CS 341, Spring 2025
Homework 06
Due: Monday 4/14/2025 at 11:59pm

Overview

For this homework assignment, you will implement a parser for VerySimpleC, a much simpler version of SimpleC, which is an already simplified version of C/C++ that was presented in lecture. Your program should use the technique known as **recursive-descent parsing**, also presented in lecture, which is designed to help you practice recursion in F#. The goal is to determine two results:

1. Is the input program a valid VerySimpleC program?
2. If the program is not valid, find the first syntax error and output an error message of the form “expecting X, but found Y”.

VerySimpleC

The VerySimpleC language is a very simple subset of C/C++. The only possible statements are empty lines, which consist of a single semicolon (;), and output statements with **`cout`**, which can only print one thing at a time.

Below you will find the BNF (“Backus–Naur Form”) definition of the VerySimpleC syntax. Notice that the first rule ends with $, which is the EOF token. In other words, a valid VerySimpleC program ends with ‘}’ followed immediately by EOF.

```
<verySimpleC>   -> void main ( ) { <stmts> } $

<stmts>     -> <stmt> <morestmts>
<morestmts> -> <stmt> <morestmts>
             | EMPTY

<stmt> -> <empty>
        | <output>

<empty>        -> ;
<output>       -> cout << <output-value> ;

<output-value> -> <expr-value>
                | endl

<expr-value> -> int_literal
              | str_literal
              | true
              | false
```

The idea of **recursive descent parsing** is to write a function for each rule in the BNF, where each rule parses that aspect of the language. With VerySimpleC, there will be

functions for **<stmt>, <empty>, <output>,** etc. When writing a given function, the approach is to match each token and call the recursive-descent function for each rule.

In F#, every recursive-descent function will be passed a list of tokens representing the VerySimpleC program. As you parse the program, you will consume tokens, remove them from the list, and return a resulting list of tokens for the next function to process.

Here is the provided **matchToken** function in F#. The purpose of the "private" keyword is to hide this function from other components of the compiler (this is an internal function for use by the parser only):

```
let private matchToken expected_token (tokens: string list) =
  //
  // if the token matches the expected token, keep parsing by
  // returning the rest of the tokens.  Otherwise throw an
  // exception because there's a syntax error, effectively
  // stopping compilation at the first error.
  //
  let next_token = List.head tokens

  if expected_token = next_token then
    List.tail tokens
  else
    failwith ("expecting " + expected_token + ", but found " + next_token)
```

If the match is successful, the function consumes the token and returns the remaining tokens. Otherwise, the function fails by throwing an exception that effectively stops the compilation process with a syntax error.

How is the list of tokens built? In a compiler, this is the job of the lexical analyzer ("lexer"). A lexer is provided, and called for you by the main function which is also provided:

```
let tokens = compiler.lexer.analyze filename
```

The lexer returns the list of tokens, which is passed to the parser.

Getting Started

To help you get started, you are being provided with the main function, the lexical analyzer, and an initial skeleton of the parser. The main function calls the lexer and parser. Your job is to modify the **verySimpleC** function in the parser module ("parser.fs") and call the other recursive-descent functions that you will need to write.

CS 341, Spring 2025
Homework 06
Due: Monday 4/14/2025 at 11:59pm

Let us look at another example, to get you started. Consider the first rule from the BNF for VerySimpleC:

```
<verySimpleC>   -> void main ( ) { <stmts> } $
```

In this case, we need to write two recursive-descent functions: **verySimpleC** and **stmts**. For now, let us define the function for **stmts** to do nothing but return the tokens back:

```
let rec private stmts tokens =
  tokens
```

Now let us write the function for **verySimpleC.** You write this directly from the BNF rule, matching tokens and calling functions. Recall that matchToken returns the remaining tokens after matching the next token, so we need to capture the return value and feed that into the next step.

```
let private verySimpleC tokens =
    let T2 = matchToken "void" tokens
    let T3 = matchToken "main" T2
    let T4 = matchToken "(" T3
    let T5 = matchToken ")" T4
    let T6 = matchToken "{" T5
    let T7 = stmts T6
    let T8 = matchToken "}" T7
    let T9 = matchToken "$" T8   // $ => EOF, there should be no more tokens
    T9
```

Next, we need a test case that matches what we are parsing: a VerySimpleC program with no statements. Create a file called “main.c” and enter this code:

```
void main()

{

}
```

Now compile and run your program, enter “main.c” when prompted, and the compiler should successfully parse the file.

As another test, run again using one of the provided input files, such as “main1.c”. This will yield a syntax error because your parser is not yet complete.

CS 341, Spring 2025
Homework 06
Due: Monday 4/14/2025 at 11:59pm

The **matchToken** function in “parser.fs” will need to be modified so that it makes a special check when matching a string literal token or an integer literal. The token created by the lexer contains both the token type and the token’s value separated by a colon as noted above. Thus the **matchToken** function will need to check that the actual token starts with the following when matching for a string literal token or an integer literal:

- “str_literal”
- “int_literal”

The **StartsWith** method of the F# string class requires that it knows that the calling object is actually a member of the string class before it lets you call it. The following F# function is a wrapper function that can be used with the StartsWith method to inform F# the parameters are actually of the string class:

```
let beginswith (pattern: string) (literal: string) =

    literal.StartsWith (pattern)
```

When parsing **<expr-value>**, what if the token is not a literal? Output an error message of the form

```
failwith ("expecting literal, but found " + next_token)
```

Running the Program

Once you have the files downloaded, create an F# project using the **dotnet new** command (if your IDE does not automatically do so). Make sure that lexer.fs, parser.fs, and main.fs are all being used in compilation (you may need to modify the .fsproj file). Then you can run the program! Make sure that any VerySimpleC program files that you are using for testing are in the same directory as your .fs files.

Feel free to ask any of the instructional staff via drop-in hours and/or Piazza if you are having trouble with this.

CS 341, Spring 2025
Homework 06
Due: Monday 4/14/2025 at 11:59pm

Requirements

Do not use imperative style programming: no mutable variables, no loops, and no data structures other than F# lists. Use recursion and higher-order approaches, but **you must adhere to the recursive-descent approach, which implies that recursion is the dominant strategy**. Do not change “main.fs” and do not change “lexer.fs”. You must work with those components as given.

Submission

Login to Gradescope.com and look for the assignment “Homework 06”. Submit your “parser.fs” file to that assignment. You can upload your entire program, but the autograder will only run and test your parser component. You cannot change the main program, or the lexical analyzer (any changes will be ignored by the autograder).

Grading will be based on the correctness of your compiler. We are not concerned with efficiency at this point, only correctness. Note that your compiler will be tested against a suite of VerySimpleC input files, some that compile and some that have syntax errors.

You have unlimited submissions. Keep in mind that any manual grading for requirements, etc. will be based on your last submission unless you select an earlier submission for grading. If you do choose to activate an earlier submission, you must do so before the deadline.

The score reported on Gradescope is only part of your final score. After the project is due, the TAs will manually review the programs for style and adherence to requirements (0-100%). **Failure to meet a requirement, e.g. use of mutable variables or loops or not adhering to the recursive-descent parsing approach, will trigger a large deduction.**

No late submissions will be accepted for this assignment.

CS 341, Spring 2025
Homework 06
Due: Monday 4/14/2025 at 11:59pm

Academic Integrity

All work is to be done individually — group work is not allowed.

While we encourage you to talk to your peers and learn from them, this interaction must be superficial with regards to all work submitted for grading. This means you cannot work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own, etc. The University's policy is available here: https://dos.uic.edu/community-standards/

In particular, note that **you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance**. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums (e.g. you cannot download answers from Chegg). Other examples of academic dishonesty include emailing your program to another student, sharing your screen so that another student may copy your work, copying-pasting code from the internet, working together in a group, and allowing a tutor, TA, or another individual to write an answer for you.

Academic dishonesty is unacceptable, and penalties range from a letter grade drop to expulsion from the university; cases are handled via the official student conduct process described at the link above.