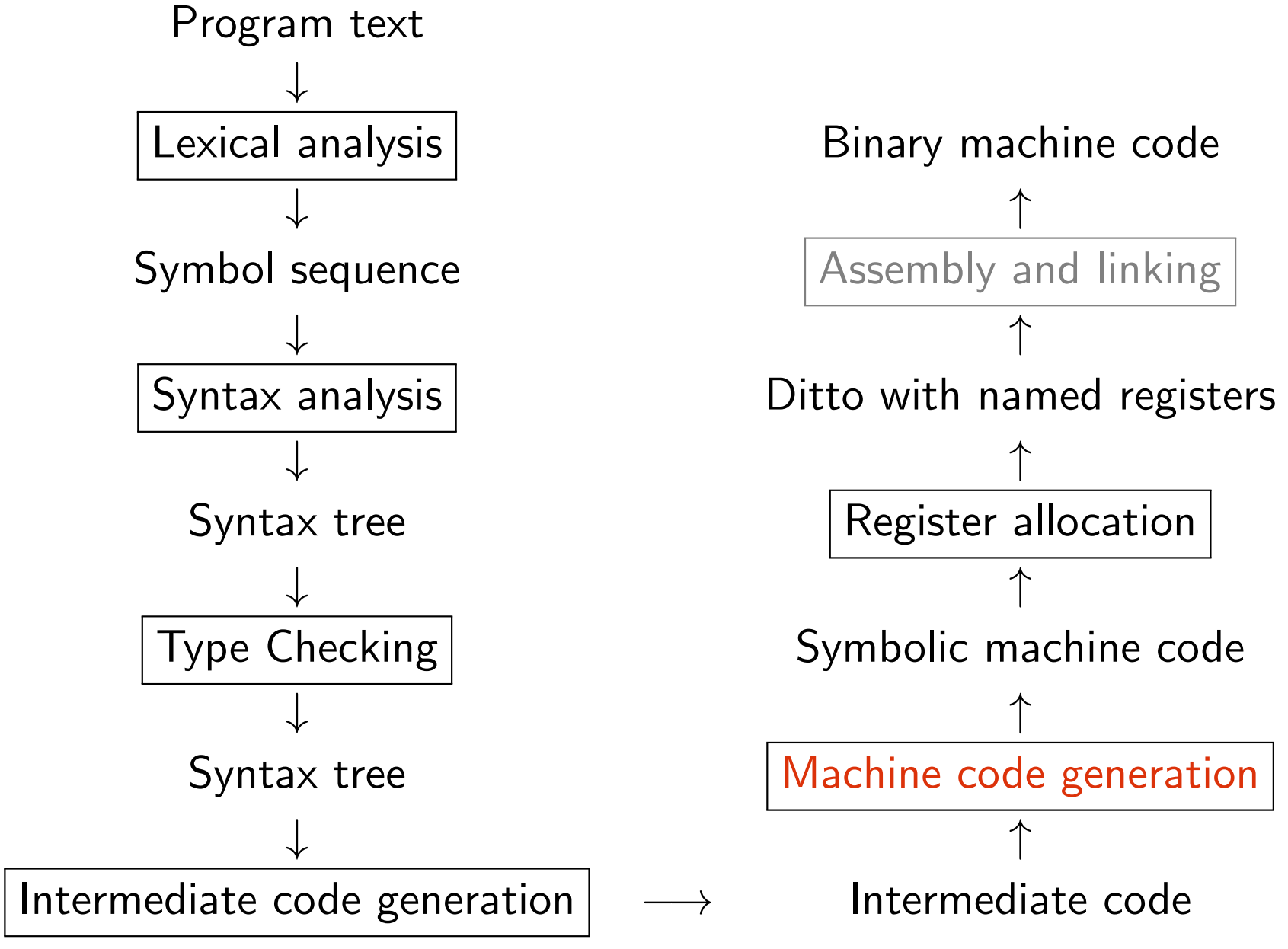
Machine Code Generation

Cosmin E. Oancea
cosmin@diku.dk

Department of Computer Science (DIKU)
University of Copenhagen

December 2013 Compiler Lecture Notes

Structure of a Compiler



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*,
- "symbolic" typically means infinite number of registers, to be mapped later via register allocation.

Remember 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
```

Remember 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
```

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

Paladim's MIPS Header

- stack grows from high to low mem, heap is statically alloc,
- user defined funs & CT strings, helper funs, errors, etc.

Entry Point to Compiler.sml; See also Mips.sml

```
fun compile funs =
 let val () = stringTable := []
 val funsCode = List.concat
           (List.map compileFun funs)
 (* magic init of prg's CT strings *)
 val (stringinit, stringdata) = ... in
 [  Mips.TEXT "0x00400000",
    Mips.GLOBL "main"      ]
 (*init heap & string pointers*)
@ (Mips.LA (HP, "_hp_")::stringinit)
@ [ Mips.JAL ("main",[]),(* run pgm *)
    Mips.LABEL "_stop_",(*end pgm&err*)
    Mips.LI ("2","10"),(*syscall exit*)
    Mips.SYSCALL ]

@ funsCode   (* code for functions *)
(* ord : char -> int *)
@ [Mips.LABEL "ord", (*trunc ints<0*)
   Mips.ANDI("2", "2", makeConst 255),
   Mips.JR (RA,[]), ...] ...
@ [Mips.LABEL "readInt",
   Mips.LI ("2","5"),
   Mips.SYSCALL,(*read_int syscall*)
   Mips.JR (RA,[]) ]
@ [Mips.LABEL "_IllegalArrErr_",
   Mips.LA ("4","_IllegalArr_"),
   Mips.LI ("2","4"),
   Mips.SYSCALL, (*print str*)
   Mips.MOVE("4","5"),Mips.LI("2","1")
  ,Mips.SYSCALL, (*print line*)
   Mips.J "_stop_"]
@ [ (* String CT & Err Msg *)
   Mips.DATA "", Mips.ALIGN "2",
   Mips.LABEL "_IllegalArr_",
   Mips.ASCIIZ "Error: Illegal " ]
 @ (* program's string literals *)
   List.concat stringdata
  (* Heap (array allocation) *)
 @ [Mips.ALIGN "2",Mips.LABEL "_hp_"
   ,Mips.SPACE "100000" ]
```

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 two issues are solved in the next two lessons.

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

Fasto Implementation

```
fun compileExp( vtable, Literal(BVal (Num n), _), place ) =
        if n < 32768
        then [ Mips.LI (place, makeConst n) ]
        else [ Mips.LUI (place, makeConst (n div 65536)),
               Mips.ORI (place, place, makeConst (n mod 65536)) ]
```

Paladim does not support negative constants (for the moment).

Exploiting Complex Instructions

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

- load/store instruction 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*; we write:

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

then we can safely translate it to: `lw r3, 116(r1)`.

Intermediate-Code Patterns

- Describe each machine instruct by one or more IL instructs,
- and `IF-THEN-ELSE` IL instrucs by one or more machine instructs.
- Take advantage of complex-machine instructions via *patterns*:
  - a sequence of IL instrs replaced by one (several) machine instr,
  - try to match first the longer pattern, i.e., the most profitable one.
- Variables marked with *last* in the IL pattern *must* be matched by variables that are used for the last time in the IL code.

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

- The converse is not necessary (last-used var loaded to a register).

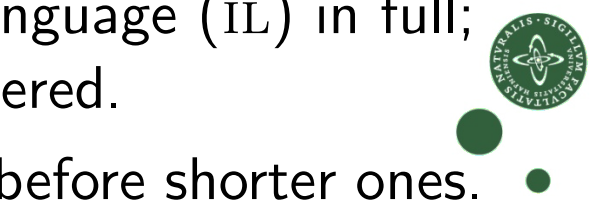
t, r_s and r_t can match arbitrary IL variables, k can match any constant (big constants have already been eliminated);

Pattern-Replacement Pairs 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*

The list of patterns must cover the intermediate language (IL) in full; in particular each single IL instruction must be covered.

Longest patterns appear in the list, and are tested before shorter ones.



Pattern-Replacement Pairs 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 $label$	$label$:

Labels are retained as they might be referenced from somewhere else.
(Cost nothing; could be eliminated in the same way as dead variables.)

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 often 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.

We talk more about optimizations next lecture and in the New Year!

`Mips` Representation is in File `Mips.sml`

Several differences:

- Mips.MOVE("r_d", "r_s") $\equiv$ $r_d := r_s$;
- Mips.LW("r", "r_m", K) $\equiv$ lw r, K(r_m) $\equiv$ $r := M[r_m + K]$;
- Mips.SW("r", "r_m", K) $\equiv$ sw r, K(r_m) $\equiv$ $M[r_m + K] := r$;

Machine Code For Paladim's Statements

compileExp(vtab : VTab, e : Exp, place : string) : mips list

```
fun compileStmt(vtable, s, exitLabel) = case s of
      Return (NONE, p) => [ Mips.J exitLabel ]
    | Return (SOME e, p) => (* one result placed in $2 *)
        let val t     = "_return_"^newName()
            val code0 = compileExp(vtable, e, t)
        in code0 @ [ Mips.MOVE ("2",t), Mips.J exitLabel ] end
    | IfThEl (e,blockT,blockF,p) =>
        let val (ereg,els,endl)=("_th"^newName(),"_el"^newName(),"_end"^newName()
            val codeE = compileExp(vtable, e, ereg)
            val codeT = compileStmts blockT vtable exitLabel
            val codeF = compileStmts blockF vtable exitLabel
        in codeE @ [ Mips.BEQ (ereg, "0", els) ] @ codeT @ [ Mips.J endl]
         @ ( Mips.LABEL els) :: codeF @ [ Mips.LABEL endl ]            end
    | While (e,block,p) =>
        let val (eReg,entry)=("_wh" ^ newName(),"_en"^newName(),"_ex"^newName())
            val eCode = compileExp(vtable, e, eReg)
            val bCode = compileStmts block vtable exitLabel
        in [ Mips.LABEL entry ] @ eCode @ [ Mips.BEQ (eReg, "0", exit) ]
           @ bCode @ [ Mips.J entry, Mips.LABEL exit ]                  end
```

Each fun/proc associated to an exit label, where code jumps on return. `While` and `If` similar to IL generation, but use Mips.BEQ

Machine Code For Paladim's LValue Expressions

compileExp(vtab : VTab, e : Exp, place : string) : mips list

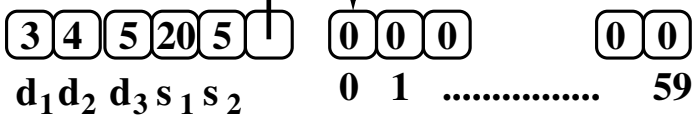
```
datatype Location = Reg of string (* in given register *)
                  | Mem of string (* at memory address held in register *)

fun compileExp( vtable, Literal(BVal (Chr c), _), place ) =
      [ Mips.LI (place,makeConst (ord c)) ]
  | compileExp( vtable, LValue (lv,pos), place ) =
      let val (code, loc) = compileLVal(vtable, lv, pos)
          val tp = typeOfExp( LValue(lv,pos) )
      in case loc of
           Reg r => [ Mips.MOVE (place,r) ]
         | Mem x => case tp of                    (* lb place, 0(x)*)
                      BType Char  => code @ [Mips.LB (place,x,"0")]
                    | BType Bool  => code @ [Mips.LB (place,x,"0")]
                    | other       => code @ [Mips.LW (place,x,"0")]
      end
and compileLVal( vtab, Var (n,_): LVAL, pos ): Mips.mips list * Location =
      ( case SymTab.lookup n vtab of
          SOME reg => ([], Reg reg) (* returns a value in a register *)
        | NONE     => raise Error ("unknown variable "^n, pos) )
  | compileLVal( vtab : VTab, Index ((n,t),inds) : LVAL, pos : Pos ) = ?
```

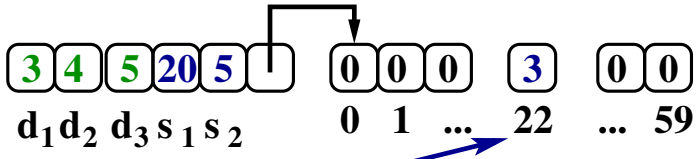
Project Task.

Paladim's Arrays

```
a : array of array of array of int;
a = new(3, 4, 5);
```




```
a[1,0,2] := 3;
```




```
index_flat = 1 * 20 + 0 * 5 + 2 = 22
index_ok = (1 >= 0 && 1 < 3) &&
           (0 >= 0 && 0 < 4) &&
           (2 >= 0 && 2 < 5) = true
```

compileExp(vtab : VTab, e : Exp, place : string) : mips list

```
datatype Location = Reg of string (* in given register *)
                  | Mem of string (* at memory address held in register *)

and compileLVal( vtab, Var (n,_): LVAL, pos ): Mips.mips list * Location =
      ( case SymTab.lookup n vtab of
          SOME reg => ([], Reg reg) (* returns a value in a register *)
        | NONE     => raise Error ("unknown variable "^n, pos) )
  | compileLVal( vtab : VTab, Index ((n,t),inds) : LVAL, pos : Pos ) = ?
```

How to implement compileLVal?

Perhaps looking at new, already implemented, would help.