SFU CMPT 379
Compilers
Fall 2022
Milestone 3

Assignment due Friday, December 2, by 11:59pm.

For this assignment, you are to expand your Bilby-2 compiler from assignment 2 to handle Bilby-3. Project submission format is the same as for milestone 2.

Bilby-3 is backwards compatible with Bilby-2. All restrictions and specifications from Bilby-2 are still in force, unless specifically noted. I also omit unchanged productions. If "**...**" is listed as a possibility for a production, it means "all possibilities for this production from Bilby-2".

Language Bilby-3

S → *globalDefinition** **main** *blockStatement*
globalDefinition → *functionDefinition*

statement → *returnStatement* // allowed inside a function definition only.
callStatement
forStatement
breakStatement
continueStatement
…

functionDefinition → **func** *type identifier* **(** *parameterList* **)** *blockStatement*
parameterList → *parameterSpecification* ⋈ **,** // 0 or more comma-separated parameterSpecifications
parameterSpecification → *type identifier*

type → **void** … // **void** usable only as a return type

returnStatement → **return** *expression*? **;**
callStatement → **call** *functionInvocation* **;**

forStatement → **for (** *identifier* **in** *expression* **)** *blockStatement* // expression must have a char or int range type
breakStatement → **break ;**
continueStatement → **continue ;**

expression → *functionInvocation*
…

functionInvocation → *identifier* **(** *expressionList* **)**
expressionList → *expression* ⋈ **,**

1. Function definitions

The productions

functionDefinition → **func** *type identifier* **(** *parameterList* **)** *blockStatement*
parameterList → *parameterSpecification* $\bowtie$ **,**
parameterSpecification → *type identifier*

define the syntax of function definitions. All functions are declared before the **main** block.
The *identifier* in the production for *functionDefinition* is called the name of the function, the *type* is called the return type of the function, and the *blockStatement* is called the body of the function. These terms carry roughly the same meaning as they do in java or C++.

The following is a valid *functionDefinition*:

```
func float weight(int numHydrogen, int numOxygen) {
        imm g := 6.02E23 ;
        imm hy := 1.008 ;
        imm ox := 15.999 ;
        imm molecularWeight := hy * numHydrogen + ox * numOxygen ;
        return molecularWeight * g ;
}
```

The name of the function is weight, and its return type is floating. It has a signature of "(int, int) -> float".

Function defintions are *visible* (can be used) anywhere inside the program, unless they are shadowed by a variable of the same name. This includes the parts of the program *before* and *during* its defintion. Function defintions count as a defintion of their name, so one may not have another function with the same name.

A variable declaration in any scope (including in the function definition itself) may shadow a function definition. While the function definition is shadowed, the function is not visible.

```
func int fin( int a, int b ){
        return pax(5*a, b) ;                    // legal : can reference pax before its definition.
}
func int pax( int a, int b)  {
        if (a < b ) {
                return a ;
        }
        return b + pax(a, 3*b) ;                // legal : recursion is okay.
}
func float pax( float a, float b) {             // illegal : overloading is not okay.
        if (a < b ) {
                return a ;
        }
        return b + pax(a, 3*b) ;                // legal, if this were a legal function : recursion is okay.
}

main {
        imm a := 11;
        imm b := 14;
        imm c := 16;

        if( a > 5) {
                imm m := pax(a, b);             // legal
```

```
        imm pax := pax(a, b);                    // also legal but bad practice

        … // in here, the function called pax is shadowed and may not be used

    }
    imm x := pax(a, b);                          // legal to use the function pax again.
}
```

2. Scoping

2.1. Scope terminology

A scope is what we call a block of code in which variables can be declared and outside of which the variables do not exist (or cannot be referenced). We can classify scopes according to when we create memory for their variables at run-time. The following is a general discussion of these scopes; this is not particular to Bilby.

A *static* scope is any scope that we can know the location of at compile time (remember, the word "static" *means* compile-time). Static scopes include the global scope, and, for instance in java or C++, a static scope holding any variables (inside objects and/or methods) explicitly declared as `static` in these languages.

> The *global* scope has variables visible throughout a program, and there is only one copy of each of the global scope variables active at runtime. Because there is only one of these scopes, we can specify where to put it in memory, and therefore know the addresses of variables in it at compile time.

A *dynamic* scope is any scope that is not static. We typically cannot determine the location of the memory for the variables in a dynamic scope at compile-time; this is most often because we may have more than one live instance of the variables in this scope at run-time. (Consider, for instance, the variables associated with a recursive procedure or an object class with multiple instances.) To access a variable in a dynamic scope, we maintain a pointer to the memory block for the scope, and add a fixed (statically-determined) **offset** to this pointer.

> An *object* scope is a dynamic scope for variables that exist for the life of an object instance. In C++ these variables are called the *member variables*, and in java they are called *fields*. The object may be user-created or it may be a language-created object. Typically, memory for object scopes are allocated from the heap, because objects can be created or destroyed at any point in a program.
>
> A *procedure* scope (sometimes called a *frame scope*) is for variables that exist for the life of a procedure call. In most languages, the variables in these scopes are called *local* variables.
>
> Some people separate out a *parameter* scope from the procedure scope. This scope contains only the parameters to the procedure call.

A *local scope* (sometimes *block* scope) is a scope that is particular to some piece of code under consideration. This piece of code may be an entire procedure call, or it could be the "else" clause of an if-then-else statement. A local scope can be dynamic (a procedure scope or inside of a procedure scope, for instance) or static (such as the block statements following the `static` keyword in java).

A scope may be *nested* in another scope; this means it is (lexically) completely inside the other scope. Examples are parameter and function scopes nested inside an object scope in java, or the local scope of a block statement inside of a member function in C++. The *parent* of a nested scope is the smallest scope containing it. The nested scope is called a *subscope* of any scope containing it.

Do not confuse *dynamic and static scopes* with *dynamic and static scoping*. In static scoping (also called *lexical scoping*), a unit of code (e.g. a procedure or block) has access to the variables in any code that statically (lexically) encloses it.

For instance, in a java method you have access to variables declared in the containing class. In dynamic scoping, a procedure has access to the variables in any code that calls it (in other words, access to variables in its *dynamic parent*). Dynamic scoping is found in languages such as Lisp and Logo. (Dynamically scoped programs are difficult to get correct, so dynamic scoping has fallen into disfavor.)

2.2. Allocation of offsets for variables in scopes

Memory for variables is typically allocated at the global level, the object level, and the procedure level. Any memory needed for a subscope of one of these levels is included in the memory needed for the closest enclosing one of these levels. For instance, if a procedure scope holds five local scopes, the memory needed for the local scopes is included in the memory allocated for the procedure.

Since each variable is held inside a block of memory allocated at one of these levels, each variable is typically accessed by adding a pointer to the block (called the *base pointer*) to an offset for the variable in the block. The base pointer is (for dynamic scopes) a run-time variable, but the offset is fixed at compile time. Typically the semantic analyzer will step through the variables in a scope, allocating successive offsets to the variables it encounters. Complicating this process are three concerns:

(1) Variables are of different sizes, so *successive* above means "changed by the size of the allocation."
(2) We want to conserve memory, so non-overlapping subscopes should be assigned to the same memory.
(3) Sometimes we have a pointer to the *top* rather than the bottom of a block of memory, so we sometimes allocate positive offsets and sometimes allocate negative ones.

The Bilby symbolTable package (which handles scoping) is able to accommodate these concerns. However, I suggest changes below to make things a bit more clear for Bilby.

I suggest using four different types of scopes in Bilby-3.

program scope	for the base level of statically allocated memory (the scope attached to the ProgramNode, or root of the AST). This scope has a positive (postincrement) allocation scheme. In this scheme, when a scope receives a request to allocate *n* bytes, its current offset is returned (allocated), and then it is incremented by *n*. These use the global memory block (called GLOBAL_VARIABLE_BLOCK in the compiler) as their base pointer.
parameter scope	for the base level (parameter scope) of each procedure. These scopes are attached to the function definition node. They have a negative (predecrement) allocation scheme, where first the current offset is decremented by the requested size, then it is returned. These scopes use the *frame pointer* (see sections below) as their base pointer.
procedure scope	for the function body of each procedure. These scopes are attached to the body node (blockStatementNode) of the procedure. They have a negative (predecrement) allocation scheme. These scopes use the *frame pointer* (see sections below) as their base pointer.
subscope	for all other scopes. These scopes defer to their static parent scope for their allocations.

Actually, the procedure scope can be implemented as a subscope of the parameter scope. In Scope.java, you have already been provided with program scopes and subscopes (the first two factories). Note that createProgramScope() is static, whereas createSubscope is a method. You must implement parameter and possibly procedure scopes.

Both parameter and procedure scopes have slight wrinkles in their use. For parameter scopes, it will be necessary to modify the allocated offsets when the scope ends, by adding the scope size to each offset. (This will give positive offsets, with offsets starting at the most positive and decreasing.) If you implement procedure scopes as subscopes of the parameter scopes, then they also will need to have the offsets increased. Details are found in the sections below.

You will probably need to create a class ParameterMemoryAllocator to handle parameter scopes. The simplest way to implement this is for ParameterMemoryAllocator to be just like NegativeMemoryAllocator, except that it keeps track of all of the MemoryLocations it has allocated, and then, in restoreState, if there are no bookmarks after the remove, it adds the maximum allocated size to all of those MemoryLocations. **This will necessitate putting a mutating method on MemoryLocation.**

> **An aside:** Making a class mutable often gives a large increase in complexity, so don't do it lightly. Having a different solution (for example, a Builder pattern) might be better here, as MemoryLocation should cease mutating after the first mutation.

For procedure scopes, the runtime system will need 8 bytes of memory, so this memory should be allocated when the scope is created. This can be done in the semantic analyzer or in scopes—your choice.

2.3. Implementation of visibility for function invocations

The Bilby programmer can use functions that are declared *after* their use. In a normal semantic-analysis tree traversal, these functions would not be in a symbol table at the time when we are checking the expression that uses them. Because they are not in a symbol table, we don't know their parameter types or their return type, so we can't typecheck the expression. This is a problem.

There are two approaches to solving this problem. The first is to defer the checking of the expression using the function until after the function is defined. This means leaving not only the tree node of the function-call itself unchecked, but also most nodes above that node in the tree. When the definition of the function is finally found, then the function-call and its tree ancestors are checked. This is called *backpatching*, and it involves a lot of bookkeeping to ensure that everything eventually gets correctly checked. Backpatching code is difficult to get correct as it jumps around the tree rather than following a standard traversal.

The second approach is to do two traversals of the tree for semantic analysis. The first traversal is simple, and it looks for function declarations only. It computes the signature for each of these functions and puts a binding for it in the immediately-enclosing scope's (here, the global scope's) symbol table. Then the second traversal is the usual one, except that scopes already have all of the functions installed in the symbol table (for now, in Bilby, only the global scope is affected). These symbol-table entries can then be used to get the information needed to typecheck the uses of these functions.

I recommend the second approach. It involves creating a second visitor for semantic analysis, and executing that visitor before the current one.

So, in the first visitor, on reaching the program node (in its visitEnter method), the global scope can be created and attached to the node. The names and signatures of function definitions can then be recorded in the scope's symbol table. In the second visitor, we process the variable definitions within the global scope (during the traversal below the scope's node), and then close the scope (in the corresponding visitLeave).

To look at it from the point of view of the global scope: first, create the scope. Second, add any global definitions to that scope's symbol table. Third, in the second visitor, add any other definitions (this will probably be in subscopes of the global scope). Fourth (and last), close the scope. Add no further definitions to the scope. [Note that we have an allocation scheme where we put an allocation made in a subscope into its parent scope, and

use bookmarks to measure and reuse memory. The bookmark mechanism assumes scope creation/leave have LIFO (parenthesis) structure. If we were to allow function declarations in other scopes with use-before-define semantics, then we would have to change this mechanism. [For we could get a situation where a function definition in a subscope sets a bookmark and allocates in the superscope in the first visitor, then the superscope itself adds some allocations in the second visitor, followed by the closing (leave()) of the subscope in the second visitor, which would tear off any allocations back to the bookmark on the superscope, resulting in the loss of space for the allocations that the superscope made. Congratulations if you followed that.]

If *scope* is a scope, and *parent* its parent scope, then creating *scope* and *scope.leave()* should be called between creating *parent* and *parent.leave()*. If this guideline and those of the previous two paragraphs are not followed, it may cause the MemoryAllocator mechanism to fail, and result in memory corruption.

Because functions are not executed where they are defined, you should find a way to either emit the function code before or after all the other code, or issue a jump statement to get any other code to avoid executing the function code where it is defined.

2.4. Scopes for function declarations

Each functionDefinition denotes two scopes: one that includes the parameters and function body, and another that contains only the variables in the function body. We will call scope that includes the parameters the *parameter scope*, and the one that excludes it the *body scope*. The only definitions appearing in the parameter scope are the parameters of the function, which in Bilby are treated as if they are declared **imm** (immutable). On the other hand, definitions can also appear in the body scope, for example:

```
func float root(float a, float b, float c) {
        imm discriminant := b*b - 4*a*c;
        . . .
}
```

The parameter scope should be implemented with the parameter scope type from Sec. 2.2, and the body scope with the procedure scope or subscope type.

We don't have the following situation in Bilby, but it comes up when you allow local (not just global) function or lambda definitions. To avoid problems with referencing local variables, parameter scopes are often made *opaque* to any definitions outside them which are not at global scope. This means that code inside the scope cannot access local variables that are outside the scope. Global-scope variables (currently only function names) can be accessed from inside a procedure.

2.5. Implementing a function call; memory allocation for functions; the frame stack

Use a frame stack to implement local storage space, argument passing, and return values for functions. There are two main pointers for a frame stack:

- the **frame pointer**, which contains the location of (informally, "points at") the byte directly above the frame for the procedure currently executing, and
- the **stack pointer,** which points at the bottom byte of the frame (or below if arguments are being set up in order to call a function).

There is only one frame pointer, and one stack pointer, in existence in a program. Thus, you may allocate space for them globally (typically in Runtime.java). Initialize them both to the location above the top of memory; there is an instruction (opcode) that will help with this. Read ASMOpcode until you find it.

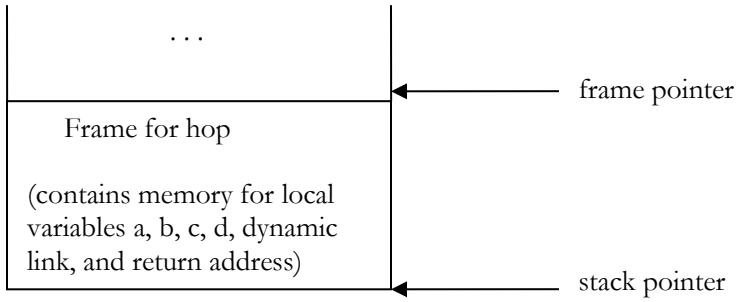
Each currently "live" function invocation (function invocation that has started but not completed) has a *frame* (or *activation record*) on the frame stack. Each frame contains the previously-executing function's frame pointer (this is called the **dynamic link**), the return address for the Call instruction that called the function, and space for local variables. Above the frame, we will store the function's arguments.

To set up a function invocation, place the arguments to the function, from *left* to *right*, on the frame stack. To place an argument on the frame stack, first decrease the stack pointer by the size of the argument, and then write the argument value to the stack pointer location. **You are not allowed to pass arguments or return values on the ASM accumulator stack** (you'll get a very hefty penalty on the assignment if you do and we notice)**.** This is because real stack machines actually have limited stack size and recursive procedure arguments would eat it all up if they were to be kept on the stack. (The ASM has a large stack size, but I want you to get practice setting up a full stack frame.)

For instance, if you have:

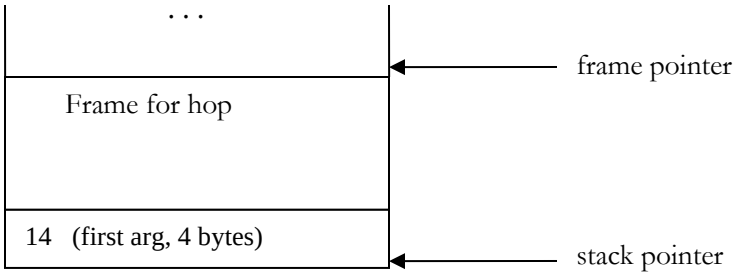
```
func int hop ( ) {
        imm a := 14;
        imm b := 1.23;
        imm c := 20;
        imm d := barge( a, b, c);
        . . .
}
```

Then, at the start of some invocation of hop( ), the frame stack looks like

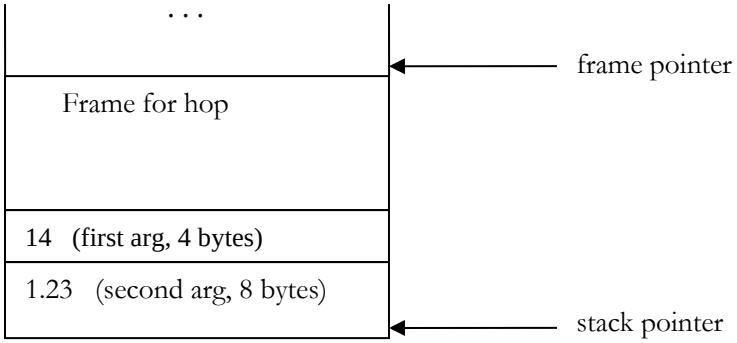


In these diagrams, high memory (larger addresses) is at the top, and memory continues both above the diagram and below the diagram.

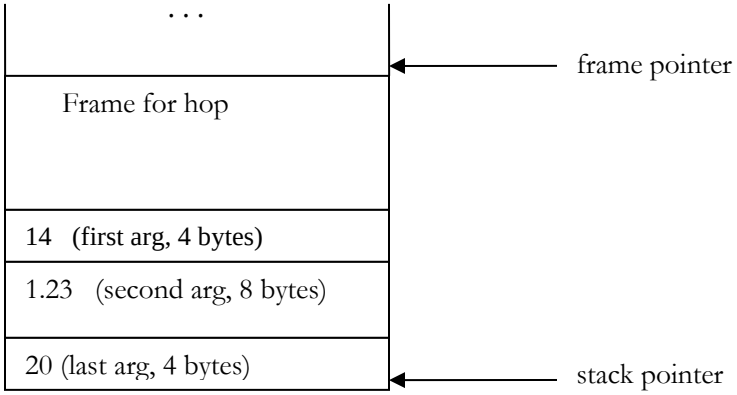
When starting to process the call to barge, one visits the first argument (a) first, placing it on the stack below the frame:



Then one visits the second argument, placing it below the first:

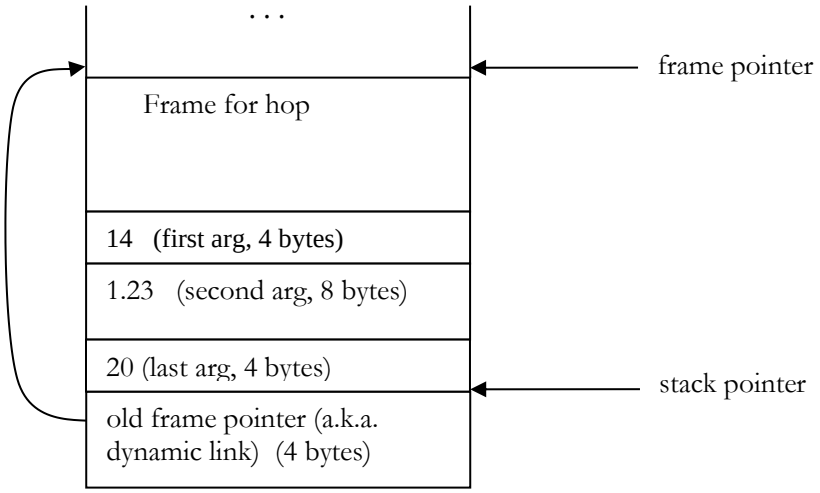


And similarly for the last:

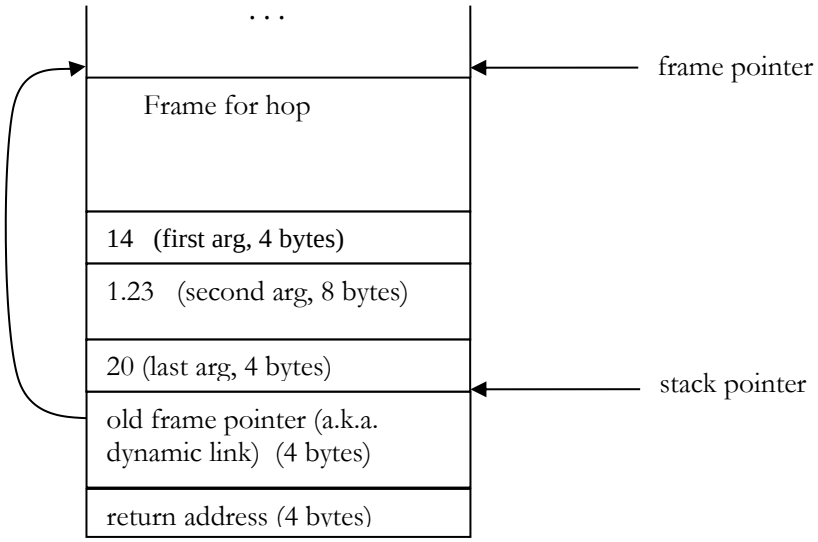


At this point, one calls the subroutine for barge() using ASMOpcode **Call**.

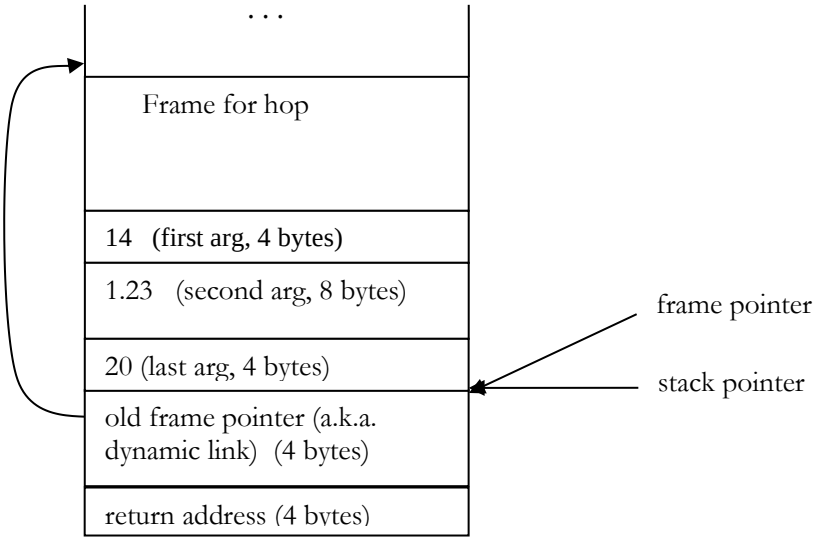
The subroutine barge() first stores the value of the frame pointer (which points to the top of hop's frame) below the current stack pointer. (This area below the stack pointer will become the top of barge's frame.)



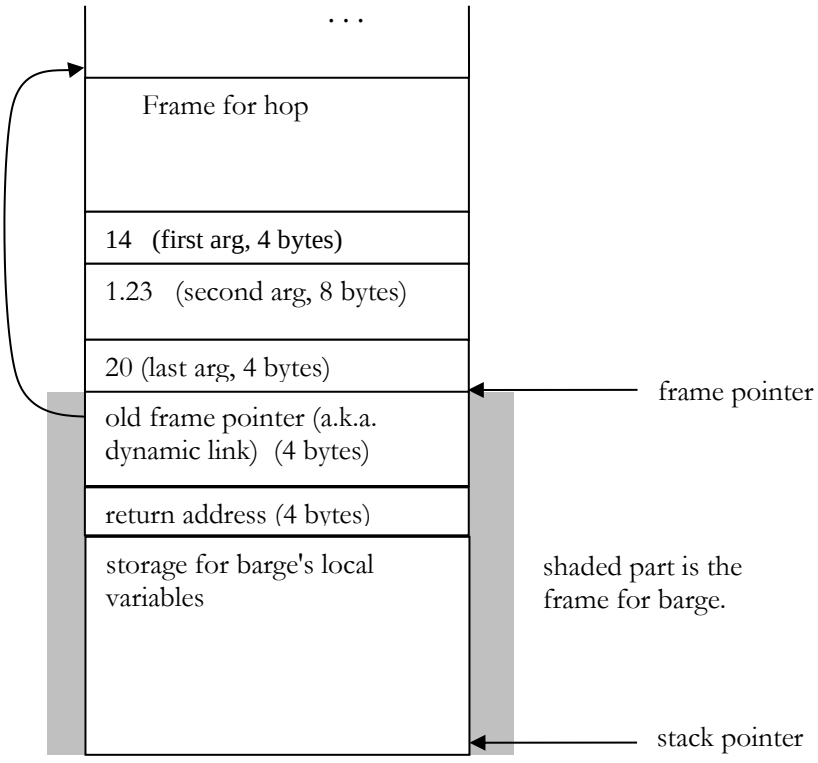
Then it stores the return address (which is placed on the ASM stack by the ASMOpcode **Call** or **CallV**) below the dynamic link.



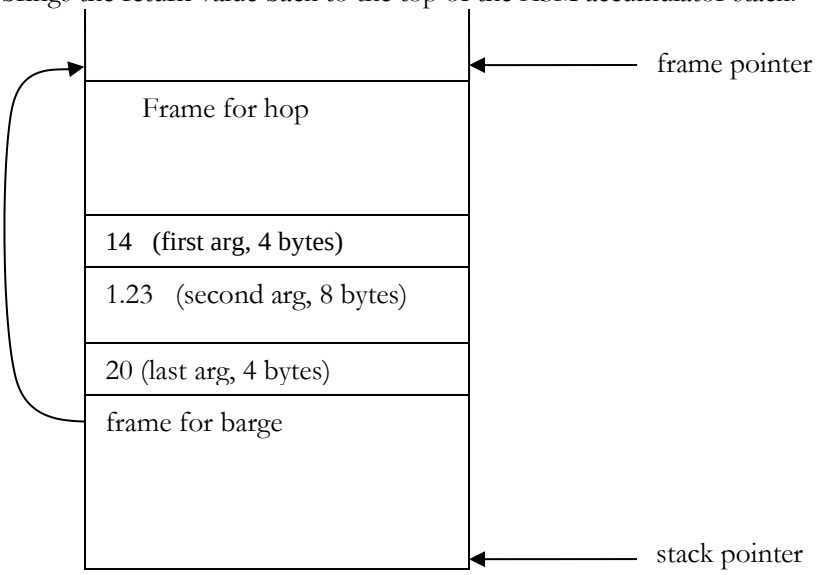
Next, the frame pointer is set to be equal to the stack pointer.



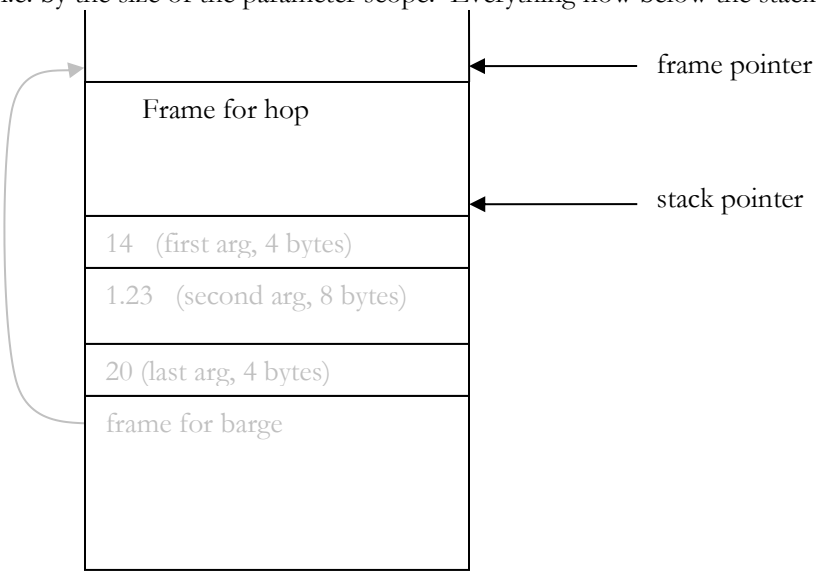
Next, the size of the frame for barge() is subtracted from the stack pointer. This frame size should include the 4 bytes for the dynamic link, the 4 bytes for the return address, and all memory necessary for the storage of local variables.



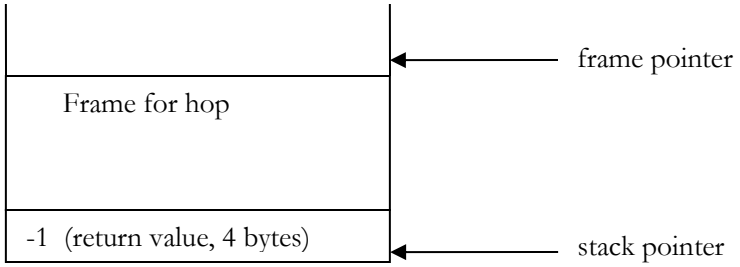
Now the user code for barge( ) starts executing. At some point, barge( ) may execute a **return** statement, with, say, the integer value -1. This value is placed on top of the ASM accumulator stack by the **return** statement. The exit handshake code for barge( ) should first push the return address (stored at framePointer-8) onto the accumulator stack, then replace the frame pointer with the dynamic link. An **Exchange** operation brings the return value back to the top of the ASM accumulator stack.



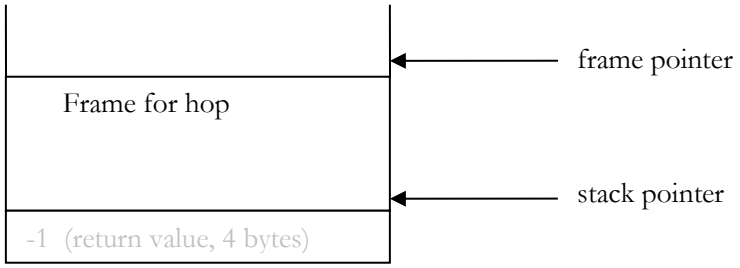
Then the code should increase the stack pointer by the size of barge's frame + the size of barge's arguments, i.e. by the size of the parameter scope. Everything now below the stack pointer is no longer needed.



Finally, it should decrease the stack pointer by the return value size (in our example, the return value is an int, having a size of 4 bytes). Then it should store the return value at that location.



We then return from barge() using the return address on the ASM stack, transferring control back to hop(). hop() then should take the return value from the location pointed at by the stack counter and place it on the ASM stack. Finally, it moves the stack pointer up by the size of the return value:



The user code in hop() then continues.

Again, a reminder: You may not pass the arguments or return value on the ASM stack. When you transfer control between the callee and caller (either way), these values must be on the frame stack, as shown above. This is because typical stack machines have a relatively small stack and it is expensive to overflow it. Nested and recursive procedures can end up pushing a lot of arguments and return values on a stack, so the stack machine stack (ASM stack) is an inappropriate place for them.

2.6. Offsets for variables in parameter and procedure scopes, revisited

We use the abbreviations FP for the frame pointer and SP for the stack pointer.

Examine the diagram above showing when the frame for barge is finished being set up. From this, one can see that the arguments to barge are at offsets 12, 4, and 0 from FP (parameters 1, 2, and 3 respectively). In general, the offsets of the arguments from FP will start at some positive number and decrease to zero. Thus, when you are in semantic analysis on a parameter scope, you should allocate offsets in a negative direction.

So start a negative allocation scheme at zero offset (decrement before allocating), and allow the offsets to go negative. When you are finished with the parameters, determine the total size of the parameters, and *add this to each offset*. In the example, we would start allocating with a 0 offset, and when encountering parameter 1, we would subtract 4 (its size) to get an offset of -4. For parameter 2, we subtract 8 (it's a float…8 bytes) to get -12. For parameter 3, we subtract 4 to get -16. Now we add the total size (16) to these offsets, and end up with offsets of -4 +16 = 12, -12+16 = 4, and -16 + 16 = 0 for parameters 1 and 2 and 3, respectively

Again examining the same diagram above, one can see that the dynamic link is at offset -4 from FP, and the return address is at -8. Local variables are below that. Thus, when you analyze a function body scope, the allocation is in a negative direction, decrementing before allocating, starting at -8, with the frame pointer as base. Somewhere (perhaps in visitEnter for the function body?) you will need to allocate the 8 bytes for the dynamic link and return address.

Since FP is stored in static memory, and at compile time you only have its address, not its value, the FP (base pointer for memory access) is said to be *accessed indirectly*. This means that you must insert a LoadI to get the value of the base. Thus, accessing a variable at offset -24 would be done as follows:

```
PushD    $framePointer
LoadI
PushI    -24
Add
```

Okay, now we know the code that we need to access local variables, but where do we put it? Variable access code is generated in symbolTable.MemoryAccessMethod. There, INDIRECT_ACCESS_BASE is already set up for you. You just need to connect it to the proper scopes in Scope.java.

2.7. Promotion and procedure invocation

Procedure invocations in Bilby do not promote any of their arguments. All arguments must be cast to appropriate types.

3. Break and continue statements

These statements are only allowed inside the body of a **while** or **for** loop. This includes nested inside statements in the loop body.

A **break** statement immediately jumps to the code immediately after the closest (most deeply nested) loop that contains it. A **continue** immediately jumps to the code of the closest loop—if that's a **while** loop, it jumps to the code for checking the condition; if that's a **for** loop, it jumps to the code for the increment.

4. For statements

forStatement → **for (** *identifier* **in** *expression* **)** *blockStatement*

The **for** statement is a loop with an identifier iteratively taking on the values in a range expression. The expression must be of type **range(char)** or **range(int)**.

It is somewhat equivalent to (this is not proper bilby):

```
{                                   // new scope
    imm identifier = low expression;
    while ( identifier <= high expression ) {
        blockStatement;
        identifier++;
    }
}
```

except that the *expression* is only evaluated once, the identifier is mutable only at the increment, and the increment operates on either an int or a char identifier.

So, **for** statements should have a scope associated with them other than the *blockStatement's* scope. There is only one binding placed in this scope, which is the binding for the identifier. It has an immutable binding because we don't want the *blockStatement* to be able to change it. However, the increment does change it, but no adjustment is required in the semantic analyzer because the semantic analyzer sees a tree corresponding to the **for** loop above, not the **while** loop. The code generator is where the adjustments need to be made, emitting the code for the identifier's initialization, its check against the upper bound, and increment. And placing the *blockStatement* code in the correct place.

As an example, the following for loop prints "4 9 16 25 ":

```
for(i in <2..5>) {
        print i*i $s;
}
```

5. Function invocations

functionInvocation → *identifier* **(** *expressionList* **)**

One approach to parsing function invocations is to use an *expression* in place of the *identifier* in the production. Then, in the semantic analyzer, check that the expression has a function type that matches the *expressionList*. (That is, the function parameter types match the types of the expressions in the list, in order.) This function invocation has the type that is the return type of the expression's function type.

Function invocations are never targetable. Expressions in the *expressionList* are not promoted.
Details on implementing function invocation are given in section 2.5 above.

6. Return statement

A *returnStatement* specifies the value that the enclosing function returns to its caller. You may have more than one returnStatement inside a function body. Any returnStatement that is not the last statement of the body terminates execution of the body after it is executed, as we expect return statements to do. In code generation, this is implemented as a jump to the start of the exit handshake after evaluating the expression in the return statement.

You must check that all return statements inside a function return the same type, and that this type is the declared return type of the function. Returned expressions are not promoted. You must also check that all return statements are within a function. You do not need to statically check that every control path in a function ends in a return. Issue a runtime error if code runs off the end of a non-**void** function. Do a return if it runs off the end of a **void** function.

7. Void type and void return statement

The **void** type may only be specified as the return type of a function. That is, it can be used as the *type* in the *functionDefinition* only. It may not be used as a parameter type, nor may it be cast to or from, nor may it be used as the base type of an array. A *functionInvocation* of a function having void return type may not be used as an expression. Such an invocation may be used in a **call** statement only.

To return a void type from a function, use the form of *returnStatement* that omits the *expression.* That is, use:

return;

Just to be clear, this return statement is said to be returning a value of type void. A value of void type cannot be the operand of an operator, and it cannot be assigned to a variable. For the purposes of the compiler, an "instance" of the void type consumes zero bytes.

8. Call statement

The call statement allows the bilby programmer to invoke functions that return **void**. They may also use a call statement when they do not care about the value returned from a non-**void** function. In this case, it is the responsibility of the compiler to issue instructions that remove the returned value from the accumulator stack.

9. Operator precedence

The precedence of operators is

Highest precedence	parentheses, populated array creation empty array creation casting range creation function invocation	**( ) [ ] alloc[]() [as] <..>** id**( )**
	array indexing	**[ ]**
(prefix unary operators are right-associative)	prefix unary operators	**+ - ! length low high**
	multiplicative operators	***** /
	additive operators	**+ –**
	comparisons	**< > <= >= == != in**
	and	**&&**
Lowest precedence	or	**||**

These are all left-associative operators, except as noted.