Architecture of the Léa compiler

Michael Färber

November 14, 2012

1 Roadmap

I propose to implement little bits of language functionality one after another. As a start, we should be able to make an AST for the following source code:

```
procedure main()
{
        writeln("Hello world!");
}
```

Then, we might add constants as such:

```
MEANING = 42;

procedure main()
{
        writeln("Hello world!");
}
```

After this, conditionals might follow:

```
MEANING = 42;

procedure main()
{
        if (MEANING = 42)
        {
                writeln("Hello world!");
        }
}
```

Then variable declarations/assignments:

```
procedure main()
{
        meaning : int;
        meaning := 42;
```

```
        if (meaning = 42)
        {
                writeln("Hello world!");
        }
}
```

Then function calling:

```
procedure print_greetings()
{
        writeln("Hello from print_greetings()!");
}

procedure main()
{
        writeln("Hello world!");
        print_greetings();
}
```

Then function returns:

```
function calculate_meaning() : int
{
        return 42;
}

procedure main()
{
        meaning : int;
        meaning := calculate_meaning();

        if (meaning = 42)
        {
                writeln("Hello world!");
        }
}
```

And so on ...

2 Syntax

The syntax part is concerned with Cup/JFlex. The JFlex part is the interface to the semantics part. It has to:

- add constant definitions to the constant table
- add type definitions to the type table
- add function definitions to the function table

For these tasks, the JFlex part needs access to following classes:

- SyntaxTree
- Environment
- Type
- Constant
- Function

Furthermore, it needs access to the type and constant tables.

3 Semantics

This part is concerned with the checking of the abstract syntax tree (AST) and the generation of the intermediate code.

3.1 Abstract syntax tree

JFlex constructs an AST for each function in a Léa source file and adds it to the function table. During the construction of the AST, the AST has to check if its instructions are valid, i.e. type consistent.

We use several classes derived from a basic SyntaxTree class to represent the different instructions that may appear in a Léa program. For example, we may have the following classes:

- Assignment.java
- FunctionCall.java
- List.java: unary syntax tree which represents a list
- Return.java
- Succ.java
- Tuple.java
- Variable.java
- ...

We might also use interfaces to differentiate different types of syntax tree elements:

- Expression.java: might provide function getType()
- Instruction.java

Also, to implement drawing, we might have base classes like:

- BinarySyntaxTree
- UnarySyntaxTree

Note that declarations of new variables are not present as instructions in the SyntaxTree, as the information about variables is already contained in the environments belonging to each AST.

We should also provide functions to create a graph from an AST. At the end of the parsing process, we may output the graphs for all functions to separate files.

3.2 Environment

An environment stores tuples (var_name, var_type). Internally, this might be implemented as a Map<String, Type>. Each AST has a reference to an environment, by which the AST may access information about objects in its environment.

3.3 Type

The Type class is the superclass of several derived classes, such as:

- StructType
- IntType
- FloatType
- ListType
- ...

Each of these classes has to be able to check equality with another type.

Because objects may have member variables or functions (think foo.toString() or structVar.structMember), the type class needs functions to search for member variables respectively functions.

3.4 Constant

The Constant class saves constant values. It is the superclass of:

- ListConstant
- TupleConstant
- IntConstant
- FloatConstant
- StringConstant

- ...

Each of these classes has to provide a function like getType().

Note that ListConstant and TupleConstant will only be used for constant definitions; if we construct a list/tuple inside a function, we use the List/Tuple class instead, which allows more complex, non-constant expressions inside list/tuple definitions, e.g.:

```
foo_list := [1, 1+1, mult(1,2)]
```

3.5 Function

The function class saves information about a function, which is:

- input arguments: list of tuples (var_name, var_type)
- output type
- syntax tree of function

3.6 Constants table

The constants table saves tuples (const_name, const_value). It may be implemented as Map<String, Constant>. For reasons of simplicity, I propose that we do not allow constant definitions like

```
foo = 1+1;
```

because this might again require a syntax tree for foo, which we would have to check for non-constant expressions.

3.7 Type table

The type table stores type definitions, such as:

```
int_list = struct {
        next: list
        elem: int
}

color = enum (RED, GREEN, BLUE)
```

For this, we have to store tuples (type_name, type) in a table, which is the type table. This may be realised as Map<String, Type>.

3.8 Function table

The function table keeps track of functions defined in a Léa source file. The function table may be realised as Map<String, Function>.