Code formatter

Compiler Construction '13 Final Report

Yann Gabbud
EPFL
yann.gabbud@epfl.ch

1. Introduction

In the first part of the project we built the global architecture of the compiler.

First of all, we created the lexical analysis phase, the lexer. This phase reads the input file and converts it to a list of tokens. A token is the smallest unit in an input file which is meaningful. For example, a token is the keyword *while* or a bracket. The longest matching rule is used for lexing and all useless informations such as *whitespace* are dropped.

The second phase is the parsing. The parser takes the sequence of tokens produced by the lexer and converts them into an *Abstract Syntax Tree*. The parser consists of two stages. The first one takes the sequence of tokens and converts them into a *parse three* with the help of a *LL1* grammar that we defined. Then, the second stage converts the parse three into an *Abstract Syntax Tree*. The second stage only keeps the useful informations. Commas, parentheses, brackets, etc are dropped.

The third phase is the name analysis. The name analyzer checks that the program follow the Amy naming rules, assigns a unique *identifier* to every name and finally populates the *symbol table*. The symbol table is a map of identifiers to information that is useful later in the compilation.

The fourth phase is the type checking. The type checker checks that the program manipulates correct kind or shape of values. For example, it checks that a boolean is not used where an int should be. If a program passes this phase without errors, it is correct from a compiling point of view.

The last phase is the code generation. The code generator takes the abstract syntax tree and uses it to generate *WebAssembly* bytecode. Finally we can execute the result with nodejs.

In the second part of the project, I add a code formatter to the compiler. When the compiler is run with the argument *–format*, it goes through a different pipeline than the default pipeline explained above. This pipeline result in the printing of the code and comments of the given input files according to formatting rules.

To achieve its task, the code formatter goes through three pipeline stage. The first stage is the lexer. The lexer achieves the same task as before but is modified in order to also stores the comments into a list. This list is then passed to the parser and used later during the printing phase. Note that if the compiler is run by default this step is skipped. The second stage is the parser. The parser do the same job but is slightly modified in order to accept the list passed by the lexer and to pass it further. Finally the last step is the prettyprinter. The prettyprinter receives from the pipeline a *NominalTreeModule* and a *list of comments* and prints everything according to rules in order to have a nice formatted code.

2. Examples

The code formatter is useful to keep the code well structured and easily readable, mainly when there is a lot of nested expression and a lot of indentation. Moreover it allows to have a uniform code if several people working on the same project use different coding styles.

Here is an example that show how it works :

```
/*
 * Hello object defines foo method
 */
object Hello {
  // foo method
  def foo(a: Boolean, x: Int, y: Int, z: Boolean): Boolean = {
    val add: /* weird comment */ Int = x+y;
    if (0<x) { true }
    else {
```

```
      if (a) {



        add match { // a match
          case 0 => false
          case 1 => true // case one
          case _ =>
            // if 1
            if (z) { // if 2
              // if 3
              true
            }
            // else 1
            else { // else 2
              // else 3
              error("wrong input")
            }
        }
      } else { false }
    }
  }
}
```

```
/*
 * Hello object defines foo method
 */
object Hello {
  // foo method
  def foo(a: Boolean, x: Int, y: Int, z: Boolean): Boolean = {
    val add: Int = x + y; /* weird comment */
    if (0 < x) {
      true
    }
    else {
      if (a) {
        add match { // a match
          case 0 =>
            false
          case 1 => // case one
            true
          case _ =>
            // if 1
            // if 2
            if (z) {
              // if 3
              true
            }
            // else 1
            // else 2
            // else 3
            else {
              error("wrong input")
            }
          }
        }
        else {
          false
        }
      }
    }
  }
}
```

The first block is the original code and the second is the formatted code. The first thing to note is that the structure changed. Indeed the formatter applies its own formatting rules so the very compact original code is now more aerated. The big space below the *if(a)* is removed. The *else* is now below the brace. The right part of the case is also below. Etc. These rules can be simply changed by editing the *PrettyPrinter* object that ensures the formatting of the code.

The second thing to note is the comments. The comments that are above code expression or at the end of a line are simply kept at the same place. But as you can see, if a comment is situated at a unusual position such as inside an expression like in the *val* definition in the example, this comment is pushed outside the expression at the end of the line. The same logic applies for all other expressions if a comment is written inside. I decided to do it like that to make the implementation of the *PrettyPrinter* a lot more simple, and because it is really rare to find a comment in such a position.

The last thing to note is that the management of comments around an *if* or an *else* differs a little bit from the other expression. The above comments i.e. *if 1* and *else 1* stay at the same place like as for other expressions. On the other hand, the comments *if 2* and *else 2* do not stay at the end of the line but are pushed above. Finally *else 3* is not at the correct position but *if 3* is. I will explain in the implementation part why are these comments formatted like this.

3. Implementation

3.1 Theoretical Background

As my extension do not use theoretical concept this section is empty.

3.2 General description

I will explain here the general operation of the code formatter. The trickier part of the implementation will be describe in the next section. To do this, I will follow the path taken by the formatter and describe the

changes and additions if there are any. Note that I will not describe the elements and terms introduced in the first six labs. I will only describe how I changed them and how I used them.

3.2.1 Main

The *Main* is slightly modified to take into account the optional argument *–format*. During the parsing of the command line input, if this keyword is detected, a new context is created where the new argument *format* is set to true. Then the compiler follow a new pipeline called *formatPipeline*. This pipeline consists of three steps : *Lexer*, *Parser* and *PrettyPrinter*. If the keyword is not found it follow the default pipeline i.e. it compiles the code.

3.2.2 Lexer

The *Lexer* includes a new method called *extractComment*. As indicated by its name, this method extract the command of the input files and store them in a list of *COMMENTLIT*. *COMMENTLIT* is a new token specifically created to store single and multiple line comments.

The output arguments of the pipeline changed. Now a pair *(Stream[Token], List[COMMENTLIT])* is passed to the parser. Before only a *Stream[Token]* was passed to the parser.

Finally, the *run* method is slightly modified. If the *format* argument of the context is set up, the *lexFile* method and the *extractComment* method are run and their output are passed to the parser. If not, only the *lexFile* method is run and an empty list is passed as second argument of the pair.

3.2.3 Parser

The only changed of the *Parser* is its input and output. Now it additionally receives a list of *COMMENTLIT* and passes it further to the *PrettyPrinter*. There is no other changed.

3.2.4 PrettyPrinter

This is a new element of the compiler which takes care of formatting and printing. This part is a modified version of the *Printer* that was used in the lab three to print the abstract syntax tree. It almost works the same way but also manages comments.

Let's describe how it works. First, it receives from the pipeline a pair *(N.Program, List[COMMENTLIT])*. Then the *run* method calls the master method of the *PrettyPrinter*, the *print* method. This method takes care of formatting and printing. It takes as argument the pair received from the pipeline, returns a *Document* and is composed of one variable : *comments*, which simply stores the comments and four sub methods : *binOp*, *insertEndOfLineComments*, *insertElzeComments* and *createDocument*. The *binOp* method is the same helper method as for the *Printer*. The *insertEndOfLineComments* method is an helper method to print the comments positioned at the end of a line. Moreover this method extract the unusual comments as explain in the *Examples* section. The *insertElzeComments* method is an helper method to print the comments positioned around an *else*.

Then the *print* method calls the *createDocument* method. This method takes as input an abstract syntax tree and returns a *Document*. The task of this method is to iterate over the ast and the comments to create a *Document* that will be printed into the console. First, it checks if the head comment of the variable *comments* is positioned above the root of the tree i.e. this comment goes above the top expression of the tree. If it is the case, it adds this comment in the document and checks if the next comment in the variable *comments* is also positioned above the top of the tree. If it is the case it adds it to the document and repeats the same task. If it is not the case it goes into a method named *rec*. This method is the same as the *rec* method of the *Printer* i.e. it creates a document given an ast but is also extended to insert comments with the helper methods.

Finally, when the iteration over the ast is finished, The *Document* is returned and printed into the console.

3.3 Implementation Details

The first thing to describe is the use of a variable to store the comments instead of passing the list of comments into argument of the methods. It was necessary to use a global variable accessible by all the method to store the comments because the *createDocument* or the *insertEndOfLineComments* method potentially remove the first comments of the comments list and these two methods are often called several time into the *rec* method. For instance :

```
case Program(modules) =>
  Stacked(modules map (createDocument(_)),
  emptyLines = true)
```

A call to *createDocument* has no way to know if a previous call has removed some comments to the list.

So to keep consistency I used a global variable so that all changes can be seen.

The second thing is the used of a sentinel to mark the end of the comments list. Instead of checking if the list is empty before getting the head of the list, I had at the end of the list an empty comment that belong to no file. Because this comment belong to no file it will never be added into the document. Note that a comment is added to the document if its position is correct and it belongs to the same file as the tree. With this trick if there is no more comment to add to the document, getting the head of the list will not throw a *noSuchElementException* and it avoids a lot of checking so the formatter is way faster.

The third thing to explain is why and how I extract the unusual comments and move it at the end of the line. At first I try to insert them at the correct place but that was way too complicated and absolutely not effective. The problem is that there is not enough information inside the abstract syntax tree to find the correct position of the comments. So I decided to simply push them at the end of the line. To do that I simply check if there is comments in the same line than the expression and print them at the end of the line.

The fourth thing to explain is why the comments around the *if* and the *else* are pushed above. The reason is again a lake of information. I have no way to know if the comment *else 1*, *else 2*, *else 3* are above, next or below the *else* { so I decided to print all the comments that are in these position above the *else* { and I did the same for the *if* { so that the comment are printed similarly for the *if* and the *else*.

To sum up, the integration of the comments was the harder part of the extension. I took a lot of time to try to find a solution but I finally realized that with my implementation I was limited. So I decided to make thing more simple and I made compromise on the comments to have something functional.

4. Possible Extensions

As the comments are not well integrated a good thing would be to change the code formatter to work at the tokens level. At this level there is a lot more of information and every kind of comment, even the most unlikely, could be correctly printed. Moreover at this level it could be possible to take into account the whitespace or the new lines that the user inserted.

Finally a cool feature would be to allow the user to give its own rules directly and that the code formatter takes it into account to format the code. For instance the rules could be stored inside a file and given as command line argument.