Extending Punkt0 with Read-Eval-Print-Loop

Compiler Construction 2022 Final Report

Chengyang Huang
KTH Royal Institute of Technology
chehuang@kth.se

1. Introduction

Read-eval-print-loop (REPL) is an extension to a programming language that allows programmers to construct their code and get the execution result in an interactive and incremental way. Such extension is useful when trying out new ideas or learning the language and APIs.

In this project, I implemented a REPL over the programming language that was developed on previous assignments. To enable splitting compiling units and redefinition, changes are applied to most of the compiler stages, which I will give more detail about in the following sections.

The code is available at the repository of Group-7, branch Lab7.

2. Examples

In this section, I will give a compact example that consists of several interesting behaviors of the REPL. These examples reflect how this extension works internally.

```
punk0> 1;println(2);3
2
3

punk0> if
     |   (true
     | ) 1 else 2
1

punk0> var a:Int = 1;
Defined variable a

punk0> class A {
     |   def printA(): Unit = {
     |     println (a)
     |   }
     |   def updateA(x: Int): Unit = {
     |     a = x
     |   }
     | }
Defined class A

punk0> var aInstance:A = new A();
Defined variable aInstance

punk0> aInstance.printA()
1

punk0> a = 2
punk0> a
2
punk0> aInstance.printA()
2

punk0> var a:Int = 3;
Defined variable a

punk0> aInstance.printA()
2
punk0> aInstance.updateA(4)
punk0> aInstance.printA()
4

punk0> class A {
     |   def printA(): Unit = {
     |     println (a * 2)
     |   }
     | }
Defined class A

punk0> var aInstance:A = new A();
Defined variable aInstance

punk0> aInstance.printA()
6
```

In line 1, an expression sequence is evaluated, but only the value of the last expression is printed to the user.

In line 4, an expression is split into multiple lines, and our REPL can decide when the user input is complete and ready for evaluation, or wait for more input.

Starting from line 13, we present a more complicated example that shows that both classes and variables can be redefined. What is interesting is that, from line 35 to line 42, the *printA* function seems to be still pointing to the old variable *a*. Modifying the old *a* through *updateA* function will affect the result of *printA* (line 40), while modifying redefined variable *a* will not have any effect.

These examples lead us to several questions:

- When can we consider the user's input as complete?
- How does redefining a symbol affect other symbols that have reference to it?

My answer is as follows, for the first question, if the user input can be parsed into a complete expression, then we should move on to evaluate that. For the second question, redefining a symbol should not affect any previous reference to this symbol. For example, if the redefined symbol has a different type, then it would mean nothing for old symbols to point to it.

3. Implementation

3.1 REPL Runtime

This section is put first so the reader can have an overall understanding of how the REPL works. Similar to what has been implemented in previous assignments, a pipeline consisting of lexing, parsing, name analysis, type checking, and code generation is used. But there are a few differences:

- User input that forms a complete compiling unit (we will define later) will be classified as expression, class definition, or variable declaration. They will all be wrapped into a complete program to fit into the existing interfaces of name analysis, type checking, and code generation phase.
- If the input compiling unit is a class or variable declaration, we will maintain them globally. The name analysis and type checking phase will need to be modified to also search for symbols in this global store.
- If the input compiling unit is an expression sequence, we will also execute it using Java reflection after the code generation phase.
- The code generation and name analysis phase will be modified so field accesses points to the correct version (will discuss later).
- Each compiling unit that passes all the phases will be represented as a compiled Java class file stored in the local file system. The stored path needs to be under classpath so JVM could find it.
- For simplicity, we didn't support static method declaration.

3.2 Parsing on Stream Input

As discussed in the example section, one challenge of implementing the REPL comes from the streaming style input. Before this extension is implemented, the parser treats user input as a batch, which requires an explicit end-of-file token to terminate the parsing. However, when user input comes per line, the parser needs to decide when we have enough information to have the current input evaluated. Also, the parser needs to be implemented in a separate thread for not to block the main thread that interacts with the user.

The lexing stage is kept unchanged, which keeps emitting tokens as long as there is any input. We then send these tokens to the parser and query if these tokens end with a complete expression. However, as the parser runs asynchronously, it should not send back a response until it has visited all the tokens fed to it. We solved this by attaching an ID to each token and the query carries a destination id indicating after processing which token is the parser allowed to send back a response. The parser will enqueue the progress queue with its current progress once it tries to consume the next token.

To define when the user input is read to be evaluated, we here define **Compiling Unit**, which is essentially a sequence of tokens that is either a complete class definition, variable definition, or expression sequence. Only when the user input makes the parser ends up in a state where a compiling unit is parsed entirely, and it is not parsing a next compiling unit, do we consider the user input ready to be evaluated.

The core logic is shown as the following pseudo-code.

```
// REPL Thread
var tokenIndex = 0
```

```
val tokens = Lexer.run(lineInput )
while (tokenIt . hasNext) {
    REPLParser.feedToken(tokens.next(), tokenIndex)
    tokenIndex += 1
}

// read from Parser's progress queue
// until it reaches the latest token
while (progressQueue.take() != expectedProgress) {}
if (resultQueue. isEmpty) {
    // current input is not a complete compiling unit
} else {
    // consume the result from resultQueue
}

// Parser Thread
while (true) {
    try {
    tokens. peek(). _1. kind match {
            case CLASS => // try prase class
            case OBJECT => // try prase object
            case VAR => // try parse var decl
            case _ => eatExprSeq().foreach(
              e => resultQueue.enqueue
                (Some(REPLExprTree(e))))
        }
    } catch {
        // if met unexpected token
        case e: REPLUnexpectedTokenException =>
            resultQueue. enqueue(None)
    }

}
```

If the input line does not get the parser to a a complete expression, the parser will simply block at some calling stack of recursive descent parsing, when it tries to read more tokens from a blocking queue. If an unexpected token is met, an exception will be thrown, which helps us to jump out of the calling stack and re-initiate the parsing.

Also, note that the parsing rule of REPL is different from the original language since we consider a single class definition, variable declaration, or an expression sequence valid.

3.3 Symbol Redefinition

We implemented symbol redefinition by assigning a version number to each symbol at the name analysis phase and the code generation phase. At the name analysis and the type checking phase, only the new compiling unit needs to be checked, which always points to the newest version of other versioned symbols.

The reference relationship for the example section should look like this.

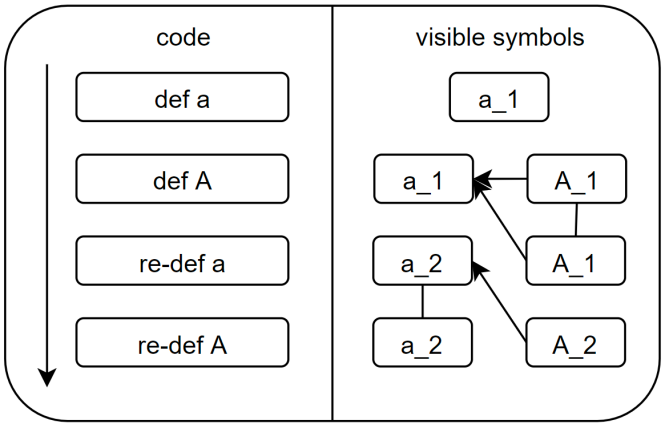

Figure 1. Reference relationship for the example, an undirected line means they point to the same object

4. Possible Extensions

These are the problems of the current implementation:

- A Compiling Unit does not allow a mixture of class definition, variable definition, and expression sequence.
- We do not support global functions. Because static function does not exist in the language, this makes it requires more effort to be implemented.

These are some functionalities that would be useful to implement in future work:

- Support importing libraries.
- Save the REPL session on disk for future usage.