CS4212 Programming Assignment 2

Bernard Teo Zhi Yi

[Note: I have submitted both the markdown (.md) file and the equivalent HTML and PDF file containing the same information. You can use either one, but the HTML version has the best formatting.]

Introduction

This submission builds on the parser from assignment 1, and processes the syntax tree with a type checker and a distinct-name checker. The type-checked syntax tree is then converted into IR3 code as specified in the task statement. You can input a Jlite source file, and it will produce the IR3 three-address code, and print it with neat formatting and proper indentation. There are no known bugs — it produces the correct IR3 code on all valid inputs, and it identifies all syntactic and semantic errors correctly.

The implementation of the type checker and IR3 generator goes beyond the requirement for assignment 2. Firstly, **overloaded methods** are allowed to occur, and they are handled in a similar way to Java. Secondly, each local variable is given an integer id in increasing order (unique within a function), allowing efficient manipulation and representation, and perhaps making it easier to do optimisations and generate assembly at a later stage. The ids are mapped back to strings when printing the IR3 code.

All kinds of errors (e.g. syntax errors, type errors, duplicate name checking, etc) will be detected, and my program will **print out details about the error, including highlighting the relevant text from the source file**. In most situations, my compiler will also provide more information, such as the relevant declaration and its location. After an error, the compiler will continue processing the rest of the program, **allowing multiple errors to be reported at one go**.

Content of the submission

The `src` directory contains all the files of my submission, including appropriate versions of the jflex and cup binaries.

Subdirectories:

- `src` : code that drives the compiler
- `src/tree` : classes to represent the syntax tree of the program
- `src/ir3` : classes to represent the IR3 code and type-checking functions
- `src/util` : some utility classes for error reporting
- `src/test` : contains tests
- `src/test/errors` : contains tests that are semantic errors
- `src/test/asg1` : contains tests from assignment 1 (you can ignore this)
- `src/jflex` : contains jflex spec and jflex binaries
- `src/cup` : contains cup spec and cup binaries

Important!: Before continuing any further, you must go into the directory first:

```
cd src
```

Compile

Firstly, we need to chmod the java archives for jflex and cup, and the scripts:

```
chmod u+x cup/*.jar
chmod u+x jflex/jflex-1.8.2/lib/*.jar
chmod u+x *.sh
```

Then type `make`, like this command:

```
make
```

Then will take a while to compile. Be patient, and it should be done after a while.

Run

You can use the `run.sh` script, which will use the specified Jlite source file from the test folder. For example to run the `test/fast_exp.j` file, you can do:

```
./run.sh fast_exp
```

Remember that you do not add the file extension here.

Alternatively, you may type in the command manually (see the content of `run.sh` for the exact syntax).

Tests

All the successful tests are in `src/test`. Those tests meant to be semantic errors are in `src/test/errors`.

There are 6 successful tests (apart from the default), and 9 semantic error tests.

To run all the successful tests, do:

```
./test.sh
```

You should get an output like this:

```
Testing test/default ... OK
Testing test/expressions ... OK
Testing test/fast_exp ... OK
Testing test/loops ... OK
Testing test/multiclass ... OK
Testing test/overloading ... OK
Testing test/statements ... OK
```

The `default` test is the one that came with the assignment.

The `expressions`, `loops`, `multiclass`, `overloading`, and `statements` tests are extensive tests that test all possible variations of the various parts of the compiler. The `fast_exp` test is a program that does exponentiation in logarithmic time (by repeatedly squaring the number).

Brief overview of the compiler

The following is a brief overview of how the compiler works.

We start with a syntax tree (that was produced by the parser).

Semantic checking and IR3 code generation is done together. The complier processes the syntax tree, generating type information along the way (stored in a separate data structure), and uses that information to generate IR3 code.

A first pass is done on the classes to generate the metadata for each class (containing the list of fields and their types, as well as the list of functions and their signatures). It keeps the fields of each class in an ordered list, and the methods are extracted from the class and stored as a global list (with the extra 'this' parameter prepended). Overload resolution tables are also generated at this stage. We can think of the metadata as an extended version of class descriptors (it is the class "ClassDescriptor" in my code). Since fields and methods are in lists, they are implicitly indexed, allowing the second pass to look up names and refer to fields and methods by their indices. Duplicate class names, field names, and method names are checked at this point.

A second pass is done over each of the methods of each class, doing type-checking and then emitting the IR3 code for each method. Type-checking is done on every statement and expression in the method, and it verifies whether each operator or function call can be done with the given argument types. The type-checking is done from the leaf nodes up to the entire statement (with the exception of "null", where the type is "coerced" by the parent, see the next section for more information). Because we have access to the class descriptors from the first pass, we can lookup the type of each variable we encounter.

Each expression (or sub-expression) has its IR3 code generated immediately after it is type-checked. It is actually a more compact form of IR3 code, where fields and methods are referenced by index instead of by name.

If both passes succeeds, my compiler will then print out the IR3 code, converting the indices back to human-readable field/method names to comply with the output requirements.

In both passes, the compiler tries to recover from semantic errors as soon as possible (by ignoring the offending item), and so is able to report multiple errors in a single execution. However, if the first pass has errors, the compiler will not go on to the second pass. This is because a failure of the first pass means that some fields or methods might be missing, and this will lead to many errors on the second pass if we still attempt to proceed on. See the error handling section for more details.

Features

Some of the more interesting features of my compiler are highlighted here, in the hope of gaining me more marks :)

Of course, my compiler does produce correct IR3 code for all valid Jlite programs, but that should be expected from all compilers. This section is for the more interesting features.

Handling of "null"

As required by the Jlite specification, null represents two possible things depending on the context:

- Indicating the lack of an object, or a "null object", which is usual Java behaviour
- Indicating an empty string.

In particular, this means that `null == ""` should return `true`, and `null + null` should return `null` (or `""`) (since string concatenation is allowed with the `+` operator).

My compiler treats the `null` expression as a special kind of type (which I call the "null type"). The null type can be present in the syntax tree, but cannot exist after type-checking is completed. Specifically, as part of type-checking, the actual type of each `null` expression is determined, and the actual type is attached to the expression (the actual type can be either `String` or some class type).

The actual type is determined via a type coercion process, which depends on the context that the value is used in. These are some examples:

- When assigning to an variable (e.g. `x = null;`), the `null` is coerced to the type of the variable being assigned to
- When used as an argument to an operator (e.g. `null + "test"`), the `null` is coerced to the type required by the operator
- When used as an argument to a method call (e.g. `f(null)`), the `null` is coerced to the type required by the method being called (see the method overloading section for specifics)
- When used in a return statement (e.g. `return null;`), the `null` is coerced to the return type of the method

The coercion process works in all cases where it is reasonable to do so. (If you want to look at the compiler code, search for the `imbueType` function.)

Method overloading

My compiler implements the bonus requirement of proper method overloading.

Method overloading works like how it does in Java. For example, we could have the following class definition:

```
class A {
    Void f(Bool b) { /* some code */ }
    Void f(String s) { /* some code */ }
    Void f(String s1, String s2) { /* some code */ }
    Void f(A a) { /* some code */ }
    Void f(Main m) { /* some code */ }
}
```

Suppose `x` is an object of type `A`. Then, `x.f(true)` will call the first overload, `x.f("test")` will call the second overload, etc. We could also have overloads with different number of arguments, as per the example class above.

In a scenario without overloading, we might distinguish methods by just their class and name. However, to make overloading work, we distinguish methods by the types of their arguments as well. For example, the first overload would be treated as something like "A.f(Bool)", or equivalently, after conversion to regular free functions, it is "f(A,Bool)". As another example, the third overload above would become "f(A,String,String)". This transformation is done during the first pass.

In the second pass, when we actually encounter call expressions and call statements, we look at the list of actual argument types (some of which could be `null`), and find the list of matching function declarations

(also considering null coercions where possible, since `null` can coerce to `String` or to any class type). If there is exactly one match, then the call is binded to that unique matching function. If there are no matches or multiple matches, then it is an error and the compiler reports the appropriate error. Error information is detailed, and will display the list of matching candidates and their locations in the source file.

There is a test specially for overloading, called `test/overloading`. This test would have been run when you did `./test.sh` earlier.

Name mangling

A side effect of allowing method overloading is that we need to mangle function names in the IR3 code, in order to ensure that the function names are unique. We use a simple mangling scheme, separating the class name, method name, and parameters types by `%`. For ease of human consumption, the parameter types are separated from the method name by `%%%`. For example, `A.f(String, String)` is mangled as `%A%f%%%A%String%String` (note: the additional `A` is the implicit "this" parameter; it may be redundant but it makes it easier to understand the actual list of function parameters, and it does not cause any harm).

In theory, this name mangling scheme would be able to support static methods too (i.e. those methods that do not take an implicit 'this' parameter), although we do not have static methods in Jlite.

The main function is exempt from name mangling, since it is required to be the only method in its class.

Implicit "this"

Like Java, the programmer can omit the "this" keyword when using a field from the same class, unless there is a local variable of the same thing. In other words, if the source code uses the name `name`, my compiler will first check for a local variable (or parameter) called `name`; failing which, it will check for a field (or method) of the current class that is called `name`.

In IR3, all the implicit "this" is converted to the explicit version, because the the "this" parameter is made explicit there.

Fields and methods with the same name

It is possible to differentiate between fields and method calls in the parser. My compiler takes advantage of this, allowing fields and methods to have the same name.

For example, we might have the following class definition:

```
class A {
    Int f;
    Void f(Bool b) { ... }
}
```

Then we will interpret the name `f` as a field or method depending on the context:

```
f = 1; // field
f(true); // method
a = new A();
a.f = 2; // field
a.f(false); // method
```

By syntactically differentiating fields and methods, they do not share the same namespace, and hence you can have a method that has the same name as a field.

Java proper also has this feature.

Void type

It is not useful to have the `Void` type as a local variable or field, because it does not contain anything.

As we want to do the "natural" thing, the `Void` type is only allowed as the return type of a method. My compiler exlicitly checks that `Void` is only used as the return type of a method, so the use of `Void` in all other contexts will lead to a semantic error, like this one:

```
Error: Void type not allowed here ( line 4 col 13 - line 4 col 16 ) :
4 |    Int j;  Void v;
  |            ^^^^
Note: The Void type is only allowed in the return type of a method
Semantic checking aborted due to above errors.
```

Error handling

My compiler comes with top-of-the-class error handling and reporting functionality. These are three reasons why the error reporting is excellent:

- After an error is detected, the compiler still proceeds on to process the rest of the program, allowing multiple errors to be reported. Hence, the programmer can see all their errors at one go, and fix all of them before re-compiling the code.
- Every error message comes with location information (line and column) of the offending expression or statement, and furthermore the line(s) in the source containing the relevant expression/statement is printed out. This allows the programmer to quickly identify the problem.
- Various notes and hints are also provided, together with location information where sensible. This could, for example, be a note showing the location of a previous declaration, or a hint suggesting a method of the same name when the programmer mistakenly thought that it was a field. There are many such notes and hints, and some of the more interesting ones are highlighted below.

Since this report is for assignment 2, we will not discuss syntax errors here. Syntax error detection is equally excellent, and also has the behaviour of the first two points above.

The tests related to semantic errors are in the `test/errors/` subdirectory. You can follow the examples below to try them.

Duplicate classes, fields, methods, parameters, or local variables

Duplicate items (such as classes, fields, methods, parameters, or local variables) print an error showing the item that was double-declared, its location, and the location of the previous declaration.

For example, we could try `test/errors/duplicate_method.j`:

```
./run.sh errors/duplicate_method
```

The source file looks like this:

```
class Main {

Void main(Int i, Int a, Int b,Int d){
    i = a;
}
}
class Dummy {
    Int x;
    Void f() { x = x;}
    Void f(String y) { x = y;}
    Void f(String more_arg) { more_arg = x;}
}
```

We will get a message like this (in stderr):

```
Error: Duplicate declaration of method "f(String)" in class "Dummy" ( line 11 col 5 - line
11 |     Void f(String more_arg) { more_arg = x;}
   |     ^^^^^^^^^^^^^^^^^^^^^^^^
Note: Previously declared here ( line 10 col 5 - line 10 col 20 ) :
10 |     Void f(String y) { x = y;}
   |     ^^^^^^^^^^^^^^^
Semantic checking aborted due to above errors.
```

The above error message demonstrates that:

1. Error messages will print out the line and column number, and will print out the offending line, and place '^' under the offending token. The number on the left hand side is the line number.
2. An additional note is produced that prints out the original declaration that caused the new declaration to be classified as a duplicate.

Method calls and overloading

Semantic errors might occur when making a method call, because either the method does not exist, or there are multiple overloads that match (in which case we say that the call is ambiguous). My compiler handles all these cases well, and will even print out other helpful information.

For example, we could try `test/errors/ambiguous_call.j`:

```
./run.sh errors/ambiguous_call
```

The source file looks like this:

```
class Main {

Void main(Int i, Int a, Int b,Int d){
    (new A()).f(null);
    (new A()).f(123);
    (new A()).g();
}
}

class A {
    Int g   ;
    Void f(Bool b) { b = b; }
    Void f(String s) { s = s; }
```

```
    Void f(String s1, String s2) { s1 = s2; }
    Void f(A a) { a=a;}
    Void f(Main s) {s=s;}
}
```

We will get a message like this (in stderr):

```
Error: Ambiguous method call to "f(null)" of class "A" ( line 4 col 5 - line 4 col 22 ) :
4 |     (new A()).f(null);
  |     ^^^^^^^^^^^^^^^^^^
Hint: 3 candidate overloads matched
Hint: Candidate overload "f(String)" matches ( line 13 col 5 - line 13 col 20 ) :
13 |     Void f(String s) { s = s; }
   |     ^^^^^^^^^^^^^^^^^
Hint: Candidate overload "f(A)" matches ( line 15 col 5 - line 15 col 15 ) :
15 |     Void f(A a) { a=a;}
   |     ^^^^^^^^^^^^
Hint: Candidate overload "f(Main)" matches ( line 16 col 5 - line 16 col 18 ) :
16 |     Void f(Main s) {s=s;}
   |     ^^^^^^^^^^^^^^
Error: Method "f" does not exist in class "A" ( line 5 col 5 - line 5 col 21 ) :
5 |     (new A()).f(123);
  |     ^^^^^^^^^^^^^^^^^
Error: Method "g" does not exist in class "A" ( line 6 col 5 - line 6 col 18 ) :
6 |     (new A()).g();
  |     ^^^^^^^^^^^^^^
Hint: There is a field of the same name, did you intend to refer to it instead? ( line 11 c
11 |     Int g   ;
   |     ^^^^^
Semantic checking aborted due to above errors.
```

The above error message demonstrates that:

1. Function calls may be ambiguous when the argument is `null`, because `null` can represent a `String` or any class type. This is the case when there are multiple overloads in which `null` can be coerced to.
2. When overload resolution is ambiguous, an error message will be produced by my compiler. Further, the compiler will print additional hints, specifying all the candidates that match.
3. If a method name does not exist, an error message is printed.
4. If there is a field of the same name as a method that does not exist, then an additional hint will be provided, showing the declaration of that field. Note that (as mentioned above) it is perfectly valid to have a field and a method with the same name, but the hint is produced in case the programmer actually wanted to refer to the field. Similarly, when there is a method of the same name as a field that does not exist, an additional hint will also be provided.

If and While statement condition type

The condition of 'if' and 'while' statements are required to be `Bool`.

For example, we could try `test/errors/if_condition.j`:

```
./run.sh errors/if_condition
```

The source file looks like this:

```
class Main {

Void main(){
    Int i; Int j; String cond;
    Int ir;
    String sr;
    if (i) {
        j = 1;
    }
    else {
        j = 2;
    }
    while (cond + "b") { cond = cond + "a"; }
}
}
```

We will get a message like this (in stderr):

```
Error: Expected expression to have type "Bool", but got type "Int" instead ( line 7 col 9 -
7 |     if (i) {
  |         ^
Note: Condition of if-statement must have type "Bool"
Error: Expected expression to have type "Bool", but got type "String" instead ( line 13 col
13 |     while (cond + "b") { cond = cond + "a"; }
   |           ^^^^^^^^^^
Note: Condition of while-statement must have type "Bool"
Semantic checking aborted due to above errors.
```

The above error message demonstrates that:

1. The conditions of 'if' and 'while' statements are checked, to ensure that they have type `Bool`. If they have a wrong type, my compiler will print a friendly error message highlighting the exact location of the error.

Assignment type checking

This test contains some general assign statements, to check that type-checking detects type errors for local variables, parameters, and fields, even in other classes.

For example, we could try `test/errors/type_imbue.j`:

```
./run.sh errors/type_imbue
```

The source file looks like this:

```
class Main {

Void main(Int i, Int a, Int b,Int d){
    Int j;
    i = "abc";
    j = true;
    (new A()).a = "ghi";
    j = null;
}
}
class A {
```

```
        Int a;
    }
```

We will get a message like this (in stderr):

```
Error: Expected expression to have type "Int", but got type "String" instead ( line 5 col 9
5 |     i = "abc";
  |         ^^^^^
Note: Variable "i" was declared as type "Int" here ( line 3 col 11 - line 3 col 15 ) :
3 | Void main(Int i, Int a, Int b,Int d){
  |               ^^^^^
Error: Expected expression to have type "Int", but got type "Bool" instead ( line 6 col 9 -
6 |     j = true;
  |         ^^^^
Note: Variable "j" was declared as type "Int" here ( line 4 col 5 - line 4 col 9 ) :
4 |     Int j;
  |         ^^^^^
Error: Expected expression to have type "Int", but got type "String" instead ( line 7 col 1
7 |     (new A()).a = "ghi";
  |                   ^^^^^
Note: Variable "a" was declared as type "Int" here ( line 12 col 5 - line 12 col 9 ) :
12 |     Int a;
   |         ^^^^^
Error: Expected expression to have type "Int", but got type "null" instead ( line 8 col 9 -
8 |     j = null;
  |         ^^^^
Note: Variable "j" was declared as type "Int" here ( line 4 col 5 - line 4 col 9 ) :
4 |     Int j;
  |         ^^^^^
Semantic checking aborted due to above errors.
```

The above error message demonstrates that:

1. All assignment type errors are detected and reported appropriately, with appropriate context from the source code.
2. A note is provided that shows the declaration of the variable being assigned, also with appropriate context from the source code.

Operator "+"

The operator "+" is "overloaded" for String+String and Int+Int, so we need to do overload resolution to determine the correct kind of operator "+". Furthermore, `null` can be coerced to `String`, which means that things like `null + null` are valid and evaluate to the empty string.

My compiler handles all these cases correctly, and prints out the relevant errors if there are issues.

For example, we could try `test/errors/operator_plus.j`:

```
./run.sh errors/operator_plus
```

For brevity, the code and output are omitted from this report, but they can be accessed in the `test/errors/` subfolder.

All error tests

There are in total 9 tests that will lead to semantic errors. They demonstrate the resilience of my compiler to errors, and the friendly output that is produced. Each can be run separately using the `run.sh` script, e.g.:

```
./run.sh errors/ambiguous_call
./run.sh errors/duplicate_class
./run.sh errors/duplicate_field
./run.sh errors/duplicate_method
./run.sh errors/if_condition
./run.sh errors/operator_plus
./run.sh errors/read_print
./run.sh errors/return_type
./run.sh errors/type_imbue
```

The expected output is also available in the `test/errors` directory.