Compilation (#8) : Functions: syntax and code generation

MIF08

Laure Gonnord
`Laure.Gonnord@univ-lyon1.fr`

nov 2016

Big picture

So far:

- All variables were global.
- No function call.

Inspiration: N. Louvet, Lyon1 (archi part), C. Alias (code gen part).

Outline

1. Front-end
2. Syntax-Directed Code Generation

Concrete syntax 1/2

- we add variable declaration (with the `var` keyword):

```
vardecl
 : VAR ID ASSIGN expr
 ;
```

- blocks are like before:

```
block
 : stat*   #statList
 ;
stat_block
 : OBRACE block CBRACE
 | stat
;
```

- procedures declaration:

```
declproc:
 : PROC ID IS stat
;
```

Concrete syntax 2/2

And now there will two new kinds of statements:

```
stat
 : assignment
 | if_stat
 | while_stat
 | log
 | CALL ID
 | BEGIN declvar* declproc* block_stat END
 ;
```

- We can declare local procedures inside local procedures.

On board : add new concrete syntax for **functions**.

Abstract syntax

WLOG, we will only consider programs with procedures:

$$
\begin{array}{lcl}
S & \in & \textbf{Stm} \\
S & ::= & x := a \mid \text{skip} \mid S_1 ; S_2 \mid \\
 & & \text{if } b \text{ then } S_1 \text{ else } S_2 \\
 & & \text{while } b \text{ do } S \text{ od } \mid \text{begin } D_V\ D_P ; S \text{ end} \mid \text{call } p \\
D_V & ::= & \text{var } x := a;\ D_V \mid \epsilon \\
D_P & ::= & \text{proc } p \text{ is } S;\ D_P \mid \epsilon
\end{array}
$$

Exercise

EX : syntax for functions

Outline

1. Front-end
2. Syntax-Directed Code Generation
   - Procedure call in LC-3
   - Code Generation for functions

A bit about Typing

Two important remarks:

- Now that variables are local, the typing environnement should also be updated each time we enter a procedure.
- Type checking for functions: construct the type from definitions, check when a call is performed (see the course on typing ML).

Outline

1. Front-end
2. Syntax-Directed Code Generation
   - Procedure call in LC-3
   - Code Generation for functions

Routines

A procedure/routine in assembly is just a piece of code

- its first instruction's address is known and tagged with a label.
- the JSR instruction jumps to this piece of code (routine call).
- at the end of the routine, a RET instruction is executed for the PC to get the address of the instruction after the routine call.

Slides coming from the architecture course, N. Louvet

Routines in LC-3, how? JSR

When a routine is called, we have to store the address where to come back:

- syntax : JSR label
- action : R7 <- PC ; PC <- PC + SEXT(PCoffset11)
  - -1024$\leq$Sext(Offset11)$\leq$1023.
  - if adI is the JSR instruction's address, the branching address is:

$$\text{adM} = \text{adI}+1+\text{Sext(PCOffset11)}, \quad \text{with}$$
$$\text{adI} - 1023 \leq \text{adM} \leq \text{adI} + 1024.$$

Routines in LC-3, how RET

Inside the routine code, the RET instruction enables to come back:

- syntax : RET
- action : PC <- R7

Writing routines

Call to the `sub` routine:

```
 ...
JSR sub    ; R7 <- next line address
...
```

The last instruction of the routine is RET :

```
; sub routine
sub:  ...
      ...
      RET        ; back to main
```

An example - strlen, without routine

```
        .ORIG x3000
        LEA R0,string   ;
        AND R1,R1,0     ;
loop:   LDR R2,R0,0     ;
        BRz end         ;
        ADD R0,R0,1     ;
        ADD R1,R1,1     ;
        BR loop
end:    ST R1,res
        HALT
; Constant chain
string: .STRINGZ "Hello World"
res:    .BLKW #1
        .END
```

String length routine 1/2

`strlen` call (the result will be stored in R0).

```
        .ORIG x3000
; Main program
        LEA R0,string  ; R0 <- @(string)
        JSR strlen      ; routine call
        ST R0,lg1
        HALT

; Data
string: .STRINGZ "Hello World"
lg1:    .BLKW #1
```

String length routine 2/2

```
strlen: AND R1,R1,0     ;
loop:   LDR R2,R0,0     ;
        BRz end         ;
        ADD R0,R0,1     ;
        ADD R1,R1,1     ;
        BR loop
end:    ADD R0,R1,0     ; R0 <- R1
        RET             ; back to main (JMP R7)

        .END            ; END of complete prog
```

Routines in LC-3: chaining routines

If a routine needs to **call another one**:

- Some temporary registers may have to be stored somewhere
- Its return address (in R7!) needs also to be stored.

► Store in the stack (R6) before, restore after.

Outline

1. Front-end
2. Syntax-Directed Code Generation
   - Procedure call in LC-3
   - Code Generation for functions

Rules of the game

We still have our LC3 machine with registers:

- general purpose registers R0 to R5.
- a stack pointer (SP), here R6.
- a frame pointer (FP), here R7

Simplification: **no imbricated** function declaration.

- when `call p`, there is a unique `p` code labeled by $p:$

Key notion: activation record - Vocabulary 1/2

(picture needed)

- Any execution instance of a function is called an **activation**.
- We can represent all the activations of a given program with an **activation tree**.

Key notion: activation record - Vocabulary 2/2

During execution, we need to keep track of alive activations:

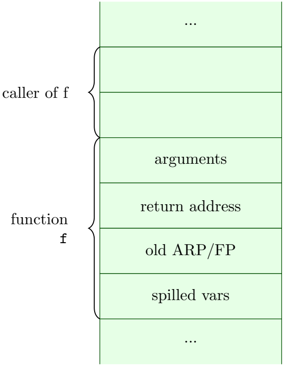
- Control stack
- An activation is pushed when activated
- When its over, it is poped out.

► **Notion of activation record** that stores the information of one function call at execution.

► **The compiler** is in charge of their management.

Slides inspired by C. Alias

Activation record of a given function



The frame pointer (ARP or FP) points to the current activation record (first spilled variable).

Code generation 1/2

For functions, we have to reserve (local) place before knowing the number of spilled variables!

int f(x1,x2) S;
return e

```
code.addMacro(PUSH R7)        #store @ret
code.addcopy(R6,R7)           #R7<-R6
code.addCode(ADD R6 R6 xx)    #xx= future nb of spilled vars
code.addCode(LDR tmp1 R7 -1)  #arg1
code.addCode(LDR tmp2 R7 -2)  #arg2 (in rev order)
CodeGenSmt(S)                 #under the context x1->tmp1...
dr<-CodeGen(e)                #same!
code.addcopy(dr,R0)  #convention return val in R0
code.addMacro(RET,2+xx) #desalloc args + spilled vars + retur
```

- `CodeGenSmt` must be called with a modified map.

Code generation 2/2

call f(e1,e2)

```
Gencodesaveregisters()   #save current values of reg.
dr <- newtmp
dr1=Gencode(e1)
code.addMacro(PUSH dr1)
dr2=Gencode(e2)
code.addMacro(PUSH dr2)
code.add(JSR f)     #return @ in R7
code.addcopy(r0,dr) # dr <- returned value
Gencoderestoreregisters() #restore curr values of reg.
return dr
```

A simple example 1/3

Generate code and draw the activation records during the call execution of f:

```
int f(x) {return x+1;}
```

```
main:
z:=f(7);
```

A simple example 2/3

```
main:
PUSH(R0,R1....R5)   #should be replaced by R6 manipulation.
AND tmp1 tmp1 0
ADD tmp1 temp1 7
PUSH(tmp1)
JSR f
AND tmp2 tmp2 0
AND tmp2 R0 0
pop(R5... ,R1,R0) #but not the register associated to temp2
[use of temp2 here]
```

A simple example 3/3

```
f: PUSH(R7)
   COPY(R6,R7)
   ADD R6 R6 xx       #xx=number of spilled vars
   LDR tmp1 R7 #1     #first argument
   ADD tmp2 tmp1 1
   COPY(tmp2,R0)      #store result in R0
   COPY(R7,R6)        #this is postlude
   ADD R6 R6 -1      #1 argument
   POP(R7)
```

Register allocation gives $tmp1, tmp2$ are allocated in $R1$ (or R0 if we are clever). Thus `xx`=0.

To go further

- How to implement the different calling conventions? (here, call by value)?
- How to implement imbricated functions (dynamic link, static link).
- How to store more complex types (arrays, structs, user defined types)?