Project 05 : Checking simple C programs

(doc v1.0)

Assignment: F# program to semantically check simple C programs
Evaluation: Gradescope followed by manual execution & review
Policy: Individual work only
Complete By: Thursday, April 27th @ 11:59pm CDT for full credit

Late submissions: 10% penalty for code that is submitted up to 24 hours late (Friday, April 28st)
30% penalty for code that is submitted up to 48 hours late (Saturday, April 29th)
48 hours after due date, code cannot be submitted for credit

Background

This is part 2 of a multi-part programming project in F#. Part 1 was Project 04. Here in part 2 we're going to check **simple C** programs for semantic errors. In particular, we're going to perform two semantic checks:

1. Does the program properly define all variables?
2. Is the program using types correctly?

A sample program is shown to the right. Note that the syntax of simple C has expanded to allow the declaration of real variables (line 7) and the use of real literals (line 19). The program passes all semantic checks. Note that the assignment of an integer to a real variable (line 9) is allowed, otherwise all types must match exactly (e.g. the comparison on line 19). However, the program on the right does yield one warning: "*comparing real numbers with == may never be true*" (line 19).

If you were unable to finish part 1 (i.e. Project 04), a solution will be provided; see the "Getting Started" section for more information. Note that this solution will not become available until after the last late deadline date for Project 04.

```
 1 void main()
 2 {
 3   int x;
 4   cin >> x;
 5
 6   int  y;
 7   real z;
 8   y = 3^2;  // 3 squared:
 9   z = x-y;
10
11   cout << "x: ";
12   cout << x;
13   cout << endl;
14
15   cout << "y: ";
16   cout << y;
17   cout << endl;
18
19   if (z == 1.0)
20     cout << "z is 1.0";
21   else
22     cout << z;
23
24   cout << endl;
25 }
```

Updated BNF definition of simple C

The BNF from project 04 has been extended to add two features: the declaration of real variables in **<vardecl>** and support for real literals in **<expr-value>**. Changes to the BNF are shown below in **boldface**:

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

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

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

<empty>        -> ;
<vardecl>      -> int identifier ;
                | real 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
              | real_literal
              | str_literal
              | true
              | false

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

Getting Started

We are providing all necessary files to build the compiler, including our parser solution from project #04. There are a total of five F# files involved in this project:

1. analyzer.fs
2. checker.fs
3. lexer.fs
4. parser.fs
5. main.fs

The files are available on replit.com in the project "**Project 05**". Solution version of parser.fs for Project 04 will become available after Project 04 late deadline date.

Feel free to delete the provided "parser.fs" file and substitute your own version from project #04. You'll be reusing the parser code in both "analyzer.fs" and "checker.fs", so you might find it easier to work with your own solution instead of ours. The project workflow is as follows:

1. leave main.fs alone --- minor changes for debugging purposes are fine, but do *not* change how lexer, parser, analyzer and checker functions are called
2. modify lexer.fs
3. test
4. modify parser.fs
5. test
6. modify analyzer.fs to build symbol table
7. test
8. modify checker.fs to perform type checking
9. test
10. submit

Step 1: modify lexer

Modify the **lexer** to recognize the following two tokens: (1) the keyword "real", and (2) real literals. A real literal is defined as one or more digits followed by a '.' followed by 0 or more digits. Examples of valid real literals: 12345.12345, 1. and 0.5.

Recognizing a new keyword is as simple as adding to the list of keywords; you'll see the list in "lexer.fs". To recognize real literals, the hint is to modify the code that recognizes integer literals. After finding an integer literal, use lookahead to see if the next token is a ".". If so, you have a real literal, otherwise you have an integer literal.

Test.

Step 2: modify parser

Modify the **parser** to accept the BNF shown on page 2. Since you have a working parser, you only need to handle the additions shown in **boldface**. Note that the additions will impact other rules beyond <vardecl> and <expr-value>, since other rules use lookahead to determine what to do. Example: "real" can now denote the start of a statement, in particular the start of a <vardecl>.

Test. And test some more. It is very important that you test the parser before continuing with the next step, because you're going to copy-paste the parser functions for steps 3 and 4. If the parser code is wrong, you'll copy those errors into other files.

Step 3: modify analyzer

In step 3 you're going to build a semantic analyzer that collects variable declarations into a data structure called a symbol table. In this case a symbol is a list of tuples, where each tuple is of the form (name, type). Both name and type are strings. For example, the program shown on the right declares 3 variables: x, y, and z. The analyzer will successfully analyze this program and build the following symbol table: [("z", "real"); ("y", "int"); ("x", "int")]. Note that the variables are listed in reverse order of declaration --- "x" was declared first, but appears last in the symbol table. **You must adhere to building a data structure of this exact structure and order.**

```
1 void main()
2 {
3   int x;
4   cin >> x;
5
6   int  y;
7   real z;
8   y = 3^2;  // 3 squared:
9   z = x-y;
10
11  cout << "x: ";
12  cout << x;
13  cout << endl;
14
15  cout << "y: ";
16  cout << y;
17  cout << endl;
18
19  if (z == 1.0)
20    cout << "z is 1.0";
21  else
22    cout << z;
23
24  cout << endl;
25 }
```

Write your analyzer in the provided file "analyzer.fs". Do not change the provided public function **build_symbol** --- work with the design as given. The easiest way to write this module is to copy-paste the parser functions from "parser.fs" and modify them as follows:

1. delete any code that checks / reports syntax errors
2. add logic to build and return the symbol table
3. add logic to check for duplicate variable declarations
4. add logic to check that all identifies/variables are declared before they are used.

The logic for #2 should focus on variable declarations. Assume that variable declarations will occur only at the main level of scoping, and will *not* occur as part of an if-then-else statement. In other words, the following code is legal in simple C, but assume it will not occur:

```
if (x == 0)
  int var1;
else
  int var2;
```

Since simple C doesn't support block statements, variable declarations make no sense as part of an if-then-else. The logic for #3 should throw an exception using "failwith", much like we did in project #04 when reporting a syntax error. Error message format:

```
failwith ("redefinition of variable '" + name + "'")
```

The logic for #4 should throw an exception using "failwith" much like we did in project #04 when reporting a syntax error. When working with a BNF function what uses an identifier except for <vardelc>, you will need to verify that the identifier name is part of the symbol table that has been created at the point in the program. Obviously, we need to find the error of using an identifier that is never declared; however, we also need to find the error when an identifier is used BEFORE it is declared. Error message format:

```
failwith ("variable '" + name + "' undefined")
```

Step 4: modify checker

In step 4 you're going to perform another semantic analysis, in this case type-checking. In particular, you are going to type-check assignment statements, if conditions, and expressions to make sure they are type-compatible. To perform type-checking, you'll also need to make sure that all variables have been declared. The rules for simple C are straightforward:

1. simple C has four types: "int", "real", "string", and "bool".
2. arithmetic operations --- +, -, *, /, and ^ --- must involve "int" or "real", and both operands must be the same type. The result of a valid arithmetic operation is the type of either operand.
3. comparison operators --- <, <=, >, >=, ==, and != --- must involve operands of the same type. Note that comparisons involving true and false are legal, e.g. true < false. The result of a valid comparison is the type "bool".
4. when assigning a value X to a variable Y, the types of X and Y must either be (a) the same, or (b) X may be "int" and Y may be "real".
5. if conditions must evaluate to a type of "bool"
6. issue a warning for expressions involving "==" with operands of type "real"

Modify "checker.fs" to enforce rules 2 – 6. Do not change the provided public function **typecheck** --- work with the design as given. As you did in step 3, the easiest approach is to copy-paste the parser functions and modify as needed. **When programming in F#, avoid using the name "type" as a name, since it is a keyword in F#.** In other words, do not write code such as:

```
let type = "int"  // this will fail because type is an F# keyword
```

If any of rules 2-5 are broken, throw an exception using "failwith" to immediately stop and report the error. Error message formats (these are in order for breaking of rules 2 – 5):

```
failwith ("operator " + next_token + " must involve 'int' or 'real'")

failwith ("type mismatch '" + left_type + "' " + operator + " '" + right_type + "'")

failwith ("cannot assign '" + expr_type + "' to variable of type '" + id_type + "'")

failwith ("if condition must be 'bool', but found '" + expr_type + "'")
```

In the case of rule 6, issue a warning by printing a message with the following format:

```
printfn "warning: comparing real numbers with == may never be true"
```

The compiler should keep running after the warning is output.

Hint: you'll need to pass around the tokens as well as the symbol table. Type-checking is typically performed by modifying the functions such as <expr-value> to return the type, e.g. "string" or "int" or "real" or "bool". Then, for expressions with an operator, you obtain the types of the left-hand and right-hand operands, and then perform the type-check. For an assignment statement, you obtain the types of the right-hand value and the left-hand identifier, and then perform the type-check. And so on. Since the functions need to return the tokens along with the type, **use tuples** to return multiple values.

Test.

Requirements

No imperative programming. In particular, this means no mutable variables, no loops, and no data structures other than F# lists. No file I/O other than what is performed by the lexer.

Also, no global/module level variables/values other than what is provided in the lexer. For example, the symbol table built in step 3 by the analyzer should be returned and passed to the checker in step 4 --- do not try to store the symbol table into some sort of global or module-level variable to avoid parameter-passing. One of the goals of this assignment (and this class) is to learn functional programming, and parameter-passing is an important aspect of functional programming. Taking shortcuts like global / module-level variables will lead to a project score of 0.

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 your name appears in the header comments of any file you modify. You also need to comment the body of any files you modify, as appropriate.

When you are ready to submit your program for grading, login to Gradescope and upload your program files. We will ignore "main.fs", and grade the remaining four files:

1. analyzer.fs
2. checker.fs
3. lexer.fs
4. parser.fs

When testing your analyzer and checker modules, we will use our own solution for the other files to make sure you are adhering to the overall design guidelines for this assignment. For example, when we test your "checker.fs" module, we will use *our* lexer, parser, analyzer, and main program files.

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. We will grade the last submission 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 generally fail the submission with a project score of 0.

Policy

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 .