

G53CMP: Lecture 11

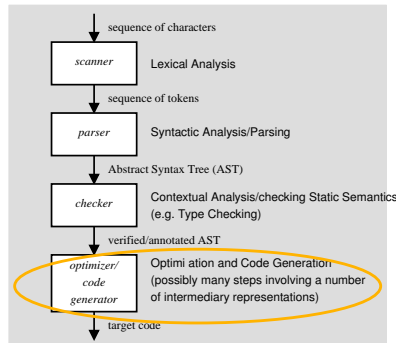
Code Generation I

Henrik Nilsson

University of Nottingham, UK

G53CMP: Lecture 11 - p.1/30

Where Are We?



G53CMP: Lecture 11 - p.2/30

Code Generation: Subproblems (1)

The code generator must address the following issues:

- **Code Selection:** Which code sequence to generate for each source code phrase? For example, for an expression (phrase) like

```
y := 3 + x * 5
```

the code for a register machine might be:

```
MUL R7, R1, #5
ADD R2, R7, #3
```

G53CMP: Lecture 11 - p.3/30

Code Generation: Subproblems (2)

- **Storage Allocation:** Where and how to store data? E.g.
 - Global variables
 - Local variables
- **Register Allocation:** How to allocate registers for variables and other purposes?

G53CMP: Lecture 11 - p.4/30

Run-Time Organisation (1)

Code generation is intimately related to the **Run-Time Organisation**. This includes:

- **Memory Organisation:** How to organise the memory into data structures for different kinds of storage; e.g.
 - Global static storage
 - Stacks
 - Heaps

G53CMP: Lecture 11 - p.5/30

Run-Time Organisation (2)

- **Calling conventions:** protocols for procedure/function/method calls and returns, including how to
 - pass arguments
 - return results
- **Data Representation:** How to represent high-level data types (integers, records, arrays, objects, ...) as sequences of bits?

G53CMP: Lecture 11 - p.6/30

This Lecture

- Code selection
- Specifically, code selection for the Triangle Abstract Machine (TAM), a **stack machine**.
- Stack machines:
 - simplify code selection
 - allow us to defer a more in-depth treatment of run-time organisation until later
 - but we will cover the basics of TAM calling conventions

G53CMP: Lecture 11 - p.7/30

The Triangle Abstract Machine (1)

Watt & Brown use the **Triangle Abstract Machine** (TAM) to illustrate code generation. We will use a variant.

- TAM is a simple **stack machine**.
- Dedicated registers define the stack: ST, LB, SB.
- Operands and results for all instructions on the stack.
- Register allocation is thus a non-issue.

G53CMP: Lecture 11 - p.8/30

The Triangle Abstract Machine (2)

Stack machines in perspective:

- Hardware CPUs (e.g. x86, SPARC, ARM) tend to be **register machines**, not stack machines.
- Code for a stack machine thus has to be either
 - interpreted
 - compiled further
- The Java Virtual Machine (JVM) is a prominent, real-world example of a stack machine.
- JVM code is typically Just-In-Time (JIT) compiled for execution speed.

G53CMP: Lecture 11 - p.9/30

TAM Registers

The TAM has a number of registers related to the stack. Among others:

- **SB**: Stack Base
- **ST**: Stack Top
- **LB**: Local Base

G53CMP: Lecture 11 – p.13/30

TAM Instructions (3)

- **JUMP** *l*: jump unconditionally to label *l*.
- **JUMPIFZ** *l*: pop value on top of stack, jump to label *l* if it is 0.
- **JUMPIFNZ** *l*: pop value on top of stack, jump to label *l* if it is not 0.
- **CALL** *f*: call function at label *f*, arguments and result on stack.
- **RETURN** *m n*: return to caller from routine with *n* arguments with the *m* top stack locations replacing the activation record.

G53CMP: Lecture 11 – p.13/30

Exercise: TAM Code Selection

Assuming the variable

- *x* is stored at address $[SB + 1]$
- *y* is stored at address $[SB + 2]$

write code for

```
x := y; y := 17
```

and

```
repeat
```

```
  y := y + x;  
  x := x + 1
```

```
until x == 10
```

G53CMP: Lecture 11 – p.13/30

TAM Instructions (1)

- **LOADL** *c*: push constant *c* onto stack.
- **LOADA** *a*: push *address a* onto stack. Address *a* can be e.g. $[SB + d]$ or $[LB + d]$.
- **LOAD** *a*: push *contents* at address *a* onto stack. Address *a* can be e.g. $[SB + d]$ or $[LB + d]$.
- **STORE** *a*: pop value from stack and store at address *a*.
- **LOADI** *d* and **STOREI** *d*: *indirect* load and store; target address = top stack elem. + *d*.
- **POP** *m n*: pop *n* values below the top *m* values off the stack.

G53CMP: Lecture 11 – p.13/30

TAM Instructions (4)

All of the following take argument(s) from the stack and leave the result on the stack:

- **Arithmetic**: ADD, SUB, MUL, DIV, NEG
- **Comparison**: LSS, EQL, GTR
- **Logical**: AND, OR, NOT

(There are also subroutines for these operations (and more) in the MiniTriangle standard library. E.g. **CALL mul** is an alternative to **MUL**. This allows for a uniform treatment of functions, facilitating code generation.)

G53CMP: Lecture 11 – p.14/30

TAM Calling Conventions (1)

```
var n: Integer;  
...  
fun f(x, y: Integer): Integer =  
  let  
    z: Integer  
  in begin  
    z := x * x + y * y;  
    return n * z  
  end
```

(Not quite current MiniTriangle as function body must be an expression.)

G53CMP: Lecture 11 – p.17/30

TAM Instructions (2)

- **LOAD** $[SB + d]$: fetch the value of the (global) variable at address *d* relative to *SB*.
- **STORE** $[SB + d]$: store a value in the (global) variable at address *d* relative to *SB*.
- **LOAD** $[LB + d]$: fetch the value of the (local) variable at address *d* relative to *LB*.
- **STORE** $[LB + d]$: store a value in the (local) variable at address *d* relative to *LB*.

Displacements may also be negative; e.g. **LOAD** $[SB - d]$ etc.

Addressing relative to *ST* also possible.

G53CMP: Lecture 11 – p.12/30

Example: TAM Code Selection

Example of code selection for TAM:

```
x := x * 2
```

TAM code, assuming *x* stored at $[SB + 1]$:

```
LOAD [SB + 1]  
LOADL 2  
MUL  
STORE [SB + 1]
```

Let's do a live demo ...

G53CMP: Lecture 11 – p.12/30

TAM Calling Conventions (2)

TAM activation record layout:

address	contents
LB - <i>argOffset</i>	arguments
...	...
LB	static link
LB + 1	dynamic link
LB + 2	return address
LB + 3	local variables
...	...
LB + <i>tempOffset</i>	temporary storage

where

$argOffset = size(arguments)$

$tempOffset = 3 + size(local variables)$

G53CMP: Lecture 11 – p.18/30

TAM Calling Conventions (3)

TAM code for the example:

```

LOADL 0      ; z      ADD
LOAD  [LB - 2]; x     STORE  [LB + 3] ; z
LOAD  [LB - 2]; x     LOAD   [SB + 42]; n
MUL                    LOAD   [LB + 3] ; z
LOAD  [LB - 1]; y     MUL
LOAD  [LB - 1]; y     POP    1 1
MUL                    RETURN 1 2
    
```

Note: all offsets are in **words** (4 bytes).

G53CMP: Lecture 11 - p.19/30

Execution of the Example (1)

On entry:

address	contents
...	...
SB + 42	n: n
...	...
LB - 2	x: x
LB - 1	y: y
LB	static link
LB + 1	dynamic link
LB + 2	return address
ST	

G53CMP: Lecture 11 - p.20/30

Execution of the Example (2)

After LOADL 0:

address	contents
...	...
SB + 42	n: n
...	...
LB - 2	x: x
LB - 1	y: y
LB	static link
LB + 1	dynamic link
LB + 2	return address
LB + 3	z: uninitialized
ST	

G53CMP: Lecture 11 - p.21/30

Execution of the Example (3)

After LOAD [LB - 2]; LOAD [LB - 2]:

address	contents
...	...
SB + 42	n: n
...	...
LB - 2	x: x
LB - 1	y: y
LB	static link
LB + 1	dynamic link
LB + 2	return address
LB + 3	z: uninitialized
LB + 4	x
LB + 5	x
ST	

G53CMP: Lecture 11 - p.22/30

Execution of the Example (4)

After MUL:

address	contents
...	...
SB + 42	n: n
...	...
LB - 2	x: x
LB - 1	y: y
LB	static link
LB + 1	dynamic link
LB + 2	return address
LB + 3	z: uninitialized
LB + 4	x^2
ST	

G53CMP: Lecture 11 - p.23/30

Execution of the Example (5)

After LOAD [LB-1]; LOAD [LB-1]; MUL:

address	contents
...	...
SB + 42	n: n
...	...
LB - 2	x: x
LB - 1	y: y
LB	static link
LB + 1	dynamic link
LB + 2	return address
LB + 3	z: uninitialized
LB + 4	x^2
LB + 5	y^2
ST	

G53CMP: Lecture 11 - p.24/30

Execution of the Example (6)

After ADD:

address	contents
...	...
SB + 42	n: n
...	...
LB - 2	x: x
LB - 1	y: y
LB	static link
LB + 1	dynamic link
LB + 2	return address
LB + 3	z: uninitialized
LB + 4	$x^2 + y^2$
ST	

G53CMP: Lecture 11 - p.25/30

Execution of the Example (7)

After STORE [LB + 3]:

address	contents
...	...
SB + 42	n: n
...	...
LB - 2	x: x
LB - 1	y: y
LB	static link
LB + 1	dynamic link
LB + 2	return address
LB + 3	$z: x^2 + y^2$
ST	

G53CMP: Lecture 11 - p.26/30

Execution of the Example (8)

After LOAD [SB + 42]; LOAD [LB + 3]:

address	contents
...	...
SB + 42	n: n
...	...
LB - 2	x: x
LB - 1	y: y
LB	static link
LB + 1	dynamic link
LB + 2	return address
LB + 3	$z: x^2 + y^2$
LB + 4	n
LB + 5	$x^2 + y^2$
ST	

G53CMP: Lecture 11 - p.27/30

Execution of the Example (9)

After MUL:

address	contents
...	...
SB + 42	n: n
...	...
LB - 2	x: x
LB - 1	y: y
LB	static link
LB + 1	dynamic link
LB + 2	return address
LB + 3	z: $x^2 + y^2$
LB + 4	$n(x^2 + y^2)$
ST	

Execution of the Example (10)

After POP 1 1:

address	contents
...	...
SB + 42	n: n
...	...
LB - 2	x: x
LB - 1	y: y
LB	static link
LB + 1	dynamic link
LB + 2	return address
LB + 3	$n(x^2 + y^2)$
ST	

POP is used here to tidy away the storage for local variables, preserving only the overall result.

Execution of the Example (11)

After RETURN 1 2:

address	contents
...	...
SB + 42	n: n
...	...
Stack Top	$n(x^2 + y^2)$
ST	

RETURN tidies away the rest of the activation record and returns to the caller.

Stack Top is in *f's caller's* activation record, at some offset from *f's* caller's LB.