Lund University
Computer Science
Niklas Fors, Görel Hedin, Christoff Bürger

Compilers
EDAN65
2018-09-29

Programming Assignment 5
Interpreter and Static Analysis

The goal of this assignment is to implement an interpreter and a simple static analysis for the SimpliC language. You will use reference attribute grammars for computing program properties, and inter-type methods for implementing the interpreter.

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 Interpreter for SimpliC

You will implement the interpreter as an extension of your SimpliC compiler from assignment 4. The interpreter will execute SimpliC programs by traversing the corresponding AST. It will be implemented using inter-type methods, with the aid of attributes. The implementation will be done in small steps, each step supporting interpretation of more complex programs.

▶ Make sure the code you start from builds and tests correctly.

1.1 Initial interpreter and activation records

To evaluate a program, the interpreter will create an *activation record* for each new function call (i.e., an instance of the function). An activation record stores the parameters and local variables of the function instance. You will introduce a class `ActivationRecord` to represent the activation record, with methods to store and retrieve the values of parameters/variables.

Only valid programs with no compile-time errors should be interpreted.

The following methods can be added in a new aspect to implement the interpreter.

```
public void Program.eval();
public int FunctionDecl.eval(ActivationRecord actrec);
public void Stmt.eval(ActivationRecord actrec);
public int Expr.eval(ActivationRecord actrec);
```

In the API above, the evaluation of an expression returns an integer. To handle Boolean values, they can be mapped to integers. The class `ActivationRecord` should map names (`String`) to integers (`Integer`). You should add methods to `ActivationRecord` to store and retrieve variable/parameter values, for example `put` and `get`.

▶ Implement an initial version of the interpreter aspect that has the `eval` methods above. For the time being, you can implement them by just letting them throw an unchecked runtime exception (`throw new RuntimeException();`). You can implement the `ActivationRecord` class in the same aspect. Implement a main program `Interpreter.java` that parses in an AST, checks if it has no compile-time errors, and if so, runs the program by calling `eval` on a `Program` AST.

▶ Now enhance the interpreter by letting the method `Program.eval` invoke `eval` on the function called `main` (if it exists). Use the attributes from the name analysis to retrieve the `main` function. Report a run-time error if the `main` function is missing. Note that a `FunctionDecl` should be given a new instance of `ActivationRecord` when it is evaluated.

1.2 Literals and arithmetic expressions

▶ Extend the interpreter by adding support for integer literals and arithmetic expressions. Let functions invoke the `eval` method on all their statements and let variable declarations evaluate their expressions. As a temporary step in implementing the interpreter, let the variable declarations print the initialization results, instead of storing them. For example, running the following program could result in printing `7 20`:

```
int main() {
        int a = 1+2*3;
        int b = 4*5;
}
```

1.3 The predefined function print

▶ Now, extend the interpreter to support the predefined function `print`, which prints the value of its argument. Running the following program should print `7 10`:

```
int main() {
        print(1+2*3);
        print(10);
}
```

We now have an interpreter that correctly executes simple programs!

1.4 Testing

Before continuing to extend the interpreter to handle more parts of the language, we will add a framework for parameterized unit testing, in the same style as for previous assignments. This will allow us to easily add new automated tests by just adding `.in` and `.expected` files.

The `print` function prints a value to the standard output. To test the interpreter, we can use the `print` function and compare the output of the interpretation of a program with an expected output. The expected output is located in the `.expected` file. To do the comparison, we need to redirect the standard output to an object, and then compare it with an expected output. The code below illustrates how this can be done. Note that we need to reset the standard output after the test case is run.

```
public void runTest() throws Exception {
        PrintStream out = System.out;
        try {
                Program program = (Program) parse(inFile);
                ByteArrayOutputStream baos = new ByteArrayOutputStream();
                System.setOut(new PrintStream(baos));
                program.eval();
                compareOutput(baos.toString(), outFile, expectedFile);
        } finally {
                System.setOut(out);
        }
}
```

▶ Implement the testing framework for the interpreter. Look for inspiration in previous test classes. Note that we should also check that the program is free from errors before the program is interpreted. Continuously add test cases as the interpreter is extended to support more language constructs.

1.5 Variables

▶ Implement support for variable assignments and variable uses. Use the class `ActivationRecord` to store the values of the variables. Note that a variable can be assigned in both a variable declaration and in an assignment. You do not need to consider variable shadowing or scoping at this point.

It can happen that a variable is used before it is assigned a value. There are several ways to handle such situations. Some common ways are listed below:

- Return a default value (for example, 0). Fields in Java are automatically set to default values.
- Throw an exception and abort the execution. This is what happens in Python.
- Add static analysis that requires variables to be *definitely assigned* before use. Definite assignment is a static (compile-time) analysis that checks if a variable is guaranteed to always have a value assigned to it for all possible program executions. Local variables in Java are required to be definitely assigned before use. The analysis cannot be done precisely in the general case, since the program execution can depend on input. However, the analysis used in a Java compiler is a safe conservative approximation that will reject all invalid programs, but might also reject some valid programs. Figure 1 shows an example program that will be rejected by a Java compiler, but that would actually be valid for any execution.
- Some languages do not define the behavior. For example, using uninitialized local variables in functions in C has undefined behavior. The value returned can be any value (what happens to be in memory at that position and at that time).

```
int m(boolean b) {
        int a;
        if (b) a = 1;
        if (!b) a = 2;
        return a;
}
```

Figure 1: A method in Java that causes a compile time error. The variable `a` is not definitely assigned according to the Java Language Specification (JLS).

▶ Handle uninitialized variables. Choose one of the four approaches described above.

1.6 Boolean expressions, and `if` and `while` statements

▶ Implement support for boolean expressions. Map boolean values to integer values.

▶ Implement support for `if` and `while` statements.

1.7 Function calls

When a function is called, a new activation record should be created containing the mapping of actual arguments to formal parameters. As an example consider the following program:

```
int add(int a, int b) {
        return a + b;
}

int main() {
        int i = 10;
        print(add(i, 20)); // Call with 2 arguments: first value of i, second value 20.
}
```

When the function `add` is called, the initial mapping should be $\{a \mapsto 10, b \mapsto 20\}$. The actual arguments are evaluated and mapped to the formal parameters. Note that the value of `i` is the first argument of the `add(i, 20)` function call; before the called function is executed the `a` parameter therefore must be initialised with a copy of the value of `i` (call by value). The actual variable `i` is not visible in the context of `add`.

After a function call has been processed, execution proceeds in the calling function. This means that the activation record of the calling function must be restored. This is straightforward since each `eval` has the activation record to use as an argument.

▶ Implement function calls. First, create a new activation record and initialize it with the formal parameters of the call, each initialised with the corresponding given argument. Then evaluate the called function's body, passing it the new activation record. Add tests cases. Since you have not yet implemented the return statement, you can test that execution works by letting it print a value instead of returning it.

Optional task. Implement support for global variables. One solution is to maintain a special global activation record that is initialized before calling `main`.

1.8 Return statements

The `return` statement can specify an expression that is evaluated and returned to the caller of the function. A `return` statement can be nested inside, for example, a `while` statement. When the `return` statement is evaluated, the evaluation of the `while` statement should also stop. Thus, the `return` cancels the current evaluation in the function and returns to the caller of the function. Here are some ways to implement this behaviour:

- A field in the activation record can be used to indicate if the current method should return, with another field giving the return value. While evaluating statements in a statement list or block, the return state must be checked to see if the previous statement returned.
- Throw an exception (subtype of `Exception`) in the `return` statement and include the return value in the exception object. The function catches this exception and returns the value. Other statements should only forward the exception, using the `throws` keyword in their `eval` methods.
- Change the method `Stmt.eval(ActivationRecord actrec)` so that it returns a value. Use this value to propagate information about if the statement has executed a `return` statement or not, and if so, include the value computed by the `return` statement. Other statements should look at this value, and in the case a return had been executed, cancel their current evaluation, and return the same value to their caller.

▶ Implement `return` statements. You can use one of the above techniques. In many languages, a return statement is required on every path from the start of the function to an end of the function – if the function is declared to return a value. In our interpreter, we will not require this, but instead return 0 if no `return` statement is executed throughout a function call.

▶ Ensure the previous activation record is restored after a function call terminates. Test this using the recursive `fac` function of Figure 3 (a).

1.9 Variable shadowing

The SimpliC language supports variable shadowing inside functions: if there are two variable declarations of the same name, but in different blocks, this is not considered an error, but the inner variable will instead shadow the outer one in the rest of the inner block. See the example in Fig. 2.

The current implementation of the interpreter does not support these semantics, since the different variables will share the same name in the activation record, and running the above program would result in the printing of three 2s.

```
int main() {
  int i = 0;           // Declaration of i
  if (i == 0) {
    int i = 1;         // Another declaration of i
    if (i == 1) {
      int i = 2;       // A third declaration of i
      print(i);        // Prints 2
    }
    print(i);          // Prints 1
  }
  print(i);            // Prints 0
}
```

Figure 2: Example variable shadowing inside a function

To support variable shadowing, we can assign unique names to all variables in a function. One way is to prefix the variable name with the statement index recursively. For example, we could give the three declarations of `i` the following unique names (assuming statements are numbered from 0 and up):

`0_i`	meaning the declaration i that is statement 0 in the function
`1_0_i`	meaning the declaration i that is statement 0 in statement 1 in the function
`1_1_0_i`	meaning the declaration i that is statement 0 in statement 1 in statement 1 in the function

If we wanted a fast interpreter, we would not use a map from names to values in the activation record. Instead, we would assign each variable a unique number that is an integer between 0 and the total number of variables in the function. We could then represent the variables as an array, using the unique number as an index in the array. This is what we will do in the next assignment, when we are generating Assembly code. We will then represent the array as a sequence of words in memory.

▶ Implement an attribute `uniqueName` on `IdDecl` that returns a unique name of the variable (unique within the function). Modify the interpreter to use this attribute for the variable names, to handle variable shadowing.

Note that the scope of a variable declaration does not extend past the block in which it is declared. For example, in the following program, the interpreter should find that the value of `a` is undefined in the print statement:

```
int main() {
  if (...) { int a = 10; }
  if (...) { int a; print(a); }
}
```

Optional task. Consider what would be needed to extend SimpliC to support nested function declarations, where inner functions could access variables in outer functions. How would your implementation need to be changed with respect to the static name analysis? How would the interpreter need to be changed to support the appropriate dynamic semantics? *Hint!* Use a static link in the activation record.

1.10 Reading input

▶ Implement the predefined function `read()` so that it reads an integer from standard input. *Hint!* You might use the class `java.util.Scanner` and its method `nextInt()`.

In case the user inputs something that is not an integer, you can either interpret this as `0` and continue execution, or as a runtime error where you halt the program execution and output some suitable runtime error message.

You should create some test programs for `read()`, but you don't need to automate them. Place them in a separate directory `examples`, and run them interactively to test them.

1.11 Complete the interpreter and add missing test cases

Your interpreter should handle any program that can be parsed without static-semantic errors.

- ▶ Make sure you have automated test cases for all the previous tasks (except for `read()`, where you instead have example programs).
- ▶ Add one automated test case that covers all syntactic constructs in your language.
- ▶ Add the `gcd` program (see 3.1 in assignment 2) as an example program that you can run interactively.
- ▶ Add an automated test case for a variant of the `gcd` program where you have replaced the two `read()`s with the constants `24` and `20`.

Optional task. Add more example programs that do useful computations. For example, write a SimpliC program that reads a fractional number and uses a `gcd` function to simplify it, e.g., simplifying `8/12` to `2/3`.

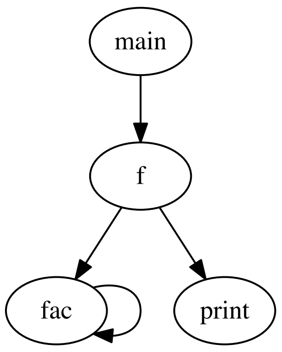
To be able to write more useful programs, you might find it convenient to extend SimpliC with the possibility to print strings and to call a predefined function `random()`.

2 Call graph for SimpliC

A call graph describes the relationship between functions as a directed graph, where an edge (`f`, `g`) represents that the function `f` calls `g`. An example of a call graph is shown in Figure 3. *Direct recursion* (`f` calls `f`) corresponds to a self-loop in the graph. *Indirect recursion* (e.g., `f` calls `g` and `g` calls `f`) corresponds to a cycle in the graph involving two or more edges.

```
int fac(int n) {
  if (n <= 1) { return 1; }
  else { return fac(n-1) * n; }
}
int f() {
  print(fac(5));
}
int main() {
  f();
}
```

(a)



(b)

```
digraph G {
  fac -> fac;
  f -> print;
  f -> fac;
  main -> f;
  fac;
  f;
  main;
}
```

(c)

Figure 3: A call graph of a program calculating the factorial of 5. Figure 3c shows the textual representation of the visualized graph expressed in the `dot` language.

The graph in Figure 3b has been created using the program `dot`, which is included in the Graphviz package[1]. The `dot` program takes as an input a textual representation of a graph expressed in the dot language. An example is shown in Figure 3c. This example describes a directed graph (`digraph`), a number of edges (`a -> b`) and a number of nodes (`a`).

A call graph can either be *static* or *dynamic*, where a static call graph is an approximation of all dynamic call graphs. Dynamic call graphs describe the call graph for one program execution with a given input, that is, the relationship between the functions that were actually executed for this input. Language analysis can be divided into two groups: *static analysis* and *dynamic analysis*. Static analysis is about analyzing the source code and dynamic analysis is about analyzing a program execution.

- ▶ Implement a collection attribute `functionCalls` that is the set of functions called by a function (in the static sense). The value should be a set of function declarations, since several function calls of the same

[1]See `http://www.graphviz.org`

function should correspond to one element in the set. It can be useful to add an inherited attribute `enclosingFunction` for Statements, which returns the function a statement is part of. Use test cases to check that your `functionCalls` attribute computes the correct value.

Optional task. The attribute `functionCalls` can be used to generate call graphs expressed in the `dot` language. When the dot file is generated, a visualization can be created using the `dot` program. Generate a call graph and visualize it. A PDF file can be created as follows.

```
$ dot -Tpdf call-graph.dot -o call-graph.pdf
```

2.1 Reachability

The reachability relation is the transitive closure of the `functionCalls` relation, that is, the set of functions that are reachable from a function `f` in the call graph. For example, the reachability of the `main` function in Figure 3 is {`f`, `fac`, `print`}. We can define the reachability for a function f as follows.

$$\text{reachable}(f) = \bigcup_{c \in \text{functionCalls}(f)} (\{c\} \cup \text{reachable}(c))$$

Note that this definition is circular, since call graphs can be circular. If we define the reachability as an attribute, then the attribute is circular, meaning that the value of the attribute can depend on itself. The evaluation engine evaluates circular attributes using a fixed-point iteration. A circular attribute is well-defined if the domain of the attributes involved in the cycle can be arranged in a lattice of finite height and all equations involved are monotonic functions. This condition is fulfilled for the `reachable` attribute since the values involved are sets of functions, the operation used is union, and the lattice will be of finite height since the set of all functions is finite for any given SimpliC program.

Consult the JastAdd reference manual for the definition syntax of circular attributes (`http://jastadd.org/web/documentation/reference-manual.php`).

▶ Implement the circular attribute `reachable`. Use test cases to check that your `reachable` attribute computes the correct value.

2.1.1 Identifying dead functions

Optional task. We can use the `reachability` attribute to identify functions that are not used in a program, that is, functions that are not in the reachability set of the `main` function. Identify and print functions that are not used in a program.

3 Topics for Discussion

1. In this assignment we have implemented the interpreter as a method, but the call graph and reachable sets as declarative attributes. When should we use declarative attributes and when should we use methods?