National University of Singapore

CS4212 Project Assignment 2 – Part B

AY 2015/2016 Semester 1

Due Date: 18^{th} October 2015 (Sunday, 23:59 Hrs)

*** This is Part B – the final part – of Assignment 2 ***

For Assignment 2, you are required to construct the second part of the front end of the **mOOL** compiler. Given a parse tree representing a **mOOL** program, you perform static checks over the parse tree, and then generate the corresponding intermediate code, defined by an intermediate representation language called IR3.

In Part A of this assignment, you were asked to perform static check over parse trees of **mOOL** programs, and create new parse trees annotated with type information. We call them *typed-parse trees.*

In Part B of this assignment, you will develop a code generator that accepts well-typed **mOOL** programs and generate intermediate codes in IR3. A well-typed **mOOL** program is one which has successfully passed the tests given by static checker.

Please note that **the deadlines for submitting Part A and Part B of the Assignment 2 are the same.**

1 IR Generator

The Intermediate code generator which you are going to construct is called "IR3-Gen". It takes in the typed parse tree (as produced by the static checker in Part A of your assignment), and generates three-address code of the form called IR3.

We normally think of the code generated to be a string of characters. However, in this project, your IR3-Gen does not directly generate the character string representing the intermediate code. Rather, it generates a data structure as specified by the IR3 grammar, as shown in Appendix A. As a data structure, we can further take it as input and transform it to some low-level machine code (as will be done in Assignment 3). Certainly, generating code from the IR3 data will not be difficult – you just need a special display function. Fortunately, both the data structure and the display function for processing IR3 code are made available to you.

IR3 certainly looks very similar to three-address code format. Please note that all complicated expressions have been transformed into either a binary operation or just an identifier. For instance, instead of having `return` $\langle exp \rangle$, in IR3 we have simply `return` $\langle id3 \rangle$, where the argument for the IR3 return statement `return` must a variable.

In the following sections, we consider various issues in converting **mOOL** programs to IR3 codes.

2 Converting classes to records

IR3 specification does not have the concept of "classes"; it only has the concept of records with field names, constructed by the data constructor `Class3`. When converting a class to a record, attributes declared in the class will become fields in the record, and with the same names too! In addition, the record will use the same name (type) as the class name.

For methods declared in classes, we elect to have all methods declared in class to be taken out of the class, and become global methods. Specifically, each declared method will be given a *globally unique names*, which will be used for method invocations. While full details of method declarations will not stay in the converted record, we maintain a table – one for each class – associating **mOOL**'s method names (with some variants, as seen later) with their globally unique names. (In IR3 specification, we append this method table to the record of the corresponding class, for easy reference.)

Several concerns need to be addressed with this conversion, as described below.

2.1 How to handle method overloading?

Two overloaded method must have the same method name, but different method signatures (ie., the types of the method parameters.) For instance, the following two methods declared in a class, say `C` are overloaded:

```
Int m1 (Int p1, C1 p2, Bool p3) { ... }
Int m1 (Bool p1, C1 p2, Bool p3) { ... }
```

Suppose further that these two methods gave been assigned a globally unique name, say `_C_0` and `_C_1` respectively. Then, we generate two pairs of identifiers to be kept in the corresponding method tables for `C`:

```
(m1~Int~C1~Bool, _C_0) ;
(m1~Bool~C1~Bool, _C_1) ;
```

Note that the first component of each pair above is a special name obtained by "concatenating" the original method name and the corresponding method signature. This thus makes referencing to each method unique and distinguishable.

In summary, all methods (irrespective of whether the method will be overloaded) will refer to their global method names through special names composing from the original names and the signatures.

2.2 What happens to class inheritance when classes are turned into records?

As mentioned earlier, since IR3 does not have the concept of class hierarchy, a record representing a class does not only need to capture information about the class (ie., its attributes and methods), but also need to collect information about all its superclasses.[1]

For instance, consider the following three classes:

[1] Recursively, A class C is a *superclass* of another class D if either D extends C in its class declaration, or D extends another class D' of which D is the superclass.

```
class A {      |  class B extends A {  |  class C extends B {
 Int a1 ;      |    Int b1 ;           |    Bool c1 ;
 Void ma () {  |    Int mb () {        |    Bool mc () {
   ...         |     ...               |     ...
 }             |    }                  |    }
}              |   }                   |   }
```

Under this scheme of converting classes to records, we obtain the following three records for `A`, and `B` and `C` respectively:

```
class3 A {     |  class3 B {      |  class3 C {
 parent: None  |   parent: A      |   parent: B
 Int a1 ;      |   Int a1 ;       |   Int a1 ;
 (ma, _A_0)    |   Int b1 ;       |   Int b1 ;
}              |   (ma, _A_0) ;   |   Bool c1 ;
               |   (mb, _B_0) ;   |   (ma, _A_0) ;
               |  }               |   (mb, _B_0) ;
               |                  |   (mc, _C_0) ;
               |                  |  }
```

Note the following arrangements:

1. **All attributes** of the superclasses are kept in the children records, in the order according to the class hierarchy. Note that the ordering of the attributes is important, as it is possible to have two fields in the record having the same name, and even the same type, because they are attributes from different classes.

2. **All methods** from the superclasses are also kept in the children records (forming the method table), **with special consideration given to handling method overriding** (see Sec 2.3). Again, the ordering of the methods in the table in important, as they reflect the hierarchy of the corresponding classes.

2.3 How to handle overriding methods?

Method overriding, in object oriented programming, is a language feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclasses or parent classes.[2] Two methods, from a child class and a parent class respectively, are considered overriding if they have (1) the same name, (2) the same signature, and (3) the same return type.

In our implementation, when two methods are overriding, there is no need to create two entries of the methods in the method table; one entry suffices. Thus, if we look at the following class declaration where methods (`m1`) has been overridden:

```
class D {          |   class E extends D {
 Int a1 ;          |     Int b1 ;
 Int a2 ;          |     Int a1 ;
```

[2]Reference: `https://en.wikipedia.org/wiki/Method_overriding`.

```
 Void m1 (C p) {   |     Void m1 (C p) {  // overriding
   ...             |      ...
 }                 |     }
 Void m2 () {      |     Int m3 () {
   ...             |      ...
 }                 |     }
}                  |    }
```

Under this scheme of converting classes to records, we obtain the following three records for `D`, and `E`, respectively:

```
class3 D {    |  class3 E {
 parent: None |   parent: D
 Int a1 ;     |   Int a1 ;      // from class D
 Int a2 ;     |   Int a2 ;      // from class D
 (m1~C, _D_0);|   Int b1 ;      // from class E
 (m2,_D_0);   |   Int a1 ;      // from class E
}             |   (m1~C, _E_0); // from class E overriding
              |   (m2,_D_0);    // from class D
              |   (m3, _E_1);   // from class E
              |  }
```

2.4 How to handle attributes and/or methods with "private" modifier?

There is no issue, as this has been handled by static checkers. Is that right?

3 Method invocation

During code generation of method invocation, note that:

1. The order of evaluating method call arguments is **leftmost applicative evaluation order.** (You may wish to "google" this for details.)

2. At source code, you can call a method m belonging to an object o of class C as follows: `o.m(`e_1`,` e_2`)` ; The corresponding method invocation code in IR3 will *only take variables and constants as arguments*, and not expressions as arguments. This means that we need to have IR3 code to evaluate the arguments e_1 and e_2 respectively, and assign the results to some temporary variables (if necessary), and then pass them to the IR3 method call.

3.1 The first Argument to Method invocation

In IR3, there is no notion of classes (and objects). As such, we should consider the object o as an argument passing to method m during invocation. The reason that we need this special argument is that we may wish to refer to the attributes in the object o or other methods present in o when we are computing the invocation of $o.m$. As a uniform conversion from object-oriented to object-less method invocation, we elect to pass the record representing o as the first argument of the method.

Specifically, for the following method declaration,

```
class C1 {
   Int x ;
   Int m1(Int a) {
        return a + x ;
   }
}
```

Suppose the global unique name of `m1` is `_C1_0`, then the generated method declaration will be:

```
Int _C1_0(C1 this, Int a) {
        Int t1 ;
        Int t2 ;
        t1 = this.x ;
        t2 = a + t1 ;
        return t2 ;
}
```

To summarize, a call to m which takes n arguments (for $n \geq 0$) will be converted to one that takes $n + 1$ arguments, with the first argument *always* designated to be "C `this`", where C is the class to which m belongs.

3.2 Dynamic Dispatch

Dynamic dispatch is the process of selecting which implementation of a particular method to call at *runtime*. It aims to support cases where the appropriate implementation of a method cannot be chosen until run time.[3]

To see how dynamic dispatch is necessary to execute **mOOL** programs, consider the following method declaration:

```
Int md(C p1, D p2) {
     ....
     p2.m1(p1) ;
     ....
}
```

Let's suppose that class `D` is the class declared in Section 2.3, which has been extended to class `E` with method `m1` being overridden. During the compilation of the code for method `md`, we first perform static check to ensure that method `m1` can be invoked from any object, denoted here by `p2`, of class D (which is the case for the example of class D given above). In the IR generation phase, recall that we would like to replace method `m1` by a corresponding globally unique named method. But, we don't really know what this unique name is. For instance, when `md` is called with `p2` being assigned an object of class `D`, then the global method name should be `_D_0` ; however, if instead an object of class `E` is assigned to `p2` during `md`'s invocation, then `m1` should be connected to the global method named `_E_0`. Thus, at the point when we are generting IR3 code for the body of `md`, we don't really know which method should be attached to. This decision has to be delayed until some later stage.

[3]Reference: `https://en.wikipedia.org/wiki/Dynamic_dispatch`.

Now that we understand that the method dispatch has to be delayed at IR generation phase, we keep the existing method invocation information during conversion to IR3 code. Suppose `md` is declared in a class `K`, and the global name for `md` is set to `_K_1`, the resulting code will look like the following:

```
Int _K_1 (K this, C p1, D p2) {
     ....
     p2.m1~C (p2, p1) ;
     ....
}
```

where `m1~C` is the name created at IR3 code generation time for `m1` declared in class `D`. (Finally, note that `p2` is passed in as the first argument to `m1~C`, as what has been described in Section 3.1.

4 Other Issues

1. *Object Instantiation*: An object of class `C` is created by invoking the method `C()`, which is translated to `new C()` in IR3. Complication arises when methods in class `C` uses the keyword `super`. Specifically, when `super` is encountered, control will be transferred from the current object to its parent object. But, where is the parent object? When are they created?

   We leave the answer to Assignment 3. During IR3 code generation, we simply output the `new C()`.

2. *Boolean short-circuting*: This will be applied as and when necessary. This, you have learned in the lecture; and you may also read it from the textbook.

3. *memory space allocation*: While we have discussed about determining the memory space required for storing values of identifiers, and computing offsets so that we can refer to identifiers by their relative addresses, you do not have to do this computation here. You can delegate this job to the phase of machine code generation – to be done in Assignment 3. In this assignment, you just need to (a) introduce temporary identifiers so that simple expressions receptive to IR3 specification can be created; (b) maintain the type information of the temporaries, so that you don't have to compute the types once again at the later phase of the compiler process (aka., machine code generation.)

4. `return` *statement*: Make sure that the expression returned from the IR3 `return` statement is just a variable, not any other expression.

5. `super` *vs. Upcasting to Parent class*: Consider again the class `D` and the derived class `E` declared in Section 2.3. Suppose an object `obj` belonging to class `E` has been created. Do the following two statements refer to the same object?

   (a) Call made via `super` keyword: `obj.super.m2()`.

   (b) Perform upcasting operation before calling:

```
D v = (D) obj ;
v.m2() ;
```

We claim here that the two calls **behave differently**. Here's why.

In the second case, while `obj` has been upcast from class `E` to `D`, it remains an instance of class `E` and not `D`, despite the upcasting. Consequently, `v.m2()` triggers the invocation of method `m2` in class `E`. On the other hand, in the first case, the `super` keyword forces us to pass the control to the parent of `obj`, and later on invoke a call to method `m2` residing in class `D`.

5 What've Been Given to You?

You should have all the resources necessary for developing static checker. If you don't, please refer to Part A of this Assignment for detail.

In addition, you are given, in Part B, the following information:

1. A module, `ir3mOOL_structs.ml`, containing type definitions of all syntax specified for IR3 language. Inside this module, there is also a display function for displaying IR3 code. You must use this code for displaying results produced by IR3-Gen.

2. Two sample **mOOL** source codes, called `ex1.mo` and `sample1.mo` respectively, and the results of running the static checker on them. The results are stored in the file `ex1.all` and `sample1.all` respectively. (More may be uploaded to the workbin.) Note that each of these "`.all`"-suffixed files contains the original **mOOL** program, the result of static check, and the IR3 code generated.

3. A makefile, `Makefile`, which allows you to create the executable for static checker and IR3-Gen. In Unix, running "make" on `Makefile` will produce an executable named `mOOL_ir3_Gen`. **You must prepare a Makefile for generation of executable.** You can be severely penalized if your submission does not have a working and faithful `Makefile`.

4. A sample IR3-Gen module, `mOOLtoir3.ml`, that helps you to jump start your development of IR3-Gen.

5. The main program, `mOOL_main.ml`, that links all pieces together, from lexical analysis, through both syntax and semantic analysis, till intermediate code generation.

Given a **mOOL** program as input, your static checker is required to generate the corresponding typed parse tree, which contains type annotation at each node of the parse tree. Any **mOOL** programs that fail the static check will be rejected, and the compiler should come to a halt gracefully. On the other hand, any **mOOL** programs that pass the static check will have their typed parse tree produced and undergo intermediate code generation. Consequently, your IR3-Gen will work directly on the typed parse trees.

6 Deliverable

Submission of your product

At the end of this project assignment, you are required to submit **one directory containing your code from both Part A and Part B** of this assignment, and other relevant information. The details are as follows:

1. The directory name should be of the form `p2_xxxxxxx`, where `xxxxxxx` is your student id, such as `A1234567`.

2. The directory must include a `README` file, which includes details about what you have done to complete Part A and Part B , and how we should process your directory to assess your accomplishment. Any important details which you would like to highlight to us must be included in this file. Treat this as the report of your Assignment 2.

3. The directory must include a `Makefile` file, which can be the same as the one you received in this Assignment. Grading of your assignment will begin by running this `Makefile` program to obtain the executable. Therefore, please ensure the content of this file is an accurate reflection of the work done by you.

4. The directory must include a sub-directory named `TESTCASES` containing the test cases which you have created for testing during development. It will be good to divide the test cases into sub-directories, each of which aims to test one of the development tasks above. Each task should have at least three interesting and distinctive test cases.

   Please submit the directory in zipped format to `IVLE CS4212` website under the "`Project Assignment 2 Submission`" folder.

7 Late Submission

We strongly discourage you from submitting your assignment beyond deadline. This is to ensure that you have time to prepare for other modules, as well as time for other assignments handed out in this module.

Any submission after the deadline will have its mark automatically deducted by certain percentages. Your submission time is determined by the time recorded in IVLE submission folders. If you have multiple submissions, we will take the latest submission time. Any submission to places other than the appropriate submission folders (such as to the instructor's email account) will be discarded and ignored.

Here is the marking scheme:

Submit by 23:59HRS of	Maximum Mark	If your score is ... %	It becomes ... %
18th Oct (Sun)	100	80	80
19th Oct (Mon)	80	80	64
20th Oct (Tue)	60	80	48
After 20th Oct (Tue)	0	-	0

A Syntactic Specification of IR3

A.1 Lexical Issues

- ⟨*CName3*⟩ ∈ **Class Names**. It is defined similarly as ⟨*CName*⟩ in **mOOL**.
- ⟨*id3*⟩ ∈ **Identifiers**. This includes ⟨*id*⟩ as defined in **mOOL**. In addition, it also includes `this` (referring to specific object) and `super` (referring to object's parent), as well as newly generated temporary variables, which are of the format `(_[a-z]+[0-9]+)` for temporary variables created, `(_[a-z]+_[0-9]+)` for globally unique method names. and names composed by joining method names and method signatures using the terminal symbol ' ~ '.

A.2 Syntactic Specification

$\langle Program3 \rangle \rightarrow \langle Class3 \rangle^{+}\ \langle Mtd3 \rangle\ \langle Mtd3 \rangle^{\star}$

$\langle Class3 \rangle \rightarrow$ `class3` $\langle CName3 \rangle$ `{ parent:` $\langle CName3 \rangle$ `;` $\langle VarDecl3 \rangle^{\star}$ $\langle MtdIdPair3 \rangle^{\star}$ `}`

$\langle VarDecl3 \rangle \rightarrow \langle Type3 \rangle\ \langle id3 \rangle$ `;`

$\langle MtdIdPair3 \rangle \rightarrow$ `(` $\langle id3 \rangle$ `,` $\langle id3 \rangle$ `) ;`

$\langle Mtd3 \rangle \rightarrow \langle Type3 \rangle\ \langle id3 \rangle$ `(` $\langle FmlList3 \rangle$ `)` $\langle MdBody3 \rangle$

$\langle FmlList3 \rangle \rightarrow \langle CName3 \rangle$ `this` $\langle FmlL13 \rangle$

$\langle FmlL13 \rangle \rightarrow$ `,` $\langle Type3 \rangle\ \langle id3 \rangle\ \langle FmlL13 \rangle^{\star} \mid \epsilon$

$\langle Type3 \rangle \rightarrow$ `Int` | `Bool` | `String` | `Void` | $\langle CName3 \rangle$

$\langle MdBody3 \rangle \rightarrow$ `{` $\langle VarDecl3 \rangle^{\star}\ \langle Stmt3 \rangle^{+}$ `}`

$\langle Stmt3 \rangle \rightarrow \langle Label3 \rangle$ `:` | `if (` $\langle Exp3 \rangle$ `) goto` $\langle Label3 \rangle$ `;` | `goto` $\langle Label3 \rangle$ `;`
| `readln (` $\langle id3 \rangle$ `) ;` | `println (` $\langle idc3 \rangle$ `) ;`
| $\langle id3 \rangle$ `=` $\langle Exp3 \rangle$ `;` | $\langle id3 \rangle$ `.` $\langle id3 \rangle$ `=` $\langle Exp3 \rangle$ `;`
| $\langle id3 \rangle$ `.` $\langle id3 \rangle$ `(` $\langle VList3 \rangle$ `) ;`
| `return` $\langle id3 \rangle$ `;` | `return ;`

$\langle RelExp3 \rangle \rightarrow \langle idc3 \rangle\ \langle Relop3 \rangle\ \langle idc3 \rangle \mid \langle idc3 \rangle$

$\langle Exp3 \rangle \rightarrow \langle idc3 \rangle\ \langle Bop3 \rangle\ \langle idc3 \rangle \mid \langle Uop3 \rangle\ \langle idc3 \rangle \mid \langle id3 \rangle . \langle id3 \rangle \mid \langle idc3 \rangle$
| $\langle id3 \rangle$ `.` $\langle id3 \rangle$ `(` $\langle VList3 \rangle$ `)` | `(` $\langle CName3 \rangle$ `)` $\langle id3 \rangle$ | `new` $\langle CName3 \rangle$ `()`

$\langle Relop3 \rangle \rightarrow$ `<` | `>` | `<=` | `>=` | `==` | `!=`

$\langle Bop3 \rangle \rightarrow \langle Relop3 \rangle$ | `||` | `&&` | `*` | `/` | `+` | `-`

$\langle Uop3 \rangle \rightarrow$ `!` | `-`

$\langle VList3 \rangle \rightarrow \langle idc3 \rangle\ \langle VRest3 \rangle^{\star} \mid \epsilon$

$\langle VRest3 \rangle \rightarrow$ `,` $\langle idc3 \rangle$

$\langle idc3 \rangle \rightarrow \langle id3 \rangle \mid \langle Const \rangle$

$\langle Const \rangle \rightarrow$ `true` | `false` | `INTEGER_LITERAL` | `STRING_LITERAL` | `NULL`