CS 341, Spring 2024
Project 5 – SimpleC Type Checking
Due: Wednesday 4/24 at 11:59pm

Overview

In this project, you will be writing an F# program to check SimpleC programs for semantic errors. The goal is to determine two results:

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

SimpleC

The syntax of SimpleC has expanded from what was used in the previous project to allow the declaraction of real variables (line 7) and the use of real literals (line 19).

An example SimpleC program is shown below. The program shown passes all semantic checks. Note that the assignment of an integer to a real variable (line 9) is allowed, but 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).

```
 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 }
```

CS 341, Spring 2024
Project 5 – SimpleC Type Checking
Due: Wednesday 4/24 at 11:59pm

Updated BNF Definition of SimpleC

The BNF ("Backus–Naur Form") definition of the SimpleC syntax 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 highlighted in yellow.

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

CS 341, Spring 2024
Project 5 – SimpleC Type Checking
Due: Wednesday 4/24 at 11:59pm

Getting Started

There are a total of five F# files involved in this project: analyzer.fs, checker.fs, lexer.fs, parser.fs, and main.fs.

Begin by taking your parser.fs solution from the previous project and paste it into the parser.fs file for this project.

Step 1 – Modify the Lexer

Modify the lexer to recognize the following two tokens:

1. The keyword “real”
2. Real literals. A real literal is defined as one or more digits followed by a '.' followed by zero or more digits. Examples of valid real literals are 12345.12345, 1. and 0.5.

Recognizing a new keyword is as simple as adding it to the list of keywords that is already present in “lexer.fs”.

To recognize real literals, the hint is to modify the code that recognizes integer literals. After finding an integer literal, look ahead to see if the next token is a '.' – if it is, this is a real literal and if not, then it is an integer literal. Tokens for real literals should begin with “real_literal” just like “int_literal” and “str_literal”.

Test to check that your modifications to the lexer are working correctly.

Step 2 – Modify the Parser

Modify the parser to accept the updated BNF shown above. Note that the additions will impact other rules beyond `<vardecl>` and `<expr-value>`, since other rules look ahead to determine what to do. As an example, the keyword “real” can now denote the start of a statement.

Test to check that your modifications to the parser are working correctly. It is very important that you test the parser before continuing with the next step. You will be copying and pasting the parser functions for steps 3 and 4 so if they are wrong, those errors will be copied to other files.

Step 3 – Modify the Analyzer

Build a semantic analyzer that collects variable declarations into a data structure called a symbol table. The symbol table will be implemented as a list of tuples, where each tuple is of the form (name, type). Both name and type are strings. For example, the sample SimpleC program shown earlier declares three variables: x, y, and z. The analyzer will successfully analyze this program and build the following symbol table:

[(“z”, “real”); (“y”, “int”); (“x”, “int”) }

CS 341, Spring 2024
Project 5 – SimpleC Type Checking
Due: Wednesday 4/24 at 11:59pm

Note that the variables are listed in *reverse* order of declaration: "x" was declared first in the program, but appears last in the symbol table. **You must adhere to building a data structure of this exact structure and order.**

Write your analyzer in the provided file "analyzer.fs". Do not change the provided public function **`build_symboltable`** but work with the design as given. The easiest way to write this part 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 identifiers/variables are declared before they are used.

The logic to build and return the symbol table (#2 above) 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 SimpleC, but assume that it will not occur:

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

Since SimpleC doesn't support block statements, variable declarations make no sense as part of an if-then-else.

The logic for checking for duplicate variable declarations (#3 above) should throw an exception using "failwith". The format of the error message should be as follows:

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

The logic for checking that all identifiers/variables are declared before they are used (#4 above) should throw an exception using "failwith" as well. When working with a BNF function that uses an identifier (except for **`<vardecl>`**), 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. The format of the error message should be as follows:

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

Test to check that your modifications to the analyzer are working correctly. It is strongly recommended that you test after each function that you write/modify to be sure that it works as intended before moving on to the next one.

CS 341, Spring 2024
Project 5 – SimpleC Type Checking
Due: Wednesday 4/24 at 11:59pm

Step 4 – Modify the Checker

In this step, perform another semantic analysis: type-checking. In particular, type-check assignment statements, if conditions, and expressions to make sure they are type-compatible. To perform type-checking, you will also need to make sure that all variables have been declared.

These are the rules for SimpleC, which are rather straightforward:

1. SimpleC 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 the file "checker.fs" so that it enforces rules 2-6. Do not change the provided public function **`typecheck`** but work with the design as given. As you did when modifying the analyzer, 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. The format of the error message should be as follows (in order for the rules 2-5 above):

```
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 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 to check that your modifications to the checker are working correctly. It is strongly recommended that you test after each function that you write/modify to be sure that it works as intended before moving on to the next one.

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 all five F# files are 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.

CS 341, Spring 2024
Project 5 – SimpleC Type Checking
Due: Wednesday 4/24 at 11:59pm

Requirements

Do not use imperative style programming: no mutable variables, no loops, and no data structures other than F# lists. No file input/output other than what is performed by the lexer.

In addition, no global/module level variables/values other than what is provided in the lexer. For example, the symbol table built by the analyzer (step 3) should be returned and passed to the checker (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 is to learn functional programming, and parameter-passing is a large component of that. **Taking shortcuts like global or module level variables will lead to a project score of zero.**

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 05”. Submit your program files to that assignment. You can upload your entire program, but the autograder will only run and test the 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.

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.

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, will trigger a large deduction.**

CS 341, Spring 2024
Project 5 – SimpleC Type Checking
Due: Wednesday 4/24 at 11:59pm

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.

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.