Chapter 3: Functions

3.1 Introduction

The previous chapter assumed that all of your Bali code would be written inside a sole main function. But, as you have learned from previous programming courses, modularizing code produces clearer, more reusable code. This chapter introduces translating ***functions*** to Samcode. If all you want are the templates for Samcode and functions, jump ahead to Section 4.

3.2 Background

This section provides some reminders and some terminology with which you might not be familiar.

3.2.1 Functions

Although you are well acquainted with functions from your programming courses, recall that assembly programming requires "low-level" thinking. So, first you should review some quick higher-level concepts, below:

- **Purpose**: Functions encapsulate procedural information. Actually, there are terminology differences for *subroutine*, *function*, *procedure*, and *method*. We will stick to ***function***. For specific definitions, investigate http://foldoc.doc.ic.ac.uk/foldoc/index.html.
- **Structure**: A function has a ***header*** (or ***signature***) and ***body*** (or, ***primary block***).
- **Control flow and call order**: When encountering a function call, the code that contains the function call (as an expression and/or statement) is the ***caller***. The function being called is the ***callee***. We will use this terminology when writing Samcode for a function. Note that the function body is associated with the callee, and the function signature is associated with the function header.
- **Scope**: The grammar we have provided follows a mostly standard approach in which a programmer can declare local variables "inside" a function. So, each function can reuse variable names without interfering with variable names elsewhere in the code. A ***global variable*** is visible to the entire program and should be treated with care. Note that Java does not have a global variable, though `static` fields roughly provide a similar functionality.
- **Parameters**: Although it may seem obvious, remember that the caller passes ***actual arguments*** to a function's ***formal parameters***, which are declared by the callee. In general, we will use the condensed term ***parameter***. You will treat parameters differently than local variables (callee's scope) because actual arguments (from the caller) are passed to the parameters (in the callee).
- **Recursion**: A ***recursive function*** can call itself. Why the reminder? You need to treat each function call as a separate invocation, as if you have invoked another function that just happens to use the same code.
- **More?** There are issues of ***overloading*** and ***overriding***. Recall that an overloaded function uses the same as another function, whereas an overriden function redefines a function for a particular subclass.

In the next section, we review the notion of the ***Stack***, which provides a structure with which functions can be implemented with the above features.

3.2.2 The Stack

The ***Stack*** (or, ***call stack***) is a data structure in which information *about* a function and its data is stored. We usually model the stack as a collection of ***frames***, which are subdivisions of the stack, as shown in Figure 1.1. Each time you call a function, another frame is added to the *top* of the stack, resulting in bottom-up construction.

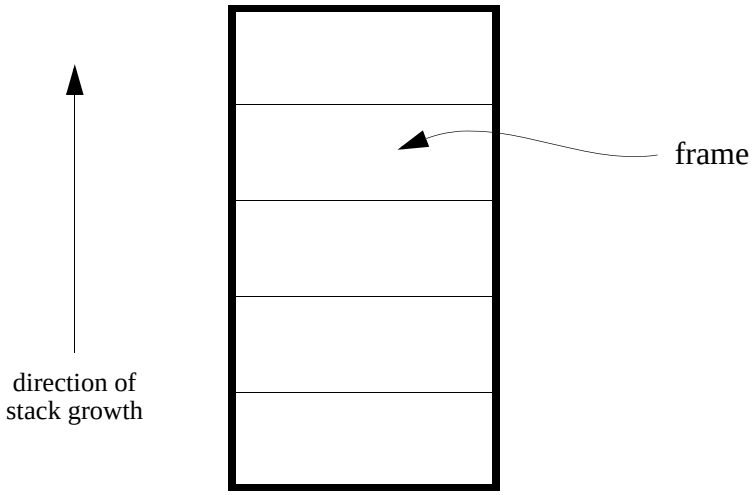

Figure 1.1: The Stack

For now, you can split each frame into a collection of ***cells***, each having a unique address, as shown in Figure 1.2. For SaM, the first cell has address zero and works upwards using integers. Each cell stores a value. Refer to Chapter 1 for specific cell types and values.

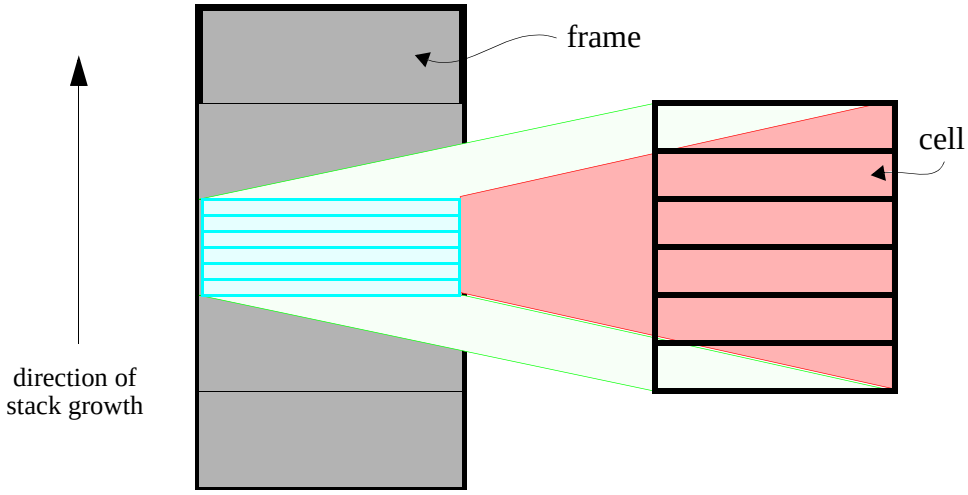

Figure 1.2: Frames and Cells

Each frame contains an assembly translation of the data associated with a function's code and operation, all of which stored in cells: a return variable, parameters, administrative information,

local variables, and data used by expressions. This assignment will take you step-by-step through the construction process.

3.2.3 Registers (Revisited)

Recall from Chapter 1 that SaM models not just memory needed for functions and classes, but smaller portions of a CPU's main memory. These small areas are ***registers***, which usually hold administrative information. For SaM, the registers store integers with which you can conveniently consider as memory addresses. Since a register can store an integer address, you can think of the register as ***pointing*** to the cell/memory location with the address in that register's memory. For this assignment, we use three registers, which were introduced in Chapter 1:

- ***SP*** (Stack Pointer): address of next free cell on stack.
- ***FBR*** (Frame Based Register): address of current frame, which is discussed in great detail in this assignment.
- ***PC*** (Program Counter): Label of current Samcode instruction to process. SaM stores all loaded Samcode instructions from your input SaM file in a data structure, which has integer address. The PC is also reviewed in more detail later in this assignment.

In general, the administrative portions of each frame will use other registers to help keep track of the current frame and control from caller to callee back to caller.

3.2.4 Caller and Callee

We have just a bit more to explain before delving into Samcode. The construction and destruction of each frame is a *collaborative* process between the caller and callee of a function. Refer to Figure 1.3. Since the caller has to call the callee ("-ers" precede "-ees"), the caller fills the first part of each frame. Then, the callee takes over and fills in the rest. As a function finishes, the callee removes its data from the frame, followed by the caller doing the same until the frame is utterly destroyed.

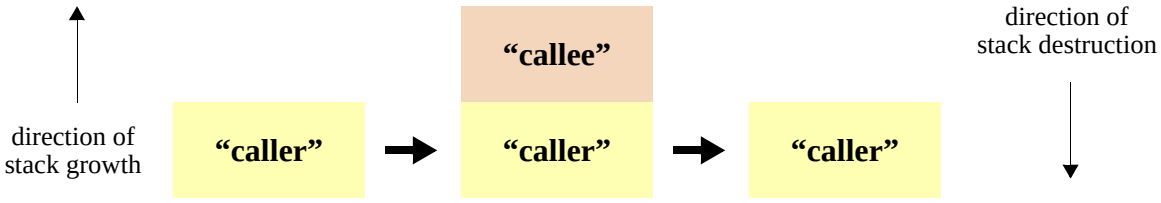

Figure 1.3: Caller/Callee Collaboration

3.3 Frame Structure

This section explains the model that we use to translate Bali functions into Samcode. We will start with the bottom of the frame (the caller cells) and work our way to the top (the callee cells). So, there will be a focus on constructing a frame. We weave the explanation of frame destruction into both the caller and callee development because of the collaborative process.

3.3.1 Return Value

Bali functions always return something. So, we need a consistent place in each frame to store the ***return value*** (sometimes interchanged with ***return variable*** and ***rv***) of each function. In fact, the expression in caller that calls the callee is replaced by the return value of the callee. Refer to Figure 1.4. Since the caller will be using the return value and frames have a bottom-up construction, the return variable is stored at the bottom of each frame. The last step of destroying a frame is popping the return value sitting on the top of the stack as part of the caller's code. In turn, that caller is the callee of some previous caller, and so forth.

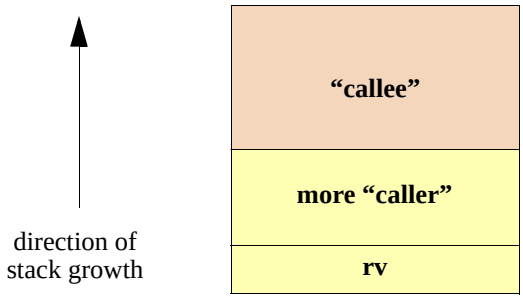

Figure 1.4: Return Variable

For Samcode, upon encountering a function to execute, the caller (which is the current function) must first allocate space for the return variable of the callee. As you have seen in previous chapters, you would use `ADDSP 1` or `PUSHIMM value`. How should you choose?

- When your language has default returns, use `PUSHIMM`, as in `PUSHIMM 0`.
- When your language has no specific default, use `ADDSP`.

Also, when compiling Bali to Samcode, you will find it very handy to label the first instruction of a function in your Samcode with the Bali function name. For instance, given the following generic Bali code,

```
int f ( ) { g ( 2 ) ; }
int g ( int x ) { /* whatever */ }
```

your compiler would produce the following Samcode for function `g`:

```
f:
// possibly other code in f
ADDPSP 1 // start building g's frame
// more Samcode coming!
```

Why use label `f`?

- We provide labels so that we can jump to a specific function elsewhere in Samcode.
- We use f here, because `f` is the caller of `g`, and `g(2)` is an expression in `f`'s frame!

So, where's `g`'s frame? You will see a label for `g` when we reach the callee portion of the frame. Note that you also place the label and the first instruction on the same line as `f: ADDSP 1`.

3.3.2 Parameters

After allocating a return variable, there are some choices we could have made for our model. We choose to push parameters (or just ***params***) onto the frame before handling the remaining caller

code, which is the "administrative information" to which we previously alluded. Pushing the actual arguments gets the caller's Samcode out of the way. Figure 1.5 summarizes the structure of the caller portion stack that we have developed, so far.

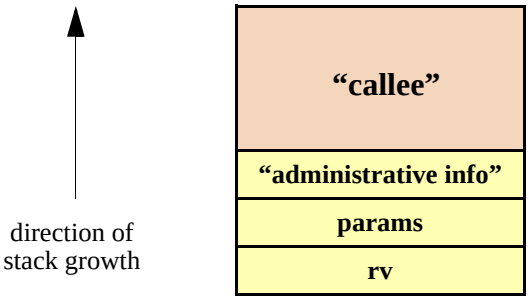

Figure 1.5: Parameters

When pushing the parameters, your caller must resolve each expression first! So, code such as

```
g ( ( 1 + 2 ) ) ;
```

involves writing Samcode for `( 1 + 2 )`. The *result* of each of expression, the actual argument, is pushed onto the frame for function `g`, as follows:

```
f:
ADDSP 1 // rv of g
PUSHIMM 1
PUSHIMM 2
ADD // pop 1 and 2, push param value 3
// more Samcode
```

If your function has no formal parameters, you would simply not have any Samcode nor cells for this portion of the frame.

3.3.3 Frame Based Register

To distinguish a specific frame from other frames, we will use the ***frame based register*** (***FBR***). The FBR stores the address of the *current* frame (and thus, function) being processed, as shown below in Figure 1.6. The FBR points to a location in each frame because the FBR stores an integer that is a cell address, as discussed in Section 2.3. Do not be tempted to think of the FBR as pointing to the bottom of the frame or even holding a simple count of the current frame! Why not use these values? As you will discover, the FBR provides a convenient way to access a function's local variables and parameters, which are stored *relative* to an FBR.

To get a feel for the model, recall from Chapters 1 and 2 that we used FBR's default initial value of 0. Assume, instead, that the current FBR is address 12345. When we say `STOREOFF 2`, we really have placed the top of the stack in address 12345+2. Perhaps the next function's frame has an FBR of 13579. When executing `STOREOFF 4`, you are now accessing that a variable of the other function. So, we now have the ability to store and access a variable within the scope of a particular function!

3.3.4 Saved FBR

So, where exactly should the FBR point inside a function's frame? We will set the FBR to point to the cell *just after* all the parameters. Since SaM keeps track of the stack pointer from each

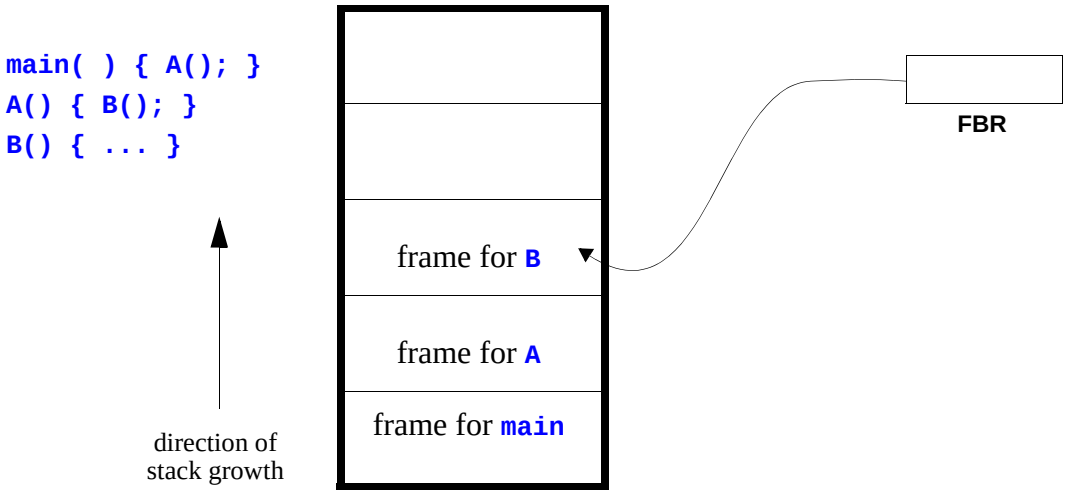

Figure 1.6: FBR Model

allocation of stack memory, SaM knows the cell address above the parameters. In fact, this cell will serve double duty, as discussed below.

Suppose SaM is finishing processing code of the following form:

```
f ( ) { g ( 1 ) ; }
```

When the current frame (the callee, **g**) finishes executing, control needs to return to the caller **f**. The caller needs to remove the parameter and use the return value. But returning to the caller means using **f**'s frame, which is below **g**'s frame. To restore the FBR to point inside **f**'s frame, SaM needs to remember **f**'s address. Given each frame's ability to store values, we can simply store **f**'s address inside **g**'s frame! So, the cell just above the parameters is called the ***saved FBR***, which stores the *previous* frame's FBR value.

Fortunately, the SaM instructions make the saving of the caller's FBR and setting of the callee's FBR fairly easy. The **LINK** instruction takes the current FBR (the FBR of the caller) and pushes it onto the stack. Then, **LINK** takes the address of the just-saved FBR and stores it in the FBR. Thus, the FBR always points to a saved FBR cell. Isn't that convenient?

For example, when calling **g(1)**, you would have the following Samcode:

```
f:
ADDSP 1     // return value for g
PUSHIMM 1   // push parameter 1 for g(1)
LINK        // push f's FBR and set current FBR to g
// more code
```

However, you are not finished learning about the frame!

3.3.5 Saved PC

You have almost finished the caller portion…just one more cell to allocate! So far, we have focused on the caller's Samcode, which is code used to set up the frame. We need a way to execute the actual callee code, which is technically *inside* the body of the callee. As with other control structures, we need to jump to the callee's code. Likewise, we will need to remember a "place" to return, which in this case, is the caller. So, the next address on the stack is reserved to restore control to the caller.

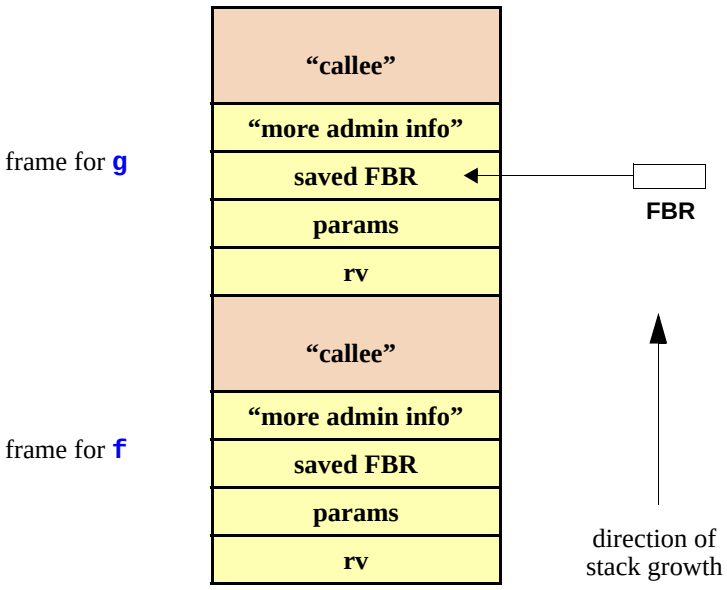

Figure 1.7: Saved FBR

What value is stored in this slot? Recall that the ***program counter*** (PC) is a register that keeps track of the current instruction. To help identify an instruction, SaM uses ***instruction labels***, as shown in Figure 1.8.

```
int main ( ) {
   int x , y;
   x = 10 ;
   y = 20 ;
   return ( x + y ) ;
}
```

```
ADDSP 3
PUSHIMM 10
STOREOFF 1
PUSHIMM 20
STOREOFF 2
PUSHOFF 1
PUSHOFF 2
ADD
STOREOFF 0
ADDSP -2
STOP
```

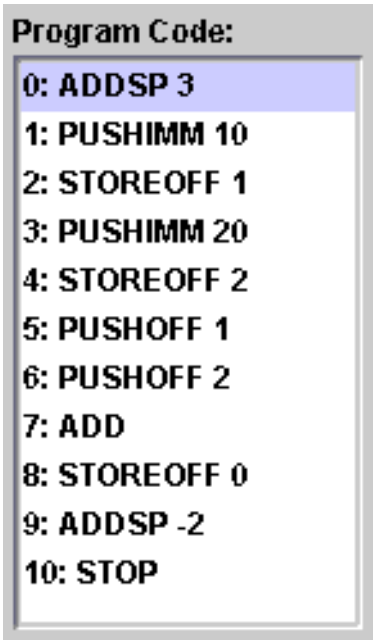

Figure 1.8: SaM Labels

Starting with 0, SaM internally numbers each instruction with an integer. Don't confuse the Samcode label with a cell address! The instruction labels are a completely different numbering scheme, which SaM uses to keep track of each instruction inside the SaM "engine." But, you too can use the labels, as you do with `JUMP` and `JUMPC`. For functions, you use `JSR label` (jump to subroutine `label`). Note that SaM automatically converts your labels into integers.

Before giving the details on **JSR**, note that the general pattern in Samcode for **f** calling **g** is

```
f:
caller Samcode
JSR g
caller Samcode
g:
callee Samcode
return to Samcode after the "JSR g"
```

The **JSR** forces SaM to jump to the **g** label, which starts the callee code. At the end of the callee code, SaM needs to finish the caller's Samcode. The remaining Samcode is a list of instructions that are written just below the **JSR g**. Above the saved PC, the callee will continue building the frame, as shown in Figure 1.9.

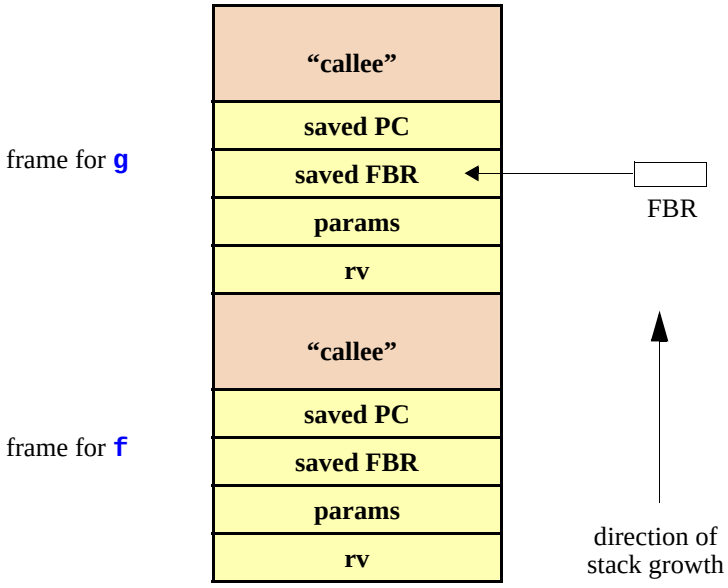

Figure 1.9: Saved PC

Now, we can reveal exactly what **JSR** does:

- **Stack[SP]←PC+1**
  Store the address of the next Samcode instruction to process on the stack. This cell is called the ***saved PC***.
- **SP++**
  Increment the stack pointer to the cell *just above* the saved PC.
- **PC←*label***
  Sets the PC to ***label***, which means executing instructions elsewhere in your Samcode. So, you effectively jump to another location in your Samcode file, which should be the callee's Samcode.

How does SaM return control to the caller?

- The stack is building up and tearing down as the callee executes the block of statements inside its body. The callee must also "tear down" the frame as the block finishes processing. When the callee's body finishes, the SP should be pointing to the saved PC cell.

- SaM is jumping back and forth in its stored collection of Samcode instructions. When SaM executes **JSR g**, SaM literally jumps to another instruction. When the callee finishes its work, control needs to return to the caller. Since the SP now points to the saved PC, a call to a special instruction **RST** (return from subroutine) sets the PC to the value on the stack. So, SaM can jump to instruction just below the **JSR g**!

Do you see how SaM uses cell addresses and program labels in a collaborative fashion? Formally, **RST** does the following:

- **PC←Stack[SP]**
  Sets the PC to the current value on the stack. If the callee properly tore down the callee portion of the frame, the PC should now be the *saved PC*.
- **SP--**
  The saved PC is popped.

Refer also to Figure 1.10, which shows the same pattern between caller **f** and callee **g**. **JSR** goes to the location of the callee Samcode inside SaM, and **RST** returns to the remaining instructions of the caller.

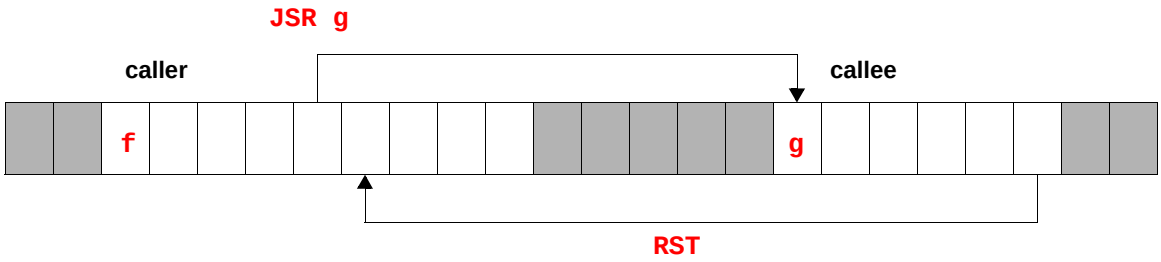

Figure 1.10: Program Instructions In SaM's Memory

The Samcode, below, gives a bit more detail to show the pattern for **f ( ) { g (1) ; }**:

```
f:
// statements inside f
ADDSP 1    // rv for g
PUSHIMM 1  // push parameter
LINK       // save f's FBR; set FBR to g
JSR g      // call g
// code to tear down g's frame
// continue with f's frame
g:
// statements inside g
RST        // return
```

Now that you can build the caller portion of the frame, we focus on the callee.

3.3.6 Callee Code

The callee portion of the frame is pretty straightforward as opposed to the **JSR** and **RST** insanity, which won't seem so crazy once you get used to it. There are two portions of the callee:

- Local variables, or just ***locals***, which are allocated just above the saved PC.
- Cells, called ***temporaries***, to store general code in the body of the callee.

These cells correspond to the body of the callee, which is the primary block of statements that constitute the actual function. As hinted at in Section 2.1, the callee is effectively the function body. So, all of the Samcode you learned in Part 2 is effectively the callee. Of course, the callee may in turn call yet another function, which will build another frame on top of the stack. In Figure 1.11 we now show the complete convention for the frames on the stack.Also, we can now completely describe the process of destroying a function's frame.

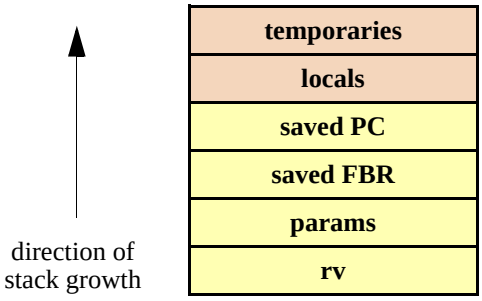

Figure 1.11: Complete Frame Order

The complete process of the callee is as follows:

- Perform the primary body of the callee code.
- Store the return value with **STOREOFF**. Note that the offset will be at least -1 because the FBR is set to the saved FBR, which is above the parameters and return variable.
- Tear down the frame.
- Return to the caller with **RST**.

So, you will have a general pattern that looks something like the following template:

```
Label:
callee Samcode
JUMP LabelEnd:
LabelEnd:
STOREOFF -(p+1)   // store rv
ADDSP -locals     // deallocate locals
RST               // return to caller
```

Why do we use a ***LabelEnd*** to identify a portion of the callee's Samcode? Since the callee might involve control structures, this style provides a way to distinguish the code for the final phase in which the callee finishes tearing down itself and returning to the caller.

3.3.7 The Rest Of The Caller Code

After returning to the caller from the callee, the remaining Samcode needs to tear down the rest of the frame, as shown in Figure 1.3. Since everything up to, and including the saved PC, is popped by the callee, the caller needs to do the following:

- Restore the FBR. Use the instruction **UNLINK**, which sets the FBR to the saved FBR address (thus moving the FBR to saved FBR of the previous frame) and deallocates the cell.
- Remove the parameters. Use **ADDSP -*p***.

The remaining cell on the stack is the return value, which is used by the previous callee. Eventually, the final return value will be that of the program.

Our running example of `f ( ) { g ( 1 ) ; }` is finished below:

```
f:
// statements inside f
ADDSP 1      // rv for g
PUSHIMM 1    // push parameter
LINK         // save f's FBR; set FBR to g
JSR g        // call g
UNLINK       // restore FBR to f's saved FBR
ADDSP -1     // pop param 1
// continue with f
g:
// statements inside g
JUMP gEnd:
gEnd:
STOREOFF -2 // store rv
RST         // return
```

3.4 Samcode Patterns

This section summarizes the algorithm and templates needed to build up and tear down the stack.

3.4.1 Program and Main Code

There is an element of "chicken-and-the-egg" at work herein. Namely, who calls `main`? Well, you could say, "the program." But who calls the program? Another program? And who calls whatever that is…?

What we *suggest* that you do is limit the pattern to "Program" who calls the `main` function. The model that we follow pretends that "Program" is a degenerate function, or a function with limited features:

- The FBR of "Program" is address 0.
- The return variable of "Program" is address 0.
- The variables of "Program" are any global variables. (See next section.)

As shown in Figure 1.12, "Program" calls `main`. So, the new FBR points to `main`. Eventually, program will need to store `main`'s return value in the first cell. Following this reasoning, you may use the following template:

```
Program:
ADDSP 1       // return value for main (and program)
ADDSP g       // allocate global variables
// set up main:
ADDSP 1       // allocate main's return variable
LINK          // save old FBR (Program) and set new FBR (main)
JSR main      // jump to main
// clean up main:
UNLINK        // pop FBR
STOREOFF 0    // store return value of Program (use main's rv)
ADDSP -g      // pop globals
STOP          // done!
main:
```

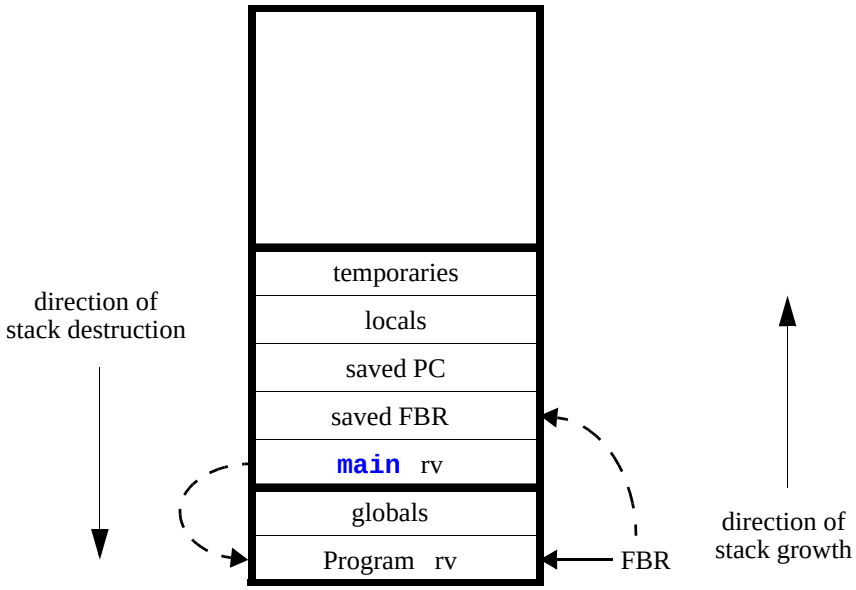

Figure 1.12: "Program" Frame

```
ADDSP v       // allocate main's local variables
code for main's statements
code for return expression
JUMP MainEnd  // finish main
mainEnd:
STOREOFF -1   // store rv
ADDSP -v      // deallocate locals
RST           // return to program
```

There are other models, which you may use. Regardless of what you choose, remember that SaM always starts at the top of your Samcode file and that you must have a **STOP** to finish.

3.4.2 Global Variables and Functions

If your grammar has global variables, then you are strongly encouraged the suggested model in the previous section. Thus, the globals will likely reside in the "Program" frame. As opposed to **STOREOFF** and **PUSHOFF**, which rely on the FBR for relative addresses, you now must use **STOREABS** and **PUSHABS**, which are described in Part 1. Your compiler will need to maintain a symbol table with the absolute cell addresses of the global variables so that all functions can access the information.

Do you notice the similarity between global variables and how functions are called? Somewhere in your compiler, you also need to keep track of function names because functions in Part 3 are not part of classes. Instead, functions exist in the scope of the program!

3.4.3 Caller Pattern

Figure 1.13 summarizes the Samcode caller pattern, like `g ( e1 , … , en )`.

Figure 1.13: Caller Pattern

`ADDSP 1`	return value for new function
`code for e1`	resolve *e1* to push param value
…	…
`code for en`	resolve *en* to push param value
`LINK`	save FBR of caller and set FBR to callee (*g*)
`JSR` *`g`*	jump to subroutine *g* (the callee)
`UNLINK`	pop saved FBR and store as new FBR
`ADDSP -`*`p`*	after returning, pop parameters

3.4.4 Callee Pattern

Figure 1.14 summarizes the Samcode pattern for a function call, like

```
int g ( t1 p1, t2 p2, ... , tn pn) {
   locals
   statements
   return statement
}
```

Figure 1.14: Callee Pattern

`g``:`	callee name
`callee Samcode`	code for primary body in callee
`JUMP` *`g`*`End:`	process clean-up of callee
`g``End:`	portion of Samcode to finish callee
`STOREOFF -(`*`p`*`+1)`	store rv
`ADDSP -`*`locals`*	deallocate local variables
`RST`	return to caller

3.5 Examples

The following examples uses an expanded Bali-- (not your real Bali!), but the Samcode follows the required pattern as explained in this document. We are also using an alternative program structure.

3.5.1 Add

```
main() {
   { int x, int y; }
   { x = 10; y = 20;
     return add(x,y); }
}
add(int p1, int p2) {
   { }
   { return (p1+p2); }
}
```

```
main:    PUSHIMM 0      // return slot for main and program
         ADDSP 2        // allocate 2 local vars
         PUSHIMM 10     // push 10
         STOREOFF 1     // store val 10 in address 1 (x<-10)
         PUSHIMM 20     // push 20
         STOREOFF 2     // store val 20 in address 1 (y<-20)
         PUSHIMM 0      // allocate return value for add
         PUSHOFF 1      // push value of x for p1
         PUSHOFF 2      // push value of y for p2
         LINK           // save old FBR (0) and update FBR (6)
         JSR add        // jump to function "add"
         UNLINK         // restore FBR after returning from "add"
         ADDSP -2       // pop parameters (p1, p2) of "add"
         JUMP mainEnd   // prepare to end main
mainEnd: STOREOFF 0     // store program's rv
         ADDSP -2       // remove x,y
         STOP           // end program
add:     PUSHOFF -2     // get p1
         PUSHOFF -1     // get p2
         ADD            // push x+y
         JUMP addEnd    // begin to end add
addEnd:  STOREOFF -3    // store x+y as rv
         RST            // return to main
```

3.5.2 Selection

```
main() {
   { int val; }
   { val = 6;
     return (check(val)); }
}
check(int val) {
   { int flag; }
   { if ( (val > 5) )
      flag = 10;
     else
      flag = 20;
     return flag; }
}
```

```
main:     PUSHIMM 0     // space for program rv
          PUSHIMM 0     // space for val
          PUSHIMM 6     // value to store in val
          STOREOFF 1    // val <- 6
          PUSHIMM 0     // space for check's rv
          PUSHOFF 1     // param to check
          LINK          // save and update FBR
          JSR check     // call check
          UNLINK        // done with check; pop FBR
          ADDSP -1      // get rid of local param
          JUMP mainEnd  // prepare to end program
mainEnd:  STOREOFF 0    // store rv
          ADDSP -1      // get rid of local var
          STOP          // end program
check:    ADDSP 1       // allocate flag
          PUSHOFF -1    // push val
          PUSHIMM 5     // push 5
          GREATER       // Is Vbot > Vtop (val>5)?
          JUMPC correct // true?
          PUSHIMM 20    // false, push 20
          STOREOFF 2    // flag = 20
          JUMP continue // continue with program
correct:  PUSHIMM 10    // true, push 10
          STOREOFF 2    // flag = 10
          JUMP continue // continue
continue: PUSHOFF 2     // push the value of flag
          JUMP checkEnd // begin to end add
checkEnd: STOREOFF -2   // store flag as rv
          ADDSP -1      // ditch flag
          RST           // return to main
```