An implementation of function calls in the Mini compiler

This document describes the implementation of function calls in a compiler that translates programs from a small, imperative programming language called Mini directly into simple IA32 assembly code.

1 An overview of the abstract syntax classes

We start by taking some time to become familiar with overall structure of the compiler. For the purposes of CS322, you are not likely to need to study every aspect of the code (in particular, all portions of the code having to do with front end tasks such as lexical analysis, parsing, and type checking relate to topics that were already covered in CS321). Nevertheless, it will still be very useful to develop some familiarity with the abstract syntax classes that are used to provide a representation for expressions, as illustrated in the inheritance hierarchy diagram in Figure 1. In this diagram here, the classes representing specific Mini language constructs are labeled not just with the class name, but also a short fragment of concrete syntax to further illustrate what the class represents. This structure also includes some additional abstract classes like `UnExpr` (a base class for all unary expressions), `BinExpr` (a base class for all binary expressions), and `BinArithExpr` (a base class for expressions using one of the four main arithmetic operators). This structure makes a lot of sense because it allows us to organize the classes in a helpful way, and to avoid duplicating common code by placing a unique copy in the appropriate class. For example, all binary expressions have both a left and a right argument, so the code for defining those fields will fit nicely in to the `BinExpr` class.

There are also similar (albeit less complex) hierarchies for different forms of Mini types (with the base class `Type`) and Mini statements (with the base class `Stmt`). For the purposes of this document, however, our main focus will be the `Call` class (which is used to represent function call nodes in abstract syntax trees), the `Args` class (which is used to represent the list of arguments in a function call). We will also add some small extensions to the `Assembly` class, which is used to represent assembly language output files.

Source code for all of the classes mentioned here is included in the `mini` folder/package. The distributed version of the Mini compiler also includes a second package called `compiler`, which provides reusable infrastructure for building standard compiler phases and for supporting error handling functionality.

2 Stack frames in the Mini compiler

Before we discuss the specifics of the implementation of function calls in our Mini compiler, we will review the details of the stack frame layout that it uses.

At the point the point where we begin the process of calling a function (or indeed, evaluating any expression), we can expect a stack layout that looks something like the following:

...	... scratch ...	old	ret	...
	esp	ebp		s

In this diagram, the `ebp` register identifies the stack frame for the current function, including the return address, `ret` and the value of the base pointer, `old`, for the enclosing function. At any given point during

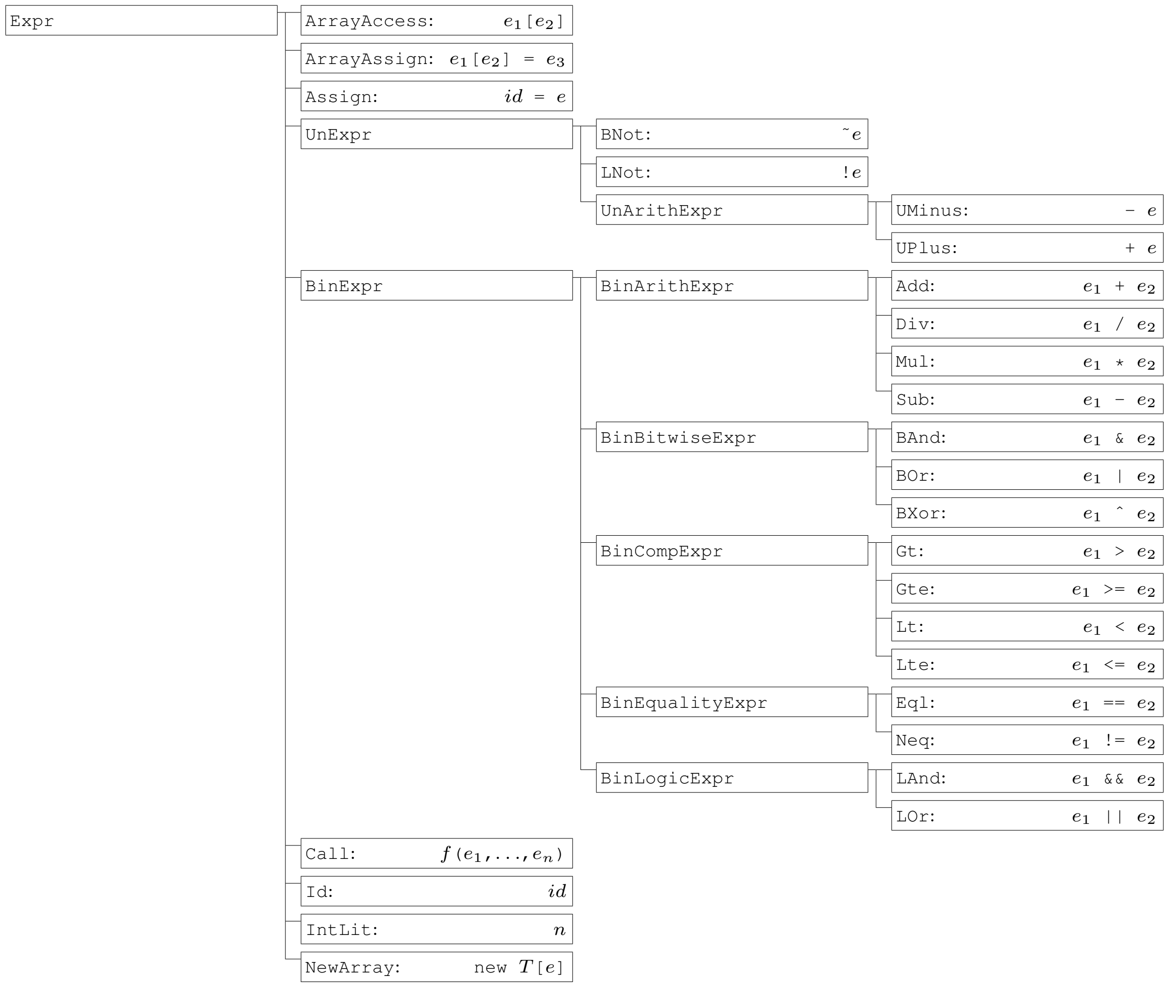

Figure 1: Inheritance hierarchy for abstract syntax of Mini expressions

the execution of the current function, the area between the current stack pointer (in `esp`) and the `old` field, marked as `scratch` in this diagram, will contain zero or more words of temporary data that have been saved on the stack as a result of register spilling. One further detail that we will draw attention to here is the first word after the `ret` field, indicated here by the address `s`. This address is important because we want the code produced by the Mini compiler to run on Linux, Windows, and Mac OS X platforms, and to comply with the Mac OS X calling convention, in particular, we must ensure that `s` is a multiple of 16. (This is sometimes summarized by saying that every stack frame must be "aligned on a 16 byte boundary".)

From this point, the first step to take before calling a function is to save the values of any registers that are being used. This is necessary because, in general, we don't know what effect the function that we are calling will have on those registers, and so it is necessary for us to ensure that they are preserved during the call. If the particular registers being used are r_i, ..., r_j, then it is very easy for us to save their values by pushing them on to the stack, resulting in a stack layout of the following form:

...	r_i	...	r_j	... scratch ...	old	ret	...
	esp				ebp		s

Of course, any register values that are spilled on to the stack in this way before the function call will need to be unspilled once the function call is done, returning those registers to their original contents.

At this point, we might hope that it would be possible to simply evaluate the arguments of the function and push their results on to the stack, creating a stack layout something like the following, and ready for the actual function call.

...	a_1	...	a_n	r_i	...	r_j	... scratch ...	old	ret	...
	esp							ebp		s

This process would be sufficient for Linux and Windows, but it is not enough, in general, for Mac OS X because of the previously mentioned alignment constraint. More specifically, to fit the calling convention on Mac OS X, we need to make sure that the value in `esp` immediately before a `call` is a multiple of 16. We can satisfy this constraint by adding some padding (i.e., unused locations) on the stack, inserting between 0 and 3 extra words. But we cannot expect the function that is being called to know what padding was required in any given setting, so we must insert it on the stack between the argument values and the saved registers creating a layout that looks like this:

...	a_1	...	a_n	... padding ...	r_i	...	r_j	... scratch ...	old	ret	...
	esp								ebp		s

Assuming that we have inserted the correct padding, we are now ready for the main function call!

If you are not a Mac OS X user, then you may be frustrated by the need to deal with an alignment restriction that is not relevant on your preferred platform. It should be noted, however, that we only have to worry about this complication in our code generator, and we will use a zero byte padding area on platforms that do not require the 16 byte alignment so that there is no runtime overhead. (Even on the Mac, the overhead is really very small, and can be expected to average at around 8 extra bytes of stack space per function call.) One way that this complication shows up in our code generator implementation is that we need to add an extra argument, `pushed`, to the compilation schemes that we use to track the number of bytes that are on the stack (i.e., between the stack pointer, `esp`, and the address `s` in the previous diagrams) at each point in the program. For example, you will find that the two parameter `compileExpr()` method that was described in class (taking an `Assembly` argument for the output, as well as the number, `free`, of the next available register) now includes `pushed` as a third argument:

```
    public void compileExpr(Assembly a, int pushed, int free)
```

In practice, it is not difficult to extend the compilation schemes that we have seen in lectures to use this extra parameter correctly. (The main challenge is just to make sure that we account for changes in the amount of data that is pushed to the stack in situations where register spilling is required.) By calculating the number of bytes that have been `pushed` at each point in the program, it is easy to determine how many additional bytes must be added to the stack to satisfy the alignment restriction. We will use the following method to perform the necessary calculation, but should note that as long as you understand the purpose of this function, it is not really necessary to dig in to the details of exactly how the calculation works.

```
    // in Assembly.java:
    int alignmentAdjust(int pushed) {
        // For platforms that need it (i.e., Mac OS X), we determine
        // how many extra bytes must be added to the stack to ensure
        // alignment on a 16 byte boundary.  For other platforms, we
        // can just return zero.
        return ((platform & ALIGN16)==0)
                 ? 0
                 : ((16 - (pushed + WORDSIZE)) & 15);
    }
```

3 Implementing functions

Now, at last, we can turn our attention to describing the steps that the Mini compiler follows to produce code for a function call. The main steps in this process have already been described in the previous section: save the values of any registers that are in use; if necessary, add some padding to the stack to meet the alignment constraint; evaluate the arguments of the function and save the results on the stack, and then call the function. At the end of all this, we will also need to produce code to remove the arguments and any padding that was added to the stack, and to restore the values of any registers that were saved at the beginning of the process.

We can see this particular sequence of steps quite clearly in the implementation of the `compileExpr()` method for `Call` objects: operations:

```
    // in Call.java:
    public void compileExpr(Assembly a, int pushed, int free) {
        // spill registers if necessary
        pushed += a.spillAll(free);

        // calculate number of argument bytes to be pushed
        int argBytes = Args.argBytes(args);

        // calculate an appropriate stack alignment
        int adjust = a.alignmentAdjust(argBytes+pushed);

        // reserve the necessary space on the stack
        a.insertAdjust(argBytes+adjust);

        // evaluate args and save results on the stack
        Args.compileArgs(a, argBytes+adjust+pushed, args);

        // do function call and move result into free
        a.call(name.toString(), free);
        // remove parameters and alignment adjustment from the stack
        a.removeAdjust(argBytes+adjust);

        // recover spilled registers
        a.unspillAll(free);
    }
```

This code uses several new methods, which we will define in the following subsections. It is not essential that we break the code down in this way, but it does help to keep it cleaner and easier to understand. Moreover, it turns out that we can reuse several of these new methods elsewhere in our compilation rules. (For example, this occurs in the compilation rule for the Mini `print` statement, which is actually implemented by a special form of function call).

3.1 Spilling and unspilling

In this subsection, we will tackle the code for spilling and unspilling at the beginning and the end of a function call. Spilling is necessary to preserve temporary values that are being held in CPU registers while we call some other piece of code that might use those registers in unpredictable ways. Spilling can be implemented by pushing register values on to the stack, and then those registers can be restored after the function call by a process of unspilling, which pops them back off in the appropriate order. There are `numRegs` physical registers, so the largest number of registers that we will ever need to save is `numRegs-1`; we don't need to save the specified free register because the whole purpose of the compilation scheme is to produce code that will overwrite that register with the result of the function call. And if `free<(numRegs-1)`, then we don't need to save the values in all of the registers, because only some of them are being used. As a result, we can calculate the total number of registers that should be spilled using `Math.min(free, numRegs-1)`, which is just using a standard Java library function to compute the minimum of its two arguments.

The methods that implement the spill and unspill actions are mostly straightforward. The only slightly tricky detail here is that, because we are saving their values on a stack, we need to make sure that the order in which the registers are restored in `unspillAll()` is exactly the reverse of the order in which they are saved in `spillAll()`. We handle this by making sure that the loop counter in the former counts down, while the loop counter in the latter counts up.

```
    // in Assembly.java:
    public int spillAll(int free) {
        int toSpill = Math.min(free, numRegs-1);
        int bytes   = 0;
        for (int i=1; i<=toSpill; i++) {
            emit("pushl", reg(free-i));
            bytes += Assembly.WORDSIZE;
        }
        return bytes;
    }

    public void unspillAll(int free) {
        int toSpill = Math.min(free, numRegs-1);
        for (int i=toSpill; i>=1; i--) {
            emit("popl", reg(free-i));
        }
    }
```

Notice also that we have arranged for `spillAll()` to return the total number of bytes of data that are pushed on to the stack during the spilling process; this information is used in the `compileExpr()` implementation for `Call` to update the `pushed` count that tracks the total number of bytes that have been pushed on to the stack since the last function call.

3.2 Pushing arguments

Our next task is to describe the code that is responsible for saving the values of the arguments to the call on the stack frame. It is easy to calculate the total number of bytes of stack space that are required for this by using a simple loop over the list of arguments.

```
    static int argBytes(Args args) {
        int bytes = 0;
        for (; args!=null; args=args.rest) {
            bytes += Assembly.WORDSIZE;
        }
        return bytes;
    }
```

(One complication to this scheme would occur if we were working with a more complicated compiler where different argument types might require different amounts of storage on the stack. However, for the simple language that we are working with here, all values fit in a single machine word that is `Assembly.WORDSIZE` (i.e., 4) bytes long.)

Now let's talk about how we're going to get those argument bytes on to the stack! One important detail that was not addressed above is the order in which the arguments in a function call should be evaluated and pushed on to the stack. A natural way to handle the first of these issues is to assume that the arguments will be evaluated from left to right. (Although some language designs intentionally leave this detail unspecified so that different compiler writers can choose other evaluation orders if that enables them to produce better code.) But if we follow the simple approach that was described in class, where each argument is evaluated and then immediately pushed on to the stack, then we will end up with a parameter order in which the first argument is at the highest address instead of the lowest. This is not the correct order for the System V calling convention. And, in fact, we can run a simple test to confirm that other parts of the Mini compiler (specifically the code in static analysis that determines the position of each formal parameter in each function's stack frame) are designed so that the first argument in each call will be placed at the lowest address within the stack frame. To verify this, we can construct and compile the following simple Mini program, annotated on the right with comments that show the generated assembly code.

```
  void mini_main() { }      // compiler requires mini_main

  int foo(int x, int y) {   //    foo: pushl   %ebp
    return x;               //         movl    %esp,%ebp
  }                         //         movl    8(%ebp),%eax
                            //         movl    %ebp,%esp
                            //         popl    %ebp
                            //         ret

  int bar(int x, int y) {   //    bar: pushl   %ebp
    return y;               //         movl    %esp,%ebp
  }                         //         movl    12(%ebp),%eax
                            //         movl    %ebp,%esp
```

```
                        //          popl    %ebp
                        //          ret
```

The two functions `foo` and `bar` shown here are the same except that the former returns its first argument, while the latter returns its second. By inspecting the assembly code, we can see that these arguments appear at offsets `8` and `12` in the stack frame, confirming that the Mini compiler is expecting to use the same argument ordering as the System V calling convention.

It is hard to satisfy both of these constraints—left to right evaluation order and first argument at the lowest address—if we stick to using the `pushl` instruction to add each argument to the stack. And so, instead, we use a different approach that starts by decrementing the stack pointer by the total number of bytes that are needed to hold all of the arguments and then using indexed addressing to fill in each of the values. The examples below contrast these two approaches, assuming a function call of the form `f(x,y,z)`. The code sequences in both columns write the values of the parameters on to the stack with the first parameter at the lowest address. However, the code in the left hand column evaluates the arguments from right to left, while the code on the right evaluates the arguments in the desired left to right order.

```
                          subl $12, %esp
        movl  z, %eax     movl x, %eax
        pushl %eax        movl %eax, (%esp)
        movl  y, %eax     movl y, %eax
        pushl %eax        movl %eax, 4(%esp)
        movl  x, %eax     movl z, %eax
        pushl %eax        movl %eax, 8(%esp)
```

One nice feature of the approach shown on the left is that it is very easy to incorporate the need for padding to meet the alignment restrictions of Mac OS X: the `compileExpr()` implementation for `Call` accomplishes this with the statement: `a.insertAdjust(argBytes+adjust)`. The `insertAdjust()` method produces code that will decrement the stack pointer by the specified number of bytes and the argument ensures that we provide the right amount of space for both the padding and the function arguments.

Following this approach, we can construct the following method in the `Args` class that will evaluate each of the arguments in a given list and push the results onto the stack:

```
    /** Generate code to evaluate each of the arguments in the given list
     *  and save the results on the stack.
     */
    static void compileArgs(Assembly a, int pushed, Args args) {
        for (int offset=0; args!=null; args=args.rest) {
            // compile this argument, writing final value on the stack
            args.arg.compileToStack(a, pushed, 0, offset);

            // compute the offset for the next argument
            offset += Assembly.WORDSIZE;
        }
    }
```

This code uses another new compilation scheme, `compileToStack()`, that we have defined specifically for use in contexts where we are evaluating an expression whose result is supposed to be saved directly on the stack. In general, this will require a sequence of instructions that calculates the value of the expression in a register and then uses a simple `movl` instruction with indexed addressing, `n(%esp)`, to write the result into the appropriate stack location.

```
    // in Expr.java, a default implementation
    public void compileToStack(Assembly a, int pushed, int free, int offset) {
        compileExpr(a, pushed, free);
        a.emit("movl", a.reg(free), a.indirect(offset, "%esp"));
    }
```

In the special case of an integer literal, however, we do not need to use an intermediate register, and can instead write an immediate value directly to the appropriate stack location.

```
    // in IntLit.java, a special case
    public void compileToStack(Assembly a, int pushed, int free, int offset) {
        a.emit("movl", a.immed(Integer.parseInt(num)),
                       a.indirect(offset, "%esp"));
    }
```

Although there are only two cases in this compilation scheme—a default and one special case override—it is a useful, and easy addition to the compiler because it is quite common to see code in which functions are called with integer literals as arguments.

Notice also that the `compileToStack()` call in the `compileArgs()` method shown previously only ever uses a `free` value of `0`. This reflects the fact that our first step in compiling a function was to push all of the registers that were in use at the time on to the stack, which also means that we are now free to use as many of those registers as we need to compute the values of arguments. This may help to avoid the need for spilling while we are evaluating the arguments, which might even compensate a little for the cost of having to save the registers at the start of the call.

(You might also wonder: if `free` is always zero in every call to `compileToStack()`, then why do we bother passing it in as a parameter instead of just fixing the value to `0` in the body of `compileToStack()`? Certainly, it would be possible to do that, but for the time being we have kept the `free` parameter in place in case we find other uses for `compileToStack()` later on where the special case of `free=0` does not apply.)

3.3 Calling the function

With the argument values in place and the stack appropriately aligned, we are now ready to transfer control to the function that is being called, which just requires a standard `call` instruction. The calling convention that we are using assumes that the result of the function will be in the `eax` register when the function returns. But to meet the specification of the `compileExpr()` compilation scheme that we are using, we need to ensure that the result is placed in the specified `free` register. Fortunately, it is very easy to handle this by emitting an extra `movl` instruction in cases where the `free` register is not `eax`, as shown in the following implementation:

```
    // in Assembly.java:
    public void call(String lab, int free) {
        emit("call", name(lab));
        if ((free%numRegs)!=0) {
            emit("movl", reg(0), reg(free));
        }
    }
```

(In fact, as a special case, it would be valid to skip this final `movl` instruction if we knew that the function being called did not return a value (because it was declared as having `void` return type) or because its value was not going to be used (because the function call expression appears in a context, such as a statement, where the final result is discarded). It would not be very difficult for us to detect these conditions during static analysis and to record this information in a flag stored in each `Call` node so that it can be retrieved and used during code generation, but we have not implemented that feature in the code shown here.)

Once the call is over and the result is in the correct register, we need only adjust the stack to remove the argument values and padding (by calling `a.removeAdjust(argBytes+adjust)`), and then restoring the spilled registers (by calling `a.unspillAll(free)`), and then our task is finally done!

4 A final example

We end this document with a final example that shows a fragment of Mini source code together with corresponding IA32 assembly code (in comments on the right hand side) that has been generated using the compilation rules described above.[1]

```
void mini_main() {       // _mini_main:
    print foo(1,2);      //      pushl   %ebp
                         //      movl    %esp,%ebp
                         //      subl    $24,%esp
                         //      movl    $1,(%esp)
                         //      movl    $2,4(%esp)
                         //      call    _foo
                         //      addl    $20,%esp
                         //      pushl   %eax
                         //      call    _print
    print bar(3,4);      //      subl    $16,%esp
                         //      movl    $3,(%esp)
                         //      movl    $4,4(%esp)
                         //      call    _bar
                         //      addl    $20,%esp
                         //      pushl   %eax
                         //      call    _print
    print baz(5,6,7);    //      subl    $16,%esp
                         //      movl    $5,(%esp)
                         //      movl    $6,4(%esp)
                         //      movl    $7,8(%esp)
                         //      call    _baz
                         //      addl    $20,%esp
                         //      pushl   %eax
                         //      call    _print
                         //      movl    %ebp,%esp
                         //      popl    %ebp
}
```

The reader is encouraged to study this example carefully because it illustrates many of the points that have been described previously in a more abstract setting. Clearly, the source program includes three distinct function calls, but in fact the Mini `print` statement is also implemented by generating a special, single

[1]This particular sample was generated using the settings for Mac OS X, so it ensures appropriate alignment at each function call, but essentially the same code sequence would work on any of the platforms that we are interested in here.

argument function that is referenced here as `_print`. The `compileExpr()` implementation for the `Print` class uses a simple `pushl` instruction, which is why you can see a mixing of the two styles in the code above.

Another small detail is that the implementations of the `insertAdjust()` and `removeAdjust()` methods in the `Assembly` class—which conceptually just output instructions to decrement or increment the stack pointer by some fixed number of bytes—have actually been written in a way that allows adjacent stack adjustments to be combined in to a single instruction. This prevents the compiler from generating assembly code that includes two consecutive stack adjustment instructions such as `addl $4,%esp; subl $12,%esp`, which would instead be combined in to a single `subl $8,%esp` instruction. This is a simple example of a *peephole optimization*, and we will be seeing more examples of those (and other optimization techniques) in the second half of the term ...