COMP 520: Compilers
Compiler Project – Assignment 4 (complete)

Assigned: Thu Mar 21
Due: Thu Apr 11 **extended to** Tue Apr 16

The fourth checkpoint in your compiler is to construct a code generator for miniJava targeting *mJAM*, an abstract machine like TAM (as described in Appendix C of our text), with a few modifications to more directly support miniJava execution. The code generator can be implemented using the AST visitor interface to make a single traversal of the AST, generating runtime entity descriptions (RED) for all declarations top-down, and emitting instructions using the RED and the mJAM code generator interface bottom-up. A program passing miniJava contextual analysis should be compiled to an mJAM program that, when executed, has Java execution semantics. As a consequence of our PA1-PA3 restrictions, a valid program will have a unique main method where execution should start, and on termination of this method, execution should halt.

1. miniJava modifications

The length field of arrays should be added and implemented.

2. Target machine: mJAM

The mJAM package will be available through the course web page. mJAM differs from TAM in the following respects.

(1) mJAM does not use or maintain static links, because they are not needed for a language that does not permit nested procedure declarations. Thus registers L1, … , L6 are not part of mJAM, and the CALL instruction does not require a static link argument.

(2) mJAM has a new register OB (Object Base) that holds the value of **`this`** (i.e. the address of the object instance) during execution of a non-static method.

(3) The CALL instruction performs a method call and requires the address of the object instance at stack top. CALL saves the OB register of the caller in the callee activation record and sets OB to the address at stack top. The remainder of its functionality follows the TAM CALL instruction (except that it does not use or save a static link). To call a method *x.foo*(..) with *k* parameters, the *k* argument values followed by the address of *x* should be on the stack when the CALL instruction specifying the code address of *foo* is executed. OB can be used as the value of **`this`** during execution of *foo*.

(4) The RETURN instruction performs method return much like the TAM RETURN. In addition it restores the caller's OB.

(5) 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.

(6) The method invocation `System.out.println(int x)` (defined in the standard environment) should be implemented using the mJAM primitive `putint`.

(7) mJAM primitive operations are essentially static methods and are *not* passed an address of an object on the stack. mJAM has all TAM primitive operations as well as the following additions:

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

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

newarray 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 *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) of the array starting at address *a*. Execution of the instruction will fail with a nullPointer exception if the array object address is null, or an array index error when *i* is not a valid index or *a* does not refer to an integer array.

- 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*. The execution of the instruction will fail with a nullPointer error if the array object address is null, or with an array index error when *i* is not a valid index or *a* does not refer to an array.

- 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. In this assignment you need not implement inheritance, and hence the class object (which is used to dispatch the correct virtual method) is never used, so its address can be -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 $0 \leq i <$ #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 $\leq i$.

- 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 $0 \leq i <$ #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 $\leq i$.

3. Compiler operation

Your compiler should accept a source file to be compiled as a command line argument. The file name may have a `.java` or `.mjava` extension. Given an input file, say `foo.java`, then if the file contains a valid miniJava program (i.e. passes 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 syntactic or contextual analysis, a relevant diagnostic message should be issued and the compiler should terminate using `System.exit(4)`. No object file should be written in this case.

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

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

A `putint` instruction in mJAM called with e.g. value 15 on the stack will produce ">>> 15\n" 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 using 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:

```
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. Sample miniJava Programs

The PA3 `passxxx.java` testcases for xxx > 340 can be used to exercise miniJava features. The features go from simple to complex with increasing value of xxx and can be used to incrementally develop and test the code generator.