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. The idea is that the compiler runs with the argument *–prettyPrint* 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.

CHANGE NAME : PRETTYPRINT-¿ FORMAT

2. Examples

The code formatter is useful when there is a lot of nested expression and a lot on indentation. It allows the programmer to have a good structured code easily readable. Moreover it allows to have a uniform code if several people use different coding styles.

For example, let's say we have the following code :

```
/*
 * Hello object defines foo method
 */
object Hello {
  // foo method
  def foo(a: Boolean, x: Int, y: Int, z: Boolean): Boolean = {
    val add: /* addition */ Int = x+y;
```

```
    if (0<x) { true }
    else {
      if (a) {
        add match {
          case 0 => false
          case 1 => true
          case _ =>
            if (z) {
              true
            } else {
              error("wrong input")
            }
        }
      }
      else { false }
    }
  }
}
```

This code is really

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

EXPLAIN HOW THE COMMENT GO OUTSIDE AN EXPRESSION ! EXPLAIN WHY THERE IS A LINE RETURN AFTER CLASS, DEF, OBJECT, ETC ! EXPLAIN MISTAKE : COMMENTLIT DON'T USE POSITIONED CLASS ADVANTAGE

3. Implementation

EXPLAIN IF ELSE COMMENT

This is a very important section, you explain to us how you made it work.

3.1 Theoretical Background

If you are using theoretical concepts, explain them first in this subsection. Even if they come from the course (eg. lattices), try to explain the essential points *in your own words*. Cite any reference work you used like this [**?**]. This should convince us that you know the theory behind what you coded.

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.

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.