Project Assignment 2 Readme

User guide

Compiling and running

Ensure Java 11 is installed.

Run `make` to compile.

Run `make run file=path/to/file.txt` to parse, typecheck, and generate IR3 for a file. Some test files are located in `tests/pa2/success` and `tests/pa2/failure` .

Static checks and IR3 Generation

First, all distinct name errors are reported. If there are distinct name errors, type checking is not done, because if there are duplicate names with different we don't know which one is the intended one.

Once all name errors are fixed, type checking is done and the relevant type mismatch errors are output.

Once all these type errors are fixed, the IR3 will be generated and printed out.

Design and Implementation

Type Inference Rules Assumptions

Parser vs. Type Checker

In the language specification of Jlite, it explicitly disallowed expressions such as `1 * true` , so sadly all these would result in a parse error and not even go through the type checker.

Perhaps it would be better to simply allow all types to be accepted into the language for all operators first, and then let the type checker do its job, so that better error messages can be provided.

Nonetheless, I did not redo PA1, so stuff like `1 * true` will remain parse errors.

In the test cases and subsequent examples, I circumvent this issue by wrapping the wrong types in parentheses in order to be accepted by the language, e.g. `1 * (true)` instead of `1 * true` .

`null`

There weren't any rules on how to handle `null` . So, I decided to only allow the use of null when assigning to a non-primitive variable, or returning `null` when the return type is not a primitive.

- `Int i; i = null` not allowed, since `Int` is primitive.
- `CustomClass cc; cc = null` allowed, since `CustomClass` is not primitive.

Unary operators

I just assumed that `!` only works on `Boolean` s and `-` only works on `Int` s.

Method overloading

To allow for method overloading, we don't just map the method *name* to the relevant method declaration, we map the method name **and** its parameter types to the relevant method declaration.

Implementation wise, we store the method name and its parameter types together in a class `MethodSignature` and then create a `Map` mapping a `MethodSignature` to its actual method declaration.

Then, during type checking, to check if a method call is valid, we simply create a new `MethodSignature` containing the method's name and parameter types, and see if it exists in the `Map` .

In summary, we use the method name **and** its parameter types to identify a method, not just its name alone.

Error Cascading

In early versions of my type checker, it terminated once the first error is encountered (by throwing an exception). This gave rise to a rather unsatisfying user experience, as the user is forced to fix their errors one by one.

For example,

```
class Main {
    Void main() {
        Int a;
        a = 1 + (true); // AA
        a = (true) * (true); //BB
    }
}
```

Ideally, one should see three error messages:

```
Line: 4, Col: 15: Expected type Int on right of + instead got type Bool
Line: 5, Col: 20: Expected type Int on left of * instead got type Bool
Line: 5, Col: 20: Expected type Int on right of * instead got type Bool
```

To implement this, whenever a type check fails, I set its type to an interal type `ERROR` and then pass this along. It worked well for the above example, and did produce those 3 nice error messages.

However, another problem came along. Consider this:

```
class Main {
    Void main() {
        Int a;
        a = (((1 + (true)) + 2) + (false)); // AA
    }
}
```

1. `(1 + (true))` is obviously wrong, and gives rise to the `ERROR` type.
2. `((1 + (true)) + 2)` is now treated as wrong again, since `(1 + (true))` is of type `ERROR` and then gives rise to another `ERROR` type.

This continues on, and then produces these four error messages:

```
Line: 4, Col: 18: Expected type Int on right of + instead got type Bool
Line: 4, Col: 28: Expected type Int on left of + instead got type error
Line: 4, Col: 33: Expected type Int on left of + instead got type error
Line: 4, Col: 33: Expected type Int on right of + instead got type Bool
```

This is clearly unhelpful though, since some of these errors were caused by previous errors!

Imagine an even more complicated program, one single mistake in a deeply nested computation can cause a whole bunch of redundant error messages.

Finally, I decided that whenever an `ERROR` type was encountered, I would skip the check and not print any error message, and simply pass the `ERROR` type along. The final result:

```
Line: 4, Col: 18: Expected type Int on right of + instead got type Bool
Line: 4, Col: 33: Expected type Int on right of + instead got type Bool
```

Error messages where they happen, no more, no less!

Testing

Since muliple errors can be reported at once, I split the error testing into two files: name checking with only distinct name errors, and type checking with only type errors. To add on, there is another file with more type errors.

For successful test cases, I have one minimal program with just one main class with just one return statement, and another file with many complicated expressions such as nested binary expressions, recursive calls, and nested calls such as `a.b().c().d().e()` .