8 VM code generation

- Aspects of code generation
- Address allocation
- Code selection
- Example: Fun code generator
- Representing addresses
- Handling jumps

Aspects of code generation *(1)*

- **Code generation** translates the source program (represented by an AST) into equivalent object code.
- In general, code generation can be broken down into:
  - **address allocation**
    (deciding the representation and address of each variable in the source program)
  - **code selection**
    (selecting and generating object code)
  - **register allocation** (where applicable)
    (assigning registers to local and temporary variables).

Aspects of code generation *(2)*

- Here we cover code generation for **stack-based VMs**:
  - address allocation is straightforward
  - code selection is straightforward
  - register allocation is *not* an issue!
- Later we will cover code generation for real machines, where register allocation *is* an issue *(see §15)*.

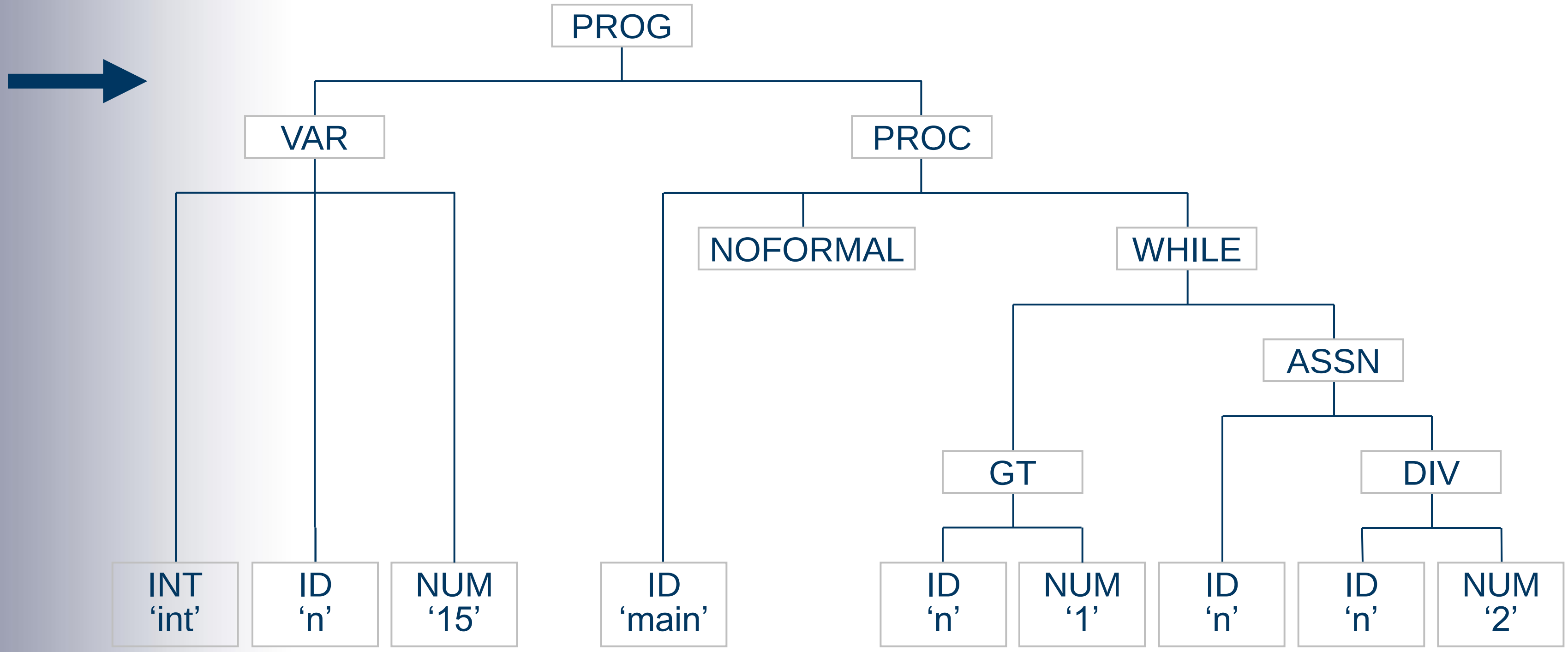
Example: Fun compilation *(1)*

- Source program:

```
int n = 15
# pointless program
proc main ():
  while n > 1:
    n = n/2 .
.
```

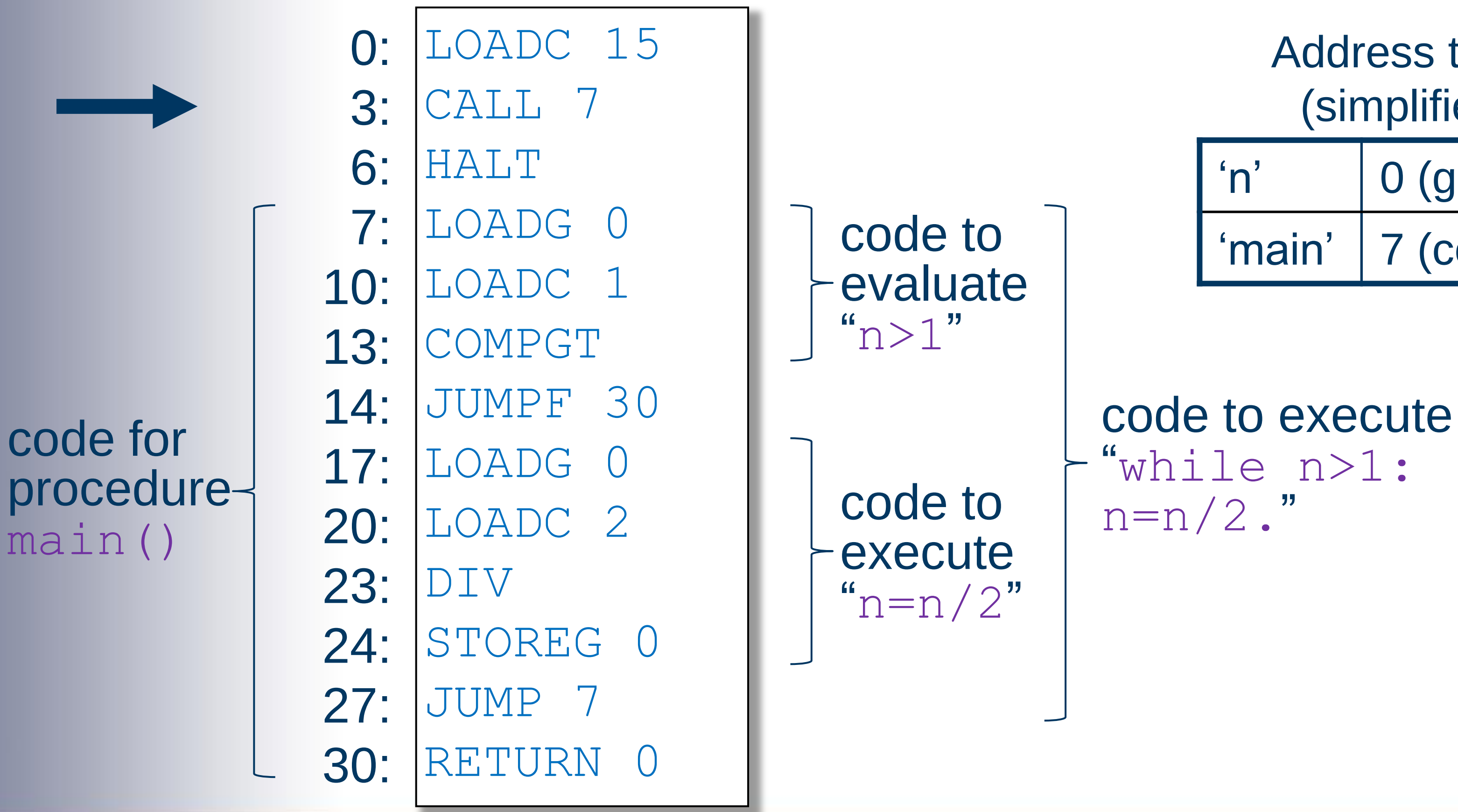
Example: Fun compilation *(2)*

- AST after syntactic analysis (slightly simplified):



Example: Fun compilation *(3)*

- SVM object code after code generation:



```
 0: LOADC  15
 3: CALL   7
 6: HALT
 7: LOADG  0
10: LOADC  1
13: COMPGT
14: JUMPF  30
17: LOADG  0
20: LOADC  2
23: DIV
24: STOREG 0
27: JUMP   7
30: RETURN 0
```

code for procedure `main()`

code to evaluate “`n>1`”

code to execute “`n=n/2`”

code to execute “`while n>1: n=n/2.`”

Address table (simplified)

‘n’	0 (global)
‘main’	7 (code)

Address allocation *(1)*

- Address allocation requires collection and dissemination of information about declared variables, procedures, etc.
- The code generator employs an **address table**. This contains the address of each declared variable, procedure, etc. E.g.:

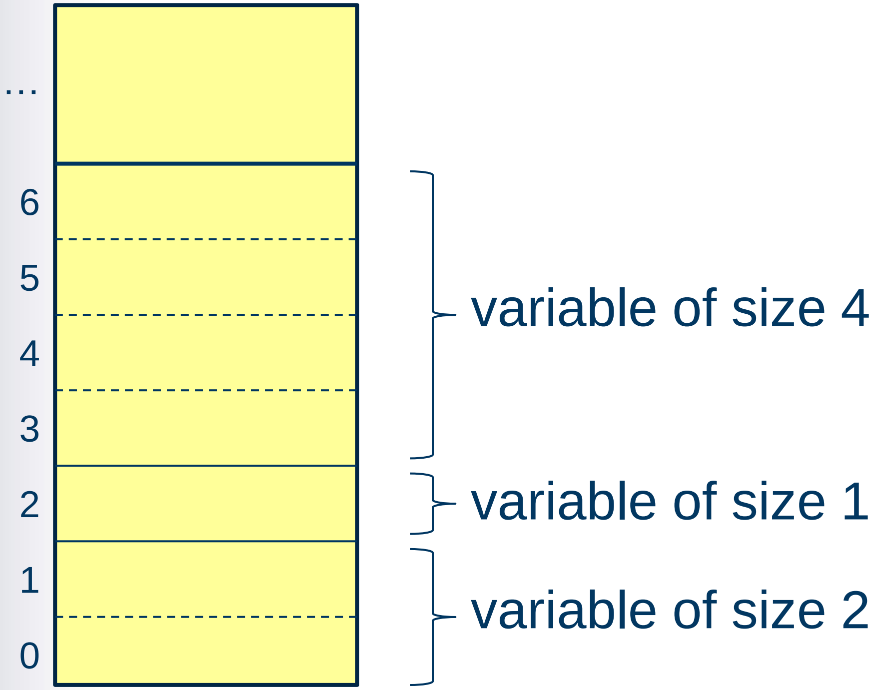
‘x’	0 (global)	variables
‘y’	2 (global)	
‘fac’	0 (code)	procedures
‘main’	7 (code)	

Address allocation *(2)*

- At each *variable declaration*, allocate a suitable address, and put the identifier and address into the address table.
- Wherever a variable is *used* (e.g., in a command or expression), retrieve its address.
- At each *procedure declaration*, note the address of its entry point, and put the identifier and address into the address table.
- Wherever a procedure is *called*, retrieve its address.

Address allocation *(3)*

- Allocate consecutive addresses to variables, taking account of their sizes. E.g.:



- *Note:* Fun is simpler: all variables are of size 1.

Code selection

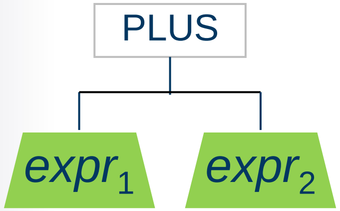
- The code generator will walk the AST.
- For each construct (expression, command, etc.) in the AST, the code generator must emit suitable object code.
- The developer must plan what object code will be selected by the code generator.

Code templates

- For each construct in the source language, the developer should devise a **code template**. This specifies what object code will be selected.
- The code template to evaluate an *expression* should include code to evaluate any sub-expressions, together with any other necessary instructions.
- The code template to execute a *command* should include code to evaluate any sub-expressions and code to execute any sub-commands, together with any other necessary instructions.

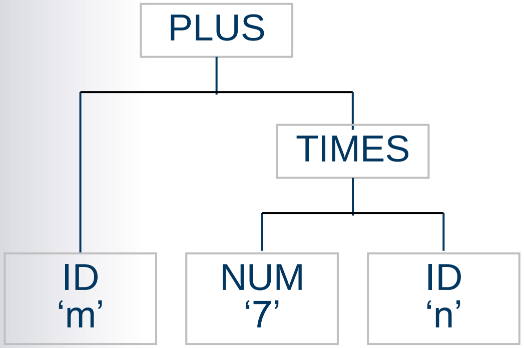
Example: Fun → SVM code templates *(1)*

- Code template for binary operator:



code to evaluate $expr_1$
code to evaluate $expr_2$
`ADD`

- E.g., code to evaluate "`m+(7*n)`":



```
LOADG 3
LOADC 7
LOADG 4
MULT
ADD
```

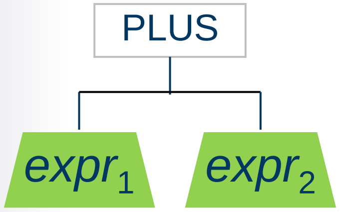
code to evaluate "`m`"

code to evaluate "`7*n`"

- We are assuming that `m` and `n` are global variables at addresses 3 and 4, respectively.

Example: Fun → SVM code templates *(2)*

- Code generator action for binary operator:

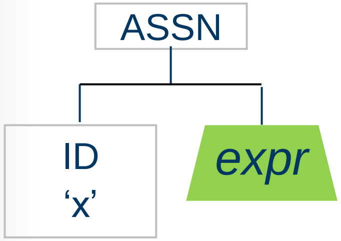


walk $expr_1$ generating code;
walk $expr_2$ generating code;
emit instruction "`ADD`"

- Compare:
  - The *code template* specifies what code should be selected.
  - The *action* specifies what the code generator will actually do to generate the selected code.

Example: Fun → SVM code templates *(3)*

- Code template for assignment-command:



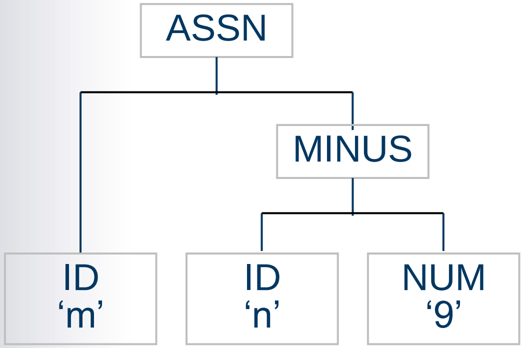
→

code to evaluate *expr*

`STOREG` *d* or `STOREL` *d*

where *d* is the address offset of 'x'

- E.g., code to execute "`m = n-9`":



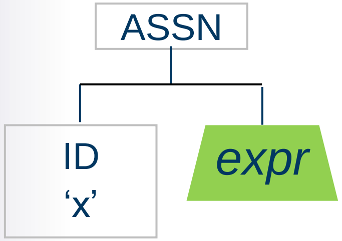
→

```
LOADG  4
LOADC  9
SUB
STOREG  3
```

code to evaluate "`n-9`" (LOADG 4, LOADC 9, SUB)

Example: Fun → SVM code templates *(4)*

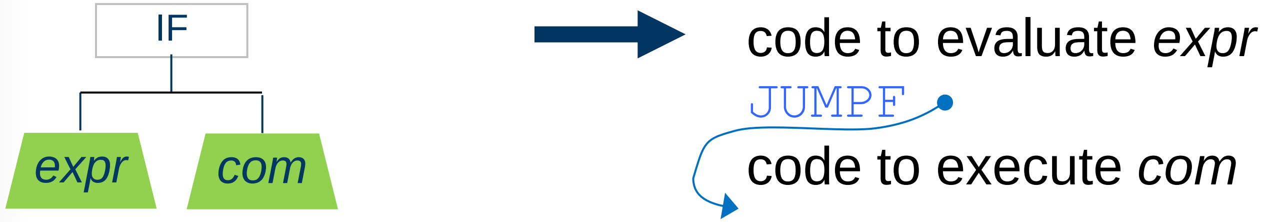
- Code generator action for assignment-command:



walk *expr* generating code;
lookup ‘x’ and retrieve its address *d*;
emit instruction “`STOREG` *d*” (if x is global)
or “`STOREL` *d*” (if x is local)

Example: Fun → SVM code templates *(5)*

- Code template for if-command:

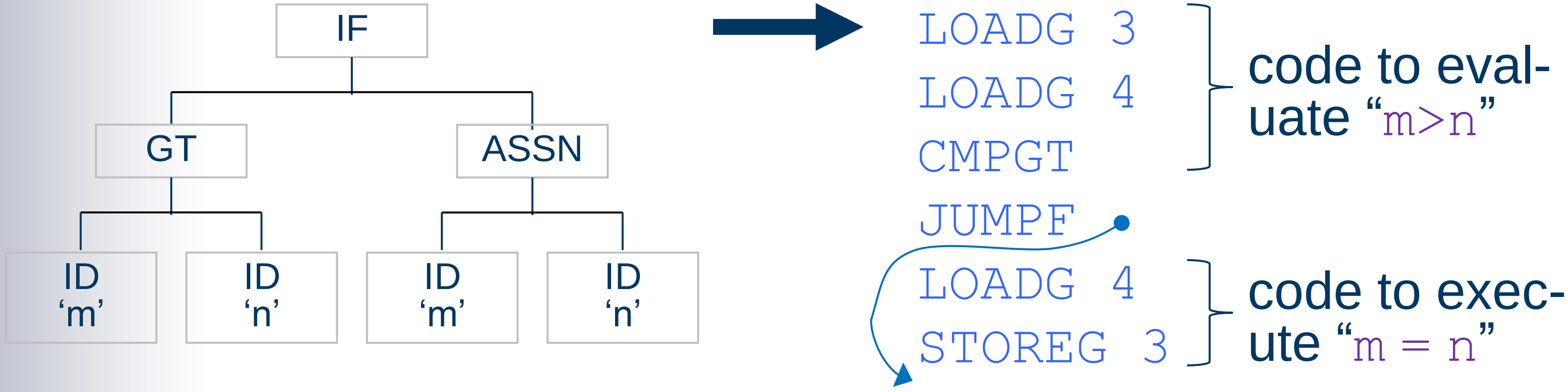


code to evaluate *expr*

```
JUMPF
```

code to execute *com*

- E.g., code to execute “`if m>n: m = n.`”:



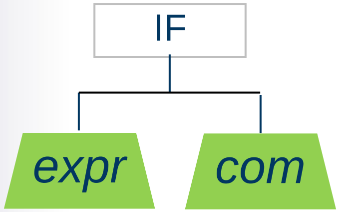
```
LOADG  3
LOADG  4
CMPGT
JUMPF
LOADG  4
STOREG  3
```

code to evaluate “`m>n`”

code to execute “`m = n`”

Example: Fun → SVM code templates *(6)*

- Code generator action for if-command:



walk *expr*, generating code;
emit instruction "`JUMPF 0`";
walk *com*, generating code;
patch the correct address into
 the above `JUMPF` instruction

Code generation with ANTLR

- *Recall:* In ANTLR we can write a “tree grammar” which describes the ASTs. Each rule in the tree grammar is a pattern match for part of the AST. From the tree grammar, ANTLR generates a depth-first left-to-right tree walker.
- To build a code generator, we enhance the tree grammar with actions to perform address allocation and code selection.
- ANTLR inserts those actions into the tree walker.

Case study: Fun tree grammar with code generation actions *(1)*

- Fun tree grammar with actions *(outline)*:

```
tree grammar FunEncoder;

options {
   tokenVocab = Fun;
   …;
}

@members {
   private SVM obj = new SVM();
   private int varaddr = 0;
   private SymbolTable<Address> addrTable;
   …
}
```

Creates an instance of the SVM. The code generator will emit instructions directly into its code store.

Case study: Fun tree grammar with code generation actions *(2)*

- Fun tree grammar with actions *(continued)*:

```
expr
      :  NUM
                        { let n = value of the numeral;
                          emit "LOADC n"; }
      |  ID
                        { lookup the identifier in
                            addrTable and
                            retrieve its address d;
                          emit "LOADG d" or
                            "LOADL d"; }
      |  …
```

Case study: Fun tree grammar with code generation actions *(3)*

- Fun tree grammar with actions *(continued)*:

```
|  ^(EQ
     expr           //generate code for left expr
     expr           //generate code for right expr
    )               {  emit "CMPEQ";  }
|  ^(PLUS
     expr           //generate code for left expr
     expr           //generate code for right expr
    )               {  emit "ADD";  }
|  ^(NOT
     expr           //generate code for expr
    )               {  emit "INV";  }
|  …
;
```

Case study: Fun tree grammar with code generation actions *(4)*

- Fun tree grammar with actions *(continued)*:

```
com
        :  ^(ASSN
             ID
             expr          //generate code for expr
            )              {  lookup the identifier in
                                 addrTable and
                                 retrieve its address d;
                              emit “STOREG d” or
                                 “STOREL d”;  }
        |  …
```

Case study: Fun tree grammar with code generation actions *(5)*

- Fun tree grammar with actions *(continued)*:

```
|  ^(IF
      expr          //generate code for expr
                    {  emit “JUMPF 0”
                        (incomplete);  }
      com           //generate code for com
     )              {  let c = next instruction address;
                       patch c into the incomplete
                        “JUMPF” instruction;  }
|  ^(SEQ
      com*          //generate code for com*
     )
|  …
;
```

Case study: Fun tree grammar with code generation actions *(6)*

- Fun tree grammar with actions *(continued)*:

```
var_decl
        :   ^(VAR
              type
              ID
              expr          //generate code for expr
             )              {  put the identifier into addr-
                                 Table along with varaddr;
                               increment varaddr; }
        ;

type
        :   BOOL
        |   INT
        ;
```

Case study: Fun tree grammar with code generation actions *(7)*

- Fun tree grammar with actions *(continued)*:

```
prog                returns [SVM objprog]
    :  ^(PROG
                    {  put ‘read’ and ‘write’ into
                        addrTable;  }
          var_decl*  //generate code for var_decl*
                    {  emit “CALL 0” (incomplete);
                       emit “HALT”;  }
          proc_decl+//generate code for proc_decl*
         )          {  lookup ‘main’ in addrTable
                         and retrieve its address c;
                       patch c into the incomplete
                         CALL instruction;
                       set $objprog to obj;  }
    ;
```

Case study: Fun compiler *(1)*

- Put the above tree grammar in a file named `FunEncoder.g`.
- Feed this as input to ANTLR:

```
…$ java org.antlr.Tool FunEncoder.g
```

- ANTLR generates a class `FunEncoder` containing methods that walk the AST and perform the code generation actions.

Case study: Fun compiler *(2)*

- Program to run the Fun syntactic analyser and code generator:

```
public class FunRun {
  public static void main (String[] args) {
     // Syntactic analysis:
     …
     CommonTree ast = (CommonTree)
        parser.prog().getTree();
     // Code generation:
     FunEncoder encoder = new FunEncoder(
           new CommonTreeNodeStream(ast));
     SVM objcode = encoder.prog();
  }
}
```

Representing addresses

- The code generator must distinguish between three kinds of addresses:
    - A **code address** refers to an instruction within the space allocated to the object code.
    - A **global address** refers to a location within the space allocated to global variables.
    - A **local address** refers to a location within a space allocated to a group of local variables.

Case study: implementation of Fun addresses

- Implementation in Java:

```
public class Address {

   public static final int
      CODE = 0, GLOBAL = 1, LOCAL = 2;

   public int offset;
   public int locale; // CODE, GLOBAL, or LOCAL

   public Address (int off, int loc) {
      offset = off;  locale = loc;
   }

}
```

Handling jumps *(1)*

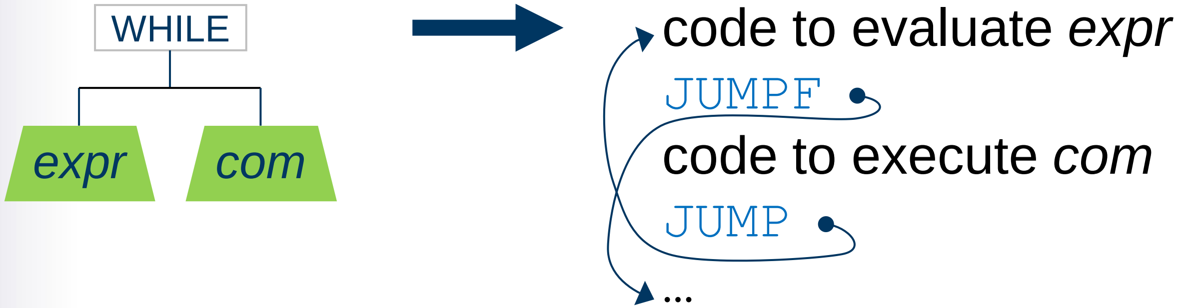
- The code generator **emits** instructions one by one. When an instruction is emitted, it is added to the end of the object code.
- At the destination of a jump instruction, the code generator must note the destination address and incorporate it into the jump instruction.

Handling jumps *(2)*

- For a *backward* jump, the destination address is already known when the jump instruction is emitted.
- For a *forward* jump, the destination address is unknown when the jump instruction is emitted. Solution:
  - Emit an incomplete jump instruction (with 0 in its address field), and note its address.
  - When the destination address becomes known later, **patch** that address into the jump instruction.

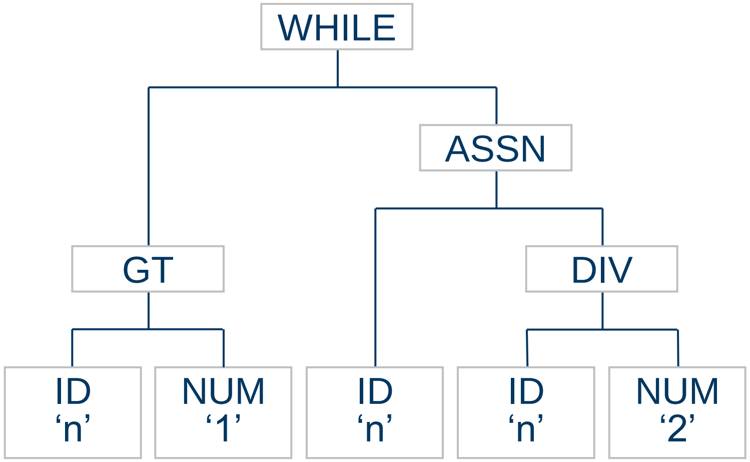
Example: Fun while-command *(1)*

- Code template for while-command:



Example: Fun while-command *(2)*

- AST of while-command “`while n>1: n=n/2.`”:



- Assume that the while-command’s object code will start at address 7.

Example: Fun while-command *(3)*

- Code generator action (animated):

```
note the current instruction address $c_1$
walk expr, generating code
note the current instruction address $c_2$
emit "JUMPF 0"
walk com, generating code
emit "JUMP $c_1$"
note the current instruction address $c_3$
patch $c_3$ into the jump at $c_2$
```

c_1 $\boxed{7}$ c_2 $\boxed{14}$ c_3 $\boxed{30}$

```
 0: …
    …
 7: LOADG  0
10: LOADC  1
13: COMPGT
14: JUMPF  30
17: LOADG  0
20: LOADC  2
23: DIV
24: STOREG 0
27: JUMP  7
30:
```