Parser combinators
Computer Language Processing ’18

Clara Di Marco
EPFL
clara.dimarco@epfl.ch

I. INTRODUCTION

In this project, we began by implementing a lexer and a parser for Amy. For the lexer, we traversed the input file character by character to create tokens. Then we defined a LL1 grammar and created a tree from it that we then processed to finally create an AST tree. My goal for this final project part is to simplify this procedure by using the scala library ”StandardTokenParsers”. All those steps will be replaced by one single Pipeline that imediatly tokenize and parse the input files into the final AST tree, which will prevent the effort of writing a lot of code and making a LL1 grammar.

II. EXAMPLES

To use the library, your class needs to extend `StandardTokenParsers`. Then you can define `Parser` objects as follows:

```
object Parser extends StandardTokenParsers {
    def program: Parser[Boolean] = ”(” ~> expr <~ ”)” ^^
        {x => !x}
    def expr: Parser[Boolean] = ”true” ^^^ true
}
```

In this small example, `program` and `expr` parse into a Boolean. `expr` is the string ”true” and with the operator `^^^` it immediately parses the expression to the Boolean `true`. Regarding `program` the input consists of several parts, separated by the operator ~. Here, I used a useful variant: ~>, which ignores the left hand side (and <~ ignores the right hand side). It uses the operator ^^ which take a function of the input (in that case, thanks to ~> and <~, the parenthesis are ignored and the input consists only of `expr`) to the Parser type. All the inputs used in a `Parser` need to be either a Parser, a delimiter or a reserved keyword. You can add strings to delimiter and reserved list as follows:

```
lexical.reserved += (”true”)
lexical.delimiters += (”(”, ”)”)
```

You can parse your file with the function `phrase`, taking as arguments your top-level `Parser` and a `Scanner` object which represents your tokens:

```
phrase(program)(new lexical.Scanner(inputString))
```

III. IMPLEMENTATION

As for this project, I erased the content of object `Parser` and nested a `SmartParser` inside it, extending `StandardTokenParsers`. To define the `Parser` objects, I based it on the non-LL1 grammar with a discrete but major change: repetitions and options have a build-in function to be represented. Instead of defining a `Program` that consists of `moduleDefs` that itself consists of a `moduleDef` followed by `moduleDefs`, Program will only consist in `rep(moduleDef)`. There also is function `repsep(parser, sep)` that represents multiple instances of `parser` seperated by `sep`, `rep1(parser)` that ensure there is at least one instance of `parser` and `opt(parser)` that represents an optional part. `IDSENT` becomes `ident`, `STRINGLITSENT` becomes `stringLit`, and `INTLITSENT` becomes `numericLit`. All the tokens are directly written as strings (after being added in reserved or delimiters).
Regarding the parse results, it is simply based on the `ASTConstructor` class. The operators, literals and types are directly parsed with `^^^`. Thanks to that, the functions used in `ASTConstructor` can be used really easily in the other parsers.
The last step was to rewrite a `run` function to replace the ones from `Lexer` and `Parser`. I first read the files into one string (separated by EOF character), then removed the comments, generated the tokens from that string and parsed it to return a `Program`.

IV. IMPLEMENTATION DETAILS

To remove the comments, I found a working argument for `replaceAll` on the internet[1].

V. POSSIBLE EXTENSIONS

Unfortunately, I haven’t had the time to set the positions as `ASTConstructor` does. This is an essential feature for debugging and would be needed for the compiler to be really usable.

VI. CONCLUSION

As I thought, this last lab was mostly understanding the library. I looked into a lot of examples, carefully read the documentation and did a few smaller scale tests. Once I was

[1]https://blog.ostermiller.org/find-comment

clear about the way it worked, the implementation was quite straight forward.
Overall the lab was really satisfying (especially erasing what was previously written) and it allowed me to make up with parsers.