Project: Language Parser

Overview

In this project, you will implement a lexer and parser for a small C-like language. Your lexer functions will convert the source text of a program (internally stored as string) into a list of tokens, and your parser functions will consume these tokens to produce an abstract syntax tree (AST). The input could be an entire program, or a fragment of a program, and your parser will produce an AST that represents one or more statements (stmt) or an expression (expr). This document describes the tokens and grammar you will use.

The only requirement for error handling is that input that cannot be lexed or parsed according to the provided rules should raise an `InvalidInputException`. Informative error messages should be used when raising these exceptions to make debugging easier.

All tests will be run on direct calls to your code, comparing your return values to the expected return values. Any other output (for example, debugging information) will be ignored. You are free and encouraged to have additional output.

Here is an example program fragment:

```
int x;
x = 2 * 3 ^ 5 + 4;
print(x > 100);
```

An example use of the parser in the OCaml top-level:

```
parse_stmt (tokenize "int x; x = 2 * 3 ^ 5 + 4; print(x > 100);")
```

Testing

You can run your lexer or parser directly on a program by running

```
  dune exec bin/main.exe lex <filename>
```

or

```
  dune exec bin/main.exe parse <filename>
```

where the `<filename>` argument is required.

All of the tests will feed C-like code into your implementation to be lexed and parsed. You can write your own tests which only test the parser by feeding it a custom token list. For example, to see how the expression `1 + 2 ^ 3 * 4` would be parsed, you can construct the token list manually:

```
parse_expr [Tok_Int 1; Tok_Add; Tok_Int 2; Tok_Pow;
            Tok_Int 3; Tok_Mult; Tok_Int 4; EOF];;
```

This way, you can work on the parser even if your lexer is not complete yet.

Part 1: The Lexer (aka Scanner or Tokenizer)

The lexer transforms source text into tokens. The goal is to transform a program, represented as a string, into a list of tokens that capture the different elements of the program. This process can be handled by using regular expressions. Information about OCaml's regular expressions library can be found in the `Str` module documentation. You are not required to use it, but you may find it useful.

Your lexer must be written in `lexer.ml`. You will need to implement the following function:

`tokenize`

- **Type:** `string -> token list`
- **Description:** Converts source code (given as a string) to the associated token list.
- **Examples:**

  ```
  tokenize "1 + 2" = [Tok_Int 1; Tok_Add; Tok_Int 2; EOF]
  tokenize "" = [EOF]
  tokenize "int main() { int x; }" =
    [Tok_Int_Type; Tok_Main; Tok_LParen; Tok_RParen; Tok_LBrace;
     Tok_Int_Type; Tok_ID "x"; Tok_Semi; Tok_RBrace; EOF]
  ```

The `token` type is implemented in `tokenTypes.ml`.

A few important notes to consider:

- Tokens can be separated by arbitrary amounts of whitespace, which your lexer should discard. Spaces, tabs (`\t`) and newlines (`\n`) are all considered whitespace.
- Tokens are case sensitive.
- Lexer output must be terminated by the `EOF` token, meaning that the shortest possible output from the lexer is `[EOF]`.
- If the beginning of a string could be multiple things, the **longest** match should be preferred, for example:
  - "while0" should not be lexed as `Tok_While`, but as `Tok_ID("while0")`, since it is an identifier
  - "1-1" should be lexed as `Tok_Int(1)` and `Tok_Int(-1)`, since "-1" is a valid integer.

In grammar given later in this project description, we use the lexical representation of tokens instead of the token name; for example, we write `(` instead of `Tok_LParen`. The following table shows all mappings of tokens to their lexical representations. `Tok_Bool`, `Tok_Int`, and `Tok_ID` are listed as regular expressions.

Token Name	Lexical Representation (in grammars below)
`Tok_LParen`	`(`
`Tok_RParen`	`)`
`Tok_LBrace`	`{`
`Tok_RBrace`	`}`
`Tok_Equal`	`==`
`Tok_NotEqual`	`!=`
`Tok_Assign`	`=`
`Tok_Greater`	`>`
`Tok_Less`	`<`
`Tok_GreaterEqual`	`>=`
`Tok_LessEqual`	`<=`
`Tok_Or`	`||`
`Tok_And`	`&&`
`Tok_Not`	`!`
`Tok_Semi`	`;`
`Tok_Int_Type`	`int`
`Tok_Bool_Type`	`bool`
`Tok_Print`	`print`
`Tok_Main`	`main`
`Tok_If`	`if`
`Tok_Else`	`else`
`Tok_For`	`for`
`Tok_From`	`from`
`Tok_To`	`to`
`Tok_While`	`while`
`Tok_Add`	`+`
`Tok_Sub`	`-`
`Tok_Mult`	`*`
`Tok_Div`	`/`
`Tok_Pow`	`^`
`Tok_Bool`	`/true|false/`
`Tok_Int`	`/-?[1-9][0-9]*/`
`Tok_ID`	`/[a-zA-Z][a-zA-Z0-9]*/`

Part 2: The Parser

The parser consists of three main functions which parse expressions, statements, and an entire program (`parse_expr`, `parse_stmt` and `parse_main`.) The three functions work together to parse a full program.

The three functions you need to implement are:

`parse_expr`

- **Type:** `token list -> token list * expr`

- **Description:** Takes a list of tokens and returns an AST representing the expression corresponding to the given tokens, along with the new, reduced list of tokens
- **Exceptions:** If the next tokens in the token list do not represent an expression, raise `InvalidInputException`.
- **Examples:**

```
parse_expr [Tok_Int(1); Tok_Add; Tok_Int(2); Tok_Semi; EOF] =
    ([Tok_Semi; EOF], Add (Int 1, Int 2))
parse_expr [Tok_Int(1); EOF] = ([EOF], Int 1)
parse_expr [Tok_Int_Type; EOF] (* InvalidInputException *)
```

`parse_stmt`

- **Type:** `token list -> token list * stmt`
- **Description:** Takes a list of tokens and returns an AST representing the statement corresponding to the given tokens, along with the new, reduced list of tokens
- **Exceptions:** If the next tokens in the token list do not represent a statement, raise `InvalidInputException`.
- **Examples:**

```
parse_stmt [Tok_Int_Type; Tok_ID("x"); Tok_Semi; EOF] =
    ([EOF], Seq (Declare (Int_Type, "x"), NoOp))
parse_stmt [Tok_ID("x"); Tok_Assign; Tok_Int(3); Tok_Semi; EOF] =
    ([EOF], Seq (Assign ("x", Int 3), NoOp))
parse_stmt [Tok_Int(3); Tok_Add; Tok_Int(4); EOF] = InvalidInputException
```

`parse_main`

- **Type:** `token list -> stmt`
- **Description:** Takes a list of tokens and returns an AST representing the list of statements in the body of the `main` function. All tokens should be consumed, so there is no need to return the remaining tokens.
- **Exceptions:** If the list of tokens does not contain only `EOF` at the end, raise `InvalidInputException`. Also, if the token list contains tokens before the beginning of the `main` function declaration, or after its last closing brace, or if the `main` function doesn't exist in the token list, raise `InvalidInputException`.
- **Examples:**

```
parse_main [Tok_Int_Type; Tok_Main; Tok_LParen; Tok_RParen; Tok_LBrace;
            Tok_Int_Type; Tok_ID("x"); Tok_Semi; Tok_RBrace; EOF] =
    Seq (Declare (Int_Type, "x"), NoOp)
parse_main [Tok_Int(3); Tok_Add; Tok_Int(4); EOF] =
    InvalidInputException
```

The parser must be implemented in `parser.ml` in accordance with the signatures found in `parser.mli`. `parser.ml` is the only file you will write code in. The functions should be left in the order they are provided, as a good implementation will rely heavily on earlier functions.

Note: the function `match_token` is provided for you which takes the list of tokens and a single token as arguments, and will either:

- Return a new token list with the first token removed IF the first token matches the second argument, or
- Raise an exception IF the first token does not match the second argument to the function.

The `lookahead` function is also provided for you which returns the first token in the list of tokens, but does NOT modify the token list. This function will raise an exception if the token list is empty.

The CFG below is for the language of C-like expressions. This CFG is right-recursive, so something like `1 + 2 + 3` will parse as `Add (Int 1, Add (Int 2, Int 3))`, essentially implying parentheses in the form `(1 + (2 + 3))`.)

`parse_expr`

Expressions represent mathematical and boolean formulas that typically evaluate to either an integer or a boolean. Because expressions are a self-contained subset of the grammar, we can implement them first, and build the rest of the language on top of them later.

```
type expr =
  | ID of string
  | Int of int
  | Bool of bool
  | Add of expr * expr
  | Sub of expr * expr
  | Mult of expr * expr
  | Div of expr * expr
  | Pow of  expr * expr
  | Greater of expr * expr
  | Less of expr * expr
  | GreaterEqual of expr * expr
  | LessEqual of expr * expr
  | Equal of expr * expr
  | NotEqual of expr * expr
  | Or of expr * expr
  | And of expr * expr
  | Not of expr
```

The unambiguous CFG of expressions, from which you should produce a value of `expr` AST type, is as follows:

$$
\begin{array}{rcl}
\mathit{Expr} & \rightarrow & \mathit{OrExpr} \\
\mathit{OrExpr} & \rightarrow & \mathit{AndExpr}\ \texttt{||}\ \mathit{OrExpr} \\
 & | & \mathit{AndExpr} \\
\mathit{AndExpr} & \rightarrow & \mathit{EqualityExpr}\ \texttt{\&\&}\ \mathit{AndExpr} \\
 & | & \mathit{EqualityExpr} \\
\mathit{EqualityExpr} & \rightarrow & \mathit{RelationalExpr}\ \mathit{EqualityOp}\ \mathit{EqualityExpr} \\
 & | & \mathit{RelationalExpr} \\
\mathit{RelationalExpr} & \rightarrow & \mathit{AdditiveExpr}\ \mathit{RelationalOp}\ \mathit{RelationalExpr} \\
 & | & \mathit{AdditiveExpr} \\
\mathit{AdditiveExpr} & \rightarrow & \mathit{MultiplicativeExpr}\ \mathit{AdditiveOp}\ \mathit{AdditiveExpr} \\
 & | & \mathit{MultiplicativeExpr} \\
\mathit{MultiplicativeExpr} & \rightarrow & \mathit{PowerExpr}\ \mathit{MultiplicativeOp}\ \mathit{MultiplicativeExpr} \\
 & | & \mathit{PowerExpr} \\
\mathit{PowerExpr} & \rightarrow & \mathit{UnaryExpr}\ \texttt{\^{}}\ \mathit{PowerExpr} \\
 & | & \mathit{UnaryExpr} \\
\mathit{UnaryExpr} & \rightarrow & \texttt{!}\ \mathit{UnaryExpr} \\
 & | & \mathit{PrimaryExpr} \\
\mathit{EqualityOp} & \rightarrow & \texttt{==} \mid \texttt{!=} \\
\mathit{RelationalOp} & \rightarrow & \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \\
\mathit{AdditiveOp} & \rightarrow & \texttt{+} \mid \texttt{-} \\
\mathit{MultiplicativeOp} & \rightarrow & \texttt{*} \mid \texttt{/} \\
\mathit{PrimaryExpr} & \rightarrow & \mathit{Tok\_Int} \mid \mathit{Tok\_Bool} \mid \mathit{Tok\_ID} \mid \texttt{(}\ \mathit{Expr}\ \texttt{)}
\end{array}
$$

As an example, see how the parser will break down an input mixing a few different operators with different precedence:

Input:

```
2 * 3 ^ 5 + 4
```

Output (after lexing and parsing):

```
Add(
  Mult(
    Int(2),
    Pow(
      Int(3),
      Int(5))),
  Int(4))
```

`parse_stmt`

Statements typically represent side-effecting actions, for example, control flow. When parsing, a sequence of statements should be terminated as a `NoOp`. The `stmt` type:

```
type stmt =
  | NoOp
  | Seq of stmt * stmt
  | Declare of data_type * string
  | Assign of string * expr
  | If of expr * stmt * stmt
  | For of string * expr * expr * stmt
  | While of expr * stmt
  | Print of expr
```

The `stmt` type isn't self-contained like the `expr` type, and instead refers both to itself and to `expr`; use your `parse_expr` function to avoid duplicate code. The following CFG is for statements:

$$
\begin{array}{rcl}
\mathit{Stmt} & \rightarrow & \mathit{StmtOptions}\ \mathit{Stmt} \\
 & | & \epsilon \\
\mathit{StmtOptions} & \rightarrow & \mathit{DeclareStmt} \\
 & | & \mathit{AssignStmt} \\
 & | & \mathit{PrintStmt} \\
 & | & \mathit{IfStmt} \\
 & | & \mathit{ForStmt} \\
 & | & \mathit{WhileStmt} \\
\mathit{DeclareStmt} & \rightarrow & \mathit{BasicType}\ \mathit{Tok\_ID}\ \texttt{;} \\
\mathit{BasicType} & \rightarrow & \mathit{Tok\_Int} \mid \mathit{Tok\_Bool} \\
\mathit{AssignStmt} & \rightarrow & \mathit{Tok\_ID}\ \texttt{=}\ \mathit{Expr}\ \texttt{;} \\
\mathit{PrintStmt} & \rightarrow & \texttt{print}\ \texttt{(}\ \mathit{Expr}\ \texttt{)} \\
\mathit{IfStmt} & \rightarrow & \texttt{if}\ \texttt{(}\ \mathit{Expr}\ \texttt{)}\ \texttt{\{}\ \mathit{Stmt}\ \texttt{\}}\ \mathit{ElseBranch} \\
\mathit{ElseBranch} & \rightarrow & \texttt{else}\ \texttt{\{}\ \mathit{Stmt}\ \texttt{\}} \\
 & | & \epsilon \\
\mathit{ForStmt} & \rightarrow & \texttt{for}\ \texttt{(}\ \mathit{Tok\_ID}\ \texttt{from}\ \mathit{Expr}\ \texttt{to}\ \mathit{Expr}\texttt{)}\ \texttt{\{}\ \mathit{Stmt}\ \texttt{\}} \\
\mathit{WhileStmt} & \rightarrow & \texttt{while}\ \texttt{(}\ \mathit{Expr}\texttt{)}\ \texttt{\{}\ \mathit{Stmt}\ \texttt{\}}
\end{array}
$$

If we expand on our previous example, we can see how the expression parser integrates directly into the statement parser:

Input:

```
int x;
x = 2 * 3 ^ 5 + 4;
print(x > 100);
```

Output (after lexing and parsing):

```
Seq(Declare(Int_Type, "x"),
Seq(Assign("x",
  Add(
    Mult(
      Int 2,
      Pow(
        Int 3,
        Int 5)),
    Int 4)),
Seq(Print(Greater(ID "x", Int 100)), NoOp)))
```

Input:

```
int main(){
  int a;
  for (a from 1 to 10){
    print(a);
  }
}
```

Output:

```
(Seq
  (Declare (Int_Type, "a"),
   Seq (For ("a", Int 1, Int 10, Seq (Print (ID "a"), NoOp)),
         NoOp)))
```

`parse_main`

The `parse_main` handles the function entry point. This function behaves the exact same way as `parse_stmt`, except for two key semantic details:

- `parse_main` will parse the function declaration for main, not just the body.
- `parse_main` validates that a successful parse terminates in `EOF`. A parse not ending in `EOF` should raise an `InvalidInputException` in `parse_main`. As such, `parse_main` does NOT return remaining tokens, since it validates ensures that the token list is emptied by the parse.

The CFG is:

$$Main \quad \rightarrow \quad \texttt{int main ( ) \{ } Stmt \texttt{ \} } EOF$$

For this slightly modified input to the example used in the previous two sections, the exact same output would be produced:

Input:

```
int main() {
  int x;
  x = 2 * 3 ^ 5 + 4;
  print(x > 100);
}
```

The output is the exact same as in the statement parser, but `parse_main` also trims off the function header and verifies that all tokens are consumed.