



**CS 341, Spring 2024**  
**Project 4 – SimpleC Parser**  
**Due: Wednesday 4/10 at 11:59pm**

## Overview

In this project, you will be writing an F# program to parse SimpleC programs using a technique known as **recursive-descent parsing**. The goal is to determine two results:

1. Is the input program a valid simple C 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”.

This project is an exercise in recursion. Higher-order approaches probably will not fit for this assignment. This is because the definition of a programming language like SimpleC is highly recursive, and thus lends itself naturally to a recursive implementation.

## SimpleC

The SimpleC language is a very simple subset of C. There are no pointers, no arrays, and no loops. Only 3 types exist: integer, string, and boolean. There is input and output with `cin` and `cout`, but you can only input or output one value within a single statement. Any expressions use one value or are binary operations which combine two values with some operator. Simple assignment statements are used. If statements can have an else but only allow for one statement to be executed if the condition evaluates to be true.

An example SimpleC program is shown below.

```
1//
2// simple C test program #1
3//
4void main()
5{
6    int x;    // define x:
7    cin >> x; // input a value:
8
9    int y;
10   int z;
11   y = 3^2;  // 3 squared:
12   z = x-y;
13
14   cout << "x: ";
15   cout << x;
16   cout << endl;
17
18   cout << "y: ";
19   cout << y;
20   cout << endl;
21
22   if (z < 0)
23       cout << "z is negative";
24   else
25       cout << z;
26
27   cout << endl;
28}
29
```



CS 341, Spring 2024  
Project 4 – SimpleC Parser  
Due: Wednesday 4/10 at 11:59pm

## BNF Definition of SimpleC

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

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

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

<stmt> -> <empty>
          | <vardecl>
          | <input>
          | <output>
          | <assignment>
          | <ifstmt>

<empty>      -> ;
<vardecl>    -> int identifier;
<input>      -> cin >> identifier;
<output>     -> cout << <output-value> ;
<output-value> -> <expr-value>
                  | endl
<assignment> -> identifier = <expr> ;
<ifstmt>      -> if ( <condition> ) <then-part> <else-part>
<condition>  -> <expr>
<then-part>  -> <stmt>
<else-part>  -> else <stmt>
                  | EMPTY

<expr>       -> <expr-value> <expr-op> <expr-value>
               | <expr-value>

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

<expr-op>    -> +
               | -
               | *
               | /
               | ^
               | <
               | <=
               | >
               | >=
               | ==
               | !=
```



## Recursive Descent Parsing

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 SimpleC, there will be functions for `<stmt>`, `<empty>`, `<vardecl>`, `<input>`, `<output>`, etc. When writing a given function, the approach is to match each token and call the recursive-descent function for each rule.

For example, consider the `<else-part>` for SimpleC:

```
<else-part>  -> else <stmt>
              | EMPTY
```

Conceptually, the `<else-part>` needs to look ahead to see if the “else” keyword is present. If so, the function matches the “else” and processes the statement. Otherwise, the `<else-part>` is missing so the function does nothing because the `<else-part>` is optional. Here is the corresponding recursive-descent function in a **C-like way**:

```
void else_part(tokens) {
    if (nextToken(tokens) == "else") then
    {
        match("else", tokens); // match and consume the "else" token
        stmt(tokens);          // parse the stmt that are supposed to follow
    }
    else
        ; // EMPTY, do nothing, just return because "else" is optional
}
```

In F#, every recursive-descent function will be passed a list of tokens representing the simple C 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 `<else-part>` recursive-descent function expressed in F#:

```
let private else_part tokens =
    let next_token = List.head tokens

    if next_token = "else" then
        let T2 = matchToken "else" tokens // match and discard "else"
        stmt T2 // parse the stmt with remaining tokens
    else
        tokens // EMPTY is legal, so do nothing and return tokens unchanged
```

First, notice that `else_part` is passed a list of tokens, and returns a list of tokens. All the recursive-descent functions operate in this manner. Second, notice if the else-part is



**CS 341, Spring 2024**  
**Project 4 – SimpleC Parser**  
**Due: Wednesday 4/10 at 11:59pm**

missing, we return the same tokens since nothing was parsed / matched (like most programming languages, else is optional in SimpleC).


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:

```
[<EntryPoint>]  
let main argv =  
    //  
    printf "simpleC filename> "  
    let filename = System.Console.ReadLine()  
    printfn ""  
    //  
    if not (System.IO.File.Exists(filename)) then  
        printfn "***Error: file '%s' does not exist." filename  
        0  
    else  
        printfn "compiling %s..." filename  
        //  
        // Run the lexer to get the tokens, and then  
        // pass these tokens to the parser to see if  
        // the input program is legal:  
        //  
        let tokens = compiler.lexer.analyze filename  
        //  
        printfn ""  
        printfn "%A" tokens  
        printfn ""  
        //  
        let result = compiler.parser.parse tokens  
        printfn "%s" result  
        printfn ""  
        //  
        0
```





CS 341, Spring 2024  
Project 4 – SimpleC Parser  
Due: Wednesday 4/10 at 11:59pm

For example, here is the list of tokens produced for the SimpleC program “main1.c” shown earlier:

```
simpleC filename> main1.c
compiling main1.c...
["void"; "main"; "("; ")"; "{"; "int"; "identifier:x"; ";"; "cin"; ">>";
"identifier:x"; ";"; "int"; "identifier:y"; ";"; "int"; "identifier:z"; ";";
"identifier:y"; "="; "int_literal:3"; "^"; "int_literal:2"; ";"; "identifier:z";
"="; "identifier:x"; "-"; "identifier:y"; ";"; "cout"; "<<"; "str_literal:x";
";"; "cout"; "<<"; "identifier:x"; ";"; "cout"; "<<"; "endl"; ";"; "cout"; "<<";
"str_literal:y"; ";"; "cout"; "<<"; "identifier:y"; ";"; "cout"; "<<"; "endl";
";"; "if"; "("; "identifier:z"; "<"; "int_literal:0"; ")"; "cout"; "<<";
"str_literal:z is negative"; ";"; "else"; "cout"; "<<"; "identifier:z"; ";";
"cout"; "<<"; "endl"; ";"; "}"; "$"]
syntax_error: expecting $, but found void
```

Most of the tokens are self-explanatory, e.g. “void” is the keyword void, and “int” is the type int. The only non-obvious tokens are those that contain values. For example, in SimpleC we might have the variable declaration

```
int x;
```

In this case we have 3 tokens: “int”, “**identifier:x**”, and “;”. Notice the “identifier” token also contains the value, i.e. the name, of the identifier. A similar strategy is used for string and integer literals:

```
cout << “the answer is”;
```

```
cout << 42;
```

Here we have 8 tokens: “cout”, “<<”, “**str\_literal:the answer is**”, “;”, “cout”, “<<”, “**int\_literal:42**”, and “;”.



CS 341, Spring 2024  
Project 4 – SimpleC Parser  
Due: Wednesday 4/10 at 11:59pm

## 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 **simpleC** function in the parser module ("parser.fs") and call the other recursive-descent functions that you will need to write.

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

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

In this case, we need to write two recursive-descent functions: **simpleC** 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 **simpleC**. 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 simpleC 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 SimpleC program with no statements. Create a file called "main.c" and enter this code:

```
void main()  
{  
}
```



CS 341, Spring 2024  
Project 4 – SimpleC Parser  
Due: Wednesday 4/10 at 11:59pm

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

```
simpleC filename> main.c
compiling main.c...
["void"; "main"; "("; ")"; "{"; "}"; "$"]
success
```

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:

```
simpleC filename> main1.c
compiling main1.c...
["void"; "main"; "("; ")"; "{"; "int"; "identifier:x"; ";"; "cin"; ">>";
"identifier:x"; ";"; "int"; "identifier:y"; ";"; "int"; "identifier:z"; ";";
"identifier:y"; "="; "int_literal:3"; "^"; "int_literal:2"; ";"; "identifier:z";
"="; "identifier:x"; "-"; "identifier:y"; ";"; "cout"; "<<"; "str_literal:x"; ";
"; "cout"; "<<"; "identifier:x"; ";"; "cout"; "<<"; "endl"; ";"; "cout"; "<<";
"str_literal:y"; ";"; "cout"; "<<"; "identifier:y"; ";"; "cout"; "<<"; "endl";
";"; "if"; "("; "identifier:z"; "<"; "int_literal:0"; ")"; "cout"; "<<";
"str_literal:z is negative"; ";"; "else"; "cout"; "<<"; "identifier:z"; ";";
"cout"; "<<"; "endl"; ";"; "}"; "$"]
syntax_error: expecting }, but found int
```

## A Few Details

The **matchToken** function in “parser.fs” will need to make a special check when matching an identifier token, 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 an identifier token, a string literal token or an integer literal:

- “identifier”
- “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#



**CS 341, Spring 2024**  
**Project 4 – SimpleC Parser**  
**Due: Wednesday 4/10 at 11:59pm**

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)
```

Other types of syntax errors are possible. For example, when parsing `<expr-value>`, what if the token is not an identifier or literal? Output an error message of the form

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

Next, when parsing `<stmt>`, what if the next token does not denote the start of a statement? Output

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

Finally, when parsing `<expr-op>`, if the next token is not a valid operator, output

```
failwith ("expecting expression operator, but found " +  
    next_token)
```

As you implement the BNF rules as F# functions, you are going to run into the problem that some of the functions are “mutually-recursive”. In particular, `<stmt>` refers to `<ifstmt>`, which in turn refers to `<then-part>` and `<else-part>`, which in turn refer back to `<stmt>`. This presents a problem in F#, which doesn’t support the notion of function prototypes (you cannot declare a function ahead of time, and then implement it later). Instead, mutually-recursive functions must be defined using the `and` keyword as follows:

```
let rec private stmt tokens =  
    ...  
  
    and private then_part tokens =  
        ...  
  
    and private else_part tokens =  
        ...  
  
    and private ifstmt tokens =  
        ...
```

Be careful with the indentation, e.g. the start of each function must be at the same indentation.





**CS 341, Spring 2024**  
**Project 4 – SimpleC Parser**  
**Due: Wednesday 4/10 at 11:59pm**

## 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 SimpleC 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.

## 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. Modify only the **simpleC** function in “`parser.fs`” though you will need to add additional recursive-descent functions (at least one per BNF rule).

It is also expected that you use good programming practices, i.e. functions, comments, consistent spacing, etc. In a 3xx class this should be obvious and not require further explanation. But to be clear, if you submit a program with no functions and no comments, you will be significantly penalized --- in fact you can expect a score of 0, even if the program produces the correct results. How many functions? How many comments? You can decide. Make reasonable decisions and you will not be penalized.

## Submission

Login to Gradescope.com and look for the assignment “Project 04”. 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 SimpleC input files, some that compile and some that have syntax errors.

**You have unlimited submissions. Keep in mind we grade 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.**



**CS 341, Spring 2024**  
**Project 4 – SimpleC Parser**  
**Due: Wednesday 4/10 at 11:59pm**

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.**

Suggestion: when you fail a test on Gradescope, we show your output, the correct output, and the difference between the two (as computed by Linux's diff utility). Please study the output carefully to see where your output differs. If there are lots of differences, or you can't see the difference, here's a good tip:

1. Browse to <https://www.diffchecker.com/>
2. Copy your output as given by Gradescope and paste into the left window
3. Copy the correct output as given by Gradescope and paste into the right window
4. Click the "Find Difference" button

You'll get a visual representation of the differences. Modify your program to produce the required output, and resubmit.

In terms of grading, note that we expect all submissions to compile, run, and pass at least some of the test cases; do not expect partial credit with regards to correctness unless your program compiles and runs.

**Late submissions for this project are allowed.** You may turn in a project up to 3 days (72 hours) late, and will receive the following penalty on your total score:

Submission made **up to 24 hours late:** **10 point** deduction

Submission made **24-48 hours late:** **20 point** deduction

Submission made **48-72 hours late:** **30 point** deduction

No submissions will be accepted after 72 hours.



CS 341, Spring 2024  
Project 4 – SimpleC Parser  
Due: Wednesday 4/10 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.