210: Compiler Design

Fall 2015

Homework 3

Due date: Nov 12, 2015, 10:00 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, and local variable, etc. Typically, these symbols are linked in the AST tree, and possibly also stored in a symbol table to allow for a symbol to be looked up by name.

In the homework fragment, 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.

Please note that the Javali language maintains separate namespaces for types and for variables, as well as for fields and for methods. As an example, consider the following legal Javali program, which declares a local variable named `Foo` of type `Foo`. There is also 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 Check for well-formed class declarations

The semantic analysis must guarantee that:

- There exists a class named `Main` with a method named `main` that takes no parameters, so that execution has a known starting point (`INVALID_START_POINT`).
- The superclass specified for each class exists. (`NO_SUCH_TYPE`)
- There is no circular inheritance (e.g., `A extends B` and `B extends A`). (`CIRCULAR_INHERITANCE`)
- No class is declared with the name `Object`. This name is reserved for a predefined class which shall serve as the root of the inheritance hierarchy. (`OBJECT_CLASS_DEFINED`)
- All classes have unique names. (`DOUBLE_DECLARATION`)
- There are no two fields with the same name in a single class. (`DOUBLE_DECLARATION`)
- There are no two methods with the same name in a single class. (`DOUBLE_DECLARATION`)
- Overridden methods must have the same number of parameters as the method from the superclass. (`INVALID_OVERRIDE`)
- The types of the parameters and the return type of overridden methods must be the same as for the corresponding method in the superclass. (`INVALID_OVERRIDE`)
- No method may have two parameters with the same name. (`DOUBLE_DECLARATION`)
- No method may have two local variables with the same name. (`DOUBLE_DECLARATION`)

2.3 Check method bodies

Method bodies can contain the following kinds of errors.

- The built-in function `write()` should take a single integer argument. Since the number of arguments should be enforced by your parser, only the type of the expression must be checked by the semantic analyzer. (`TYPE_ERROR`)
- The built-in functions `read()` and `writeln()` should take no arguments. In the framework, this is enforced by the parser and therefore does not cause a semantic failure (`WRONG_NUMBER_OF_ARGUMENTS`).

- The condition for an if or while statement must be of boolean type. (`TYPE_ERROR`)
- For assignments, the type of the right-hand side must be a subtype of the type of the left-hand side. (`TYPE_ERROR`)
- Binary and unary arithmetic operators (`*, /, %, +, -`) must take arguments of integer type, and produce a result of integer type. Binary arithmetic operators cannot be applied to values of two different types (`TYPE_ERROR`).
- Binary and unary boolean operators (`!, &&, ||`) must take arguments of boolean type, and produce an boolean type result. (`TYPE_ERROR`)
- Relational operators (`<, <=, >, >=`) must take arguments of integer type and produce an boolean type result. Relational operators cannot be applied to values of two different types. (`TYPE_ERROR`)
- Equality operators (`==, !=`) must take arguments of types `L` and `R` where either `L` is a subtype of `R`, or `R` is a subtype of `L`. (`TYPE_ERROR`)
- The built-in function `read()` produces an integer type and can only be used where an integer type is expected. (`TYPE_ERROR`)
- A cast from type `R` to type `C` is only legal if `R` is a subtype of `C` or vice-versa. (`TYPE_ERROR`)
- In a method invocation, the number of actual argument must be the same as the number of formal parameters in the method declaration. (`WRONG_NUMBER_OF_ARGUMENTS`)
- In a method invocation, the type of each actual argument must be a subtype of the type of the corresponding formal parameter. (`TYPE_ERROR`)
- All referenced fields must exist. (`NO_SUCH_FIELD`)
- All referenced methods must exist. (`NO_SUCH_METHOD`)
- The index to an array dereference (`x[i]`) must be of integer type, and the array must be of array type. The resulting type is the element type of the array. (`TYPE_ERROR`)
- When creating a new array, the length must be of integer type. (`TYPE_ERROR`)
- The constant `null` can be assigned to any reference type.
- The type name in a `new` statement or cast statement must be the name of a valid type. (`NO_SUCH_TYPE`)
- All referenced variables must be defined. (`NO_SUCH_VARIABLE`)
- The left-hand side of an assignment must not be a non-assignable expression (such as `this`). The only legal expressions are variables (`x`), fields (`x.f`), and array dereferences (`x[i]`). (`NOT_ASSIGNABLE`)

- There must be no attempt to access the field or method of a non-object type, such as an array or integer. (`TYPE_ERROR`)
- In a method return statement the expression type must be a subtype of the corresponding formal return type. (`TYPE_ERROR`)
- A method with a return type must have a return statement on all control flow paths. (`MISSING_RETURN`)

3 Comments

3.1 Inheritance

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

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. 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 fragment 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 sometimes it is useful to have positive tests (i.e., programs that should pass semantic analysis) as well as negative tests (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.