Compiler Construction/Fall 2014/Project Milestone 3
Some Hints

Eva Rose
evarose@cs.nyu.edu

Kristoffer Rose
krisrose@cs.nyu.edu

Saturday 13th December, 2014

1 Some Reminders

Here are some things to think about when generating code – all issues, we have discussed in class.

When trying to understand how each JST code translates into ARM32 code, keep the following in mind:

1. Think about the *recursive structure* with diagrams ("boxes") that show the assembler sequences that should correspond to each JST construct. Note that such diagrams will have *labels* and *registers*.
2. Think about what the *control flow* from each box should be. Common choices include:
   - The box always exits at the end.
   - The box always finishes by jumping to a particular label L.
   - The box reprsentes a boolean expression that jumps to one label T when true and another F when false.
3. Think about what the *data flow* from each box should be. Common choices:
   - The box does not compute values but does leave all variables stored in memory.
   - The box computes a value, which is always delivered in register R0.
   - The box computes a value, and we communicate into the box which register r it should store it in.
4. It should all be hooked together properly. A good way to do this is to, yes, write an SDD, which composes the assembler code and uses the attribute mechanism to assign the needed labels and registers.
5. Once you have your boxes all defined, it may help to have macro syntax for composing them. However, a typical instruction sequence syntax has

   sort Instructions | ⟦⟨Instruction⟩ ⟨Instructions⟩ ⟧ | ⟦⟧ ;

   which restricts rules to generate the instructions one by one. In such cases, the following declaration is often helpful:

```
// Helper “syntactic macro” to append separate Instructions groups.
sort Instructions | scheme ⟦{ ⟨Instructions⟩ } ⟨Instructions⟩ ⟧;
⟦ { } ⟨Instructions#3⟩ ⟧→#3 ;
⟦ {⟨Instruction#1⟩ ⟨Instructions#2⟩} ⟨Instructions#3⟩ ⟧
   → ⟦ ⟨Instruction#1⟩ {⟨Instructions#2⟩} ⟨Instructions#3⟩ ⟧;
```

If, for example, you have a sort with syntax for a sequence of single declarations

```
sort Decls | ⟦ ⟨Decl⟩ ⟨Decls⟩ ⟧ | ⟦ ⟧ ;
```

but would like to use a scheme D that generates a sequence of Instructions for *each* Decl, then the helper above may be used:

```
sort Instructions | scheme Ds(Decls) ;
Ds( ⟦⟨Decl#1⟩ ⟨Decls#2⟩ ⟧) →⟦{ ⟨Instructions D(#1)⟩ } ⟨Instructions Ds(#2)⟩ ⟧;
Ds( ⟦⟧ ) → ⟦ ⟧ ;
```

You can verify that every side of both rules is a legal Instructions Instance, which is what is required.