210: Compiler Design

Spring 2017

Homework 3

Due: **Tuesday**, April 17th, 2018, 10 a.m.

25 Points

1 Introduction

The objective for this homework is to develop a semantic analyzer, building on the front-end that was constructed in Homework 2.

2 Tasks

The semantic analyzer needs to guarantee that the input program can be safely executed according to the semantic rules of Javali. In order to do this, it must perform several checks.

Whenever an error is detected, your code should throw a `SemanticFailure` exception, as defined by the framework. Each `SemanticFailure` must specify a `Cause`, which is an enumeration defined in the class. The testing framework will check that your code throws a `SemanticFailure` with the correct `Cause`, but does not examine the user-readable message that is attached.

We have listed each of the semantic conditions you must check below, as well as the appropriate `Cause`.

2.1 Create symbols for all types, fields, parameters, and local variables

You will need to create symbols for every class, array type, field, method, parameter, local variable, etc. Typically, these symbols are linked in the AST, and possibly also stored in a symbol table to allow for a symbol to be looked up by name.

In the homework template, we provide a sample class called `Symbol` that can be used to represent the symbols, and we also added fields to the various AST nodes where symbols can be attached. But we do not provide a symbol table implementation.

Note that the Javali language maintains a separate namespace each for types and for methods, since they can easily be syntactically distinguished. Fields, variables, and parameters are in a common namespace and can hide each other based on scoping rules (see language specification). As an example, consider the following *legal* Javali program, which declares a local variable named `Foo` of type `Foo`. Also, there is no conflict between the field `main` and the method `main()`:

```
class Foo {
}
class Main {
  int main;
  void main() {
    Foo Foo;
    Foo = new Foo();
  }
}
```

2.2 Semantic failures and their causes

`DOUBLE_DECLARATION` Within the same namespace and scope, no two symbols can have the same name (see scoping in the language specification). That is, no two classes, no two fields, no two methods, and no two local variables (including parameters) can be declared with the same name.

`NO_SUCH_TYPE` Wherever a type name is expected in the program, it must refer to an existing type that is defined elsewhere in the program (as a class) or to a predefined (built-in) type.
For class declarations, only class types can be extended (i.e., inherited from).

`INVALID_START_POINT` A valid Javali program must define a class `Main` with a method `main` that has the signature `void main()`.

`CIRCULAR_INHERITANCE` Inheritance relationships between classes must not contain cycles (e.g., `A extends B` and `B extends A`).
Note: This is a tricky one that many groups did not get 100% right in previous years!

`OBJECT_CLASS_DEFINED` No class is defined with the name `Object`. This name is reserved for a predefined class type that serves as the root of the inheritance hierarchy.

`INVALID_OVERRIDE` Overriding methods must have the same signature as the method in the super class. A method overrides a method of the super class if they share the same name. The signature of methods consists of the return type, the number of parameters, and the type of all parameters.

`TYPE_ERROR` Not following any of the rules below causes a `TYPE_ERROR`:

- Integer literals (hex or decimal) are of type `int`, boolean literals (`true` and `false`) are of type `boolean`, `this` is of the class type of the enclosing class, and the `null` literal is of a hidden type that is a subtype of all reference types (array types and classes).
- `write(expr)` requires `expr` to be of type `int`
- `read()` produces a result of type `int`
- `if(cond)` and `while(cond)` require `cond` to be of type `boolean`
- The type of the right-hand side in an assignment must be a subtype of the type of the left-hand side.
- Binary and unary arithmetic operators (`*, /, %, +, -`) require operands of type `int` and produce a result of type `int`.
- Binary and unary boolean operators (`!, &&, ||`) require operands of type `boolean` and produce a result of type `boolean`.
- Relational operators (`<, <=, >, >=`) require operands of type `int` and produce a result of type `boolean`.
- Equality operators (`==, !=`) take operands of types `L` and `R` where either `L` is a subtype of `R`, or `R` is a subtype of `L`.
- A cast from type `R` to type `C` is only legal if `R` is a subtype of `C` or vice versa. It produces a result of type `C`.
- In an array-indexing expression (`x[i]`), the index must be of type `int` and the array must be of some array type `A[]`. The type of the whole array-indexing expression is the element type `A` of the array.
- When creating a new array with an expression `new A[length]`, the type of `length` must be `int`. The whole expression is of array type `A[]`.
- `new X()` expressions are of the type `X`.
- In a method invocation, the type of each actual argument must be a subtype of the type of the corresponding formal parameter. The whole method invocation has the type that is declared as the formal return type of the method.
- There must be no attempt to access a field or a method on a target of a non-class type, such as an array or a primitive type.

- In a method return statement, the expression type must be a subtype of the corresponding formal return type. For `void` methods, *any* return statement with an expression should be a `TYPE_ERROR`.

`WRONG_NUMBER_OF_ARGUMENTS` In a method invocation, the number of actual arguments must match the number of formal parameters from the method declaration.

`NO_SUCH_VARIABLE` All referenced variables (including parameters) must exist.

`NO_SUCH_FIELD` All referenced fields must exist. Since fields in a subclass hide fields from super classes, the type of a field expression (`expr.f`) is determined by the declaration of field `f` in the class corresponding to the static type of `expr`. Only if the field is not declared there, the analyzer should search for it in the class's supertypes recursively.

`NO_SUCH_METHOD` All referenced methods must exist. Since methods can be inherited, the method `m` in the expression `expr.m()` needs to be looked up in the class corresponding to the static type of `expr` and if not found there, all its super classes recursively.

`NOT_ASSIGNABLE` The left-hand side of an assignment must be assignable.The only assignable expressions are variables (`x`), fields (`expr.f`), and array-indexing (`expr[i]`)

`MISSING_RETURN` A method with a return type different from `void` must have a return statement on all possible control flow paths.
Note: This is another tricky one.

3 Comments

3.1 Inheritance

Your compiler is expected to handle inheritance (recall that Javali does not allow multiple inheritance!).

Note that a base class does not need to be declared before its subclasses. The ordering of class declarations is not significant in Javali.

3.2 Subtyping

In Javali, all types are subtypes of themselves. In addition, all arrays are subtypes of `Object`, and a class is a subtype of the class which it extends and this class's supertypes. Be aware that in Javali arrays are not covariant. That is, all arrays are direct subtypes of `Object`, unlike Java, where an array type `C[]` is a subtype of another array type `B[]` if `C` is a subtype of `B`.

For the `null` literal, there is a hidden type that is a subtype of all reference types (arrays as well as objects).

`void`, on the other hand, is a type with no instance. It can only be used as a return type for methods and is never a subtype of any other type.

Otherwise, there are no subtyping relationships (for example, `int` and `boolean` and `Object` are not related in any way).

3.3 Gray areas between parser and semantic analyzer

Technically, you can catch many of these errors in the parser, however we recommend that you check for them in the semantic analyzer in order to be more compatible with the reference framework. As an exception, the template we provide you catches the error of wrong number of arguments for `write()`, `writeln()`, and `read()` in the parser.

3.4 Test cases

As usual, we expect you to develop a test suite. To make it easier for us to locate the tests you have written, please place them in a subdirectory of `javali_tests` named `HW3_team`. In addition, it is very helpful if you place a comment at the top of the test file indicating what you are trying to test. Please note that it is useful to have *positive* tests (i.e., programs that should pass semantic analysis) as well as *negative* ones (i.e., programs that should fail).

3.5 Turning in the solution

Please ensure that your final submission is checked into Subversion by the deadline. Any commit after the deadline will not be considered. If you have any comments or remarks on your solution, please provide a `README` file in the `HW3` directory of your team's Subversion directory.