Miniduke – MiniJava compiler

Peter Boström – *pbos@kth.se*

10 juni 2012

Introduction

`miniduke` is a compiler for MiniJava defined by Appel, but with some minor changes introduced by the compiler-construction course followed.

`miniduke` is written in C and uses the lexical-analyzer generator flex and the parser generator GNU bison (aka Bison). Compared to other java-based compilers used in the course, `miniduke`

Building

Building `miniduke` requires `flex` and `bison` to be installed (apart from gcc). Running `make` generates intermediate files from flex and Bison and finally compiles them with gcc into `miniduke`.

Invocation

Running `miniduke` can be done as follows:

```
miniduke [file.java]
```

Use flags `-o dir` to change output directory and `-S` to only generate the final assembly.

For Tigris-submission purposes `miniduke` reads and ignores the argument `JVM`.

Tests

Apart from other students' tests, tests were written simultaneously during `miniduke`'s development. When writing code checking that the main method is called main, it's nice to write a corresponding test case which confirms its validity.

Lexical Analysis

flex is used to generate a lexical analyzer which feeds parsed tokens into the parser generated by Bison. Comments, multiline comments, spaces, newlines

etc. which do not affect the code whatsoever are simply skipped and never fed into the parser.

Keywords, such as `if`, or even less obvious MiniJava tokens such as `System.out.println` are fed into the parsers as single tokens. It's important to note that *main* is not parsed as a single token, but rather as an identifier. This is because in contexts where *main* does not identify the main method of a program it needs to be parsed as an identifier. Thus the job of rejecting main methods not called main requires an additional check manually performed by the parser.

The lexer parses the entire file and feeds it to the parser token-per-token.

Parsing

The parse step uses a bison-generated parser from a grammar file similar to the course grammar. Some additional intermediate rules are added to be able to express the language (expressions such as Var* become VarList, as Bison doesn't support repetition). One intermediate rule is added to catch the creation of multidimensional arrays which are not supported in MiniJava. Otherwise the parser would interpret `new int[3][2]` as `(new int[3])[2]` which differs from Java behavior.

This generated parser generates an Abstract Syntax Tree (AST) directly, and no CST is ever built. This method avoids the extra code needed to transform a CST into an AST. For each node generated by the parser, a linenumber from the lexer is stored. This is to be able to identify error locations later. For instance, during typecheck, the code piece `int i; i = true;` would be rejected, but without approximate line numbers for the `i = true;` statement node, debugging the program could be way more hard.

If the parser recieves a token which doesn't match the language, it generates an error.

Semantic Analysis

Symtab

Simply traverses the AST and generates a symbol table. The program symtab contains a mainclass and optionally other classes (depending on the program). Each class contains fields and methods. Methods contain parameters and local variable declarations.

Typecheck

Binds identifiers to symtab methods, classes, fields, parameters or locals to their symtab respectives. During this step, duplicate declarations or locals with the same name as other parameters etc. throw errors with linenumbers.

Binding identifiers to their respective symtab equivalents means that any subsequent lookups can be done really quickly (in constant time).

AST Binding

During "ast binding", all expression nodes are assigned a type. Leaf nodes such as method calls, constants, identifiers etc. are assigned their corresponding type. If the types of operands are valid for an operation (say `<` with operands as `int`), and the operation's expression node is assigned its corresponding type (our example `<` always yields a `boolean` result).

Any incorrect assignments or operations are caught in this step. Code may of course still have bugs (such as stack overflows etc. or uninitialized variables which is not checked by our compiler).

Jasmin

Finally it's time to output Jasmin assembly. Jasmin is an assembler for the Java Virtual Machine. Any code reaching this step will compile, as no errors are thrown during Jasmin generation (last checks are performed during AST binding).

As Jasmin (and the JVM) is stack-based, code is easily generated by parsing the AST depth first. Each expression is responsible for putting its corresponding value on the top of the stack. This means that the code (a) $<$ (b) is generated as follows: put code for generating (a), put code for generating (b), then a compare instruction with corresponding jump takes the two top values, which are the return values from (a) and (b) and jump corrispondingly. Its jump-to labels have codes for putting a boolean true-or-false value on the stack respectively.

For simplicity's sake `miniduke` generates a boolean on the stack for each boolean expression (`if(a < 3)` doesn't have to generate a boolean value, you can simply jump to the corresponding block or else-block depending on the condition). Always generating a boolean value on the stack means that `miniduke` can handle if-cases and assigning boolean expressions uniformly. if- cases take a boolean from the top of the stack and jump depending on its value, while assignments simply takes the top value from the stack and stores it.

It's important to note that Jasmin can have issues with instructions appearing as different kinds of variable names. So any method-, field- or class name needs to be escaped properly (surrounded with '). `getfield goto/goto I` doesn't work, while `getfield 'goto/goto' I` does.

Results

`miniduke` turned out to be a very quick compiler. All test cases compiled in Tigris in a few seconds at most. This compared to other implementations done in Java which took 15-30 minutes. Even if the comparison isn't really fair (with a lot of this being startup time for the JVM, but still it's a nice boast.

It's considerably (way-way) faster than `javac` or compiling a short C program in `gcc`, though that comparison isn't really fair, as any commercial-grade compiler will perform a lot more analysis and generate way more efficient code in a lot of cases, but it was nice to have the programs compile almost instantly. Most of the time spent running the tests was either on execution or assembling the Jasmin source, and not compiling the actual source code.

Feedback

The following section is for feedback and general thoughts of Pedro de Carvalho Gomez' MiniJava-compiler submission and follows the layout of his report.

Lexing

Unsurprisingly JFlex' format looks similar to Flex. The format is annotated in a way which is really easy to follow. It's also well commented.

The use of `yytext()` values for symbols other than IDs and int/longs are slightly confusing, as there is a constructor which takes no value, which should be appropriate for reserved words and operators, where the text doesn't differ. If the value variable was required, then the symbol method without value declared could have used yytext by default. This keeps the method calls smaller and easier to read.

Solving integer length in the lexing step is good. I would suggest matching `[1-9][0-9]*` generating the error from `{code blocks}` instead of having a complex regex matching it. The error could easily be generated if `Integer.parseInt()` fails to parse the int instead of having two cases for good ints and overflowing ones.

I would also suggest a simple `private void dbg(String msg);` method for all the optional debug printing, `dbg("CLASS");` doesn't interfere as much with readability as all these if-cases, and more importantly it's best to keep code DRY.

Parsing

Parsing is also straightforward. I like the uniform handling of `this` as any other variable and not generating a special node for it. If some of the typecasting could have been avoided (Exp) ex1, by annotating types for all rules, it would have been nicer. But it's really simple and generates a natural AST structure.

The parser also has an interesting technique of unreserving main as a keyword, by having an ID-identifying rule match both the terminal ID and the terminal MAIN, they both generate expression IDs. As main isn't really a reserved keyword, I would have preferred this to remain an id, have the MainClass rule match ids, and assert that the public-static main method is really called main. This would more naturally resemble that it really isn't a keyword. Still, it was a nice trick to see done none the less.

The precedence declaration for `if-else` is really clean and is declared just like operator precedence.

Semantic Analysis

Symbol-table structure also seems straightforward. A class represents a class, method a method, so on, and so forth.

Instead of doing `if(found_error)` checks, the typecheck could have thrown exceptions etc. This would have avoided generally unhelpful errors like "Found errors. Leaving". When not using exceptions however, the compiler could have done more checks and reported more errors before exiting (e.g. only check for errors when the typecheck returns). Exceptions not being thrown relates mostly to the TCVisitor code at least. A more top-level catch could handle printing of the errors and not have such nested code.

The isCompatible method comparison is clean. It could have been made a tad shorter by doing `return a.equals(b);` and `return isSubtype(a,b);` respectively.

The name 'accept' is a bit confusing. It's a bit hard at first to understand what's going on with calls like `n.sl.elementAt(i).accept(this);` because accept isn't really descriptive of anything.

The code for semantic analysis is well structured, albeit a bit long. Code that checks if both operators are of a certain type could have been put into a method to avoid code repetition. Something like `assert_type(expl, "int", expr, "int", cur_class, cur_method);` for example would do. Rewrites like this would reduce code size drastically. The typechecks for many operators are similar, so clumping $<$, $<=$, $>$, $>=$ into a tighter check would be a good idea. It would also make the code more maintainable and easier to change. As of now, the classes TCVisitor, SymTable and SymVisitor are 1700 lines, which could probably be at least halved, giving easier-to-grasp code.

Instruction Selection/MiniJava

The separation of JasminClass and InstVisitor was appreciated. The idea of separating utilities from the actual visitor was nice. Otherwise the visit-model is easy to follow, except for the usage of 'accept', which takes some time to understand what it does.

One upper bound for calculating the maximum size of the stack can be done by checking AST depth. This was not done by myself either though, but suggested by Torbjörn.