COMP 520: Compilers
Compiler Project – Assignment 4

Assigned: Thu Apr 7, 2022
Due: Wed Apr 20 (11.59 PM)

The fourth and final checkpoint in your compiler requires the construction of a code generator for miniJava that targets the *mJAM* abstract machine. mJAM extends and modifies TAM (described in Appendix C of our text), to more conveniently support miniJava execution. In PA4, a miniJava program that passes syntactic and contextual analysis according to PA3 should generate an mJAM program that, when executed by the mJAM interpreter, has Java execution semantics (provided it does not exhaust mJAM's limited resources).

The code generator can be implemented as an AST visitor making a single traversal of the AST, generating runtime entity descriptions (RED) for all declarations, and emitting instructions using the mJAM code generator interface.

1. miniJava modifications

a) The `length` field for arrays should be added to miniJava. The compiler should enforce that the `length` field cannot be assigned, it can only be read.

b) PA4 should check that the miniJava program has a unique **`public static void`** `main` method with the correct `String []` parameter. If this is not the case in the miniJava program, your compiler should issue an error and terminate via `exit(4)`.

c) (i) If you have not already done this in PA3, add a check in contextual analysis that the last statement in a non-void method is a **`return`** statement If not, issue an error and terminate via `exit(4)`. (ii) For a void method, no error should be issued if the last statement in the program is not a return statement. However the mJAM code for a **return** statement should be generated at the end of the program, if not already present.

2. The target machine mJAM

The mJAM package will be available through our course web page. mJAM differs from TAM (the Triangle Abstract Machine) in the following respects.

(1) mJAM does not use or maintain static links, because Java does not have nested procedure declarations. Thus registers L1, … , L6 are not part of mJAM, and CALL instructions specify only a target code address and do not require a static link argument.

(2) mJAM has a register OB (Object Base) that holds the value of **`this`** (i.e. the address of the object instance in the heap) when executing an instance (i.e. non-static) method.

(3) The CALL instruction is used to execute a static method at a known address. Arguments are passed on the stack. The mJAM primitives (see below) are also invoked using the CALL instruction.

(4) The CALLI instruction is used to execute an instance method at a known address. It replaces the indirect call method of TAM which is not needed in miniJava. CALLI expects arguments to the method to be on the stack followed by the address of the object instance. Thus, to call *x.foo*(..) with *k* parameters, the *k* argument values followed by the

address of *x* should be on the stack, and the CALLI instruction should specify the code address of *foo*. The CALLI instruction pops the instance address from the stack, builds the callee activation record directly above the arguments, saving the caller OB, LB and return address within it, and sets LB to the start of the activation record and OB to the instance address for the called method. OB can be used as the value of **`this`** during execution of *foo*.

(5) The CALLD instruction is used for dynamic method invocation according to the dynamic type of an instance. It is only needed in the presence of inheritance. Since this is not part of the basic miniJava project, CALLD need not be used and the data structures it requires at runtime need not be generated.

(6) The "RETURN (*n*) *d*" instruction should be used to return from a method. The same RETURN instruction can be used regardless of how the method was invoked (CALL, CALLI, or CALLD). RETURN removes the activation record (frame) of the current method, restoring the caller's OB and LB, then pops *d* arguments off the caller stack (recall the instance address does not count as an argument). If the method is non-void, it pushes the result on the stack.

(7) The displacement field *d* in mJAM may assume the full range of Java **`int`** values. Thus any integer can be pushed on the stack with a LOADL instruction.

(8) mJAM includes all TAM primitive operations as well as the additions below that provide support for miniJava objects and arrays. mJAM primitive operations are static methods and are *not* passed an implicit instance on the stack.

- The *putintnl* primitive prints the integer at stacktop with a newline at the end. miniJava's `System.out.println(…)` (defined in the standard environment) can be implemented using this mJAM primitive.

- The *newarr* primitive has stack operands and result as follows:

  …, number of elts *n* → …, addr of new array in heap

  newarr allocates *n* + 2 words on the heap, initializes the first word to -2 (to indicate this is an array), and the second word to *n* (the array length). All remaining words are elements of the array and are initialized to zero. The result is the address of the first element of the array.

- The *arraylen* primitive has stack operands and result as follows:

  …, addr of array *a* in heap → …, length of array *a*

  Execution of the instruction will fail with an invalidHeapRef or nullPointer exception if the address does not refer to an array. The result corresponds to `a.length`.

- The *arrayref* primitive has stack operands and result as follows:

  …, addr of array object *a*, element index *i* → …, value of *a*[*i*]

  The result is the value of the ith element ($0 \leq i <$ array length) in the array starting at address *a*. Execution of the instruction will fail with an invalidHeapRef or nullPointer exception if the array address does not refer to an array, or an array index error when *i* is not a valid index in *a*.

- The *arrayupd* primitive has stack operands and result as follows:

  …, addr of array object *a*, element index *i*, new value *v* → …

The three arguments are consumed and no result is produced. However, the *i*th element of *a* is set to *v*. Execution of the instruction will fail with an invalidHeapRef or nullPointer exception if the array address does not refer to an array, or an array index exception when *i* is not a valid index in *a*.

- The *newobj* primitive has stack operands and result as follows:

  …, addr of class object *a*, size (# of fields) of object *n* → …, addr of new obj

  newobj allocates *n* + 2 words on the heap, with the first word initialized to the address of the class object, and the second word initialized to *n*, and remaining words set to zero. Without inheritance, no class object is needed, and so its value should be specified as -1. The result of the primitive is the address of the allocated object.

- The *fieldref* primitive has stack operands and result as follows:

  …, addr of object *a*, field index *i* → …, value of *a*.*i*

  The result is the value of the ith field of the object at address *a* (where $0 \le i <$ #fields). Execution will fail with a nullPointer error if the object address is null, or with an array index error when $i < 0$ or #fields $\le i$.

- The *fieldupd* primitive has stack operands and result as follows:

  …, addr of object *a*, field index *i*, new value *v* → …

  All three arguments are consumed and no result is produced, but the *i*th field of *a* is set to value *v* (where $0 \le i <$ #fields). Execution will fail with a nullPointer exception if the object address is null, or with an array index exception when $i < 0$ or #fields $\le i$.

3. Compiler operation

As in previous checkpoints, your compiler should accept a source file to be compiled as a command line argument. Given an input file, say `foo.java`, holding a valid miniJava program (i.e. passing syntactic and contextual analysis), the compiler should write out object file `foo.mJAM` (using the `ObjectFile` class in mJAM) and terminate using `System.exit(0)`. If the compiler discovers an error in the input program during any phase of the compiler, a relevant diagnostic message should be issued, the compiler should terminate using `System.exit(4)`, and no object file should be written.

4. mJAM execution

To execute the mJAM program in object file `foo.mJAM`, use

```
java mJAM/Interpreter foo.mJAM
```

A `putintnl` instruction in mJAM called with e.g. value 15 on the stack will produce

```
>>> 15
```

as a separate line on the console (stdout). This will help differentiate the output from your miniJava program from any output produced by the interpreter (e.g. from `halt (4)` instructions, or when run in debug mode).

To debug an mJAM program, first generate assembler code for the object file by running the command `java mJAM/Disassembler foo.mJAM`, which will generate the file `foo.asm`. Then invoke the interpreter providing both the object code and the assembler code files as inputs as follows:

```
java mJAM/Interpreter  foo.mJAM  foo.asm
```

The debugger command prompt will appear with the program at instruction 0. The available options will be listed in response to the command “?”.

4. Submission instructions

A miniJava test program PA4Test.java is available on our website that may be used to incrementally develop and test your code generator. The submission instructions for this checkpoint will be available online closer to the due date.