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 / syntax tree) 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).

University of Glasgow

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)*.

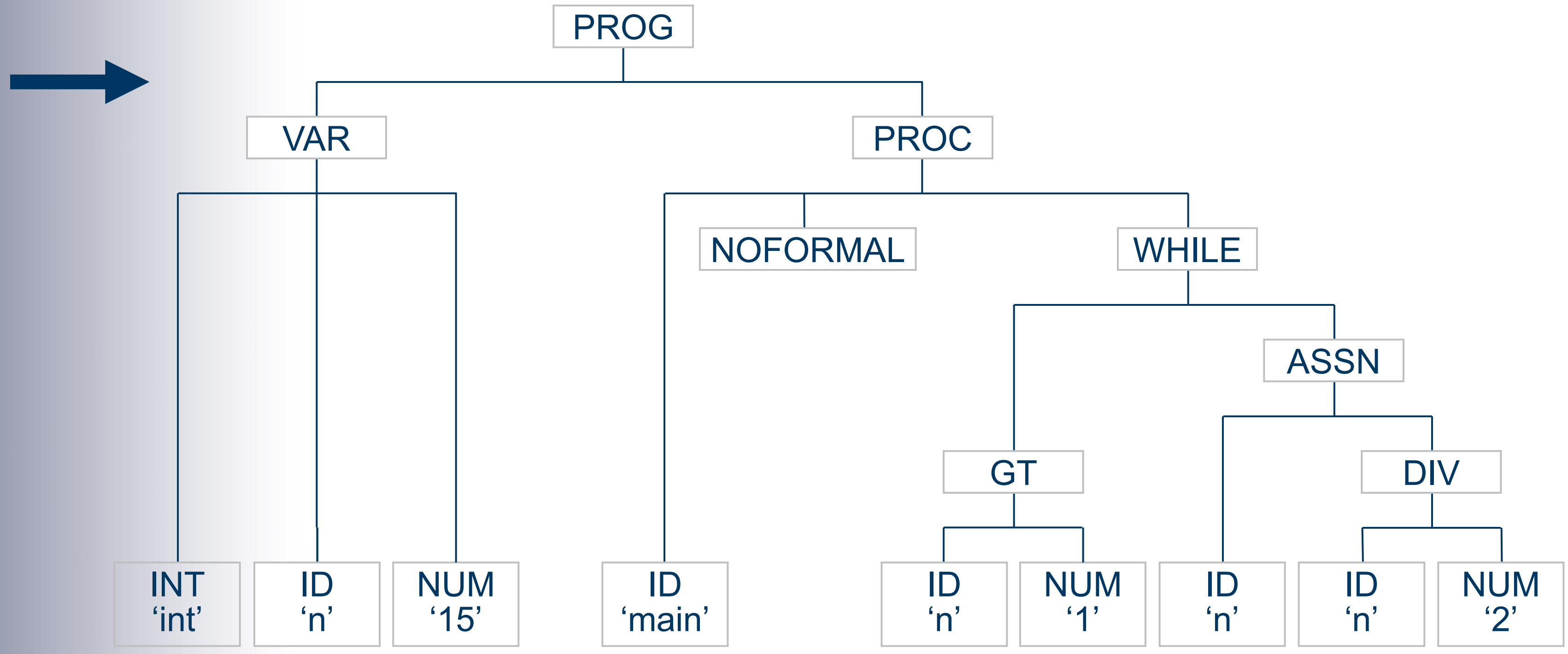
Example: Fun compilation *(1)*

- 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()` (addresses 7–30)

code to evaluate “`n>1`” (addresses 7–13)

code to execute “`n=n/2`” (addresses 17–24)

code to execute “`while n>1: n=n/2.`” (addresses 7–27)

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.:

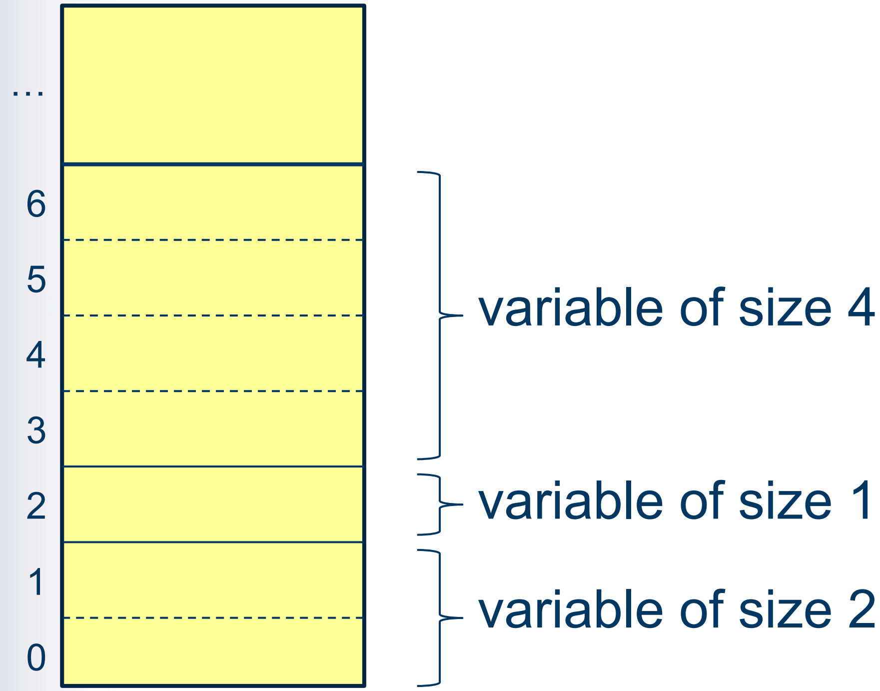
‘x’	0 (global)	variables
‘y’	2 (global)	
‘fac’	7 (code)	procedures
‘main’	30 (code)	

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.

University of Glasgow

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.

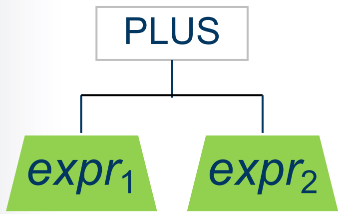
University of Glasgow

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.

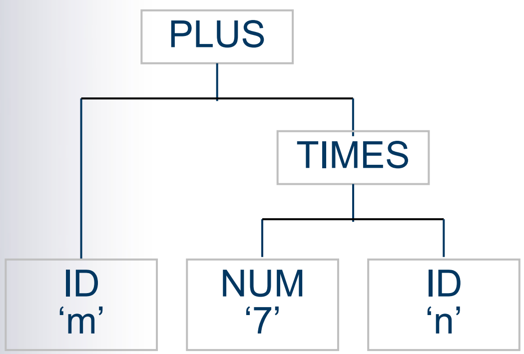
Example: Fun → SVM code templates *(1)*

- 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
```

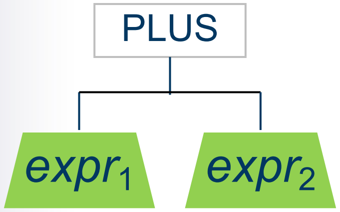
code to evaluate “`m`” (LOADG 3)

code to evaluate “`7*n`” (LOADC 7, LOADG 4, MULT)

- 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:



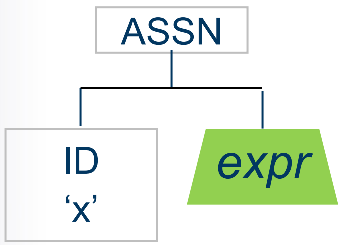
walk $expr_1$ generating code;
walk $expr_2$ generating code;
emit instruction "`ADD`"

- 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)*

赋值操作

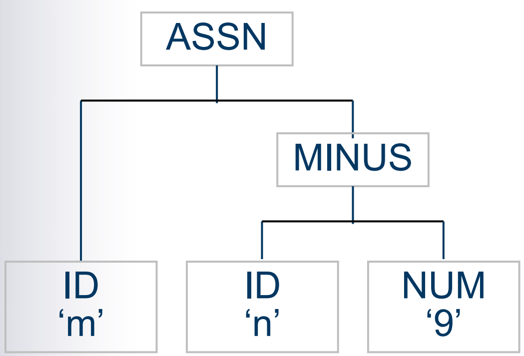
- Code template for assignment-command:



→ 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`":



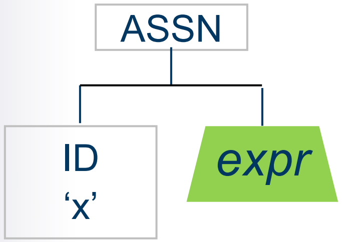
→

```
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)

Handling jumps *(1)*

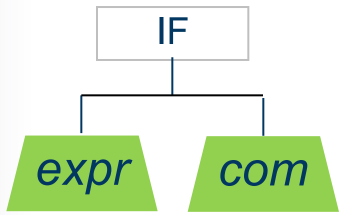
- The code generator **emits** instructions one by one. When an instruction is emitted, it is added to the end of the object code.
- At the destination of a jump instruction, the code generator must note the destination address and incorporate it into the jump instruction.

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 → SVM code templates *(5)*

- Code template for if-command:

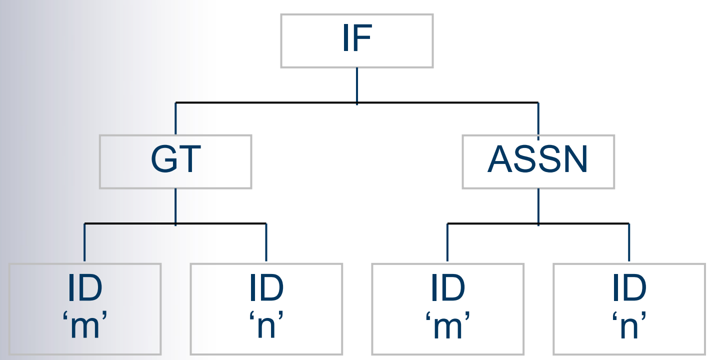


code to evaluate *expr*

```
JUMPF
```

code to execute *com*

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

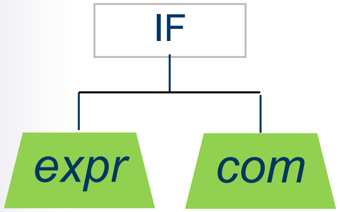



```
LOADG 3
LOADG 4
CMPGT
JUMPF
LOADG 4
STOREG 3
```

- `LOADG 3`, `LOADG 4`, `CMPGT`: code to evaluate "`m>n`"
- `LOADG 4`, `STOREG 3`: code to execute "`m = n`"

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

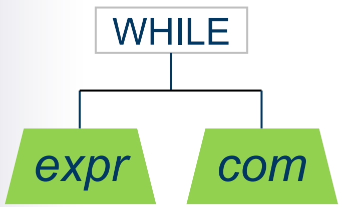
- Code generator action for if-command:



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

Example: Fun while-command *(1)*

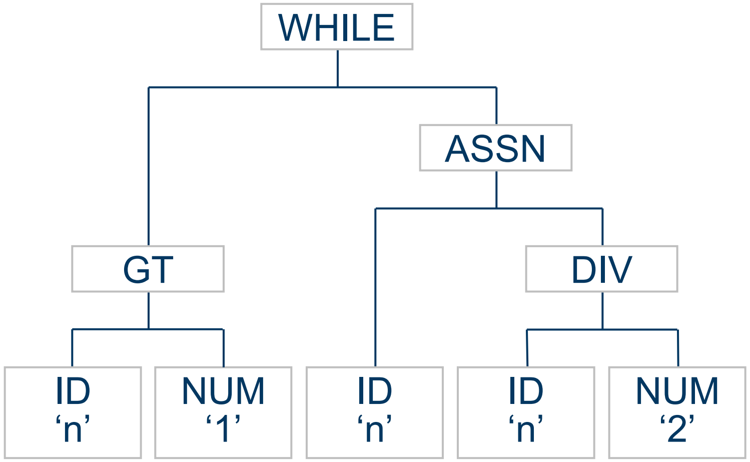
- Code template for while-command:



code to evaluate *expr*
`JUMPF`
code to execute *com*
`JUMP`
...

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:
```

Code generation with ANTLR

- The code generator is a visitor, with a similar structure to the contextual analysis visitor.
- For each type of syntax tree node, the visit method implements the code generation action.

```
class FunEncoderVisitor extends AbstractParseTreeVisitor<Void>
                        implements FunVisitor<Void> {
    SVM obj = new SVM();
    int globalvaraddr = 0;
    int localvaraddr = 0;
    int currentLocale = Address.GLOBAL;

    SymbolTable<Address> addrTable = new SymbolTable<Address>();

...
}
```

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

Case study: Fun code generation *(1)*

```
Void visitNum(FunParser.NumContext ctx) {
   int value = Integer.parseInt(ctx.NUM().getText());
   obj.emit12(SVM.LOADC, value); // emit12 means 1 opcode +
   return null;                  // 2 byte operand
}

Void visitId(FunParser.IdContext ctx) {
   String id = ctx.ID().getText();
   Address varaddr = addrTable.get(id);
   switch (varaddr.locale) {
      case Address.GLOBAL:
         obj.emit12(SVM.LOADG,varaddr.offset);
         break;
      case Address.LOCAL:
         obj.emit12(SVM.LOADL,varaddr.offset);
   }
   return null;
}
```

Case study: Fun code generation *(2)*

```
// expr : e1=sec_expr (op=(EQ | LT | GT) e2=sec_expr)?
Void visitExpr(FunParser.ExprContext ctx) {
   visit(ctx.e1); // Generate code to evaluate e1
   if (ctx.e2 != null) {
      visit(ctx.e2); // Generate code to evaluate e2
      switch (ctx.op.getType()) { // Generate an
         case FunParser.EQ:       // instruction for the
            obj.emit1(SVM.CMPEQ); // operator.
            break;
         case FunParser.LT:
            obj.emit1(SVM.CMPLT); // emit1 means 1 opcode
            break;
         case FunParser.GT:
            obj.emit1(SVM.CMPGT);
            break;
      }
   }
   return null;
}
```

Case study: Fun code generation *(3)*

```
// com : ID ASSN expr # assn

Void visitAssn(FunParser.AssnContext ctx) {
    visit(ctx.expr()); // Generate code to evaluate expr
    String id = ctx.ID().getText();
    // Find the address of the variable.
    // This always succeeds, because we assume that the
    // program has been through the contextual analyser.
    Address varaddr = addrTable.get(id);
    switch (varaddr.locale) {
        case Address.GLOBAL:
            obj.emit12(SVM.STOREG,varaddr.offset);
            break;
        case Address.LOCAL:
            obj.emit12(SVM.STOREL,varaddr.offset);
        }
    return null;
}
```

Case study: Fun code generation *(4)*

```
// IF expr COLON c1=seq_com (DOT | ELSE COLON c2=seq_com DOT) # if

Void visitIf(FunParser.IfContext ctx) {
   visit(ctx.expr());
   int condaddr = obj.currentOffset();
   obj.emit12(SVM.JUMPF, 0); // This has to be patched later.
   if (ctx.c2 == null) { // IF without ELSE
   visit(ctx.c1);
   int exitaddr = obj.currentOffset();
   obj.patch12(condaddr, exitaddr);
   }
   else {                // IF ... ELSE
      visit(ctx.c1);
      int jumpaddr = obj.currentOffset();
      obj.emit12(SVM.JUMP, 0); // This also has to be patched.
      int elseaddr = obj.currentOffset();
      obj.patch12(condaddr, elseaddr);
      visit(ctx.c2);
      int exitaddr = obj.currentOffset();
      obj.patch12(jumpaddr, exitaddr);
   }
   return null;
}
```

Case study: Fun code generation *(5)*

```
// var_decl : type ID ASSN expr  # var

Void visitVar(FunParser.VarContext ctx) {
   visit(ctx.expr());
   String id = ctx.ID().getText();
   switch (currentLocale) {
      // Adding the variable to the address table always succeeds,
      // because we assume we have done contextual analysis, so it
      // is guaranteed to be a new variable name.
      case Address.LOCAL:
         addrTable.put(id, new Address(localvaraddr++,
                                       Address.LOCAL));
         break;
      case Address.GLOBAL:
         addrTable.put(id, new Address(globalvaraddr++,
                                       Address.GLOBAL));
   }
   return null;
}
```

全局变量

University of Glasgow

Storage organization

- Each variable occupies storage space throughout its lifetime. That storage space must be:
  - allocated at the start of the variable's lifetime
  - deallocated at the end of the variable's lifetime.
- Assumptions:
  - The PL is statically typed, so every variable's type is known to the compiler.
  - All variables of the same type occupy the same amount of storage space.

University of Glasgow

Storage for global and local variables *(1)*

- Recall: A *global variable*'s lifetime is the program's entire run-time.
- For global variables, the compiler allocates **fixed** storage space.
- Recall: A *local variable*'s lifetime is an activation of the block in which the variable is declared. The lifetimes of local variables are nested.
- For local variables, the compiler allocates storage space on a **stack**.

Storage for global and local variables *(2)*

- At any given time, the stack contains one or more **activation frames**:
  - The frame at the base of the stack contains the global variables.
  - For each *active* procedure *P*, there is a frame containing *P*'s local variables.
- A frame for procedure *P* is:
  - pushed on to the stack when *P* is called
  - popped off the stack when *P* returns.

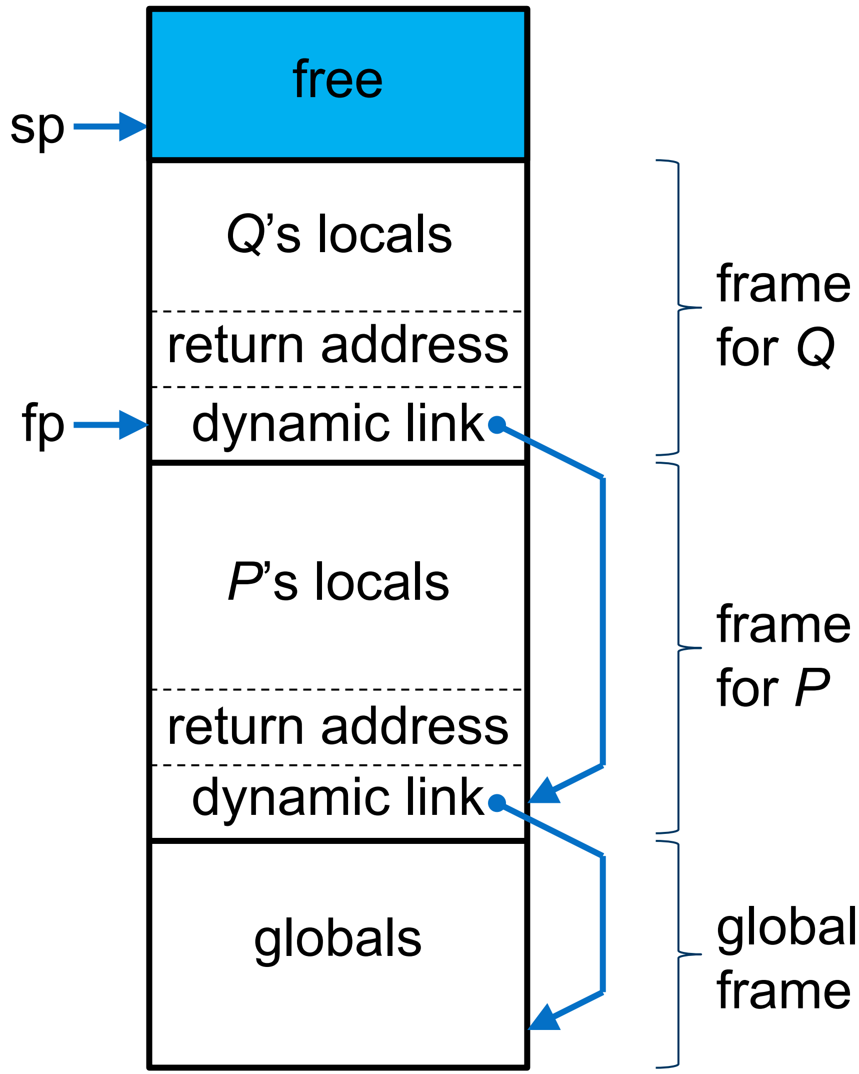
An active procedure is one that has been called but not yet returned.

Storage for global and local variables *(3)*

- The compiler fixes the size and layout of each frame.
- The offset of each global/local variable (relative to the base of the frame) is known to the compiler.

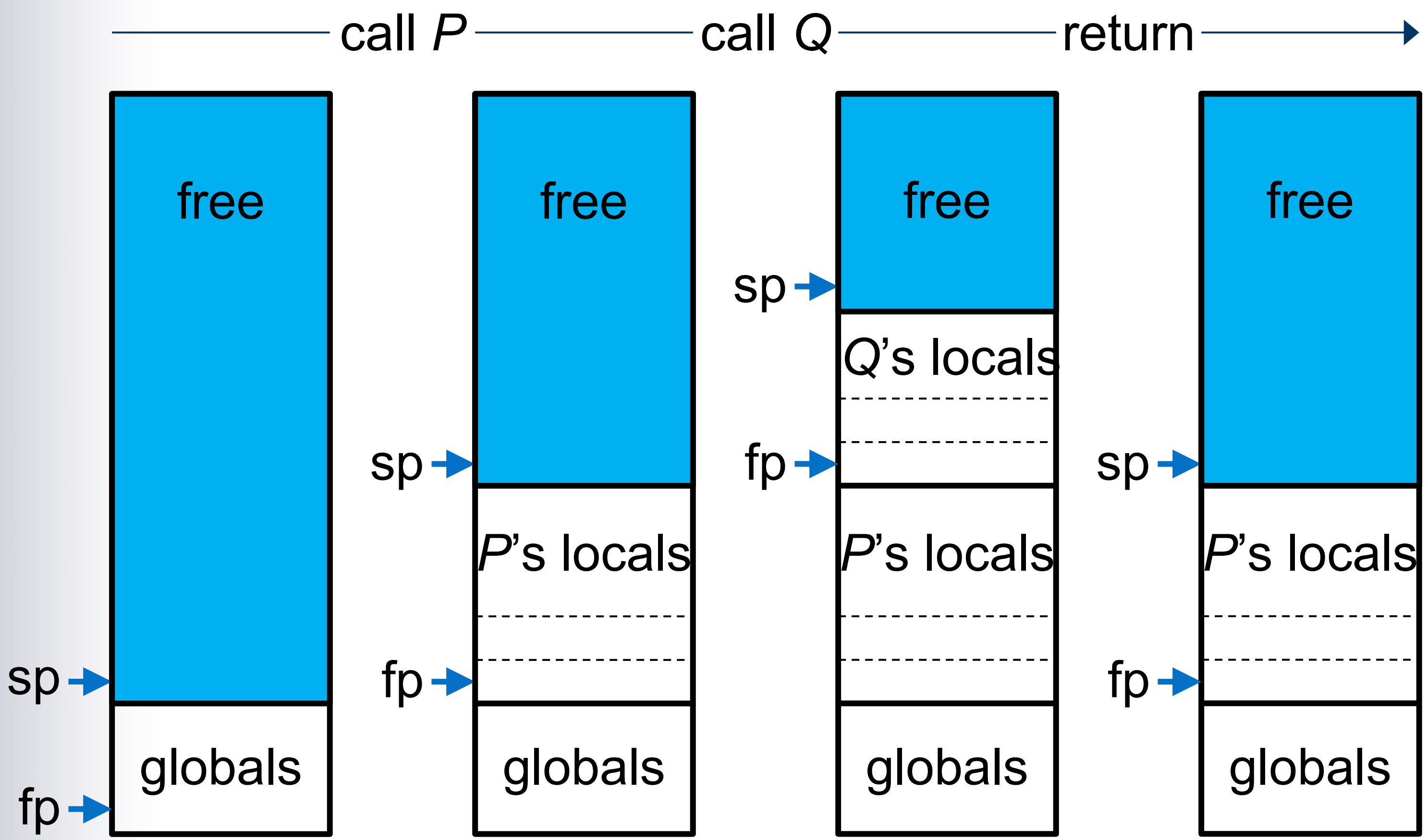
Example: storage for global and local variables in SVM *(1)*

- SVM data store when the main program has called *P*, and *P* has called *Q*:
- **sp** (stack pointer) points to the first free cell above the top of the stack.
- **fp** (frame pointer) points to the first cell of the topmost frame.



Example: storage for global and local variables in SVM *(2)*

- Effect of calls and returns:



Case study: Fun code generation *(6)*

```
// proc_decl : PROC ID LPAR formal_decl RPAR COLON
               var_decl* seq_com DOT   # proc

Void visitProc(FunParser.ProcContext ctx) {
   String id = ctx.ID().getText();
   Address procaddr = new Address(obj.currentOffset(), Address.CODE);
   addrTable.put(id, procaddr); // The address of the code for proc
   addrTable.enterLocalScope();
   currentLocale = Address.LOCAL;
   localvaraddr = 2;
   // ... allows 2 words for link data (part of the stack frame)
   FunParser.Formal_declContext fd = ctx.formal_decl();
   if (fd != null)
      visit(fd);
   List<FunParser.Var_declContext> var_decl = ctx.var_decl();
   for (FunParser.Var_declContext vd : var_decl)
      visit(vd);
   visit(ctx.seq_com());
   obj.emit11(SVM.RETURN, 0); // 0 because there is no result
   addrTable.exitLocalScope();
   currentLocale = Address.GLOBAL;
   return null;
}
```

Case study: Fun code generation *(6)*

```
Void visitFormal(FunParser.FormalContext ctx) {
   FunParser.TypeContext tc = ctx.type();
   if (tc != null) {
      String id = ctx.ID().getText();
      // A parameter is like a local variable
      addrTable.put(id, new Address(localvaraddr++, Address.LOCAL));
      // Copy arguments (actual parameters) into the stack frame
      obj.emit11(SVM.COPYARG, 1);
   }
   return null;
}
```

Case study: Fun code generation *(7)*

```
// program : var_decl* proc_decl+ EOF  # prog

Void visitProg(FunParser.ProgContext ctx) {
   predefine(); // Add read and write to the address table.
   List<FunParser.Var_declContext> var_decl = ctx.var_decl();
   for (FunParser.Var_declContext vd : var_decl)
      visit(vd);
   int calladdr = obj.currentOffset();
   obj.emit12(SVM.CALL, 0); // Call the main procedure – patch later
   obj.emit1(SVM.HALT);
   List<FunParser.Proc_declContext> proc_decl = ctx.proc_decl();
   for (FunParser.Proc_declContext pd : proc_decl)
      visit(pd);
   int mainaddr = addrTable.get("main").offset;
   obj.patch12(calladdr, mainaddr);
   return null;
}
```

Case study: Fun compiler

- FunRun contains the following definition:

```
SVM compile (String filename)
   throws Exception {
   // Compile a Fun source program to SVM code.
   FunLexer lexer = new FunLexer(
      CharStreams.fromFileName(filename));
   CommonTokenStream tokens =
      new CommonTokenStream(lexer);
   ParseTree ast = syntacticAnalyse(tokens);
   contextualAnalyse(ast,tokens);
   SVM objprog = codeGenerate(ast);
   return objprog;
}
```

University of Glasgow

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;
   }

}
```