Department of Information and Computing Sciences
Utrecht University

INFOB3TC – Assignment P3

Andres Löh

Deadline: Tuesday, 25 January 2011

The goal of this assignment is to write a code generator for a subset of Java. The target language is the "Simple Stack Machine" (SSM), a virtual machine for which a graphical simulator is available.

You are given a working framework. The task is to extend the compiler with new functionality, as specified and explained below.

Parser combinators

For this task, you once more need the parser combinators as discussed in the lectures, i.e., you need the package `uu-tc` as provided on the course Wiki.

General remarks

Here are a few remarks:

- Make sure your program compiles (with an installed `uu-tc` package). Verify that invoking `ghc --make -O Main.hs` works prior to submission. If it does not, your solution will not be graded.
- Include *useful* comments in your code. Do not paraphrase the code, but describe the structure of your program, special cases, preconditions etc.
- Try to write readable and idiomatic Haskell. Style influences the grade! The use of existing higher-order functions such as *map*, *foldr*, *filter*, *zip* – just to name a few – is explicitly encouraged. The use of all existing libraries is allowed (as long as the program still compiles with the above invocation).
- Copying solutions from the internet is not allowed. Teamwork in teams of 2 members is mandatory, exceptions only with approval from the teacher.
- Textual answers to tasks can either be included as comments in a source file, or be submitted as text or PDF files. Word documents are not accepted!

Submission Submit a zip file containing the sources (not the compiled program) via submit. *Important:* Include a `README` text file that describes where you have changed what, and what ecactly you did. Also make sure you document all your changes in the sources at the appropriate places.

Acknowledgements

This assignment is heavily inspired and to a large extent copied from an assignment Johan Jeuring has been using.

Structure

The main part of this assignment is a documentation of the given framework. In the very end, several tasks are listed.

The following files/modules are given:

- `JavaLex.hs`: a lexical scanner for Java that transforms flat input (a string) into a list of tokens.
- `JavaGram.hs`: types and functions for parsing Java.
- `JavaAlgebra.hs`: an algebra for the Java grammar, and a corresponding *fold* function.
- `SSM.hs`: types and utilities for representing SSM programs.
- `JavaCode.hs`: the code generator as an algebra, to transform Java abstract syntax into SSM code. In its current form, the code generator does not yet work correctly. It will be one of your tasks to extend the algebra in order to make the code generator more useful.
- `Main.hs`: main program that contains a driver calling the different phases in the right order. The program reads a Java file and writes an SSM result.
- `ssmui.jar`: graphical simulator for the SSM. With this, you can run the generated code and test whether your code generator is working correctly.
- `ssm.bat/ssm.sh`: wrapper script to call the Java interpreter with `ssmui.jar`.

SSM

The SSM architecture will be explained in more detail in the next lecture. Some instructions can also be found on the SSM homepage:

`http://people.cs.uu.nl/atze/SSM/index.html`

Lexical analysis

The goal of this phase, implemented in module `JavaLex.hs`, is to split the input into a sequence of *tokens*, and to discard irrelevant information such as whitespace. Lexing is a simple form of parsing, and while we could implement a DFA to perform the lexing, we use the parser combinators because they are more convenient.

Tokens are given by the following Haskell datatype:

```
data Token = POpen     | PClose    -- parentheses ()
           | SOpen     | SClose    -- square brackets []
           | COpen     | CClose    -- curly braces { }
           | Comma     | Semicolon
           | KeyIf     | KeyElse
           | KeyWhile  | KeyReturn
           | KeyTry    | KeyCatch
           | KeyClass  | KeyVoid
           | StdType   String      -- the 8 standard types
           | Operator  String      -- the 15 operators
           | UpperId   String      -- uppercase identifiers
           | LowerId   String      -- lowercase identifiers
           | ConstInt  Int
           | ConstBool Bool
           deriving (Eq, Show)
```

There are 16 proper terminal symbols, for single symbols such as braces or keywords such as “while”. There are also pseudo-terminals with extra information attached, such as the name of a standard type, an operator, a variable or a literal.

To associate the proper nonterminals with their textual representation, we use an association list:

```
terminals :: [(Token, String)]
terminals =
  [(POpen     , "("     )
  ,(PClose    , ")"     )
  ,(SOpen     , "["     )
  ,(SClose    , "]"     )
  ,(COpen     , "{"     )
  ,(CClose    , "}"     )
  ,(Comma     , ","     )
  ,(Semicolon , ";"     )
  ,(KeyIf     , "if"    )
  ,(KeyElse   , "else"  )
  ,(KeyWhile  , "while" )
  ,(KeyReturn , "return")
  ,(KeyTry    , "try"   )
```

```
    ,(KeyCatch  ,"catch" )
    ,(KeyClass  ,"class" )
    ,(KeyVoid   ,"void"  )
    ]
```

This list is then used to create a parser that can choose between any of these terminals:

```
lexTerminal :: Parser Char Token
lexTerminal =
  choice (map (λ(t,s) → const t <$> keyword s) terminals)
```

The *keyword* function is defined as follows:

```
keyword :: String → Parser Char String
keyword []                   = succeed ""
keyword xs@(x: _) | isLetter x = do
                                   ys ← greedy (satisfy isAlphaNum)
                                   guard (xs == ys)
                                   return ys
  | otherwise                = token xs
```

If passed a string that starts with a letter, it consumes alphanumeric characters greedily as long as present in the input, then compares with the desired keyword. This prevents that sequences of input such as `"classX"` can be lexed as the keyword **class** followed by the identifier *X*. For symbolic tokens, we revert to the standard *token* parser defined in the library.

For those pseudo-terminals that represent a list of possibilities (the 8 standard types and 15 operators), we use a utility function that, given the constructor of the pseudo-terminal and a list of possible strings, builds the parser:

```
lexEnum :: (String → Token) → [String] → Parser Char Token
lexEnum f xs = f <$> choice (map keyword xs)
```

We can't use the above function to handle the token types that can represent a wide class of terminals such as identifiers. Here, we use an approach based on *satisfy*. We also use *greedy* rather than *many* to make sure that a connected string of letters will under no circumstances be interpreted as two separate identifiers:

```
lexLowerId :: Parser Char Token
lexLowerId = (λx xs → LowerId (x : xs)) <$>
               satisfy isLower <*> greedy (satisfy isAlphaNum)
lexUpperId :: Parser Char Token
lexUpperId = (λx xs → UpperId (x : xs)) <$>
               satisfy isUpper <*> greedy (satisfy isAlphaNum)
lexConstInt :: Parser Char Token
lexConstInt = (ConstInt . read) <$> greedy1 (satisfy isDigit)
```

Finally, we can combine all the different functions to create a parser for a single token:

```
stdTypes :: [String]
stdTypes = ["int","long","double","float",
            "byte","short","boolean","char"]
operators :: [String]
operators = ["+","-","*","/","%","&&","||",
             "^","<=","<",">=",">","==",
             "!=","="]
lexToken :: Parser Char Token
lexToken = greedyChoice
             [lexTerminal
             ,lexEnum StdType stdTypes
             ,lexEnum Operator operators
             ,lexConstInt
             ,lexLowerId
             ,lexUpperId
             ]
```

The function *greedyChoice* is a greedy variant of choice, defined as

```
greedyChoice :: [Parser s a] → Parser s a
greedyChoice = foldr (≪|>) empty
```

The order in which operators are mentioned in *operators* is important. For instance, `">="` must occur before `">"`, otherwise the sequence `">="` might be interpeted as the operator `">"` followed by the operator `"="` rather than as a single operator. For similar reasons, the keyword parser *lexTerminal* and the standard type parser must occur before *lexLowerId*, because a keyword such as `"class"` or a type such as `"int"` could also be interpreted as lowercase identifiers.

Now that we have a parser for a single token, we have to create a parser for a list of tokens. Tokens may be separated by whitespace, so we define a parser to consume whitespace:

```
lexWhiteSpace :: Parser Char String
lexWhiteSpace = greedy (satisfy isSpace)
```

The main lexical scanner then consumes a list of tokens, where each token may be followed by whitespace, and additional whitespace may occur before the first token. All the tokens are collected, the whitespace is discarded:

```
lexicalScanner :: Parser Char [Token]
lexicalScanner = lexWhiteSpace *> greedy (lexToken <* lexWhiteSpace) <* eof
```

In this module, we also define a few functions that are useful for the actual Java parser. In the context-free parser, that is run after the lexical scanner, the tokens play the role of the symbols. For instance, a semicolon is lexed using *symbol* `';'`, but parsed using *symbol Semicolon*. We define an abbreviation for this:

```
sSemi :: Parser Token Token
sSemi = symbol Semicolon
```

For pseudo-nonterminals we define abbreviations using *satisfy*. For example, here is a parser recognizing any standard type:

```
sStdType :: Parser Token Token
sStdType = satisfy isStdType
  where isStdType (StdType _) = True
        isStdType _           = False
```

There are similar functions *sUpperId*, *sLowerId*, *sConst* and *sOperator*.

Context-free parsing

We are going to use the following grammar for Java, where *class* is the start nonterminal:

```
class       ::= class upperid { member* }
member      ::= decl ; | method
method      ::= typevoid lowerid ( decls? ) block
decls       ::= decl | decl , decls
decl        ::= type lowerid
block       ::= { statdecl* }
statdecl    ::= stat | decl ;
stat        ::= expr ;
              | if ( expr ) stat else?
              | while ( expr ) stat
              | return expr ;
              | block
else        ::= else stat
typevoid    ::= type | void
type        ::= type0 ([ ])*
type0       ::= stdtype | upperid
expr        ::= exprsimple
              | exprsimple operator expr
exprsimple  ::= const | lowerid | ( expr )
```

Terminals are written in typewriter font, nonterminals in italics. The nonterminals *upperid*, *lowerid*, *operator*, and *const* are referring to the corresponding pseudo-tokens.

Convince yourself that the file `JavaGram.hs` contains an abstract syntax and parser for the context-free grammar above.

Note that the names of classes, methods, variables etc. are stored as values of type *Token* in the abstract syntax. For example, the abstract syntax for *member* is defined as

```
data Member = MemberD  Decl
            | MemberM Type Token [Decl] Stat
  deriving Show
```

The single production for *method* has been inlined, and the *lowerid* is represented as *Token*. This also means that for instance a variable `"x"` in an expression is represented as *ExprVar* (*LowerId* `"x"`). The double constructors are in this case a bit superfluous, but are useful in the case of constants, where integer and boolean constants can both be handled via the constructor *ExprConst*.

Note also that in *Member* above, the nonterminal *typevoid* is represented as *Type*. The distinction whether `void` is allowed or not is only checked by the parser, but not reflected in the abstract syntax. There are several such small simplifications in the abstract syntax, for instance *block* is represented as *Stat*. The simplifications lead to a slightly simpler algebra type and fold function. We will discuss these next.

Algebra and fold

For the semantic functions, we are going to use a fold over the abstract syntax of Java. The definition of the algebra type and the function itself are given in the file `JavaAlgebra.hs`. They are following the general scheme for the algebras and folds of systems of datatypes. We let our algebra range over classes, members, statements and expressions. We could easily include more nonterminals (declarations and types, for instance), but these four are enough to get started.

Simple Stack Machine

In order to generate target code for the simple stack machine, we first have to know the structure of its language. We represent the abstract syntax of SSM code as a Haskell datatype, and create SSM code by producing values of that datatype, and then printing that value.

The abstract syntax and the printer are defined in `SSM.hs`. The structure of an SSM program is simple – it is a list of instructions:

```
type Code = [Instr]
```

There are 44 possible instructions, represented as constructors of the datatype *Instr*:

```
data Instr
  = STR Reg | STL Int  | STS Int  | STA Int     — Store from stack
  | LDR Reg | LDL Int  | LDS Int  | LDA Int     — Load on stack
  | LDC Int | LDLA Int | LDSA Int | LDAA Int    — Load on stack
  | BRA Int | Bra String                 — Branch always (relative/to label)
  | BRF Int  | Brf String              — Branch on false
  | BRT Int  | Brt String              — Branch on true
  | BSR Int  | Bsr String              — Branch to subroutine
  | ADD  | SUB | MUL | DIV | MOD       — Arithmetical ops on 2 operands
  | EQ   | NE  | LT |  LE  | GT | GE
                                       — Relational ops on 2 operands
  | AND  | OR  | XOR                   — Bitwise ops on 2 operands
  | NEG  | NOT                         — Bitwise ops on 1 operand
  | RET  | UNLINK | LINK Int | AJS Int — Procedure utilities
  | SWP  | SWPR Reg | SWPRR Reg Reg | LDRR Reg Reg   — Various swaps
  | JSR  | TRAP Int | NOP | HALT       — Other instructions
  | LABEL String   — Pseudo-instruction for generating a label
  deriving Show
```

Some instructions have parameters. This is either a number (*Int*), a register (*Reg*), or a label (*String*). For registers, the datatype *Reg* is defined:

```
data Reg = PC | SP | MP | R3 | R4 | R5 | R6 | R7
  deriving Show
```

For the case that you prefer to address all registers by number, abbreviations r_0 to r_7 are defined, in particular

$r_0 = PC$
$r_1 = SP$
$r_2 = MP$

The branch instructions are parametrized by a relative offset, or by a label name. Labels in the code can be specified using the pseudo-instruction `LABEL`. The final translation phase (the assembler) will resolve all labels and fill in addresses.

The printer for SSM code is given by the function *formatCode*. It uses the derived *show* function as a basis, and postprocesses the resulting output a bit. Parentheses and string quotes are removed, and all instructions except for labels are indented for better readability.

In order to compute the offsets for jumps, we have to be able to calculate the length that a piece of SSM code takes up in memory. Because of the different numbers of parameters, not every instruction has the same size. For this purpose, we define *codeSize* and *instrSize* as follows:

```
codeSize :: Code → Int
codeSize = sum . map instrSize

instrSize :: Instr → Int
instrSize (LABEL _  ) = 0   — pseudo-instruction, removed by assembler
instrSize (LDRR  _ _) = 3   — two instructions of size 3
instrSize (SWPRR _ _) = 3
instrSize (BRA   _  ) = 2   — 20 instructions of size 2
...
instrSize (SWPR  _  ) = 2
instrSize _           = 1   — the rest has size 1
```

Code generation for Java

Everything is available now to generate code for Java. We define an algebra for this purpose. In the beginning, we specify the types of the results we want to generate, and we do so for each of the four types we included in the algebra: classes, members, statements, and expressions. For the start symbol *Class* this is *Code*, because SSM code is what we want to generate. For members and statements we also choose *Code*. For expressions, we also want to generate code, but it turns out that sometimes we need to know the value, and sometimes we need to know the address of an expression. We therefore need some input of type

```
data ValueOrAddress = Value | Address
  deriving Show
```

before we can produce code. Therefore, the type of our code generation algebra becomes

```
codeAlgebra :: JavaAlgebra Code                       — result type for Class
                           Code                       — result type for Member
                           Code                       — result type for Stat
                           (ValueOrAddress → Code)    — result type for Expr
```

The algebra is a tuple of functions, so the algebra definitely has the following form:

```
codeAlgebra =
  ((fClas)
  , (fMembDecl,fMembMeth)
  , (fStatDecl,fStatExpr,fStatIf,fStatWhile,fStatReturn,fStatBlock)
  , (fExprCon,fExprVar,fExprOp)
  )
  where
    ...
```

The four datatypes that constitute the algebra have 12 constructors in total. We give each of the associated functions a name, so that we can define them one by one.

Classes A class is translated by calling the label "main" and then halting execution. The code for all the methods follows. The argument *ms* is of type [*Code*], i.e., the methods are already available in translated form, so all we have to do is to flatten the list and add the instructions at the end.

```
fClas k ms = [Bsr "main",HALT] ++ concat ms
```

Methods For methods, there are two constructors: declarations (of variables), and method-definitions. Declarations provide information to the compiler, but do not generate code:

$$\mathit{fMembDecl}\ d = []$$

A method starts with a label that can be used to call the method via s subroutine-call (`Bsr`). We then insert the code for the body of the method, and finally include a return instruction that jumps back to the place where the method has been called.

$$\begin{array}{l} \mathit{fMembMeth}\ t\ m\ ps\ s = \mathbf{case}\ m\ \mathbf{of} \\ \quad \mathit{LowerId}\ x \rightarrow [\texttt{LABEL}\ x] +\!\!+\ s +\!\!+ [\texttt{RET}] \end{array}$$

The pattern match is unsafe, because we only check for *LowerId*. We use our knowledge of the parser, where we accept only a *lowerid*, but decided to represent it as a token for simplicity. Note also that we completely ignore the parameters *ps* for now – something to be changed by you later!

Statements For statements, there are five constructors, hence five functions we have to write. Again, a declaration does not generate any code:

$$\mathit{fStatDecl}\ d = []$$

If an expression occurs as a statement, we can in principle just include the code generated for the expression and execute it because of potential side effects. The expression has a result, however, which is left on the stack, and a statement does not compute a result. We therefore discard the top element of the stack after executing the expression by adjusting the stack pointer:

$$\mathit{fStatExpr}\ e = e\ \mathit{Value} +\!\!+ [\mathit{pop}]$$

where

$$\begin{array}{l} \mathit{pop} :: \mathit{Instr} \\ \mathit{pop} = \texttt{AJS}\ (-1) \end{array}$$

Recall that expressions return results of type $\mathit{ValueOrAddress} \rightarrow \mathit{Code}$, hence we pass *Value* here in order to get at the code that computes the value. It will become clear below what the parameter affects the code generated for an expression.

For if- and while-statements, we have to join the different blocks and add jumps around them. For this, we need *codeSize* to compute the correct jump offsets:

$$\begin{array}{lll} \mathit{fStatIf} & e\ s_1\ s_2 = \mathbf{let} & c\ \ = e\ \mathit{Value} \\ & & n_1 = \mathit{codeSize}\ s_1 \\ & & n_2 = \mathit{codeSize}\ s_2 \\ & \qquad\qquad \mathbf{in} & c +\!\!+ [\texttt{BRF}\ (n_1 + 2)] +\!\!+ s_1 +\!\!+ [\texttt{BRA}\ n_2] +\!\!+ s_2 \end{array}$$

```
fStatWhile e s₁  = let c = e Value
                       n = codeSize s₁
                       k = codeSize c
                   in [BRA n] ++ s₁ ++ c ++ [BRT (−(n + k + 2))]
```

We only partially implement return statements so far. Our functions cannot yet communicate results back to the caller, therefore we compute the result, but then discard it using *pop*. Finally, we do of course return to the caller using `RET`:

```
fStatReturn e = e Value ++ [pop] ++ [RET]
```

A whole block gives us a [*Code*], which we flatten using *concat*:

```
fStatBlock ss = concat ss
```

Expressions As mentioned before, for expressions we need an extra parameter that specifies whether we are interested in the *address* or the *value* of the expression. The address is needed for the left hand side of an assignment. For example, when we write the Java code

```
x = y;
```

we want to store the *value* of *y* in the *address* of *x*.

Not all expressions are valid on the left hand side of an assignment. For the time being, we therefore ignore the extra argument in some places and assume we are interested in the value. One such case is integer constants, where we just push the constant onto the stack:

```
fExprCon c va = case c of
                  ConstInt n → [LDC n]
```

In the variable case, however, we pay attention to the argument. If we are interested in a value, we use `LDL` to load a local variable; if we are interested in an address, we use `LDLA` to load the address of a local variable – a small, but important difference! The current code generator does not calculate the locations of local variables yet, and always assumes the constant 37 as location for every variable:

```
fExprVar v va = case v of
                  LowerId x → let loc = 37
                                in case va of
                                     Value   → [LDL  loc]
                                     Address → [LDLA loc]
```

For most of the operator expressions, we compute the values of the left and right operands, and then use the instruction corresponding to that operator to compute the result. The assignment operator is an exception: it computes the value of the right operand and duplicates it using `LDS 0`, then the address of the left operand, and uses `STA` in the end to perform the assignment. The duplication causes the assigned value to be left on the stack as result of the assignment operation, such that for example $y = x = 0$; works as expected.

```
fExprOp o e1 e2 va =
  case o of
    Operator "=" → e2 Value ++ [LDS 0] ++ e1 Address ++ [STA 0]
    Operator op  → e1 Value ++ e2 Value ++ [opCodes ! op]
```

The finite map *opCodes* associates operators with their instructions, for instance `"+"` with `ADD`.

The driver

The file `Main.hs` contains the main program. It reads the command line arguments, interprets them as a list of Java files, and compiles each of them using the functionality we have just discussed. Each file is scanned, then parsed, then transformed using *foldJava codeAlgebra*. As a final step, the resulting SSM code is printed to an `.ssm` file. Such a file can be loaded into the simulator to see what happens. A valid source file `example.java` is included. Without modifications to the file and/or the code generator, however, not much can be seen when running the result.

Tasks

1. Extend the compiler so that not only integer, but also boolean and character constants can be handled. The SSM language only knows integers, so internally, you will have to map booleans and characters to integers.

2. Fix the priorities of the operators. In the starting framework, all operators have the same priority.

3 (medium)**.** Add the possibility to "call a method with parameters" to the syntax of expressions, and add the possibility to deal with such calls to the rest of the compiler. Make sure that the parameters can be used within the function body. Hint: In the algebra, you have to change the result types. You need to pass around an environment that contains the addresses of available variables.

4 (medium)**.** Extend the code generator such that methods can have a result. You may choose whether you want to pass the result via register or via the stack.

5 (difficult)**.** Adapt the code generator such that the declared local variables can be used. Hint: This is tricky, because in Java, local variables do not have to be declared at the beginning of a method body, but can be declared later as well. You should change the result type for statements in the algebra to be a *pair* of two things: the generated code, plus a list of variables declared. Due to lazy evaluation, you can use the variable list for defining the code, by writing a "circular" program:

```
let (code, vars) = ( ... vars ... — compute code using vars
                   , ...          — compute vars
                   )
in (code, vars)
```

Additional tasks

Choose and try to solve at least two of the following tasks.

6. In the lexer, discard Java comments.

7. In the parser and code generator, add assign/increment operators such as += and ++.

8. In the parser and code generator, add a *for* statement.

9. Next to the code generator, define a separate algebra that pretty-prints Java code back in text form using a standard layout (indented bodies, aligned braces, etc.).

10. Change the code generator for the logical operators, so that they are computed lazily, as is usual in Java. In other words, the right operand should only be evaluated if necessary to determine the result.

11 (difficult)**.** Add a possibility for variables declared in the class to be used in all the methods, as a sort of global variables. Hint: find out for yourself how to allocate and find these, for instance by using an additional register, or indirectly via the original mark pointer that every method stores.

12 (medium)**.** Modify the code generator so that it not only generates code, but also error messages, for a number of non-syntactic errors. Examples: undeclared variables, incorrect types of parameters, incorrect types of return values etc.

13 (very difficult)**.** Modify class variables to be proper object variables, not static variables as before. Also add a *new* construct to create new objects. Hint: every method is passed an implicit extra parameter, the pointer to the current object. Allocate objects in a separate memory segment, and use an additional register to find them.