Compiler Construction/Fall 2014/Project Milestone 3

Eva Rose
evarose@cs.nyu.edu

Kristoffer Rose
krisrose@cs.nyu.edu

Released Thursday, 12/5/2014, Due Tuesday, 12/16/2014

1 Assignment

Consider the provided *Pr3base.hx* script, reproduced in the next section. The script defines two languages:

1. A subset of the JST language that we have studied in class (without classes and methods).
2. A subset of ARM32 assembly language called MinARM32 (the full ARM32 reference manual is on the class web site).

The task of this assignment is to submit the following pair of files to the pr3 task in NYU classes:

Pr3-YourName.pdf documents what choices you have made, what inconsitencies you have found, and what you have done to remedy them.

Pr3-YourName.hx a HACS script that implements a compiler from JST to MinARM32 such that one can run

```
$ hacs Pr3-Yourname.hx
$ ./Pr3-YourName.run --scheme=Compile Test.jst
```

for some test *Test.jst* program and the output is MinARM32 assembly instructions. You should use HACS version 0.9.8.

The generated MinARM32 assembly should follow these rules:

1. Each function should generate code such that it is possible to call it with the standard ARM calling conventions:
   - On entry, R0–R3 contain parameters, SP (R13) the stack pointer, and LR (R14) the return address.
   - On return, R0–R1 can contain function results, R4–R11 and SP (R13) are restored to what they were at the time of the call.
2. Every JST program must contain a *main* function that takes one **int** argument and returns one **int** value.
3. It is alright to assume that functions never have more than four arguments.

As usual, supplementary information may be provided through *pr3 bulletins*.

2 Pr3Base.hx

Your file may be based on the HACS script *Pr3Base.hx*, which is included below (and available on the class web site).

```
// [NYU Courant Institute] Compiler Construction/Fall 2014/Project Milestone 3
//
// Skeleton
// 1. Lexical  analysis
// 2. JST Grammar
// 3. MinARM32 assembler grammar
// 4. Compiler (To Be done)
//
// See http://cs.nyu.edu/courses/fall14/CSCI-GA.2130-001/pr3/pr3.pdf

module edu.nyu.csci.cc.fall14.Pr3Base {

////////////////////////////////////////////////////////////////////////
// 1. LEXICAL ANALYSIS
////////////////////////////////////////////////////////////////////////

space [ \t\n\r] | '//' [^\n]* | '/*' ( [^*] | '*' [^/] )* '*/'   ; // Inner /* ignored

token IDENTIFIER | ⟨LetterEtc⟩ (⟨LetterEtc⟩ | ⟨Digit⟩)* ;

token INTEGER | ⟨Digit⟩+ ;

token fragment Letter    | [A-Za-z] ;
token fragment LetterEtc | ⟨Letter⟩ | [$_] ;
token fragment Digit     | [0-9] ;

////////////////////////////////////////////////////////////////////////
// 2. JST GRAMMAR
////////////////////////////////////////////////////////////////////////

// PROGRAM

main sort Program | ⟦ ⟨Declarations⟩ ⟧ ;

// DECLARATIONS

sort Declarations | ⟦ ⟨Declaration⟩ ⟨Declarations⟩ ⟧ | ⟦⟧ ;

sort Declaration
| ⟦ function ⟨Type⟩ ⟨Identifier⟩ ⟨ArgumentSignature⟩ { ⟨Statements⟩ } ⟧
;

sort ArgumentSignature
| ⟦ ( ) ⟧
| ⟦ ( ⟨Type⟩ ⟨Identifier⟩ ⟨TypeIdentifierTail⟩ ) ⟧
```

```
;
sort TypeIdentifierTail | ⟦ , ⟨Type⟩ ⟨Identifier⟩ ⟨TypeIdentifierTail⟩ ⟧ | ⟦ ⟧ ;

// STATEMENTS

sort Statements | ⟦ ⟨Statement⟩ ⟨Statements⟩ ⟧ | ⟦⟧ ;

sort Statement
| ⟦ { ⟨Statements⟩ } ⟧
| ⟦ var ⟨Type⟩ ⟨Identifier⟩ ; ⟧
| ⟦ ; ⟧
| ⟦ ⟨Expression⟩ ; ⟧
| ⟦ if ( ⟨Expression⟩ ) ⟨IfTail⟩ ⟧
| ⟦ while ( ⟨Expression⟩ ) ⟨Statement⟩ ⟧
| ⟦ return ⟨Expression⟩ ; ⟧
| ⟦ return ; ⟧
;

sort IfTail | ⟦ ⟨Statement⟩ else ⟨Statement⟩ ⟧ | ⟦ ⟨Statement⟩ ⟧ ;

// TYPES

sort Type
| ⟦ boolean ⟧
| ⟦ int ⟧
;

// EXPRESSIONS

sort Expression

| sugar ⟦ ( ⟨Expression#e⟩ ) ⟧@10 →#e

| ⟦ ⟨Integer⟩ ⟧@10
| ⟦ ⟨Identifier⟩ ⟧@10

| ⟦ ⟨Expression@9⟩ ( ) ⟧@9
| ⟦ ⟨Expression@9⟩ ( ⟨Expression⟩ ) ⟧@9

| ⟦ ! ⟨Expression@8⟩ ⟧@8
| ⟦ − ⟨Expression@8⟩ ⟧@8
| ⟦ + ⟨Expression@8⟩ ⟧@8

| ⟦ ⟨Expression@7⟩ * ⟨Expression@8⟩ ⟧@7

| ⟦ ⟨Expression@6⟩ + ⟨Expression@7⟩ ⟧@6
| ⟦ ⟨Expression@6⟩ − ⟨Expression@7⟩ ⟧@6

| ⟦ ⟨Expression@6⟩ < ⟨Expression@6⟩ ⟧@5
```

```
95  |  ⟦ ⟨Expression@6⟩ > ⟨Expression@6⟩ ⟧@5
96  |  ⟦ ⟨Expression@6⟩ <= ⟨Expression@6⟩ ⟧@5
97  |  ⟦ ⟨Expression@6⟩ >= ⟨Expression@6⟩ ⟧@5
98
99  |  ⟦ ⟨Expression@5⟩ == ⟨Expression@5⟩ ⟧@4
100 |  ⟦ ⟨Expression@5⟩ != ⟨Expression@5⟩ ⟧@4
101
102 |  ⟦ ⟨Expression@3⟩ && ⟨Expression@4⟩ ⟧@3
103
104 |  ⟦ ⟨Expression@2⟩ || ⟨Expression@3⟩ ⟧@2
105
106 |  ⟦ ⟨Expression@2⟩ = ⟨Expression@1⟩ ⟧@1
107
108 |  ⟦ ⟨Expression@1⟩ , ⟨Expression⟩ ⟧
109 ;
110
111 sort Integer  |  ⟦ ⟨INTEGER⟩ ⟧;
112 sort Identifier  |  symbol ⟦⟨IDENTIFIER⟩⟧ ;
113
114 ////////////////////////////////////////////////////////////////////////
115 // 3. MinARM32 ASSEMBLER GRAMMAR
116 ////////////////////////////////////////////////////////////////////////
117
118 // Instructions .
119 sort Instructions | ⟦ ⟨Instruction⟩ ⟨Instructions⟩ ⟧ | ⟦⟧ ;
120
121 sort Instruction
122 | ⟦ ⟨ Identifier ⟩ = ⟨Integer⟩ ¶⟧          // define identifier
123 | ⟦ ⟨ Identifier ⟩ ⟧                          // label
124 | ⟦ DCI ⟨Integers⟩ ¶⟧                       // allocate integers
125 | ⟦ ⟨Op⟩ ¶⟧                                 // machine instruction
126 ;
127
128 sort Integers | ⟦ ⟨Integer⟩, ⟨Integers⟩ ⟧ | ⟦ ⟨Integer⟩ ⟧ ;
129
130 // Syntax of individual machine instructions.
131 sort Op
132
133 | ⟦ MOV ⟨Reg⟩, ⟨Arg⟩ ⟧                     // move
134 | ⟦ MVN ⟨Reg⟩, ⟨Arg⟩ ⟧                     // move not
135 | ⟦ ADD ⟨Reg⟩, ⟨Reg⟩, ⟨Arg⟩ ⟧              // add
136 | ⟦ SUB ⟨Reg⟩, ⟨Reg⟩, ⟨Arg⟩ ⟧              // subtract
137 | ⟦ AND ⟨Reg⟩, ⟨Reg⟩, ⟨Arg⟩ ⟧              // bitwise and
138 | ⟦ ORR ⟨Reg⟩, ⟨Reg⟩, ⟨Arg⟩ ⟧              // bitwise or
139 | ⟦ EOR ⟨Reg⟩, ⟨Reg⟩, ⟨Arg⟩ ⟧              // bitwise exclusive or
140 | ⟦ CMP ⟨Reg⟩, ⟨Arg⟩ ⟧                     // compare
141 | ⟦ MUL ⟨Reg⟩, ⟨Reg⟩, ⟨Reg⟩ ⟧              // multiply
142
143 | ⟦ B ⟨ Identifier ⟩ ⟧                      // branch always
```

```
144 | ⟦ BEQ ⟨Identifier⟩ ⟧                  // branch if equal
145 | ⟦ BNE ⟨Identifier⟩ ⟧                  // branch if not equal
146 | ⟦ BGT ⟨Identifier⟩ ⟧                  // branch if greater than
147 | ⟦ BLT ⟨Identifier⟩ ⟧                  // branch if less than
148 | ⟦ BGE ⟨Identifier⟩ ⟧                  // branch if greater than or equal
149 | ⟦ BLE ⟨Identifier⟩ ⟧                  // branch if less than or equal
150 | ⟦ BL ⟨Identifier⟩ ⟧                   // branch and link
151
152 | ⟦ LDR ⟨Reg⟩, ⟨Mem⟩ ⟧                 // load register from memory
153 | ⟦ STR ⟨Reg⟩, ⟨Mem⟩ ⟧                 // store register to memory
154 | ⟦ LDRB ⟨Reg⟩, ⟨Mem⟩ ⟧                // load byte into register from memory
155 | ⟦ STRB ⟨Reg⟩, ⟨Mem⟩ ⟧                // store byte from register into memory
156 | ⟦ LDMFD ⟨Reg⟩! , {⟨Regs⟩] ⟧          // load multiple fully descending (pop)
157 | ⟦ STMFD ⟨Reg⟩! , {⟨Regs⟩} ⟧          // store multiple fully descending (push)
158 ;
159
160 // Arguments.
161
162 sort Reg        | ⟦R0⟧ | ⟦R1⟧ | ⟦R2⟧ | ⟦R3⟧ | ⟦R4⟧ | ⟦R5⟧ | ⟦R6⟧ | ⟦R7⟧
163                 | ⟦R8⟧ | ⟦R9⟧ | ⟦R10⟧ | ⟦R11⟧ | ⟦R12⟧ | ⟦SP⟧ | ⟦LR⟧ | ⟦PC⟧ ;
164
165 sort Arg | ⟦⟨Constant⟩⟧ | ⟦⟨Reg⟩⟧ | ⟦⟨Reg⟩, LSL ⟨Constant⟩⟧ | ⟦⟨Reg⟩, LSR ⟨Constant⟩⟧ ;
166
167 sort Mem | ⟦[ ⟨Reg⟩, ⟨Sign⟩ ⟨Arg⟩ ] ⟧ ;
168 sort Sign | ⟦+⟧ | ⟦−⟧ | ⟦⟧ ;
169
170 sort Regs | ⟦⟨Reg⟩⟧ | ⟦⟨Reg⟩, ⟨Regs⟩⟧ ;
171
172 sort Constant | ⟦#⟨Integer⟩⟧ | ⟦&⟨Identifier⟩⟧ ;
173
174 ////////////////////////////////////////////////////////////////////////
175 // 4. COMPILER
176 ////////////////////////////////////////////////////////////////////////
177
178 sort Instructions  |  scheme Compile(Program) ;
179
180 Compile(#1) →⟦main MOV PC,LR⟧ ;
181
182 }
```