Parsing and Type Checking

In this project, you are asked to write a parser and a type checker for a small language. The parser checks that the input is syntactically correct and the type checker enforces the semantic rules of the language. The semantic rules that your program will enforce relate to declarations and types. In addition, your program will check for unused variables and for the use of uninitialized variables and outputs corresponding error messages.

The input to your code will be a program and the output will be:

- syntax error message if the input program is not syntactically correct
- if the input program has no syntax error, the output is:
  - semantic error messages if there is a declaration error or a type mismatch in the input program,
  - error messages if there are variables that are declared but never used or variables that are used before they are assigned a value, or
  - information about the types of the symbols declared in the input program if there is no semantic error.

In what follows, I describe the language syntax and semantic rules.

1. Language Syntax

1.1. Grammar Description

program	→ scope	
scope	→ LBRACE scope_list RBRACE	
scope_list	→ scope	
scope_list	→ var_decl	
scope_list	→ stmt	
scope_list	→ scope scope_list	
scope_list	→ var_decl scope_list	
scope_list	→ stmt scope_list	
var_decl	→ id_list COLON type_name SEMICOLON	
id_list	→ ID	
id_list	→ ID COMMA id_list	
type_name	→ REAL | INT | BOOLEAN | STRING	
stmt_list	→ stmt	ll
stmt_list	→ stmt stmt_list	
stmt	→ assign_stmt	
stmt	→ while_stmt	
assign_stmt	→ ID EQUAL expr SEMICOLON	
while_stmt	→ WHILE LPAREN condition RPAREN LBRACE stmt_list RBRACE	
while_stmt	→ WHILE LPAREN condition RPAREN stmt	
expr	→ primary	
expr	→ arithmetic_expr	
expr	→ boolean_expr	
arithmetic_expr	→ arithmetic_operator arithmetic expr arithmetic expr	
arithmetic_expr	→ arithmetic_primary	
boolean_expr	→ binary_boolean_operator boolean expr boolean expr	

```
boolean_expr            → relational_operator expr expr
boolean_expr            → NOT boolean_expr
boolean_expr            → boolean_primary
arithmetic_operator     → PLUS  | MINUS  | MULT  | DIV
binary_boolean_operator → AND  | OR  | XOR
relational_operator     → GREATER  | GTEQ  | LESS  | NOTEQUAL  | LTEQ
primary                 → ID  | NUM  | REALNUM  | STRING_CONSTANT  | bool_const
arithmetic_primary      → ID  | NUM  | REALNUM  | STRING_CONSTANT
boolean_primary         → ID  | bool_const
bool_const              → TRUE  | FALSE
condition               → boolean_expr
```

Notice that expressions are in prefix notation in this language. This makes parsing easier.

If you read the grammar carefully, you will notice that it is ambiguous. Some `primary`, such as `ID`, is also `boolean_expr` and `arithmetic_expr` because some lexemes can be parsed as `boolean_primary` and as `arithmetic_primary` as well as `primary`. This is only an issue at the top level parsing of an expression (`parse_expr()`). To resolve the ambiguity, you should parse `expr` as `primary` when it is a primary.

1.2. Tokens

The tokens are included for completeness. You are provided with a `getToken()` function for the following tokens used in the grammar description:

```
LBRACE          =  {
RBRACE          =  }
COLON           =  :
SEMICOLON       =  ;
REAL            =  REAL
INT             =  INT
BOOLEAN         =  BOOLEAN
STRING          =  STRING
WHILE           =  WHILE
COMMA           =  ,
LPAREN          =  '('
RPAREN          =  ')'
EQUAL           =  =
PLUS            =  +
MULT            =  *
DIV             =  /
AND             =  ^
OR              =  '|'
XOR             =  &
NOT             =  ~
GREATER         =  >
GTEQ            =  >=
LESS            =  <
LTEQ            =  <=
NOTEQUAL        =  <>
TRUE            =  TRUE
FALSE           =  FALSE
STRING\_CONSTANT=  " (letter | digit)* "
```

```
ID          =   letter (letter | digit)*
NUM         =   0 | (pdigit digit*)
REALNUM     =   NUM . digit digit*
```

where

```
letter      =   a |b |c |... |z |A |B |C |... |Z
digit       =   0 |1 |2 |... |9
pdigit      =   1 |2 |3 |... |9
```

2. Language Semantics

2.1. Scoping Rules

Static scoping is used. I have already covered static scoping in class, so you should know the semantics and how references should be resolved. Every `scope` defines a scope.

2.2. Types

The language has four basic types: `INT`, `REAL`, `BOOLEAN`, and `STRING`.

2.3. Variables

Variables are declared explicitly in `var_decl` of the form `id_list : type_name`. The names of the declared variables appear in the `id_list` and their type is the `type_name`.

Example Consider the following program written in our language:

```
{
    x : INT;
    y : INT;
    y = x;
}
```

This program has two declared variables: `x` and `y`. Both have type `INT`.

2.4. Declaration and Reference

Any occurrence of a name in the program is either a **declaration** or a **reference**. Any occurrence of a name in an `id_list` that is part of a `var_decl` is a declaration of the name. Any other occurrence of a name is considered a reference to that name. Note that the above definitions exclude the built-in type names, which are predeclared as part of the language definition.

Given the following example (the line numbers are not part of the input):

```
01 {
02        a : INT;
03        b : REAL;
04        b = x;
05 }
```

We can categorize all occurrences of names as **declaration** or **reference**:

- Line 2, the occurrence of name `a` is a declaration
- Line 3, the occurrence of name `b` is a declaration
- Line 4, the occurrence of name `b` is a reference
- Line 4, the occurrence of name `x` is a reference

2.5. Type System

We describe which assignments are valid (type compatibility) and how to determine the types of expressions (type inference).

2.5.1. Type Compatibility Rules

Type compatibility rules specify which assignments are valid. The Rules are the following.

- **C1:** If the type of the lefthand side of an assignment is `BOOLEAN`, or `STRING`, the righthand side of the assignment should have the same type.
- **C2:** If the type of the lefthand side of an assignment is `INT`, the righthand side of the assignment should be `INT` or `BOOLEAN`.
- **C3:** If the type of the lefthand side of an assignment is `REAL`, the type of the righthand side of the assignment should be `INT` or `REAL`.
- **M1:** Assignments that do not satisfy **C1**, **C2** or **C3** are not valid. In this case, we say that there is an assigment type mismatch.

2.5.2. Type Inference Rules

- **C4:** The types of the operands of an arithmetic operator should be `REAL` or `INT` or `STRING`
- **C5:** The type of the operands of a `binary_boolean_operator` should be `BOOLEAN`
- **C6:** If neither operand of a `relational operator` has type `INT` or `REAL`, then the operands should have the same type. In this case, both types can be `STRING` or both types can be `BOOLEAN`
- **C7:** If the first operands of an `arithmetic_operator` has type `STRING`, then the second operand should also have type `STRING`.
- **C8:** If the second operand of an `arithmetic_operator` has type `STRING` and the operator is `MULT`, then the first operant should have type `STRING` or `INT`. In this case, the two operands can have different types.
- **C8':** If the second operand of a arithmetic operator has type `STRING` and the operator is not `MULT`, then the first operand should also have type `STRING`
- **C9:** If one of the operand of a `relational operator` has type `INT` or `REAL`, then the other operand should have type `INT` or `REAL`. In this case, the two operands can have different types.

- **C10:** The type of a `condition` should be `BOOLEAN`.
- **C11:** The type of the operand of the `NOT` operator should be boolean.
- **I1:** If **C4**, **C7**, **C8**, and **C8'** are satisfied, and the type of one of the operands of an arithmetic operator is `REAL`, the type of the resulting expression is `REAL`.
- **I2:** If **C4**, **C7**, **C8**, and **C8'** are satisfied, and the type of one of the operands of an arithmetic operator is `STRING`, the type of the resulting expression is `STRING`.
- **I3:** For the `PLUS`, `MINUS`, and `MULT` operators, if the types of the two operands are `INT`, the type of the resulting expression is `INT`.
- **I4:** For the `DIV` operator, if the types of the two operands are `INT`, the type of the resulting expression is `REAL`.
- **I5:** If the two operands of a `binary_boolean_operator` have type `BOOLEAN`, the type of the resulting expression is `BOOLEAN`.
- **I6:** If **C6** and **C9** are satisfied, the type of a `relational operator expr expr` is `BOOLEAN`.
- **I7:** The type of a `NUM` constant is `INT`
- **I8:** The type of a `REALNUM` constant is `REAL`
- **I9:** The type of a `bool_const` constant is `BOOLEAN`
- **I10:** The type of a `STRING CONSTANT` constant is `STRING`
- **M2:** If **C4**, **C5**, **C6**, **C7**, **C8**, **C8'**, **C9**, **C10** or **C11** are not satisfied, the type of the expressions is ERROR. In this case, we say that there is an expression type mismatch.

3. Parser

You must write a parser for the grammar, If your parser detects a syntax error in the input, it should output the following message and then it should exit:

```
Syntax Error &!#@
```

You should start coding by writing the parser first and make sure that your parser generates a syntax error message if the input program is syntactically incorrect.

The grammar of the language is not LL(1) i.e. it does not satisfy the conditions for predictive parser with one token lookahead, however, it is still possible to write a recursive descent parser with no backtracking. We can do this by looking ahead at more than one token in some cases and leftfactoring some rules. Two rules like

```
stmt_list   → stmt
stmt_list   → stmt stmt_list
```

can be parsed by first parsing `stmt` and then checking if the next token is the beginning of `stmt_list` or in the FOLLOW of `stmt_list`. In fact the two rules are equivalent to

```
stmt_list    → stmt stmt_list1
stmt_list1   → stmt_list  |
```

but you do not need to explicitly leftfactor the rules by introducing `stmt_list1` to parse `stmt_list`. You have already handled similar grammars previously.

Do not forget the remark at the end of Section 1.1 about the ambiguity of the rules for `expr`. As stated in Section 1.1, if a `expr` is a `primary`, it should be parsed as `primary` and not as `artithmetic_expr` or `boolean_expr`.

4. Semantic Checking

Your program will check for the following semantic errors and output the apropriate message when it encounters that error.

4.1. Declaration Errors

- **Variable declared more than once** (error code **1.1**)
  A variable is declared more than once if appears more than once in the same `id_list` of a `var_decl` or if it appears in two `id_list` of two different `var_decl` and *locally* looking up the name that appears in one of the two declarations returns the other declaration.
- **Undeclared variable** (error code **1.2**)
  If resolving a variable name that appears in a statement other than a declaration returns no declaration, the variable is undeclared.
- **Variable Declared but not used** (error code **1.3**)
  If a variable is declared but is not referenced, we say that the variable is declared but not used. We say that a declaration is referenced if some reference to the variable resolves to the declaration. If a declaration is not referenced, we have error code 1.3.
- **Redeclaration or reference as a variable for of a builtin type name** If a builtin type name appears in `id_list` of a variable declaration or appears in a statement other than a declaration statement, your parser should output syntax error.

For each error involving declarations, your type checker should output one line in the following format:

```
ERROR CODE <code> <symbol_name>
```

in which `<code>` should be replaced with the proper code (see the error codes listed above) and `<symbol_name>` should be replaced with the name of the variable that caused the error. Since the test cases will have at most one error each, the order in which these error messages are printed does not matter.

Note that there will only be at most one declaration error per test case.

4.2. Type Mismatch

If there are no declaration errors and any of the type constraints (listed in the Type System section above) is violated in the input program, then the output of your program should be:

```
TYPE MISMATCH <line_number> <constraint>
```

Where `<line_number>` is replaced with the line number of the line in which the violation occurs and `<constraint>` should be replaced with the label of the violated type constraint (possible values are **C1**, **C2**, **C3**, **C4**, **C5**, **C6**, **C7**, **C8**, **C8'**, **C9**, or **C11**). See the section on Type System for the details of each constraint. Note that you can assume that anywhere a violation can occur it will be on a single line.

4.2.1. Type Mistmatch and Other Errors

- **Type mismatch and C1, C2 or C3 violations** You should not declare C1, C2 or C3 violation if the expression on the RHS has a type mismatch error. If there is a type mismatch error in the expression on the RHS of an assignment (C4, C5, C6, C7, C8, C8' C9, C11), you should not also declare C1, C2 or C3 errors. I illustrate this with an example

```
1:  {     x : INT;
2:        y : BOOLEAN;
3:        z : INT;
4:        z = + x y;
5:  }
```

  Here, on line 4, we have a violation of C4 because the type of y is BOOLEAN which violates C4. Given that C4 is violated, the type of + x y is really not defined and we could conclude that C1 is violated, but you should not declare a C1 violation in this case. Similar examples can be made for other combinations.

- **Type mismatch and C10 Violations** You should not declare C10 violation if the condition has a type mismatch error.

- **Type Mismatch and Declaration Errors** If there are declaration errors and type mismatch errors, only the output for the declaration errors should be produced.

Note that there will only be at most one type mismatch error per test case.

4.3. Use of Uninitialized Variables

For this part, your program should identify variables that are used before they are *defined*. We say that a *variable is used if it appears in an expression*. We say that a *variable is defined if it appears on the lefthand side of an assignment* (it is assigned a value). A variable is used before it is defined if there is a program execution path in which there is a use of the variable that is not preceded along the execution path by a definition of the variable. Compilers typically call this "use of an uninitialized variable". In general determining use of uninitialized variables is involved and requires building a control flow graph of the program. Fortunately, for the language of this project, determining uninitialized uses is relatively easy. First, I will describe the output format, then I will give a more detailed description of how determine uninitialized uses.

The output format for use of uninitialized variables is the following

```
UNINITIALIZED <name_reference_1> <line_no_reference_1>
UNINITIALIZED <name_reference_2> <line_no_reference_2>
...
```

Where `<name reference i>` is the name of i 'th uninitialized variable and `<line_no_reference_i>` is the line number in which the i 'th uninitialized reference appears.

In what follows, I will explain how to determine uses of uninitialized variables. Consider the program below (the line numbers are not part of the input but are added for ease of reference):

```
01 {
02      x1, x2, x3, x4 , x5 : INT;
03
04      x3 = 2;
05
06      WHILE ( < x1 100 ) {
07            x1 = + x1 1;
```

```
08          x2 = 1;
09          x4 = + x2 1;
10          x1 = + x3 1;
11          WHILE ( < x1 10 ) {
12              x5 = + x1 + x3 x5 ;
13          }
14          x1 = x5;
15          x5 = 1;
16      }
17
18      x2 = + x2 1 ;
19  }
```

- The use of x1 in the expression < x1 100 is a use of an uninitialized variable. The first time the expression is evaluated, there is no previous definition (assignment) to x1. Note that later evaluations of < x1 10 happen after the assignment x1 = + x1 1; so the use of x1 in < x1 10 counts as initialized.
- The use of x1 in x1 = + x1 1; on line 07 is uninitialized because the first time + x1 1 is evaluated, x1 has no initial value.
- The use of x2 on line 09 is initialized because it is always preceded by the definition (assignment to) of x2 on line 08.
- The use of x3 on line 10 is initialized because it is always preceded by the definition (assignment to) x3 on line 04.
- The use of x1 on line 11 is initialized because it is always preceded by the definition (assignment to) x1 on line 10 (also on line 07).
- The use of x1 on line 12 is initialized because it is always preceded by the definition (assignment to) x1 on line 10 (also on line 07).
- The use of x3 on line 12 is initialized because it is always preceded by the definition (assignment to) x3 on line 04.
- The use of x5 on line 12 is not initialized because it is not preceded by a definition (assignment to) x5.
- The use of x5 on line 14 is not initialized because it is not preceded by a definition (assignment to) x5. Even though there is a definition (assignment to) of x5 on line 12, we cannot determine (in general) if the body of the loop will be executed, so we assume that the preceding loop did not execute. Note that this situation is different from the use of x1 on line 12 which is guaranteed to be preceded by the definition (assignment to) of x1 on line 10.
- The use of x2 on line 18 is not initialized because it is not preceded by a definition (assignment to) x2. Even though x2 is defined (assigned a value) on line 08, we cannot determine (in general) if the body of the loop is executed, so we have to assume that it is not executed.

It should be clear from the example, that to determine uninitialized variables, you can treat while_stmt as some kind of a scope. If a use is inside a while_stmt, then all definitions preceding it in the while_stmt together with all definitions preceding it in enclosing while_stmt should be taken into consideration.

4.4. No Semantic Errors

If there are no declaration errors, no type mismatch errors and no use of uninitialized variables, then your program should output for each reference of a declared variable name, in the order in which the reference appears in the program, the line number in which the reference appears and the line number of the declaration to which the reference of the name resolves. For this part, the format should be the following

```
<name_reference_1> <line_no_reference_1> <line_no_declared_1>
<name_reference_2> <line_no_reference_2> <line_no_declared_2>
...
```

Where <name reference i>is the name of a variable and corresponds to i'th name reference in the program. <line_no_reference_i> is the line number in which the i'th reference appears and <line_no_declared_i> is the line number of the declaration corresponding to the i'th reference.

5. More Examples

Example 1. Given the following example (the line numbers are not part of the input):

```
01 {
02      x : INT;
03      y : BOOLEAN;
04      y = x;
05 }
```

The output will be the following:

```
TYPE MISMATCH4 C1
```

Example 2. Given the following example (the line numbers are not part of the input):

```
01 {
02      x : BOOLEAN;
03      y : REAL;
04      y = x;
05 }
```

The output will be the following:

```
TYPE MISMATCH4 C3
```

Example 3. Given the following example (the line numbers are not part of the input):

```
01 {
02      x : BOOLEAN;
03      x : BOOLEAN;
04      x = x;
05 }
```

The output will be the following:

```
ERROR CODE 1.1 x
```

Example 4. Given the following example (the line numbers are not part of the input):

```
01 {
02      x : INT;
03      y, x : STRING;
04      y = 10;
05      x = 10;
05 }
```

The output will be the following:

```
ERROR CODE 1.1 x
```

Example 5. Given the following example (the line numbers are not part of the input):

```
01 {
02      x : INT;
03      y : STRING;
04      y = "abc";
05 }
```

The output will be the following:

```
ERROR CODE 1.3 x
```

Example 6. Given the following example (the line numbers are not part of the input):

```
01 {
02      x1 , x2 : INT;
03      y : INT;
04      x1 = y;
05      y = + x1 x2;
06 }
```

The output will be the following:

```
UNINITIALIZED y 4
UNINITIALIZED x2 5
```

Example 7. Given the following example (the line numbers are not part of the input):

```
01  {
02          a : INT;
03          b : INT;
04          a = 1;
05          b = 1;
06          {       a : INT;
07                  a = 1;
08                  WHILE ( > a b )
09                  {
10                          a = - a b;
11                  }
12          }
13  }
```

The output will be the following:

```
a 4 2
b 5 3
a 7 6
a 8 6
b 8 3
a 10 6
a 10 6
b 10 3
```

6. Summary of Requirements

1. If there is a syntax error, the program should output syntax error and exit.
2. If there is no syntax error and there is a declaration error, the program should output the error message for the declaration error. You can assume that there can be no more than one declaration error in the test case. If the program outputs a declaration error, the program should output no

other error message. Specifically, it should not output any type mismatch or uninitialized variable error message.

3. If there is no syntax error and no declaration error and there is a type mismatch, the program should output the error message for the type mismatch. You can assume that there can be no more than one type mismatch per test case. If the program outputs a type mismatch error message, the program should output no other error message. Specifically, it should not output any uninitialized variable error message.
4. If there is no syntax error, no declaration error and no type mismatch error, the program should output the list of variables used but not initialized in the format specified in Section 4.3.
5. If there is no error of any kind, the program should output for each reference, the line number of the reference and the line number of the declaration to which the reference resolves as explained in Section 4.4 above.

---

7. Implementation Suggestions

1. the provided code has the `main()` function in lexer. It is better to have a new file for parser and move the `main()` function to the parser.
2. You should first write the parser using the approach we studied in class.
   - The parser will require more than one lookahead in some places.
   - Do not start on any other part before finishing the parser.
   - If you do not follow the approach we studied in class methodically, you can end up with a parser that passes almost all cases and for which it is hard to find what is wrong.
   - In the past, some students could only fix their parser errors by rewriting it completely from scratch.
3. Semantic checking might affect parsing.
   - If there is syntax error, only the syntax error message should be produced and no other error messages should be produced.
   - If you produce semantic error messages while parsing, you might end up generating a semantic error message before encountering the syntax error message. That is why you should ”save” your semantic error message and wait until parsing is done to print it.
   - If your program crashes or has segmentation fault, that can affect the output of syntax checking.
4. Type checking for expressions.
   - You should have `parse_expr()` return a type (an integer value denoting the type).

- For the base case `expr -> primary` (or for the `arithmetic_primary` or `boolean_primary`), the type can be determined immediately for constants and by looking up the variable type in the case primary is `ID`.
- For the general case, it will be helpful to have a function in which all the type checking rules are implemented. The function takes as input an operator and two types (one is ignored in the case of unary operator) and returns the type of the resulting expressing or detects semantic error.