Lund University
Computer Science
Niklas Fors, Görel Hedin

Compilers
EDAN65
2017-09-22

Programming Assignment 4
Semantic Analysis using Reference Attribute Grammars

The goal of this assignment is to understand how semantic analysis can be implemented using *reference attribute grammars* (RAGs). RAGs add *computed* properties called *attributes* to abstract syntax trees (ASTs). For the language SimpliC, you will use RAGs and:

- Implement name analysis (using *synthesized and inherited attributes*)
- Implement type checking (using *non-terminal attributes* (NTAs) for primitive types)
- Implement error checking (using a *collection attribute*)

As usual, try to solve all parts of this assignment before going to the lab session.

▶ Major tasks are marked with a black triangle, like this.

1 The CalcRAG demo

The CalcRAG demo project illustrates how to implement semantic analysis using reference attribute grammars (RAGs). In contrast to the previous assignment, no explicit symbol table is needed when using RAGs. Instead, the information is stored as attributes in the AST.

▶ Download the CalcRAG demo. Run the tests and see that they pass.

1.1 Abstract grammar

▶ The abstract grammar makes a difference between declarations of names (`IdDecl`) and uses of names (`IdUse`) since uses have other properties (attributes) than declarations. For example, an `IdUse` has a `decl` attribute that points to the corresponding `IdDecl` node. Make sure you understand the difference between `IdDecl` and `IdUse` by studying the abstract grammar for CalcRAG (`src/jastadd/calc.ast`) and the following example Calc program.

```
let a = 1.5 in
  a * 2.0
end
```

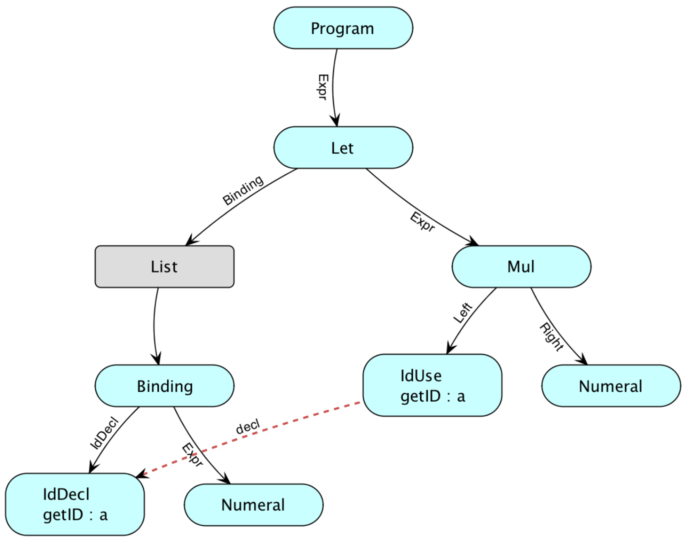

Figure 1: Example program with attributed AST

The AST above was generated by using the DrAST tool mentioned in the previous assignment. DrAST allows you to display not only the nodes in an AST, but also selected attributes and tokens. In this case, `getID` and `decl` are shown. As you see, values with primitive types, like `getID`, are drawn inside the nodes, whereas reference values, like `decl`, are drawn as edges in the tree. It can be helpful to run DrAST on your own compiler later on. The view is controlled by a *filter program* that can be edited in the top right corner of the DrAST window. The view in the figure above was obtained by the following filter program:

```
include ** show { getID() decl() }
```

This means: in the visual view, include the nodes that can be reached from the root through the path `**` (i.e., all nodes), and show all attributes named `getID()` and `decl()`. You can save the filter if you like. It will then be stored as an `.fl2` file.

1.2 Name Analysis

Name bindings are represented by an attribute `IdUse.decl` which refers to the appropriate `IdDecl` node. The name analysis module (`src/jastadd/NameAnalysis.jrag`) implements `IdUse.decl`, using the Lookup pattern, with the helper attributes `lookup` and `localLookup` that search for declarations. To implement declaration-before-use, the `localLookup` attribute has an integer argument `until` which is used to restrict the declaration search to an initial part of the declaration list of the `Let` construct.[1]

Undeclared names are handled using the Null object pattern, and are bound to the object `unknownDecl()` which is a specific `IdDecl` object defined in another aspect. The attribute `isUnknown()` can be used on `IdDecl` objects to find out if it is the `unknownDecl()` object or not.

To identify multiple declarations of the same name, an attribute `IdDecl.isMultiDeclared` is used. It checks if a lookup of the name results in another `IdDecl` node than the `IdDecl` itself.

Another error that can occur is that a variable is circularly defined, i.e., it is used to define its own value. To identify this situation, an attribute `IdUse.isCircular` is used. It checks if the `IdUse` occurs in the defining expression of its declaration.

1.2.1 A simple name binding

▶ Study the name analysis module (`src/jastadd/NameAnalysis.jrag`). Try to understand what happens when the value of the `decl` attribute is computed for the example in Figure 1. First `lookup("a")` for the `IdUse` node is evaluated. In this process, the `localLookup("a")` for the `Let` node will be evaluated. (If you are using DrAST, you can click on nodes and see the values of these attributes).

1.2.2 Nesting and undeclared uses

Note how nested scopes are implemented: the equation for `lookup` in class `Let` delegates to its own `lookup` attribute in case the declaration is not found locally.

Consider the program below that contains two undeclared uses: the first use of `b` (in `a = b`) is not within the scope of the subsequent declaration of `b`, and there is no declaration of `c`. The undeclared uses are bound to an `UnknownDecl` object, and we will see in Section 1.3 how it is implemented.

▶ Study `src/jastadd/NameAnalysis.jrag` to see how the `decl` attribute for the first use of `b` is computed. First, `lookup("b")` is evaluated, and the equation used is `Binding.getBinding(int).lookup(String)`. Note how the `int` argument is used to restrict the search to the declarations before the use. Then note how the equation calls its own `lookup` attribute to continue the search in the outer scope, in case no declaration was found. Which equation for `lookup` will be used next?

```
let
  a = b
  b = 3.0
in
  a * c
end
```

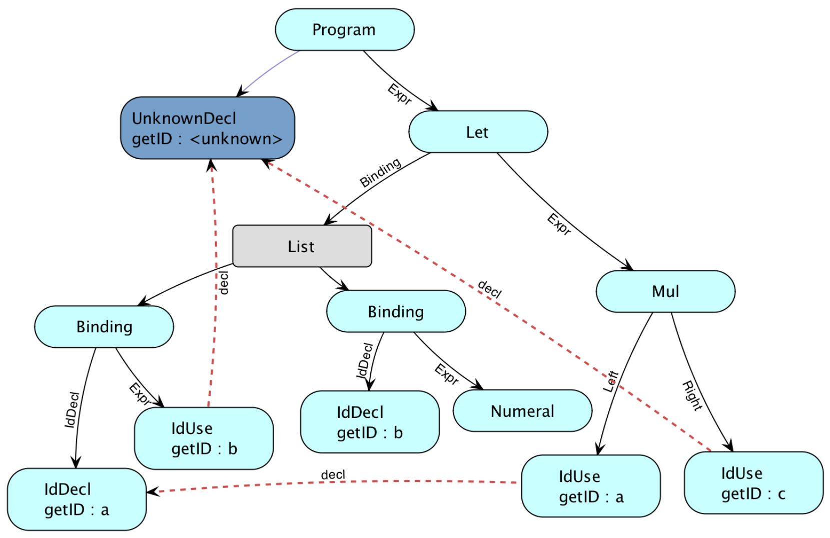

Figure 2: Nesting and undeclared variables

Some of the equations use the Java conditional expression which you are perhaps not familiar with:

cond-exp ? *then-exp* : *else-exp*

The value of this expression is *then-exp* if *cond-exp* is true, and *else-exp* otherwise.

1.2.3 Multi-declared variables

To handle multi-declared variables, the first declaration will be considered correct, and any following declarations with the same name (in the same scope) will be considered multi-declared. The attribute `IdDecl.isMultiDeclared` is used for this. An example is shown below.

```
let
  a = 1.0
  a = 2.0
in
  a
end
```

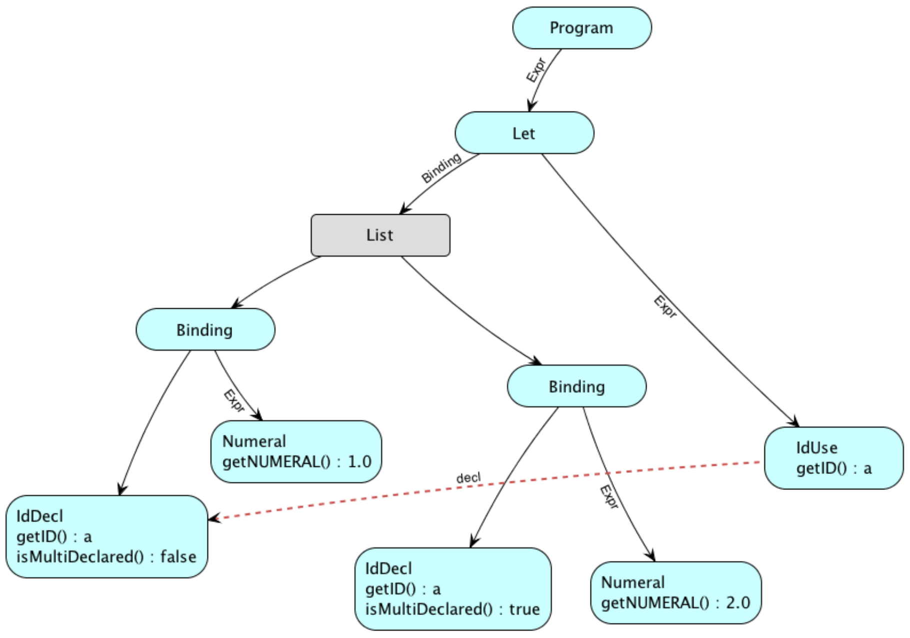

Figure 3: Multi-declared variables

[1]Note that the declarations in a `Let` are called `Bindings` in the abstract grammar. This is because in the Calc language, these declarations bind a value to a variable, like in functional programming.

The attribute `isMultiDeclared` can be computed by checking what declaration is obtained by doing a lookup at the declaration site. If another declaration is obtained, the declaration is multi-declared. Study the boolean attribute `IdDecl.isMultiDeclared` in `src/jastadd/NameAnalysis.jrag` to understand how this is computed.

1.2.4 Circularly defined variables

► A variable might be circularly defined. This happens when it is used inside its own defining expression. We will capture this by an attribute `IdUse.isCircular`. Study this attribute in `src/jastadd/NameAnalysis.jrag` and try to understand how it is implemented. An example is shown below.

```
let a =
    let b = 2.0 in a * b end
in
  a * 3.0
end
```

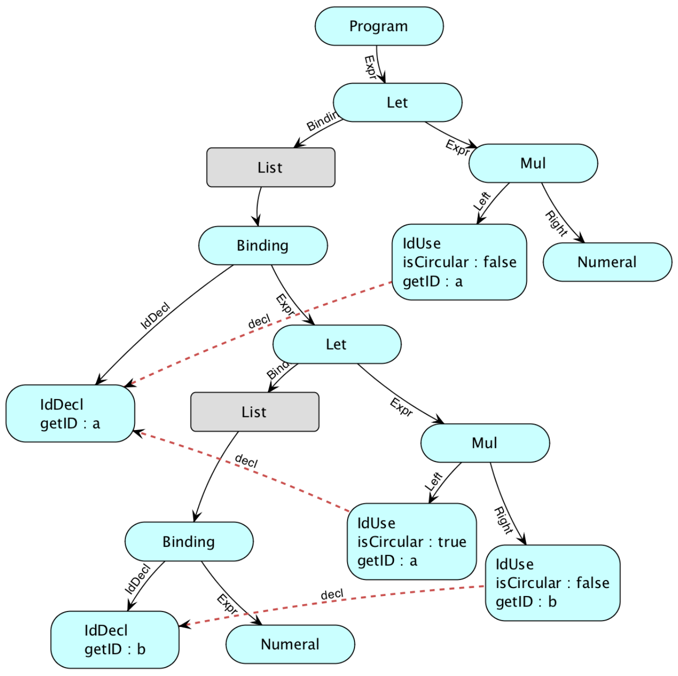

Figure 4: A circularly defined variable

1.3 Unknown declarations

Support for unknown (missing) declarations, is implemented in the module `src/jastadd/UnknownDecl.jrag`. Unknown declarations are implemented using the *Null object pattern*[2]: an object is used instead of the `null` value. The abstract grammar contains a subclass to `IdDecl` called `UnknownDecl`:

```
UnknownDecl: IdDecl;
```

All unknown declarations are represented by the same object. To create such an object, we use a *non-terminal attribute* (NTA), that is, an attribute whose value is a *new subtree*. An ordinary reference attribute refers to an existing tree node, but an NTA, in contrast, refers to a new subtree created by its defining equation. NTAs are used for creating AST nodes that were not created by the parser. The following code adds an `UnknownDecl` as an NTA to the `Program` node.

```
syn nta UnknownDecl Program.unknownDecl() = new UnknownDecl("<unknown>");
```

Later in this assignment, we will use NTAs for representing predefined functions and primitive types in the SimpliC language.

[2]See `http://en.wikipedia.org/wiki/Null_Object_pattern`

To give easy access to the `UnknownDecl` object from other places in the AST, we will use a common pattern called *Root Attribute*, which gives access to an attribute in the root from anywhere in the AST. This is accomplished by declaring the attribute as an inherited attribute for `ASTNode` (thereby giving all nodes access to it), and by an equation in the root that propagates the value to all its children. Through the JastAdd broadcast mechanism, the value is propagated throughout the complete subtree:

```
inh UnknownDecl ASTNode.unknownDecl();
eq Program.getChild().unknownDecl() = unknownDecl();
```

This attribute is used in the name analysis. We can also see another attribute `isUnknown`. This attribute is simply defined as:

```
syn boolean IdDecl.isUnknown() = false;
eq UnknownDecl.isUnknown() = true;
```

The attribute `isUnknown` illustrates the benefits of using the null object pattern: we can define attributes that are implemented differently for normal objects (`IdDecls`) and the null object (`UnknownDecl`).

▶ Study the module `src/jastadd/UnknownDecl.jrag`, and look at the example in Figure 2 to understand the definition.

1.4 Collecting errors

Compile-time errors are implemented in the module `src/jastadd/Errors.jrag`. The errors are represented by an attribute `Program.errors` that is a set of error messages. The `errors` attribute is declared as a *collection attribute*:

```
coll Set<ErrorMessage> Program.errors() [new TreeSet<ErrorMessage>()] with add root Program;
```

The value of a collection attribute is defined by *contribution* rules. The contributions are automatically collected by the attribute evaluator, traversing the AST and adding each contribution to the collection. In this case, the initial value of the collection is a `new TreeSet<ErrorMessage>()`, and the contributions are added with the method `add` on `TreeSet`. The type `ErrorMessage` contains an error message and a line number. The set is sorted by the line number.

The final part of the collection declaration, `root Program`, restricts the traversal to the subtree rooted by the `Program` node containing the `errors` attribute. In this case, the whole AST will be traversed, since `Program` is also the root of the whole AST.

The compiler checks three kinds of errors, each defined using a contribution: undeclared variables, multi-declared variables, and circularly defined variables. Consider the example for undeclared variables below. The contribution specifies

- what error message to add (in case there actually is an error)
- when to add the error message (there is an undeclared variable)
- what collection attribute it should be added to (`Program.errors()` in this case)
- which node contains the collection attribute (`program` in this case)

```
IdUse contributes error("symbol '" + getID() + "' is not declared")
when decl().isUnknown()
to Program.errors() for program();
```

The `program` attribute is an inherited attribute again implemented using the Root Attribute pattern, that propagates a reference to the root down to all nodes in the AST. The `error` method creates an object of the type `ErrorMessage` and sets the line number.

A key benefit of using collection attributes is that contributions can be spread out in different aspects. This way, if the language is extended, or if new analyses like type checking are added, new kinds of compile-time errors can easily be added. JastAdd will automatically collect all contributions to a collection attribute.

▶ Study the module `src/jastadd/Errors.jrag` for compile-time errors, and make sure you understand how the collection of error messages works.

1.5 Demand evaluation and attribute caching

Attributes in JastAdd are *evaluated on demand*, that is, computed when they are accessed. When an attribute is accessed, the corresponding equation is computed. If an attribute is evaluated several times, the equation will be computed several times, making attribute evaluation exponential in the number of attributes in the worst case. To get linear behavior, JastAdd can cache the attributes, that is, store the value of an accessed attribute for further accesses. This is also called *memoization.* In CalcRAG, all attributes are cached. By default, no attributes are cached by JastAdd, but by adding the option `--cache=all` in `build.gradle`, all attributes are cached.

Memoization is possible because the equation right-hand side must be a pure function, i.e., it always returns the same result, for a given input, and it does not have any externally visible side-effects.

1.6 Generated Java code

JastAdd takes a set of `.jrag` aspects and `.ast` abstract grammars and generates Java classes. Each attribute is translated to a method in the class the attribute is declared on. For example, for the attribute `IdUse.decl`, a method `decl()` is generated in the class `IdUse`. JastAdd generates different code depending on the kind of attribute. The simplest attributes are synthesized attributes. These correspond to ordinary methods with extra code that adds caching of attributes and a circularity check. An attribute instance that depends on itself is ill defined, unless it is explicitly declared as `circular`, and JastAdd throws an exception during execution of such an attribute. Note that an attribute can access the same attribute of another node without necessarily being circular. For example, the `lookup` attribute accesses `lookup` attributes of other nodes. Some problems are naturally circular, and in those cases, attributes on the cycle can be explicitly declared as `circular`. JastAdd will then solve the equations using iteration, running the equations over and over until they are solved. More on circular attributes in the next assignment.

▶ Look at the method for the `IdUse.decl` attribute in the generated code (`src/gen/lang/ast/IdUse.java`). You can see that JastAdd generates code for caching the attribute in an instance variable `decl_value`. This is because the option `--cache=all` is set in `build.gradle`.

▶ Write an attribute that is circular (without using the `circular` keyword). Invoke the attribute and verify that an exception is thrown.

2 Semantic Analysis for SimpliC

You will now extend your compiler for SimpliC from assignment 2 or 3 by semantic analysis using RAGs. If you are extending the assignment 3 version, you can simply create a new main program that does not call the visitors or aspects from assignment 2.

▶ Make sure the code you start from builds and tests correctly. Make a copy of your code, so that you can return to a well-defined state, in case you run into a blind alley and want to discard your changes. (Or better, use a version control system like Git.)

2.1 Name analysis and error checking

▶ Implement name analysis for SimpliC using RAGs. The semantics regarding declaration order is a bit different from assignment 3. Declare before use will now only apply for variables, and *not* functions, that is, the order in which the functions are declared is not significant. However, the semantics regarding shadowing and that both variables and functions share the same name space are the same as in assignment 3. Ignore predefined functions (`read` and `print`) for the moment. Use the Lookup pattern and the Null object pattern, as for the Calc example.

▶ Add error checking by using a collection attribute for errors, i.e., in the same way as in the Calc example. Report undeclared names and multi-declared names. Add suitable test cases. You can copy the `CalcRAG/src/java/TestNameAnalysis.java` testing framework to easily add name analysis test cases. As you see in the testing code, a source file is parsed into an AST, then the `program.errors()` attribute is accessed and printed to a file, which then is compared with the expected output file.

2.2 Predefined functions

The SimpliC language has two predefined functions: `read` and `print`. These functions can be represented as an NTA in the AST. In this case, we want the NTA to be a `List` node that contains one subtree for each of the functions. The following code illustrates how such an NTA can be specified.

```
syn nta List<FunctionDecl> Program.predefinedFunctions() {
        List<FunctionDecl> list = new List<FunctionDecl>();
        // Create objects of type FunctionDecl and add them to the list
        return list;
}
```

The Root Attribute pattern can be used for propagating the `predefinedFunctions` attribute to the whole AST, or to the places where it is needed.

▶ Extend your name analysis with support for predefined functions. Define the attribute `predefinedFunctions`. Use this attribute in the equation of the `lookup` attribute. Make sure that you have test cases that cover the use of predefined functions and that predefined functions cannot by redeclared by other functions.

2.3 Type analysis

Type analysis is about assigning types to expressions. There are three types in SimpliC: integer, boolean and the unknown type. Variables, parameters and the return type of a function can only be of integer type. This limitation exists to make the assignment easier. Arithmetic expressions are of type integer and boolean expressions are of type boolean. The statements `if` and `while` require the condition to be of type boolean. The unknown type is assigned to variable uses that refer to unknown names. An unknown type makes it easier to limit the propagation of errors – we only want to report relevant errors and not errors that are caused by other errors. The types can be modelled as the following abstract grammar fragment shows:

```
abstract Type;
IntType: Type;
BoolType: Type;
UnknownType: Type;
```

▶ Implement the primitive types using NTAs, in the same way as you did for the unknown name. Use the Root Attribute pattern to propagate references to these objects throughout the program.

▶ Add the attribute `type` to both expressions and `IdDecl`, returning the type of the expression or declaration. Give appropriate equations for the attributes.

We now want to report type errors. This can be done by adding a contribution to the `errors` attribute. One approach is to add an inherited attribute `expectedType` to expressions that describes the type to be expected. The contribution can then compare the expected type with the actual type, and if they are incompatible, an error is reported. It is also useful to add an attribute `compatibleType` on `Type` that takes another type as a parameter and compares the two types and tells if they are compatible. To limit the propagation of type errors, the unknown type should be compatible with all other types.

▶ Add the attributes `expectedType` and `compatibleType`. Add a contribution that contributes an error message if two types are incompatible.

2.4 More errors

We will now report more errors.

▶ Since variables and functions share the same name space, a variable use can refer to a function and a function call can refer to a variable. Report these errors. Add two inherited attributes `isVariable` and `isFunction` to `IdDecl` and use them in the contributions.

▶ Report if the number of actual arguments in a function call does not match with the number of formal parameters of that function. You can add an attribute `function` on `IdDecl` to access the function declaration, and from there the number of formal parameters.

3 Common Mistakes

Common coding errors that you should try to avoid:

- Creating a new instance of `IntType` or `BoolType` for each `type()` equation. There should be only one instance of `IntType` and `BoolType`, see `UnknownDecl` in CalcRAG. New instances of AST nodes should only be created to define values of NTAs.
- Using `instanceof` in the `compatibleType` attribute.

4 Optional Challenges

When you have completed the compulsory part of the assignment you may try one or more of these optional challenges:

- Add boolean literals (`true` and `false`)
- Add the type `void` that can be used as the return type of a function
- Add the type `bool` as a type that can be used for return types, variables and parameters.