Code Formatter for Amy Language

Modern Compiler Construction Spring 2024

Gu Yuyang
HKUST
ygubb@connect.ust.hk

1. Introduction

In the previous labs, we have implemented a compiler for Amy language. The compiler is composed of four parts: Lexer will convert plain text into tokens based on lexical rules of Amy; Parser will transform the list of tokens into a well-structed AST, based on syntax of Amy language; Type checker will scan AST statically to avoid type errors and other common errors in the programme; Code generator will ultimately generates WebAssembly code for this piece of code to validate efficient execution;

In this project, I will add a new feature to the compiler: code formatter. This feature can reorganize the code piece and make it clear and easy-to-read. The code formatter will perform on the basis of lexers, so that it can preserve most details of the programme (e.g., comments) before those "useless" tokens are removed by lexers.

2. Examples

Think about a messy programme like this:

```
object {def main():Unit={println(new A().foo(−41));}
}

class A{
def foo(i:Int):Int = {
var j:Int;
if(i<0) {j=0−i;}
else {j=i;}
return j+1;
}
}
```

Without indentation, it is hard to read. With the help of our code formatter, we can organize the code piece into a more clear style:

```
object
{
  def main() : Unit =
  {
    println ( new A().foo(−41));
  }
}

class A
{
  def foo( i: Int ) : Int =
  {
    var j: Int ;
    if ( i < 0 )
    {
      j = 0 − i ;
    }
    else
    {
      j = i ;
    }
    return j + 1 ;
  }
}
```

By adding proper indentation at the beginning of lines and add spaces between words, the code looks more clear now.

3. How to Use

To activate code formatter, we should add a special option `--format` to compilation command. The explicit rule is as follows: `run -d [InputFiles]`

4. Implementation

4.1 Theoretical Background

Adding spaces between words is trivial. However, we should find out a way to indent properly. After checking the syntax of Amy language, we can find that whenever we create a block which needs modification in inden-

tation, we will always apply a pair of curly brackets. Therefore, by detecting the existence of curly brackets, we can record the indentation we need for every new line in the current block.

4.2 Implementation Details

We handle the indentation by declaring a special variable `idt` in the code formatter. This variable will increment once we enter a new block (i.e., encountering an open curly bracket) and decrement once we leave the current block (i.e., encountering a closed curly bracket). With the assist of this variable, we can properly indent every line.

Another challenge compared to normal compilation is how to handling comment properly. In the original compiler, we will delete all comment tokens once we finished lexical analysis. Therefore, if we want to preserve comments, we should do it before lexical analysis is done. This is also the reason why we choose to insert the formatting phase between lexical analysis and syntax analysis: AST can definitely make the formatter works more easily, but it ignored too much information in the original code piece, which is what we want to preserve.

5. Possible Extensions

This extension will normally be ignored in compiling process, but we can activate it by specifying it as an option of compilation. This way of activation offers great flexibility to our formatter. For example, some users may perfer one tab as four spaces, whereas others may prefer one tab as two spaces. Our implementation allows use to specify this in command line, and we can simply add a variable in the file CodeFormatter.scala to store such kind of preferences and adjust the format accordingly. In this way, we can achieve high level of flexibility for users.

References