- Somewhat simpler
- Require compiler support to address dynamic libraries
- Programs and DLL are Portable Executable (PE)
- Each application has it own address
- Supports lazy bindings

Dynamic Linking Approaches

- Unix/ELF uses a single name space space and MS/PE uses several name spaces
- ELF executable lists the names of symbols and libraries it needs
- PE file lists the libraries to import from other libraries
- ELF is more flexible
- PE is more efficient

Costs of dynamic loading

- Load time relocation of libraries
- Load time resolution of libraries and executable
- Overhead from PIC prolog
- Overhead from indirect addressing
- Reserved registers

Summary

- Code generation yields code which is still far from executable
  - Delegate to existing assembler
- Assembler translates symbolic instructions into binary and creates relocation bits
- Linker creates executable from several files produced by the assembly
- Loader creates an image from executable