

5.6 Multiple Syntax Styles

Compiler Construction '22 Final Report

Donggyu Lee Jisu Seo
EPFL
donggyu.lee@epfl.ch jisu.seo@epfl.ch

1. Introduction

In 22 fall semester's Computer Language Processing (CLP) class, students were made to construct a compiler from the very base with Amy language, which is a very similar programming language to Scala. The project's goal is to translate Amy written program into WebAssembly and to execute the program.

There were several steps for this processing. First, students have to implement lexer, a phase for tokenizing sequence of characters into sequence of tokens for future steps. Second, parser phase changes the tokens into abstract syntax tree (AST) which is a tree that concise version of the code and denotes the syntactic order. This is followed by name analyser checking that the code follows the Amy naming rules and generating symbol table for each identifiers within their own scopes. Third, type checker phase examines that input Amy code follows Amy syntax rules. After this, interpreter actually executes the program with previous resulted ASTs as an input. Finally, code generator translates ASTs into WebAssembly code.

5.6 Multiple Syntax Styles extension enables the different coding styles in Amy language. Currently Amy language mixes braces style (`if`, `match`) and end-marker style (`object`) for indicating the scope of statement. By introducing this extension, a user can use both styles when programming with Amy language. This could help the newcomers of Amy adapt more easily with their previous coding habits, and can enhance the readability of the code by using adequate style in circumstances.

To implement this extension, parser phase have to be changed. In the parser phase, logics that generating AST based on end markers and braces both have to be implemented without violating the LL(1) condition of the parser.

2. Examples

Currently, the typical structure of Amy program is like **Code 1**. The program starts with the program name "object Syntax" and finishes with end-marker "end Syntax". In the `if` statement, there is brace style right after the condition parenthesis. This is same in `match` statement which have braces that covers the case part.

Now, below implementation **Code 2** is available with our extension. The styles of statements are now have changed. Now there is brace next to the name of the program and ends with the correspond brace. `if` and `match` statements now have end markers (`end if`, `end match`) instead of braces.

Code 1. Current Amy syntax

```
object Syntax
  fn foo(): Int(32) = {...}
  if(true) {
    0
  } else {
    foo() match{
      case _ => 1
    }
  }
end Syntax
```

Code 2. New possible Amy syntax

```
object Syntax {
  fn foo(): Int(32) = {...}
  if(true)
    0
  else {
    foo() match
      case _ => 1
    end match
  end if
}
```

3. Implementation

3.1 Theoretical Background

3.1.1 LL(1) Parser

LL(1) parser is a recursive descent parser that parses the sequence of tokens in LL(1) grammar which need only one token to overhead for the parsing. The conditions for a grammar to be LL(1) are like below.

First, for each non-terminal X , first sets of different alternatives of X are disjoint. It is because if more than one sets of alternatives exists, parser does not know how to parse the current token between them.

Second, if X is nullable, $\text{first}(X)$ must be disjoint from $\text{follow}(X)$ and only one alternative of X may be nullable. It is because to distinguish the X and the following token, and determine when to parse X as epsilon (null).

3.1.2 Conflicts

With the conditions explained above, there are possibly two kinds of conflicts; First/First conflict, First/Follow conflict.

```
First/First Conflict
S -> E | E 'a'
E -> 'b' | epsilon
```

```
First/Follow Conflict
S -> A 'a' 'b'
A -> 'a' | epsilon
```

For the First/First conflict, $\text{First}(E) = (b, \text{epsilon})$ and $\text{First}(E a) = (b, a)$. Thus, parser does not know how to parse S .

In addition, for First/Follow conflict, the $\text{First}(A)$ is $(a, \text{epsilon})$ and the $\text{Follow}(A)$ is (a) . Then, parser does not know the A is epsilon or not, when parsing A .

3.1.3 Left Factoring

To fix the conflicts shown at 3.1.2, Left Factoring is used in many cases. Left Factoring splits the composite of more than two tokens by making new non-terminals like below.

```
A -> X | X Y Z
```

changes to

```
A -> X B
B -> Y Z | epsilon
```

3.2 Implementation Details

The goal of implementation is to make Amy satisfy both braces style and end-marker style in `if`, `match` and `object` statements. For this goal, implementation efforts fully focused on "Parser" phase which accepts both styles and generates the same ASTs.

```
Original Amy Syntax
if ( Expr ) { Expr } else { Expr }
Expr match { MatchCase+ }
object Id Definition* Expr? end Id
```

```
Extended Amy Syntax
if ( Expr ) { Expr } else { Expr }
if ( Expr ) Expr else Expr end if
Expr match { MatchCase+ }
Expr match MatchCase+ end match
object Id Definition* Expr? end Id
object Id { Definition* Expr? }
```

3.2.1 If statement

The easiest idea of implementing new syntax is simply adding it like **Code 3**. However, this causes LL(1) error. It is because the keyword "if" is both `First` in `if_old` and `if_new`. To be specific, it causes First/First conflict.

The solution of this problem is to give priority in these syntaxes, using Left Factoring. New implementation gave priority to the common expression "if (Expr)" and then add `if_comb` which includes `if_old` and `if_new` as **Code 4**.

For the implementation details, new implementation now uses `List[Expr]` to store parameters for `if` statement. After that, "Lexer" phase had modified to add keyword token "end if". However, to consider the case of having more than one whitespaces between end-marker end and `if`, now "Parser" phase has the role of mapping keywords end and `if`, not the "Lexer" phase.

Code 3. Wrong implementation for if statement

```
val if_old: Syntax[Expr] = (kw("if") ~ ... ).map{...}
val if_new: Syntax[Expr] = (kw("if") ~ ... ).map{...}
```

Code 4. Correct implementation for if statment

```
val if_pri: Syntax[Expr] = (kw("if") ~ delimiter("(") ~
  expr ~ delimiter(")") ~ if_comb).map{...}
val if_comb: Syntax[List[Expr]] = if_old | if_new
val if_old: Syntax[List[Expr]] =
  (delimiter("{") ~ ... ).map{...}
val if_new: Syntax[List[Expr]] = (expr ~ ... ).map{...}
```

3.2.2 Match statement

The solution of Match statement is quite similar to that of If statement. To prevent LL(1) conflict, parser phase have been modified to give priority to the overlapped keyword match, and then added `mtch_comb` which includes `mtch_old` and `mtch_new`. After that, new implementation stores the parameter and matchcases in new type `Seq[MatchCase]` temporally. Then, change `Seq` to the required type `List` at `mtch_pri`.

Code 5. Correct implementation for match statement

```
val mtch_pri: Syntax[List[MatchCase]] =  
  (kw("match") ~ mtch_comb).map{...}  
val mtch_comb: Syntax[Seq[MatchCase]] =  
  mtch_old | mtch_new  
val mtch_old: Syntax[Seq[MatchCase]] =  
  (delimiter("{") ~ ...).map{...}  
val mtch_new: Syntax[Seq[MatchCase]] =  
  (many(matchcase) ~ ...).map{...}
```

3.2.3 Object statement

The new implementation for Object statement needs more consideration than previous two cases. First, to prevent LL(1) error new implementation gave priority to common expression "object *Id*" and then add `module_comb` which includes `module_old` and `module_new`. Second, storing different types of parameters was the focus. It took some times to find out the way to store parameters. This is because previous implementation used data type `List` on `if` and `match`, but `List` cannot store different type of values within it. Therefore, new implementation uses `Tuple3` type in Scala which can contain different types of variables. Third, comparing begin and end module name is needed for original implementation, but it is not for the modified version. Thus, new implementation added new string `"_"` at `Tuple3` to differentiate.

Code 6. Correct implementation for object statement

```
val module_pri: Syntax[ModuleDef] =  
  (kw("object") ~ identifier ~ module_comb).map{  
    case _ ~ id ~ mod =>  
      if (mod(2)=="_") || (mod(2)==id) then  
        ModuleDef(...)  
      else throw ...  
  }  
val module_comb: Syntax[  
  Tuple3[Seq[ClassOrFunDef],Option[Expr],String]] =  
  module_old | module_new
```

```
val module_old: Syntax[  
  Tuple3[Seq[ClassOrFunDef],Option[Expr],String]] =  
  (many(definition) ~ ...).map{  
    case defs ~ body ~ _ ~ id1 =>  
      Tuple3(defs,body,id1)  
  }  
val module_new: Syntax[  
  Tuple3[Seq[ClassOrFunDef],Option[Expr],String]] =  
  (delimiter("{") ~ ...).map{  
    case _ ~ defs ~ body ~ _ =>  
      Tuple3(defs,body,"_")  
  }
```

4. Possible Extensions

4.1 Omit Braces In One Line

Many major programming languages including **C**, **Java**, **JavaScript** let the braces optional if the **if** statement's body has only one line.

```
if (cond)  
  one_line_body()
```

```
if (cond) {  
  one_line_body()  
}
```

This feature reduces unnecessary braces for only one body statement and helps to enhance the code readability.

To implement this feature, lexer has to accept new line character as a new token. Parser has to parse a statement right after the **if** statement's condition as a body of the **if**.

4.2 Indentation Based Parsing

One step further from the end-markers and braces, compiler can use the indentation based parsing which is used in Python language or Scala 3 provides optionally. Although there are some disadvantages, indentation based parsing is easy to write and read the codes in certain circumstances.

To implement this feature, lexer now insert **indent** token in specific conditions such as after the specific keywords like **if**, **for**, **while** and **outdent** token to close the scope with similar conditions. [Scala3Docs 2022b]

4.3 Format Forcing

Allowing to use various syntax to the users is good in usual, they can embrace and learn the language more quickly and can use appropriate syntax in their circumstances. For example, with the braces style, it is very

hard to recognize where the else statement is corresponded if the code is vertically long.

object Syntax

```

if(foo){
  match x => {
    if(bar){
      match x.1 {
        if(zoo) {
          ...
        }
      }
    } else {
      ...
    }
  }
}

```

In contrast, in this below example which uses end marker, it is better to recognize where the else statement because of its explicit **end if** keyword.

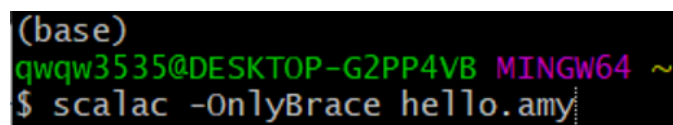
object Syntax

```

if(foo){
  match x =>
  if(bar)
    match x.1
    if(zoo)
      ...
    end if
  end match
else
  ...
end if
end match
end if

```

However, this variety of syntaxes can bring confusion when many programmers are included in a project. If some modules use brace style while other modules use end-marker style, it could be hard or awkward to read different formatted code in the same project. To prevent this, compiler could offer format forcing option when compiles just like **scalac**[Scala3Docs 2022a] compiler provides. For this feature, adding conditional variable in parser phase to activate/deactivate the braces style and end-marker style will be needed.

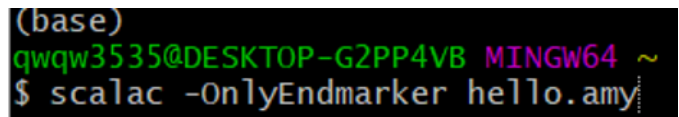


```

(base)
qwqw3535@DESKTOP-G2PP4VB MINGW64 ~
$ scalac -OnlyBrace hello.amy

```

Figure 1. Example of only brace option in compiler



```

(base)
qwqw3535@DESKTOP-G2PP4VB MINGW64 ~
$ scalac -OnlyEndmarker hello.amy

```

Figure 2. Example of only end marker option in compiler

References

- Scala3Docs. New control syntax, 2022a. URL <https://dotty.epfl.ch/docs/reference/other-new-features/control-syntax.html>. Last accessed 4 Jan 2023.
- Scala3Docs. Optional braces, 2022b. URL <https://dotty.epfl.ch/docs/reference/other-new-features/indentation.html>. Last accessed 3 Jan 2023.