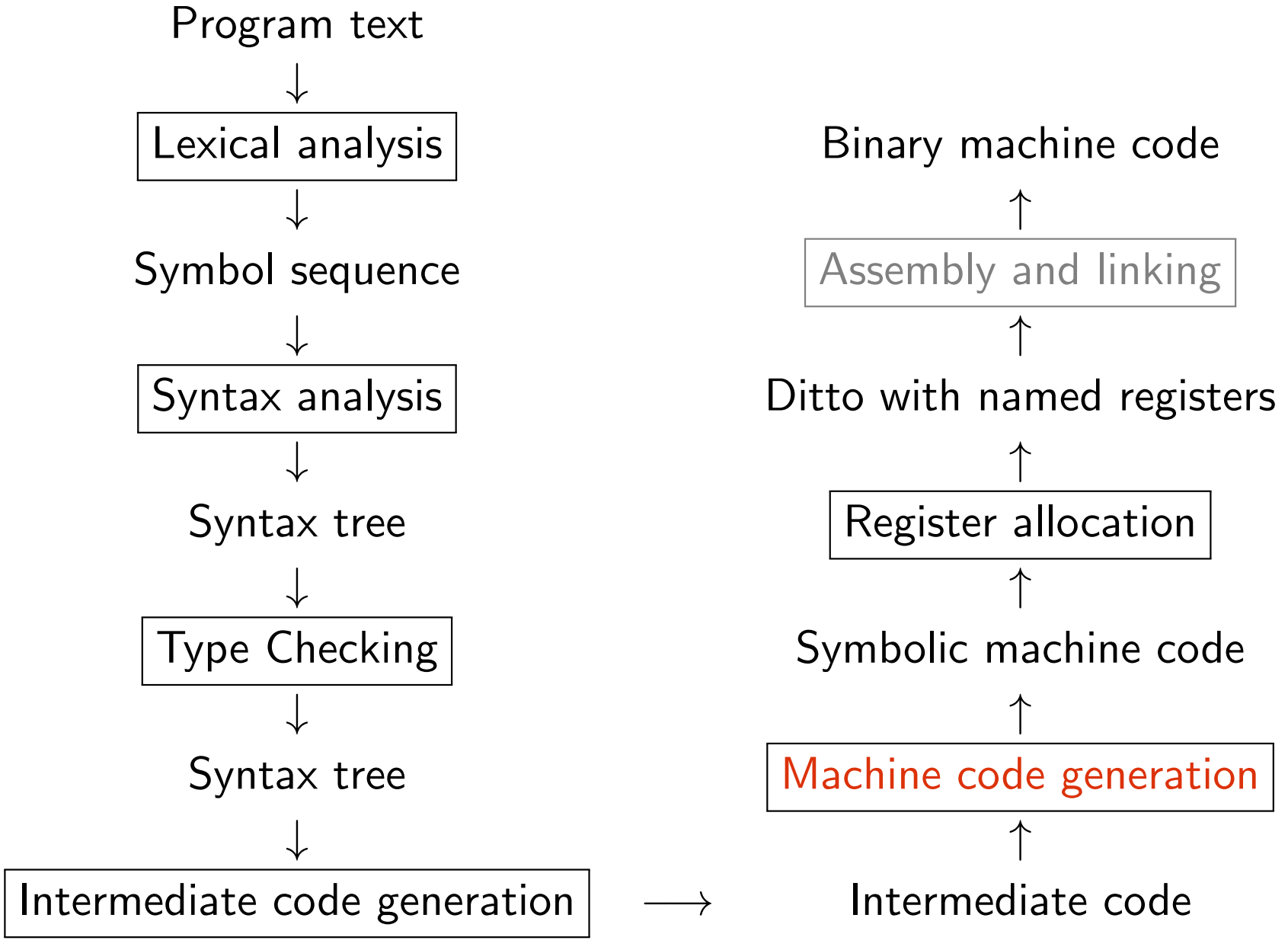
Machine-Code Generation for Functions

Cosmin E. Oancea
cosmin@diku.dk

Department of Computer Science (DIKU)
University of Copenhagen

December 2013 Compiler Lecture Notes

Structure of a Compiler



1. Problem Statement and Terminology
2. Caller-Saves Strategy
3. Callee-Saves Strategy
4. Mixed Strategy (Caller + Callee Saves)
5. Global Variables, Call by Reference, Aliasing

Problem Statement

In the translation to IL of functions (calls):

- A function call was simply translated to a `CALL` instruction. Function's result translated via a `RETURN` instruction.
- We have assumed that all variables are local.
- We have assumed that the function's parameters and result are passed via named variables (symbolic registers).

How to implement these in the machine language?

Next Lecture: Register Allocation maps named variables / symbolic registers to machine registers. Performed on the function's body.

Machine Code Translation Uses Machine Registers!

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

```
| compileExp( vtable, Plus (e1, e2, _), place ) =
    let val t1 = "plus1_" ^ newName()
        val c1 = compileExp(vtable, e1, t1)
        val t2 = "plus2_" ^ newName()
        val c2 = compileExp(vtable, e2, t2)
    in c1 @ c2 @ [Mips.ADD (place,t1,t2)]
    end
```

Machine Code Generation uses symbolic registers, which are going to be mapped to machine registers by Register Allocator (next lecture).

Register Allocation May Spill Variables To Memory

Exponential Fibonacci Implementation In Paladim

```
int fibrec(int n) =
        if      (n = 0) then 0
        else if (n = 1) then 1
        else fibrec(n-1) +
             fibrec(n-2)

int main () =
    let n = 8
    in  fibrec(n)
```

Creates an exponential (in n) number of interger-type temporaries, which will not fit in the limited number of machine registers.

Basic Idea: Only variables that are local to the most recently called function are kept in registers. The others are "spilled" to (stack) memory.

When?

Register Allocation May Spill Variables To Memory

Exponential Fibonacci Implementation In Paladim

```
int fibrec(int n) =
        if      (n = 0) then 0
        else if (n = 1) then 1
        else fibrec(n-1) +
             fibrec(n-2)

int main () =
    let n = 8
    in  fibrec(n)
```

Creates an exponential (in n) number of interger-type temporaries, which will not fit in the limited number of machine registers.

Basic Idea: Only variables that are local to the most recently called function are kept in registers. The others are "spilled" to (stack) memory.

When? Before the function's body is executed by caller or callee!

Call Stack

Callee → the fun/procedure that is called;
Caller → the fun/proc from which the callee is called.

We use a stack to store the information that connects the caller to the callee when a function call occurs:

- The registers' content is stored on the stack before the call and is restored (in registers) after the call.
- The return address is stored on the stack.
- Parameters and return value can be also passed on the stack.
- Spilled variables can be also stored on the stack.
- Local arrays/records are also typically allocated on the stack.
- Finally, non-local variables can be allocated on the stack.

Activation Records

Each function allocates a piece of memory on the stack to keep associated information. This piece of storage is called function's activation record or frame.

The hardware / operating system will dictate a calling convention that would standardize *the layout of the activation record* (so that we can call function across different compilers and languages).

However, some languages use extended calling conventions, such that only "simple" functions can be called from other languages, i.e., foreign-function interface.

We assume CALL-BY-VALUE semantics!

Prologue, Epilogue and Call Sequence

The code of a function starts with a *prologue*, that (i) retrieves parameters from the stack and places them in variables (registers), and (ii) may save the registers to be preserved.

Prologue, Epilogue and Call Sequence

The code of a function starts with a *prologue*, that (i) retrieves parameters from the stack and places them in variables (registers), and (ii) may save the registers to be preserved.

The code of a function ends with an *epilogue* that places the return value back on stack, and may restore in registers the values saved in the *prologue*, and then returns control to the calling function.

Prologue, Epilogue and Call Sequence

The code of a function starts with a *prologue*, that (i) retrieves parameters from the stack and places them in variables (registers), and (ii) may save the registers to be preserved.

The code of a function ends with an *epilogue* that places the return value back on stack, and may restore in registers the values saved in the *prologue*, and then returns control to the calling function.

A CALL instruction is replaced with a *call sequence*, which places arguments on stack, saves the registers to be preserved, saves the return address, calls the function and moves the returned value from the stack to a variable (register), restores the saved registers.

What the *prologue, epilogue and call sequence* do exactly depends on the calling convention.

Caller-Saves: Activation-Record Layout

The caller does the work of saving and restoring registers.

	...
	Next activation records
	Space for storing local variables for spill or preservation across function calls
	Remaining incoming parameters
	First incoming parameter / return value
FP $\longrightarrow$	Return address
	Previous activation records
	...

The frame pointer, FP, indicates the beginning of the activation record.

With this layout, the stack grows up in memory.

Caller-Saves: Prologue and Epilogue

Assume a function with name *function-name* and parameters $parameter_1 \ldots parameter_n$. The result is calculated in variable *result*.

We assume that the caller saves the registers: *caller-saves*.

$$\text{Prologue} \begin{cases} \texttt{LABEL}\ \textit{function-name} \\ parameter_1 := M[FP + 4] \\ \ldots \\ parameter_n := M[FP + 4 * n] \end{cases}$$

code for the function body

$$\text{Epilogue} \begin{cases} M[FP + 4] := result \\ \texttt{GOTO}\ M[FP] \end{cases}$$

We used here IL instructions, but typically, the prologue, epilogue and call sequence are introduced directly in machine language.

Caller-Saves: Call Sequence

Consider call $x := \texttt{CALL}\ f(a_1, \ldots, a_n)$.

Assume that $R0 \ldots Rk$ are used for local variables.

framesize framesize is the *size* of the current activation record.

$$
\begin{aligned}
&M[FP + 4 * m + 4] := R0 \\
&\ldots \\
&M[FP + 4 * m + 4 * (k + 1)] := Rk \\
&FP := FP + framesize \\
&M[FP + 4] := a_1 \\
&\ldots \\
&M[FP + 4 * n] := a_n \\
&M[FP] := returnaddress \\
&\texttt{GOTO}\ f \\
&\texttt{LABEL}\ returnaddress \\
&x := M[FP + 4] \\
&FP := FP - framesize \\
&R0 := M[FP + 4 * m + 4] \\
&\ldots \\
&Rk := M[FP + 4 * m + 4 * (k + 1)]
\end{aligned}
$$

Callee-Saves: Activation Records

The callee does all the work of saving and restoring registers.

	...
	Next activation records
	Space for storing local variables for spill
	Space for storing registers that need to be preserved
	Remaining incoming parameters
	First incoming parameter / return value
FP $\longrightarrow$	Return address
	Previous activation records
	...

Difference: separate space for *saved* and *spill* registers.

Callee-Saves: Prologue and Epilogue

$$\text{Prologue} \begin{cases} \texttt{LABEL}\ \textit{function-name} \\ M[FP + 4 * n + 4] := R0 \\ \cdots \\ M[FP + 4 * n + 4 * (k + 1)] := Rk \\ \textit{parameter}_1 := M[FP + 4] \\ \cdots \\ \textit{parameter}_n := M[FP + 4 * n] \end{cases}$$

code for the function body

$$\text{Epilogue} \begin{cases} M[FP + 4] := \textit{result} \\ R0 := M[FP + 4 * n + 4] \\ \cdots \\ Rk := M[FP + 4 * n + 4 * (k + 1)] \\ \texttt{GOTO}\ M[FP] \end{cases}$$

Difference: $R0 \ldots Rk$ are saved in *prologue* and restored in *epilogue*.

Callee-Saves: Call Sequence

$$
\begin{aligned}
&FP := FP + framesize \\
&M[FP + 4] := a_1 \\
&\ldots \\
&M[FP + 4 * n] := a_n \\
&M[FP] := returnaddress \\
&\texttt{GOTO } f \\
&\texttt{LABEL } returnaddress \\
&x := M[FP + 4] \\
&FP := FP - framesize
\end{aligned}
$$

Difference: $R0 \ldots Rk$ are not stored here.

Caller-Saves vs. Callee-Saves

So far, no big difference, but:

Caller-Saves vs. Callee-Saves

So far, no big difference, but:

- Caller-saves need only save the registers containing live variables.
- Callee-saves need only save the registers that are used in the function's body.

Can use a mixed strategy: some registers are caller-saves and others are callee-saves.

Use of Registers for Parameter Passing

If parameters are passed on the stack, they must be transfered from registers to stack and shortly after from stack to registers again.

The idea is to transfer some parameters and return value via registers:

- Subset of caller-saves registers (typically 4-8), used for parameter transfer (to be rarely preserved across the function call).
- A caller-saves register (often the same) is used for the result.
- The remaining parameters are transferred on the stack as shown.
- Often the return address is also transferred in a register.

Typical Register Breakdown

With a 16-register processor:

Register	Saved by	Used for
0	caller	parameter 1 / result / local variable
1-3	caller	parameters 2 - 4 / local variables
4-12	callee	local variables
13	caller	temporary storage (unused by register allocator)
14	callee	FP
15	callee	return address

Typically there are more callee-saves than caller-saves registers.

Activation Records for Register-Passed Parameters

	...
	Next activation records
	Space for storing local variables for spill and for storing live variables allocated to caller-saves registers across function calls
	Space for storing callee-saves registers that are used in the body
	Incoming parameters in excess of four
FP $\longrightarrow$	Return address
	Previous activation records
	...

Prologue and Epilogue for Register-Passed Params

Prologue

LABEL *function-name*
$M[FP + offset_{R4}] := R4$ (if used in body)
...
$M[FP + offset_{R12}] := R12$ (if used in body)
$M[FP] := R15$ (if used in body)
$parameter_1 := R0$
$parameter_2 := R1$
$parameter_3 := R2$
$parameter_4 := R3$
$parameter_5 := M[FP + 4]$
...
$parameter_n := M[FP + 4 * (n - 4)]$
code for the function body

Epilogue

$R0 := result$
$R4 := M[FP + offset_{R4}]$ (if used in body)
...
$R12 := M[FP + offset_{R12}]$ (if used in body)
$R15 := M[FP]$ (if used in body)
GOTO $R15$

Calling Sequence for Register-Passed Parameters

$M[FP + offset_{live_1}] := live_1$ (if allocated to a caller-saves register)
...
$M[FP + offset_{live_k}] := live_k$ (if allocated to a caller-saves register)
$FP := FP + framesize$
$R0 := a_1$
...
$R3 := a_4$
$M[FP + 4] := a_5$
...
$M[FP + 4 * (n - 4)] := a_n$
$R15 := returnaddress$
`GOTO` f
`LABEL` $returnaddress$
$x := R0$
$FP := FP - framesize$
$live_1 := M[FP + offset_{live_1}]$ (if allocated to a caller-saves register)
...
$live_k := M[FP + offset_{live_k}]$ (if allocated to a caller-saves register)

Interaction with Register Allocator

Register Allocation Can:

- Preferably place the variables that are not live after the function call in caller-saves registers (so that they are not saved).
- Determine which caller-saves registers need to be saved by the caller before the function call.

Interaction with Register Allocator

Register Allocation Can:

- Preferably place the variables that are not live after the function call in caller-saves registers (so that they are not saved).
- Determine which caller-saves registers need to be saved by the caller before the function call.
- Preferably place the variables that are live after the function call in callee-saves registers (i.e., the called function might not save them if those registers are not used).
- Determine which callee-saves register need to be saved by the callee, i.e., used in the callee's body.
- Eliminate the unnecessary copying of data to and from local vars.

The strategy used most often:

Interaction with Register Allocator

Register Allocation Can:

- Preferably place the variables that are not live after the function call in caller-saves registers (so that they are not saved).
- Determine which caller-saves registers need to be saved by the caller before the function call.
- Preferably place the variables that are live after the function call in callee-saves registers (i.e., the called function might not save them if those registers are not used).
- Determine which callee-saves register need to be saved by the callee, i.e., used in the callee's body.
- Eliminate the unnecessary copying of data to and from local vars.

The strategy used most often: use caller-saves only for the variables that are not live after the function call, hence they need not be saved.

This simplifies the call sequence.

Typical Function-Code-Generation Strategy

1. Generate code for the *function's body* using symbolic registers for named variables, but using numbered registers in the call sequence for parameters.
2. Add the prologue and epilogue code for moving numbered registers (and stack locations), i.e., parameters, into symbolic registers, i.e., named variables (and the opposite for the result).
3. Call the register allocator on this expanded function body.
   - `RegAlloc` is aware of the register division: caller vs callee saves,
   - Allocates live-across-call variables only in callee-saves regs,
   - Finds both the set of used callee-saves regs and of spilled vars.
4. Add to the prologue and epilogue the code for saving/restoring the callee-saves regs that `RegAlloc` said to have been used in the extended function body + updating FP (including space for saved regs and spilled vars).
5. Add a function label at prologue's start and a jump at then end.

Treating Global Variables

Global variables: allocated in memory at statically-known addresses.

Generated reading/writing code is similar to that of spilled variables:

$$x := M[address_x]$$
instruction that uses x

$x :=$ the value to be stored in x
$$M[address_x] := x$$

Various temporary variables (registers) may be used.

If a global variable is frequently used in a function, then copy-into a register in the beginning, and copy-out to memory at the end.

Copy-in/out across fun calls + extra care in the presence of aliasing.

Morale: Use global variables sparingly, local variables are preferred.

Call By Reference

The update to a function parameter is visible after the return point. Also applies to fields of records, unions, array's elements, etc.

Code generation typically treats such parameters as pointers.

Call by reference give rise to *aliasing*.

Morale: cheaper to update a variable via the returned value, rather than passing it by reference: `x = f(x);` is cheaper than `f(&x);`.

Aliasing

Aliasing: if the same memory location can be accessed via two different named variables.

This can occur when the language allows references, either:

- between two references pointing to the same place,
- between a global variable and a reference to this.

If two names may alias, then:

- before reading from one save the other to memory,
- if writing into one, then read the other again from memory before using it.

Can be sometimes optimized be means of *aliasing analysis*.

Task 5 – Call By Value Result for Procs in Paladim

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

```
and compileF (isProc, fname, args, block, pos) =
    (* at this point, we do not care about the return type (or no return) *)
    let (* make a vtable from bound formal parameters,
            then evaluate expression in this context, return it *)
        (* arguments passed in registers, "move" into local vars.
           Code generator imposes max. 13 arguments (maxCaller-minReg)
         *)
        val () = if length args <= maxCaller - minReg + 1 then ()
                 else raise Error (fname ^ ": too many arguments (max "
                                   ^ makestring (maxCaller - minReg + 1)
                                   ^")", pos)
        val (movePairs, vtable) = getMovePairs args [] minReg
        val argcode = map (fn (vname, reg) => Mips.MOVE (vname, reg)) movePairs
(** TASK 5: You need to add code to move variables back into callee registers,
 * i.e. something similar to 'argcode', just the other way round.  Use the
 * value of 'isProc' to determine whether you are dealing with a function
 * or a procedure. **)
```

To be read: at the end of procedure prologue, move formal-arguments registers back to the number registers!

Task 5 – Call By Value Result for Procs in Paladim

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

```
| compileStmt(vtable, s, exitLabel) = case s of ...
  | ProcCall ((n,_), es, p) =>
      let val (mvcode, maxreg) = putArgs es vtable minReg
      in mvcode @ [Mips.JAL (n, List.tabulate (maxreg, fn reg => makeConst reg))]
      end
and putArgs [] vtable reg = ([], reg)
  | putArgs (e::es) vtable reg =
      let val t1 = "_funarg_"^newName()
          val code1 = compileExp(vtable, e, t1)
          val (code2, maxreg) = putArgs es vtable (reg+1)
      in  ( code1 @ code2 @ [Mips.MOVE (makeConst reg,t1)], maxreg )
      end
```

Modify putArgs so that it also returns a prologue code that `MOVE`s the numbered (caller save) registers to the registers associated to any actual argument that is a variable (as destination).

Note that the latter are not the temporary registers (`_funarg_`), but are to be found from `vtable` , i.e., symbol table lookup.