UIC COMPUTER SCIENCE

Project 03 : Parsing simple C programs

(doc v1.2)

Assignment: **F# program to parse simple C programs**
Evaluation: **Gradescope followed by manual execution & review**
Policy: **Individual work only**
Complete By: **Friday March 25th @ 11:59pm CDT for full credit**

Early submission: earn 10% bonus towards pass/future projects if 100% done by Fri 3/18 @ 11:59pm
Late submissions: 10% penalty if submitted up to 24 hrs late (by Saturday 3/26 @ 11:59pm)
20% penalty if submitted up to 48 hrs late (by Sunday 3/27 @ 11:59pm)

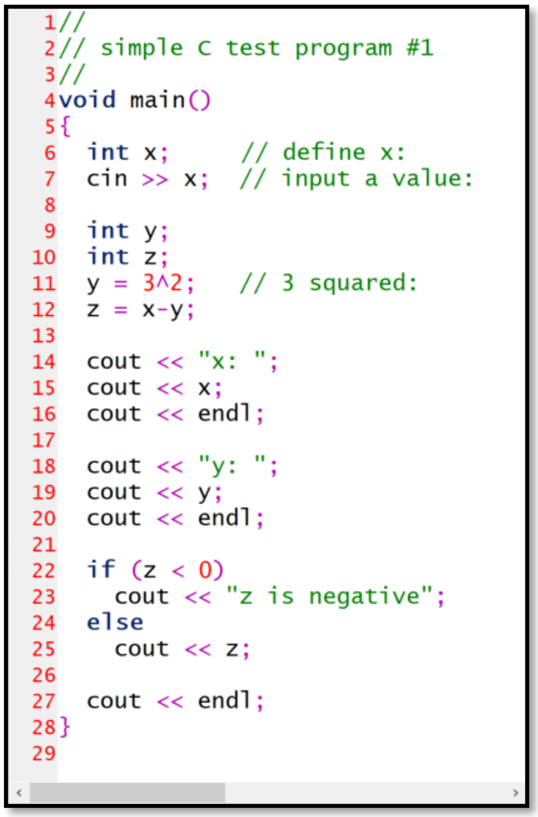
Pre-requisites: **Lecture from Wednesday, March 9th (PPT, PDF, recording on BB)**

Background

This is part 1 of a multi-part programming project in F#. Here in part 1, we're going to parse **simple C** programs using a technique known as **recursive-descent**. 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".

Note that "simple C" is a very simple subset of C --- no pointers, no arrays, only 3 types (integer, string, boolean), no loops, very simple expressions, etc. Here's an example simple C program ---------------->



```
//
// simple C test program #1
//
void main()
{
  int x;      // define x:
  cin >> x;   // input a value:

  int y;
  int z;
  y = 3^2;    // 3 squared:
  z = x-y;

  cout << "x: ";
  cout << x;
  cout << endl;

  cout << "y: ";
  cout << y;
  cout << endl;

  if (z < 0)
    cout << "z is negative";
  else
    cout << z;

  cout << endl;
}

```

The project is an exercise in recursion; a higher-order approach is probably not going to fit. Why? Because as you'll see in the next section, the definition of a programming language like simple C is highly recursive, lending itself more naturally to a recursive implementation.

BNF definition of simple C

What follows is the BNF definition of the **simple C** syntax. Notice that the first rule ends with $ --- the EOF token. In other words, a valid simple C 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

In class on Wednesday (March 9th) we introduced the concept of recursive descent parsing. The idea is to write a function for each rule in the BNF, where each rule parses that aspect of the language. In the case of simple C, 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, let's consider the <else-part> for simple C:

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

Conceptually, the <else-part> needs to lookahead 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 <else-part> is optional. Here's 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's 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'll consume tokens, remove them from the list, and return a resulting list of tokens for the next function to process. Here's 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 missing, we return the same tokens since nothing was parsed / matched --- like most programming languages, else is optional in simple C.

Here's 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):

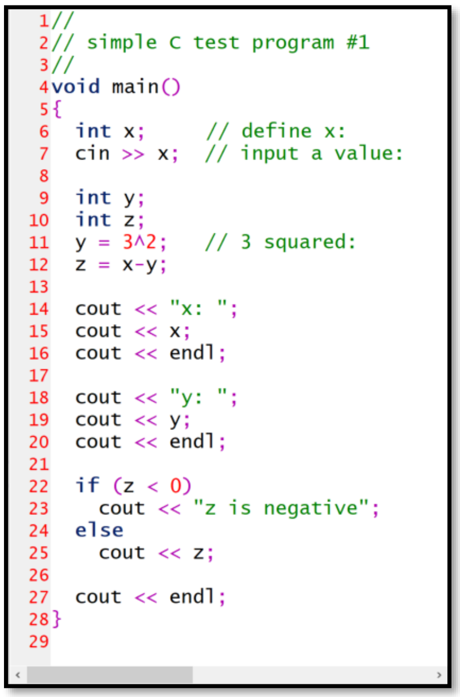
```
let private matchToken expected_token tokens =
  //
  // if the token matches the expected token, keep parsing by
  // returning the rest of the tokens.  Otherwise throw an
  // exception because there's a syntax error, effectively
  // stopping compilation at the first error.
  //
  let next_token = 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 remaining tokens, otherwise the function fails by throwing an exception that effectively stops the compilation process with a syntax error.

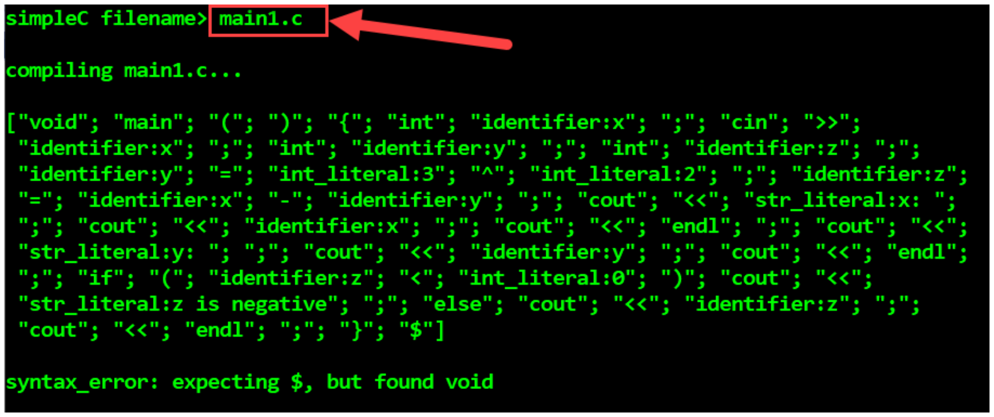
Who builds the list of tokens? In a compiler, this is the job of the **lexical analyzer** ("lexer"). A lexer is provided, and called for you by the main function which is also provided:

```
[<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
```

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



```
//
// simple C test program #1
//
void main()
{
  int x;      // define x:
  cin >> x;   // input a value:

  int y;
  int z;
  y = 3^2;    // 3 squared:
  z = x-y;

  cout << "x: ";
  cout << x;
  cout << endl;

  cout << "y: ";
  cout << y;
  cout << endl;

  if (z < 0)
    cout << "z is negative";
  else
    cout << z;

  cout << endl;
}

```



```
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 simple C 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 “;”.

Getting Started

To help you get started, we are providing the main function, the lexical analyzer, and an initial skeleton of the parser. The code is available on replit.com “**Project03 – simple in F#**”, or you can download from dropbox. The main function was shown in the previous section, and already calls the lexer and parser. Your job is to modify the **simpleC** function in the parser module (“parser.fs”), calling other recursive descent functions you will need to write. Here’s the provided code from “parser.fs”:

```
//
// Parser for simple C programs.  This component checks
// the input program to see if it meets the syntax rules
// of simple C.  The parser returns a string denoting
// success or failure.
//
// Returns: the string "success" if the input program is
// legal, otherwise the string "syntax_error: ..." is
// returned denoting an invalid simple C program.
//
// <<YOUR NAME>>
//
namespace compiler

module parser =
  //
  // NOTE: all functions in the module must be indented.
  //

  //
  // matchToken
  //
  let private matchToken expected_token tokens =
    //
    // if the next 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:
    //
    let next_token = List.head tokens

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

  //
  // simpleC
  //
  let private simpleC tokens =
    matchToken "$" tokens

  //
  // parse tokens
  //
  // Given a list of tokens, parses the list and determines
  // if the list represents a valid simple C program.  Returns
  // the string "success" if valid, otherwise returns a
  // string of the form "syntax_error:...".
  //
  let parse tokens =
    try
      let result = simpleC tokens
      "success"
    with
      | ex -> "syntax_error: " + ex.Message
```

To get started, let's look at another example. Consider the first rule from the BNF for simple C:

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

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

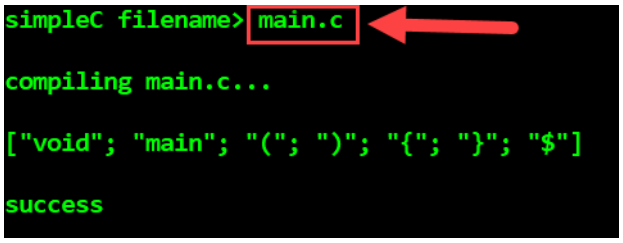
```
let rec private stmts tokens =
  tokens
```

Now let's write the function for **simpleC**, which you write 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
```

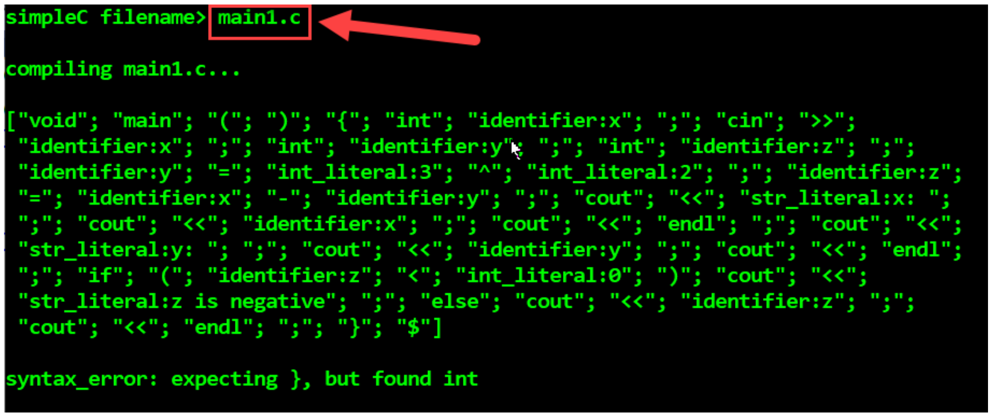
Next, we need a test case that matches what we are parsing --- a simple C program with no statements. Create a file “main.c” and enter this code into the file:

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



```
simpleC filename> main.c

compiling main.c...

["void"; "main"; "("; ")"; "{"; "}"; "$"]

success
```

Now run your compiler, 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”, which is the program shown on page 1. 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
```

Requirements

No imperative programming. In particular this means no mutable variables, no loops, and no data structures other than F# lists. Use recursion and higher-order approaches, though you must adhere to the recursive descent approach, which implies recursion is the dominant strategy. Do not change “main.fs” nor “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).

Assignment details…

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’re going to run into the problem that the 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 --- i.e. 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.

Electronic Submission and Grading

Grading will be based on the correctness of your compiler. We are not concerned with efficiency at this point, only correctness. Note that we will test your compiler against a suite of simple C input files, some that compiler and some with syntax errors. Make sure you name appears in the header comment of "parser.fs".

When you are ready to submit your program for grading, login to Gradescope and upload your "parser.fs" source file. You can upload your entire program, but we are only going to run and test your parser component; this implies you cannot change the main program, nor the lexical analyzer. You have unlimited submissions, and Gradescope keeps a complete history of all submissions you have made. By default, Gradescope records the score of your last submission, but if that score is lower, you can click on "Submission history", select an earlier score, and click "Activate" to select it. The activated submission will be the score that gets recorded, and the submission we grade. If you submit on-time and late, we'll grade the last submission (the late one) unless you activate an earlier submission.

The grade reported by Gradescope will be a tentative one. After the due date, submissions will be manually reviewed to ensure project requirements have been met. Failure to meet a requirement --- e.g. use of mutable variables or loops --- will trigger a large deduction.

Policy

Late work *is* accepted. You may submit as late as 24 hours after the deadline for a penalty of 10%, and as late as 48 hours for a penalty of 20%. Submissions are not accepted after 48 hours.

Unless stated otherwise, all work submitted for grading *must* be done individually. 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. The University's policy is available here:

https://dos.uic.edu/conductforstudents.shtml .

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. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, 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 https://dos.uic.edu/conductforstudents.shtml .