Menhir interpreter on OCaml

author: https://github.com/lastdesire
tutor: https://github.com/Kakadu

SPbU SE, 3rd semester, 2022

Chapter 1

Description of task

- Menhir + recursive descent:
- Such a kind of DSL for writing parsers. By default, it parses in the LR way, but in this project we don't care about it;
- without action code, associativity, priorities;
- translation into parser-combinators;
- with fixed left recursion;
- In general, you are doing a parser, interpreter + AST transformations;

- That is, the task is to "issue" a parser to the user, where he describes the tokens and rules of terminal and non-terminal tokens, the user passes a set of tokens to this parser, after which he receives a syntactic parse tree.

1.1 Before we started

I want to say a huge thanks to my tutor Dmitry Sergeevich and the people from the Discord channel "OCaml" for answering my questions and helping me understand the idea of this task.

Chapter 2

Grammar and AST

Our grammar which one we take in lexer from .mly file has this type: type grammar = string * (string * string list) list

Let's look at this with an example:

```
%token PLUS
%token MUL
%token INT
%token EOL
%token LBRACE
%token RBRACE
%start main
%%
main:
  | expr; EOL
  | EOL
expr:
  | LBRACE; expr; RBRACE
  | MUL; expr; expr
  | PLUS; expr; expr
  | INT
```

For this input file we have next grammar:

```
("main",
   [("main", ["expr"; "EOL"]);
   ("main", ["EOL"]);
   ("expr", ["LBRACE"; "expr"; "RBRACE"]);
   ("expr", ["MUL"; "expr"; "expr"]);
   ("expr", ["PLUS"; "expr"; "expr"]);
   ("expr", ["INT"])])
```

Our AST looks like this:

```
type parseTree =
  | Term of string
  | Nonterm of string * parseTree list
```

Chapter 3

Lexer and Parser

The lexer and parser were written using Menhir and Sedlex. Yes, it is a parser for Menhir using Menhir. A little strange, but quite interesting. There is nothing more to add here.

Chapter 4

Interpreter

With the grammar in hand, we read a set of tokens, trying to get parse_tree. The algorithm was invented by me here: first we check if we can apply at least some starting rule: the try_apply_start_nonterm function tries to apply any of the starting rules using try_apply_rule: if the first element of the rule is terminal, then we compare its with the first input token: if they match, then we go further, checking the tail of the rule and the tail of the input, if not, it returns (false, 0) (here it is worth noting that the second element of the tuple, if the first element is false, serves to inform, whether REJECT or OVERSHOOT happened: REJECT is 0, OVERSHOOT is -1, however, if the first element is true, then the second element serves to report how many incoming tokens the rule "ate"). If the first element of the rule is not terminal, then we need to check whether it is nonterminal: if it is not, then (false, 0), in another case we are trying to apply the rules of a new nonterminal symbol to the tail of the input. If it worked out, then we can continue working, if not, then the original rule cannot be applied.

The apply_rule function is called only if try_apply_rule returned true. We use both of these functions in parse to get a ParseTree. Using the parse_tree function, we can get a string describing the ParseTree that we print in REPL.

Chapter 5

TODO

^_^

*** More tests ***

*** Check Arg module in interpret part ***