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 convert 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 convert them into an *Abstract Syntax Tree*. The parser consists of two stages. The first one takes the sequence of tokens and convert 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 keep the useful informations. Commas, parentheses, brackets, etc are dropped.

The third phase is the name analysis. The name analyzer check that the program follow the Amy naming rules, assign a unique *identifier* to every name and finally populate 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 check that a boolean is not used where an int should be. If a program pass 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 store 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 (z) { // an if
              true
            } else { // an else
              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 _ =>
            // an if
            if (z) {
              true
            }
            // an else
            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 apply is own formatting rules so the very compact original code is now more aerated. The big space 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 the expressions. 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 comment *an if* and *an else* are pushed above. The *if/else* comments management differs a little from the other expression. After formatting, these comments will always be pushed above the *if* or the *else* and they will never be at the end of the line next to them. This is due to an implementation difficulty explained later in the implementation part.

3. Implementation

3.1 Theoretical Background

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

3.2 Implementation Details

Describe all non-obvious tricks you used. Tell us what you thought was hard and why. If it took you time to figure out the solution to a problem, it probably means it wasn't easy and you should definitely describe the solution in details here. If you used what you think is a cool algorithm for some problem, tell us. Do not however spend time describing trivial things (we what a tree traversal is, for instance).

After reading this section, we should be convinced that you knew what you were doing when you wrote your extension, and that you put some extra consideration for the harder parts.

EXPLAIN MISTAKE : COMMENTLIT DON'T USE POSITIONED CLASS ADVANTAGE

4. Possible Extensions

If you did not finish what you had planned, explain here what's missing.

In any case, describe how you could further extend your compiler in the direction you chose. This section should convince us that you understand the challenges of writing a good compiler for high-level programming languages.