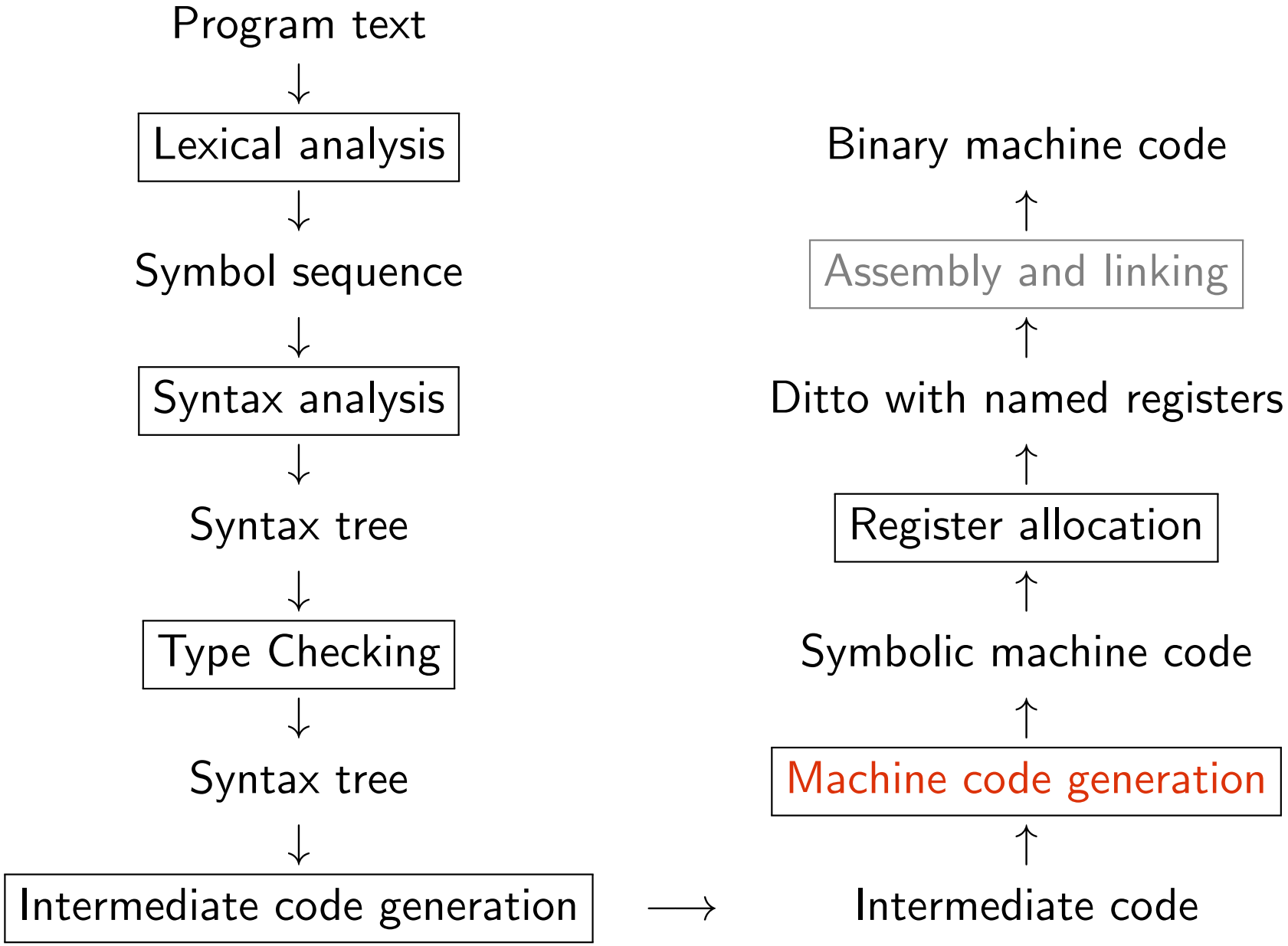
Machine Code Generation

Cosmin E. Oancea
cosmin.oancea@diku.dk
Modified by Marco Valtorta (UofSC) for CSCE 531 Spring 2020

Department of Computer Science (DIKU)
University of Copenhagen

February 2018 IPS Lecture Slides

Structure of a Compiler



Intended Learning Objectives

Students should be able to:

- translate simple, imperative programs (supporting while loops and sequential (short-circuiting) boolean operators to a low-level three-address intermediate representation (IR) code.
- apply (by hand) the pattern-based machine code generation technique in order to translate simple programs written in three-address IR code to machine code (MIPS).

Symbolic Machine Language

A text-based representation of binary code:

- more readable than machine code,
- uses labels as destinations of jumps,
- allows constants as operands,
- translated to binary code by *assembler* and *linker*.

Fast Introduction to MIPS

.data: the upcoming section is considered data,

.text: the upcoming section consists of instructions,

.global: the label following it is accessible from outside,

.asciiz "Hello": string with null terminator,

.space n: reserves n bytes of memory space,

.word w1, .., wn: reserves n words.

Mips Code Example: $ra = $31, $sp = $29, $hp = $28 (heap pointer)

```
        .data
val:    .word 10, -14, 30
str:    .asciiz "Hello!"
_heap_: .space 100000
        .text
        .global main
        la $28, _heap_
        jal main
        ...
_stop_:
        ori  $2, $0, 10
        syscall
main:
        la   $8, val    # ?
        lw   $9, 4($8)  # ?
        addi $9, $9, 4  # ?
        sw   $9, 8($8)  #...
        j    _stop_     #jr $31
```

Fast Introduction to MIPS

Mips Code Example: $ra = $31, $sp = $29, $hp = $28 (heap pointer)

```
                          _stop_:
                                  ori  $2, $0, 10
                                  syscall          # syscall 10
        .data                                      # means exit
val:    .word 10, -14, 30 main:
str:    .asciiz "Hello!"          la   $8, val     # loads start address
_heap_: .space 100000                              # of array val in reg $8
        .text                     lw   $9, 4($8)   # loads the second element
        .global main                               # of val in register $9
        la $28, _heap_            addi $9, $9, 4   # adds ct 4 to the second
        jal main                                   # element of val
        ...                       sw   $9, 8($8)   # and stores it in the
                                                   # third element of val
                                  j    _stop_      # or jr $31 (to reg $31)
                                                   # jumps to label _stop_
```

The third element of val, i.e., 30, is set to $-14 + 4 = -10$.

Intermediate and Machine Code Differences

- machine code has a limited number of registers,
- usually there is no equivalent to `CALL`, i.e., need to implement it,
- conditional jumps usually have only one destination,
- comparisons may be separated from the jumps,
- typically RISC instructions allow only small-constant operands.

The first issue will be solved by means of register allocation (Ch.8 [M]).

The second issue is solved in Ch.9 [M].

Two-Way Conditional Jumps

`IF` c `THEN` l_t `ELSE` l_f can be translated to:

```
branch_if_cond   l_t
jump             l_f
```

If l_t or l_f follow right after IF-THEN-ELSE, we can eliminate one jump:

```
      IF c THEN l_t ELSE l_f
l_t:
      ...
l_f:
```

can be translated to:

```
branch_if_not_cond l_f
```

Comparisons

In many architectures the comparisons are separated from the jumps: first evaluate the comparison, and place the result in a register that can be later read by a jump instruction.

- In MIPS both $=$ and $\neq$ operators can jump (`beq` and `bne`), but $<$ (`slt`) stores the result in a general register.
- ARM and X86's arithmetic instructions set a flag to signal that the result is 0 or negative, or overflow, or carry, etc.
- PowerPC and Itanium have separate boolean registers.

Constants

Typically, machine instructions restrict *constants' size* to be smaller than one machine word:

- MIPS32 uses 16 bit constants. For *larger constants*, `lui` is used to load a 16-bit constant into the upper half of a 32-bit register.
- ARM allows 8-bit constants, which can be positioned at any (even-bit) position of a 32-bit word.

Code generator checks if the constant value fits the restricted size:

if it fits: it generates one machine instruction (constant operand);

otherwise: use an instruction that uses a register (instead of a ct) generate a sequence of instructions that load the constant value in that register.

Sometimes, the same is true for the jump label.

Demonstrating Constants

```
let rec compileExp e vtable place =
  match e with
    | Constant (IntVal n, pos) ->
        if n < 0 then ...
        else if n < 65536 then
          [ Mips.LI (place, makeConst n) ]
        else
          [ Mips.LUI (place, makeConst (n div 65536))
          ; Mips.ORI (place, place, makeConst (n mod 65536)) ]
```

What happens with negative constants?

Demonstrating Constants

```
let rec compileExp e vtable place =
  match e with
    | Constant (IntVal n, pos) ->
        if n < 0 then ...
        else if n < 65536 then
          [ Mips.LI (place, makeConst n) ]
        else
          [ Mips.LUI (place, makeConst (n div 65536))
          ; Mips.ORI (place, place, makeConst (n mod 65536)) ]
```

What happens with negative constants?

```
let rec compileExp e vtable place =
  match e with
    Constant (IntVal n, pos) =>
      if n < 0 then
        compileExp (Negate (Constant (IntVal (-n), pos), pos)) vtable place
      else if n < 65536 then
        [ Mips.LI (place, makeConst n) ]
      else
        [ Mips.LUI (place, makeConst (n div 65536))
        ; Mips.ORI (place, place, makeConst (n mod 65536)) ]
```

Exploiting Complex Instructions

Many architectures expose complex instructions that combine several operations (into one), e.g.,

- load/store instructions also involve address calculation,
- arithmetic instructions that scale one argument (by shifting),
- saving/restoring multiple registers to/from memory storage,
- conditional instructions (other besides jump).

In some cases: several IL instructions $\rightarrow$ one machine instruction.

In other cases: one IL instruction $\rightarrow$ several machine instructions, e.g., conditional jumps.

MIPS Example

The two intermediate-code instructions:

```
t2 := t1 + 116
t3 := M[ t2 ]
```

can be combined into *one* MIPS instruction (?)

```
lw r3, 116(r1)
```

MIPS Example

The two intermediate-code instructions:

```
t2 := t1 + 116
t3 := M[ t2 ]
```

can be combined into *one* MIPS instruction (?)

```
lw r3, 116(r1)
```

IFF t_2 is not used anymore! Assume that we mark/know whenever a variable is used for the last time in the intermediate code.

This marking is accomplished by means of *liveness analysis* (Ch.8 [M]); we write:

```
t2 := t1 + 116
t3 := M[ t2^last ]
```

Intermediate-Code Patterns

- Need to map each IL instruct to one or many machine instructs.
- Take advantage of complex-machine instructions via *patterns*:
  - map a sequence of IL instructs to one or many machine instructs,
  - try to match first the longer pattern, i.e., the most profitable one.
- Variables marked with *last* in the IL pattern *must* be matched with variables that are used for the last time in the IL code.
- The converse is not necessary, i.e., if a variable is not marked with *last* in the pattern, then it still may be matched by a variable used for the last time in IL

$t := r_s + k$ $r_t := M[t^{last}]$	`lw` r_t, $k(r_s)$

t, r_s and r_t can match arbitrary IL variables, k can match any (small) constant.

Patterns for MIPS (part 1)

$t := r_s + k$, $r_t := M[t^{last}]$	`lw`	r_t, $k(r_s)$
$r_t := M[r_s]$	`lw`	r_t, $0(r_s)$
$r_t := M[k]$	`lw`	r_t, k(`R0`)
$t := r_s + k$, $M[t^{last}] := r_t$	`sw`	r_t, $k(r_s)$
$M[r_s] := r_t$	`sw`	r_t, $0(r_s)$
$M[k] := r_t$	`sw`	r_t, k(`R0`)
$r_d := r_s + r_t$	`add`	r_d, r_s, r_t
$r_d := r_t$	`add`	r_d, `R0`, r_t
$r_d := r_s + k$	`addi`	r_d, r_s, k
$r_d := k$	`addi`	r_d, `R0`, k
`GOTO` *label*	`j`	*label*

Must cover all possible sequences of intermediate-code instructions.

Patterns for MIPS (part 2)

IF $r_s = r_t$ THEN $label_t$ ELSE $label_f$, LABEL $label_f$		beq	r_s, r_t, $label_t$
	$label_f$:		
IF $r_s = r_t$ THEN $label_t$ ELSE $label_f$, LABEL $label_t$		bne	r_s, r_t, $label_f$
	$label_t$:		
IF $r_s = r_t$ THEN $label_t$ ELSE $label_f$		beq	r_s, r_t, $label_t$
		j	$label_f$
IF $r_s < r_t$ THEN $label_t$ ELSE $label_f$, LABEL $label_f$		slt	r_d, r_s, r_t
		bne	r_d, R0, $label_t$
	$label_f$:		
IF $r_s < r_t$ THEN $label_t$ ELSE $label_f$, LABEL $label_t$		slt	r_d, r_s, r_t
		beq	r_d, R0, $label_f$
	$label_t$:		
IF $r_s < r_t$ THEN $label_t$ ELSE $label_f$		slt	r_d, r_s, r_t
		bne	r_d, R0, $label_t$
		j	$label_f$
LABEL $label$	$label$:		

Compiling Code Sequences: Example

$a := a + b^{last}$
$d := c + 8$
$M[d^{last}] := a$
`IF` $a = c$ `THEN` $label_1$ `ELSE` $label_2$
`LABEL` $label_2$

Compiling Code Sequences

Example:

$a := a + b^{last}$	`add` $a,\ a,\ b$
$d := c + 8$	`sw` $a,\ 8(c)$
$M[d^{last}] := a$	
`IF` $a = c$ `THEN` $label_1$ `ELSE` $label_2$	`beq` $a,\ c,\ label_1$
`LABEL` $label_2$	$label_2$:

Two approaches:

Greedy Alg: Find the first/longest pattern matching a prefix of the IL code + translate it. Repeat on the rest of the code.

Dynamic Prg: Assign to each machine instruction a cost and find the matching that minimize the global / total cost.

Two-Address Instructions

Some processors, e.g., X86, store the instruction's result in one of the operand registers. Handled by placing one argument in the result register and then carrying out the operation:

$r_t := r_s$	`mov` r_t, r_s
$r_t := r_t + r_s$	`add` r_t, r_s
$r_d := r_s + r_t$	`move` r_d, r_s `add` r_d, r_t

Register allocation can remove the extra `move`.

Optimizations

Can be performed at different levels:

Abstract Syntax Tree: high-level optimization: specialization, inlining, map-reduce, etc.

Intermediate Code: machine-independent optimizations, such as redundancy elimination, or index-out-of-bounds checks.

Machine Code: machine-specific, low-level optimizations such as instruction scheduling and pre-fetching.

Optimizations at the intermediate-code level can be shared between different languages and architectures.

Code Hoisting

Code hoisting means moving common code in the body of a loop outside the loop, so that it executed only once, rather than every time the body is executed. This may require unrolling while loops once, to prevent executing code even when the condition of the loop is false; such execution could lead to run-time errors.

Original loop:

```
while (j < k) {
    sum = sum +a[i][j];
    j++;
}
```

After unrolling once:

```
if (j < k) {
    sum = sum + a[i][j];
    j++;
    while (j < k) {
        sum = sum +a[i][j];
        j++;
     }
}
```

Some Other Types of Optimization

Common Subexpression Elimination Simple methods for common subexpression elimination work on *basic blocks*, i.e., straight-line code without jumps or labels. More on this in Chapter 10.

Constant Propagation

Index-Check Elimination