Tool language extension - Liberal Syntax

Less pain in tool

EPFL, Computer Language Processing 2016-17
Final Lab (mini project)

Gregoire Hirt Thierry Bossy Rafael Pizarro

January 13, 2017

1. Introduction

During the semester, we implemented an interpreter and a compiler in scala for a simple object oriented programming language : Tool. The coding constraints for this language are far more restrictive as in scala. We chose to add some improvements, generally inspired from scala, to this language and permit more coding freedom. In order to add more coding freedom, we implemented 3 different problems:

- Semicolon interference
- Infix Operations
- Parameterless calls

To write the initial compiler and interpreter we divided the problem in a 5 stage pipeline.

```
Lexer → Parser → Name Analysis → Type Check
→ Code Constructor
```

1.1 Lexer

The Lexer is the step where we filter the useful parts of the input text. For each chain of characters we will define a Token, for example :

```
; −> SEMICOLON
while −> WHILE
charly −> ID
```

The challenges of the lexer are :

- Define the tokens we need
- Filter white spaces and comments

After the lexer runs, we get a chain of tokens as the result. For example :

```
def foo(): Int ...
```

Becomes

```
DEF() ˜ ID() ˜ LBRACKET()
˜ RBRACKET() ˜ COLON() ˜ INT()...
```

In the case of our extension, we had to add the following new token for the semicolon interference: LINEJUMP that will make a token out of '\r' and '\n' characters.

1.2 Parser

Parser is the step where we define the grammar of the language and then transform it into an abstract form. The grammar is the set of rules that the code structure must follow. The best way to parse a list of tokens is to generate an Abstract Syntax Tree (AST).

The challenges of the parser are :

- In order to parse the tree in linear time, we must have the grammar in LL1, this might be sometimes impossible unless we put some conditions in the grammar
- Create the tree nodes depending on the those conditions.

After the parser runs, we get an Abstract Syntax Tree (AST).

In the case of our extension, many changes had to be done in the grammar and constructor that we will explain in the other chapters.

1.3 Name Analysis

Name Analysis is the step where we check if each variable call has the right conditions.

In order to do this we pass throught the tree and create a 'Symbol' for each identifier. When creating these symbols we check that conditions as following hold:

- Scope of the variable, that means for example if we define it in a function, it must only be accesed from that function.
- Existence, which means that an accessed variable must be defined before.
- Etc..

In the case of our extension, we didn't make any changes for this part.

1.4 Type Check

Type Checking is the step where we check if the types of expressions match what we expect them to be. For example:

```
program Program {
  var b : B;
  var array : Int[];
  do(b.foo() + 1); // success, check return of super class
  do(array + b.foo()); // fail, sum of int with an int array
}
class A { def foo(): Int = {...}}
class B extends A {...}
```

The challenge of the type checker is to go through the program from the Leafs to the Root and to check that the Nodes have the right types.

In the case of our project, we changed it to allow +, -, *, / operations for Class Types.

The Parsing, Name analyzis and Type Checking steps check if a program is valid or not. If we have a valid program we are ready for the last step : code generation.

1.5 Code Generation

In this last step we end up generating JVM code from the final structure we created, to be executable by the JVM. We used the 'Cafebabe' library, which makes the JVM instructions creation much more easier. For example:

```
Node(Plus(Variable(id1),Variable(id2))
```

Generates

```
iload_n1
iload_n2
iadd
```

In the case of our project, we changed it for the infix operators part.

2. Semicolon Inference

The semicolon is not required anymore at the end of an instruction when the next instruction is in another line, like in scala. It is still needed for a statement which precedes another instruction on the same line.

2.1 Example

```
class Matrix2 {
  def meaninglessTransformation1() : Matrix2 = {
    var bias : Matrix2; bias.init(1,0,0,1);
    return this.times(bias1);
  }
}
```

Becomes

```
class Matrix2 {
  def meaninglessTransformation1() : Matrix2 = {
    var bias : Matrix2; bias.init(1,0,0,1)
    return this.times(bias1)
  }
}
```

2.2 Implementation

Up until now, line jumps were ignored by the lexer as any whitespace. However, in order to add semicolon inference, we consider them as semicolon when needed. We have defined a new token for line jumps in the lexer. We first thought we had to treat this token in the grammar, but it was a very bad idea not far from impossible. The semicolon inference has to be done after the lexing and before the parsing. We then created a huge method in Parser.scala which transforms the list of Tokens before parsing it. We helped ourselves with the following simple list of rules for scala semicolon inference [Takalkar] :

A line ending is treated as a semicolon unless one of the following conditions is true :

- *The line in question ends in a word that would not be legal as the end of a statement, such as a period or an infix operator.*
- *The next line begins with a word that cannot start a statement.*
- *The line ends while inside parentheses (. . .) or brackets [. . .], because these cannot contain multiple statements anyway.*

We noticed that two cases were not correctly handled by these rules :

1. The multiple-lines if-statements, where we don't want the line jump to be treated like a semicolon :

```
if(condition) // no semicolon
  do(expression) // semicolon
```

2. The closing braces on the same line than the statement, where we want a semicolon before '}' even if there is no line jump :

```
if(condition) { println("true"); }
else { println("false"); }
```

The 'semicolonInference' methods takes the list of tokens resulting from the token iterator then returns an adapted list with no more LINEJUMP token. This transformation consists of the following steps :

```
def semicolonInference(list: List[Token]): List[Token]
```

- Add the line jump token before each closing brace token to solve the 2nd issue described above.
- Remove line jumps before non-starting tokens and after non-ending tokens
- Remove line jumps contained in parenthesis or brackets
- Remove line jumps placed after an if-condition (1st issue above)
- Transform each remaining line jump into semicolon

This implementation is backwards compatible, as we can mix the old syntax (semicolon) and the new syntax (no semicolon) at the same time.

3. Methods as infix operators

Methods which take only one parameter can be called as an infix operator. It becomes very fun and easy to enchain such methods. With good names for objects and methods, a programming instruction can looks like a spoken sentence. We add the possibility to call such methods with +, -, *, / operators if they are named "plus", "minus", "times" or "divided" respectively.

3.1 Example

```
class Matrix2 {
  def meaninglessTransformation2() : Matrix2 = {
    var bias1 : Matrix2; bias1.init(1,0,0,1);
    var bias2 : Matrix2; bias2.init(1, 1, 0,0);
    return this.times(bias1).plus(this.times(bias2));
  }
}
```

Becomes

```
class Matrix2 {
  def meaninglessTransformation2() : Matrix2 = {
    var bias1 : Matrix2; bias1.init(1,0,0,1);
    var bias2 : Matrix2; bias2.init(1, 1, 0, 0);
    return this * bias1 + this * bias2;
  }
}
```

3.2 Implementation

The first step to implement this feature was to change the 'dot' operation definition in the grammars. For the LL1 grammar, we had to add a form to the operator, with the possibility of not having a dot, followed by an identifier (method) and one single argument with parenthesis or not.

```
'OpDot ::= DOT() ~ 'DotEnd ~ 'OpDot
| 'Identifier ~ 'ExprTerm ~ 'OpDot
| epsilon(),
'DotEnd ::= LENGTH()
| 'Identifier ~ LPAREN() ~ 'Args ~ RPAREN(),
```

To keep our grammar LL1, we had to accept only simple expressions ('ExprTerm) as argument in a method call as infix operator, and not composed expressions with operator. It seems legit but should be clarified for these non-accepted cases :

- Array Read: obj meth arr[index] is forbidden, we can use instead:
  obj.meth(arr[index]) or obj meth (arr[index])
- Bang operation: obj meth !expr is forbidden, we can use instead:
  obj.meth(!expr)) or obj meth (!expr)
- Dot operation: obj meth obj.meth is not equivalent to
  obj.meth(obj.meth) or obj meth (obj.meth)
  but it's equivalent to obj.meth(obj).meth()
  (not forbidden because of the parameterless calls feature that we'll describe after.)

A similar and simpler change was also added in the non-LL1 grammar. The two constructors had to be adjusted in consequence. The operator overloading is then done at type checking and code generation stages by allowing operators expression nodes to have Class typed operands if the left one has the method named after the operator. The changes in the Evaluator follow the same logic.

4. Parameterless Calls

To define or call a method which takes no parameter, you don't need to put parenthesis anymore. This works for 'new' instances too.

4.1 Example

```
new Matrix2();
...
class Matrix2 {
  def isInversible() : Boolean = {
    var determinant : Int;
    determinant = this.getDeterminant();
    return determinant != 0;
  }
}
```

Becomes

```
new Matrix2;
...
class Matrix2 {
  def isInversible : Boolean = {
    var determinant : Int;
    determinant = this.getDeterminant;
    return determinant != 0;
  }
}
```

4.2 Implementation

This feature was totally implemented on the parser. We had to change the definition of method declarations, dot operation and new expression on the grammar, to make another option for the arguments as following. It gives the possibility to not put parenthesis for the cases we don't need to.

```
...
'MethodDeclaration ::= DEF() ~ 'Identifier ~ 'ParamsOpt
~ COLON() ~ 'Type ~ EQSIGN() ~ LBRACE()
~ 'VarDecs ~ 'Stmts ~ RETURN() ~ 'Expression
~ SEMICOLON() ~ RBRACE(),
'ParamsOpt ::= LPAREN() ~ 'Params ~ RPAREN()
| epsilon(),
...
'ExprTerm ::= ... | NEW() ~ 'NewEnd | ...
'NewEnd ::= 'Identifier 'ParenOpt
  | INT() ~ LBRACKET() ~ 'Expression ~ RBRACKET(),
'ParenOpt ::= LPAREN() ~ RPAREN()
  | epsilon(),
...
```

Similar changes were done in the non-LL1 grammar and the constructors have been adjusted.

5. Possible Extension

We planned to add the possibility to call a statement simply with the expression, without using 'do(...)', and the possibility to call a method for the 'self' object without 'this'. We guessed that it was possible to implement these features keeping our grammar in LL1. It appeared that it was not possible and that these updates needed another grammar.

One possibility for this is to encapsulate a specific non-LL1 grammar for the cases which brakes LL1 property (simple statement, expression terminaison). The parser should have the possibility to call the 2nd grammar only when it needed, and keep using LL1 grammar otherwise.

References

J. Takalkar. rules of semicolon inference. URL `http://jittakal.blogspot.ch/2012/07/scala-rules-of-semicolon-inference.html`.