Project 4A: SmallC Parser

Due: November 7th (Late November 8th) at 11:59:59 PM

Points: 48P/52R/0S

Introduction

In this project, you will implement the lexer and parser for SmallC. Your lexer will convert an input string into a flat `token list`, and your parser will consume these tokens to produce a `stmt` and/or `expr` corresponding to the input. The only requirement for error handling is that input that cannot be lexed/parsed according to the provided rules should raise an `InvalidInputException`. We recommend using relevant error messages 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 (e.g., for your own debugging) will be ignored. You are free and encouraged to have additional output.

Ground Rules

In your code, you may use any OCaml modules and features we have taught in this class (If you come asking for help using something we have not taught we will direct you to use methods taught in this class). You may use imperative OCaml (following examples given in lecture), but are not required to.

Testing

You can run your lexer or parser directly on a SmallC program by running `dune exec bin/interface.bc lex [filename]` or `dune exec bin/interface.bc parse [filename]` where the `[filename]` argument is optional.

You can run the tests as usual with `dune runtest -f`. To test from the toplevel with `dune utop src`, import functions with `open Parser` and `open Lexer` at the prompt you get after starting `utop`.

Part 1: The Lexer (aka Scanner)

Before your parser can process input, the raw file must be transformed into logical units called tokens. This process is readily handled by use of regular expressions. Information about OCaml's regular expressions library can be found in the `Str` module documentation. You aren't required to use it, but you may find it very helpful. Note that a lexer is the same as a scanner, which is discussed in the lecture slides.

Your lexer must be written in `lexer.ml`. You will need to implement the function `tokenize : string -> token list` which takes as input the program as a string and outputs the associated token list. The `token` type is implemented in `tokenTypes.ml`.

Your lexer must meet these general requirements:

- Tokens can be separated by arbitrary amounts of whitespace, which your lexer should discard. Spaces, tabs ('\t') and newlines ('\n') are all considered whitespace.
- The lexer should be case sensitive.

- Lexer input should 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
- The version of the for loop used in this project is different from the normal version you see in the C language.

Most tokens only exist in one form (for example, the only way for `Tok_Pow` to appear in the program is as `^` and the only way for `Tok_While` to appear in the program is as `while`). However, a few tokens have more complex rules. The regular expressions for these more complex rules are provided here:

- `Tok_Bool of bool`: The value will be set to `true` on the input string "true" and `false` on the input string "false".
  - *Regular Expression*: `true|false`
- `Tok_Int of int`: Valid ints may be positive or negative and consist of 1 or more digits. You may find the function `int_of_string` useful in lexing this token type.
  - *Regular Expression*: `-?[0-9]+`
- `Tok_ID of string`: Valid IDs must start with a letter and can be followed by any number of letters or numbers. Note that keywords may be contained within IDs and they should be counted as IDs unless they perfectly match a keyword!
  - *Regular Expression*: `[a-zA-Z][a-zA-Z0-9]*`
  - *Valid examples*:
    - "a"
    - "ABC"
    - "a1b2c3DEF6"
    - "while1"
    - "ifelsewhile"

In grammars given later in this project description, we use the lexical representation of tokens instead of the token name; e.g. we write `(` instead of `Tok_LParen`. This table shows all mappings of tokens to their lexical representations, save for the three variant tokens specified above:

Token Name (in OCaml)	Lexical Representation (in grammars below)
`Tok_LParen`	`(`
`Tok_RParen`	`)`
`Tok_LBrace`	`{`
`Tok_RBrace`	`}`
`Tok_Equal`	`==`
`Tok_NotEqual`	`!=`
`Tok_Assign`	`=`
`Tok_Greater`	`>`
`Tok_Less`	`<`

Token Name (in OCaml)	Lexical Representation (in grammars below)
`Tok_GreaterEqual`	`>=`
`Tok_LessEqual`	`<=`
`Tok_Or`	`||`
`Tok_And`	`&&`
`Tok_Not`	`!`
`Tok_Semi`	`;`
`Tok_Int_Type`	`int`
`Tok_Bool_Type`	`bool`
`Tok_Print`	`printf`
`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`	`^`

Your lexing code will feed the tokens into your parser, so a broken lexer will render the parser useless. **Test your lexer thoroughly before moving on to the parser!**

Part 2: The Parser

Once the program has been transformed from a string of raw characters into more manageable tokens, you're ready to parse. The parser must be implemented in `parser.ml` in accordance with the signature for `parse_main` 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.

We provide an **ambiguous** CFG below for the language that must be converted so that it's right-recursive and right-associative. That way it can be parsed by a recursive descent parser. (By right associative, we are referring to binary infix operators—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))`.) As convention, in the given CFG all non-terminals are capitalized, all syntax literals (terminals) are formatted `as non-italicized code` and will come in to the parser as tokens from your lexer. Variant token types (i.e. `Tok_Bool`, `Tok_Int`, and `Tok_ID`) will be printed *`as italicized code`*.

`parse_expr`

Expressions are a self-contained subset of the SmallC grammar. As such, implementing them first will allow us to 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 (ambiguous) CFG of expressions, from which you should produce a value of `expr` AST type, is as follows:

- Expr -> OrExpr
- OrExpr -> OrExpr `||` OrExpr | AndExpr
- AndExpr -> AndExpr `&&` AndExpr | EqualityExpr
- EqualityExpr -> EqualityExpr EqualityOperator EqualityExpr | RelationalExpr
  - EqualityOperator -> `==` | `!=`
- RelationalExpr -> RelationalExpr RelationalOperator RelationalExpr | AdditiveExpr
  - RelationalOperator -> `<` | `>` | `<=` | `>=`
- AdditiveExpr -> AdditiveExpr AdditiveOperator AdditiveExpr | MultiplicativeExpr
  - AdditiveOperator -> `+` | `-`
- MultiplicativeExpr -> MultiplicativeExpr MultiplicativeOperator MultiplicativeExpr | PowerExpr
  - MultiplicativeOperator -> `*` | `/`
- PowerExpr -> PowerExpr `^` PowerExpr | UnaryExpr
- UnaryExpr -> `!` UnaryExpr | PrimaryExpr
- PrimaryExpr -> *`Tok_Int`* | *`Tok_Bool`* | *`Tok_ID`* | `(` Expr `)`

The transformation of the above ambiguous grammar into a parsable, non-ambiguous, grammar can be found in ambiguity.md. We encourage you to do the transformation yourself and use ambiguity.md to check your work and ensure correctness before coding.

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`

The next step to parsing is to build statements up around your expression parser. When parsing, a sequence of statements should be terminated as a `NoOp`, which you will remember as a do-nothing instruction from the interpreter. Recall 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! Again, we provide a grammar that is ambiguous and must be adjusted to be parsable by your recursive descent parser:

- Stmt -> Stmt Stmt | DeclareStmt | AssignStmt | PrintStmt | IfStmt | ForStmt | WhileStmt
  - DeclareStmt -> BasicType ID `;`
    - BasicType -> `int` | `bool`
  - AssignStmt -> ID `=` Expr `;`
  - PrintStmt -> `printf` `(` Expr `)` `;`
  - IfStmt -> `if` `(` Expr `)` `{` Stmt `}` ElseBranch
    - ElseBranch -> `else` `{` Stmt `}` | ε
  - ForStmt -> `for` `(` ID `from` Expr `to` Expr `)` `{` Stmt `}`

- WhileStmt -> `while` `(` Expr `)` `{` Stmt `}`

As with the expression grammar, the transformation to enable the grammar to be parsable can be found in ambiguity.md. 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;
printf(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){
    printf(a);
  }
}
```

Output:

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

`parse_main`

The last and shortest step is to have your parser handle the function entry point. This is where `parse_main : token list -> stmt` comes in. 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 grammar for this parse is provided here:

- Main ::= `int` `main` `(` `)` `{` Statement `}` `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;
  printf(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.

Academic Integrity

Please **carefully read** the academic honesty section of the course syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, or unauthorized use of computer accounts, **will be** submitted to the Student Honor Council, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand what you are and what you are not permitted to do in regards to academic integrity when it comes to project assignments. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. Full information is found in the course syllabus, which you should review before starting.