Compilation and Program Analysis (#9) : Functions: syntax, semantics, code generation

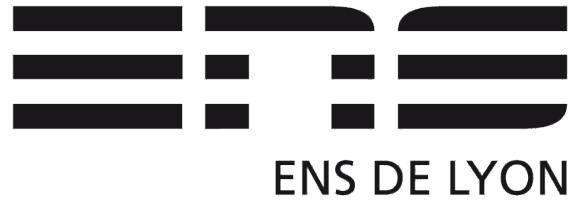
Laure Gonnord

http://laure.gonnord.org/pro/teaching/capM1.html

Laure.Gonnord@ens-lyon.fr

Master 1, ENS de Lyon

2018-2019



Big picture

So far :

- All variables were global.
- No function call.

Inspiration : Y. Lakhnesh, UGA (first part), N. Louvet, Lyon1 (archi part), C. Alias (code gen part).

1 Front-end

2 Operational Semantics for procedures

3 Syntax-Directed Code Generation

Concrete syntax 1/2

- blocks are like before :

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

- procedures declaration :

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

- variable decl can be local.

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 :

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

1 Front-end

2 Operational Semantics for procedures

3 Syntax-Directed Code Generation

Operational semantics for local variables

Variable declaration : we invent $\rightarrow_D$ a semantic for declarations, and the following definitions :

- $DV(D_V)$ is the set of variables declared in D_V
- $\sigma'[X \mapsto \sigma] = \lambda x.$ if $x \in X$ then $\sigma(x)$ else $\sigma'(x)$.

Rules :

$$\frac{(D_V, \sigma[x \mapsto \mathcal{A}[a]\sigma]) \rightarrow_D \sigma'}{(\text{var } x := a; D_V, \sigma) \rightarrow_D \sigma'}$$

$$(\varepsilon, \sigma) \rightarrow_D \sigma$$

Rules for blocs + Example

Be careful to restore everything after the local bloc :

$$\frac{(D_V, \sigma) \rightarrow_D \sigma' \qquad (S, \sigma') \rightarrow \sigma"}{(\text{begin } D_V; S \text{ end}, \sigma) \rightarrow_D \sigma"[DV(D_V) \mapsto \sigma]}$$

Seq is unchanged :

$$\frac{(S_1, \sigma) \rightarrow \sigma' \ (S_2, \sigma') \rightarrow \sigma''}{\big((S_1; S_2), \sigma\big) \rightarrow \sigma''}$$

Execute the semantics :

```
begin var y := 1;
      (x := 1;
      begin var x := 2; y := x+1 end;
      x := y+x)
end
```

Dynamic versus Static binding, an example

What will be the behavior of call q ?

```
begin   var x := 0;
        proc p is x := x * 2;
        proc q is call p;
        begin var x := 5;
              proc p is x := x + 1;
              call q; y := x;
        end;
end
```

It depends !

Dynamic versus Static binding, ex 3/4

Dynamic binding for variables and dynamic for procedures :

```
begin   var x := 0;
        proc p is x := x * 2;
        proc q is call p;
        begin var x := 5;
              proc p is x := x + 1;
              call p; y := x;
        end;
end
```

```
begin   var x := 0;
        proc p is x := x * 2;
        proc q is call p;
        begin var x := 5;
              proc p is x := x + 1;
              x := x + 1; y := x;
        end;
end
```

Dynamic versus Static binding, ex 4/4

Static for variables and procedures :

```
begin   var x := 0;
        proc p is x := x * 2;
        proc q is call p;
        begin var x := 5;
              proc p is x := x + 1;
              call p; y := x;
        end;
end
```

```
begin   var x := 0;
        proc p is x := x * 2;
        proc q is call p;
        begin var x := 5;
              proc p is x := x + 1;
              x := x * 2; y := x;
        end;
end
```

Semantics : dynamic links 1/2

How ?

- States : variable $\rightarrow$ int.
- Environment : procedure name $\rightarrow$ Stm.
- Configuration : $(Env_P \times Stm \times State) \cup State$

The dynamic link for procedures is made by calling the *current* value at the call :

$$\frac{(env, env(p), \sigma) \rightarrow \sigma'}{(env, \textsf{call } p, \sigma) \rightarrow \sigma'}$$

Semantics : dynamic links 2/2

Thus, environments are kept for the sequence :

$$\frac{(env, S_1, \sigma) \to \sigma' \qquad (env, S_2, \sigma') \to \sigma"}{(env, (S_1; S_2), \sigma) \to \sigma'}$$

but they are updated with a local declaration :

$$\frac{(D_V, \sigma) \to_D \sigma' \quad (\texttt{upd}(env, D_P), S, \sigma') \to \sigma"}{(env, \texttt{begin } D_V D_P; S \texttt{ end }, \sigma) \to \sigma"[\texttt{DV}(D_V) \mapsto \sigma]}$$

with

- $\texttt{upd}(env, \varepsilon) = env$
- $\texttt{upd}(env, \texttt{proc } p \texttt{ is S}; D_P) = \texttt{upd}(env[p \mapsto S], D_P)$

▶ Ex : test on the example !

Static links for procedures

We need to store an environnement while defining a procedure :

- Environment : procedure name $\rightarrow Stm \times Env_P$
- Configuration : $(Env_P \times Stm \times State) \cup State$

Now update is modified :

- $\texttt{upd}(env, \varepsilon) = env$
- $\texttt{upd}(env, \texttt{proc } p \texttt{ is S}; D_P) = \texttt{upd}(env[p \mapsto (S, env)], D_P)$.

```
begin   var x := 2;
        proc p is x := 0;
        proc q is begin x := 1; (proc p is call p); call p end;
        call q
end
```

Static links for procedures 2/2

Two possibilities for the call, let $env(p) = (S, env')$ in :

$$\frac{(env', S, \sigma) \to \sigma'}{(env, \texttt{call } p, \sigma) \to \sigma'}$$

or

$$\frac{(env'[p \mapsto [(S, env')], S, \sigma) \to \sigma'}{(env, \texttt{call } p, \sigma) \to \sigma'}$$

Static link for variables and procedures 1/3

Config = (env_V, env_P, S, sto) or (env_V, sto) with :

- $env_V : x \mapsto address$ “symbol table”.
- $env_P : p \mapsto (env_V, env_P, S)$ to store values during variable/proc declaration
- $sto : address \mapsto \mathbb{Z}$ (old $\sigma(x) = sto(env_V(x))$).
- $new(sto)$ gives a new address.

Static link for variables and procedures 2/3

Variable declaration :

- $(\varepsilon, env_V, sto) \rightarrow_D (env_V, sto)$
- $$\frac{(D_V, env_V[x \mapsto nc], sto[nc \mapsto v]) \rightarrow_D (env'_V, sto')}{((\texttt{var}\ \ x := a; D_V), env_V, sto) \rightarrow_D (env'_V, sto')}$$

  with $v = \mathcal{A}[a](sto \circ env_V)$, $nc = new(sto)$ such that $x \rightarrow nc \rightarrow v$

Static link for variables and procedures 3/3

$$(env_V, env_P, x := a, sto) \rightarrow (env_V, sto[nc \mapsto v])$$

with $v = \mathcal{A}[a](sto \circ env_V)$ and $nc = env_V(x)$

$$\frac{(env'_V, env'_P, S, sto) \rightarrow (env'_V, sto')}{(env_V, env_P, \texttt{call } p, sto) \rightarrow (env_V, sto')}$$

or

$$\frac{(env'_V, env'_P[p \mapsto (env'_V, env'_P, S)], S, sto) \rightarrow (env'_V, sto')}{(env_V, env_P, \texttt{call } p, sto) \rightarrow (env_V, sto')}$$

where $env_P(p) = (env'_V, env'_P, S)$.

1 Front-end

2 Operational Semantics for procedures

3 Syntax-Directed Code Generation
- Procedure call
- 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).

1 Front-end

2 Operational Semantics for procedures

3 Syntax-Directed Code Generation
- Procedure call
- 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.
- there is one instruction that 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 the target machine, how ?

When a routine is called, we have to store the address where to come back (current pc), it can be a special register.

- TARGET18 case : `call` stores the return address (in R7 ?)

this should be restored with another instruction, RET in general (jumps to the stored address).

be careful not to write in this special register

An example - strlen, without routine

```
lea r0 str
print string r0
leti r1 0
setctr a0 r0
loop:
readse a0 8 r2
add r2 r2 0
jumpif z end
add r1 1
jump loop
end:
print signed r1
;; now stores r1 in mem
lea r3 res
setctr a1 r3
write a1 16 r1
fin:
```

String length routine 1/2

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

```
lea r0 str
print string r0
call strlen
print signed r0 ;result is now in r0
lea r3 res
setctr a1 r3
write a1 16 r0
jump fin

[...]
fin:
jump fin
str:
 string "Hello, world!"
res:
```

String length routine 2/2

```
strlen: ; subroutine
leti r1 0
setctr a0 r0 ; r0 has the parameter @str
loop:
readse a0 8 r2
add r2 r2 0
jumpif z end
add r1 1
jump loop
end:
print signed r1
let r0 r1
return  ;end of subroutine
```

Routines in TARGET18 : chaining routines

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

- Some temporary registers may have to be stored somewhere
- Its return address needs also to be stored.

- Store in the stack (sp) before, restore after.

1. Front-end
2. Operational Semantics for procedures
3. Syntax-Directed Code Generation
   - Procedure call
   - Code Generation for functions

Rules of the game

We still have our TARGET18 machine with registers :

- general purpose registers
- a stack pointer (SP)
- a frame pointer (FP), (here R7 ?)

Simplification : **no nested** function declaration.

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

Key notion : activation record - Vocabulary 1/2

- 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 :

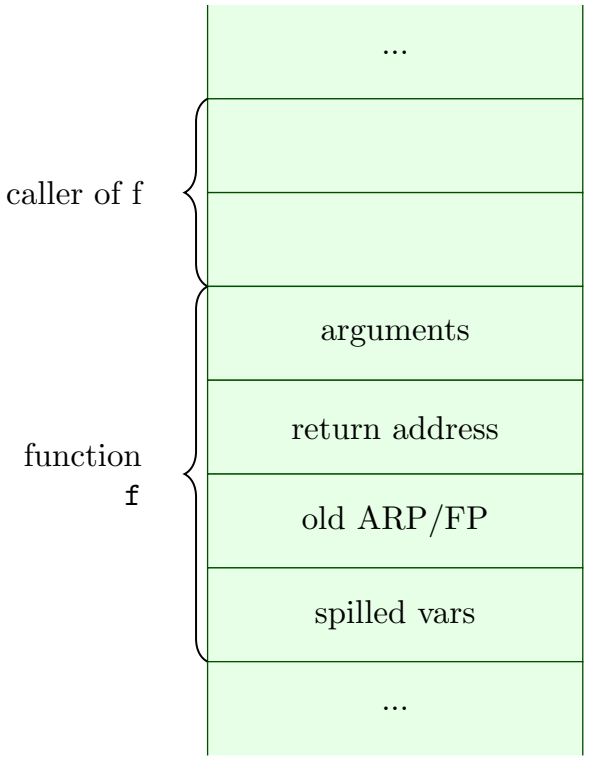
- Control stack
- An activation is pushed when activated
- When its over, it is poped out.

▶ **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(SUB R6 R6 xx)` `#xx= future nb of spilled vars`
	`code.addMacro(RMEM tmp1 R7 -2)` `#arg1 is in @[R7-2]`
	`code.addMacro(RMEM tmp2 R7 -1)` `#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(call 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)
call 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
   [RMEM] 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 xx+1    #1 argument
   POP(R7)
```

Register allocation gives tmp1,tmp2 mapsto 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 nested functions (dynamic link, static link).
- How to store more complex types (arrays, structs, user defined types) ?