COMP 520 Compiler Design
Individual Assignment #2
Language Specifications

Overview:

Given the lexical and syntactic choices from the first assignment, your second assignment is to implement the relevant semantics for each language construct. Note that as you implement your compiler, some decisions may be more difficult to implement than you first thought - or we might have missed out a key detail you need. In these cases, bring them up in class and we can discuss possible changes. By the end of this assignment you will have produced your first full compiler!

Specifications

A program in `MiniLang` consists of a list of variable declarations followed by a list of statements.

Declarations

Variable declarations must follow the following 2 rules or a compile time error "`Error: <description>`" with explanation must be thrown.

- Variable identifiers must be defined before being used

  ```
  var a : int = 0;
  b = 5; // Error: (line 2) "b" is not declared
  ```

- Variable identifiers must not be redeclared (regardless of type)

  ```
  var a : int = 0;
  var a : int = 1; // Error: (line 2) "a" is already declared
  ```

For this assignment, we will be using the following types from `C`

- Integers: `int`
- Float: `float`
- Strings: `char*`
- Booleans: `bool` (look into `stdbool.h`)

Variable Initialization

Variable initialization (i.e. the initial value specified in a declaration) must follow the assignment compatibility rules described in the section below.

Statements

- **Read** into a variable according to `C scanf` semantics. Use `%d` for integers and booleans, `%f` for floats, and `%s` for strings (buffer size of 1024).

  ```
  read x;
  ```

- **Print** an expression according to `C printf` semantics for integers, floats, and strings. For booleans use `%d`.

  ```
  print x * x;
  ```

- **Assignment** into a variable. Assignment compatibility is as follows:

  ```
  int := int
  float := float, int
  string := string
  boolean := boolean
  ```

  Note that `floats` may not be assigned to `ints`.

- **If statement**, with optional else branch. The condition `<expression>` must be an integer or a boolean

  ```
  if <expression> {
      <stmts>
  } [ else {
      <stmts>
  }]
  ```

- **While loop**. The condition `<expression>` must be an integer or a boolean

  ```
  while <expression> {
      <stmts>
  }
  ```

Expressions

Literals

Literals in `MiniLang` have their corresponding types. i.e. an integer literal is of type `int`, etc.

Binary Operations

Given a binary expression `<expr1> <op> <expr2>` where `<op>` is one of `+, -, *, /`:

- `int <op> int` is `OK` and results in type `int`
- `float <op> float` is `OK` and results in type `float`
- `int <op> float` (and vice-versa) is `OK` and results in type `float`

- `string + string` is `OK` and results in type `string`. The semantics of this operation are string concatenation.
- `string * int` (and vice-versa) is `OK` and results in type `string`. The semantics of this operation are string repetition.

  ```
  "derp" * 3 -> "derpderpderp"
  ```

  Note that a runtime check to ensure the integer is ≥ 0 will be required. A value of 0 results in the empty string. A negative value produces a runtime exception.

Given a binary expression `<expr1> <op> <expr2>` where `<op>` is one of `&&`, `||`:

- `<expr1>` and `<expr2>` can either be of type `int` or `bool`
- The resulting expression has type `bool`

Given a binary expression `<expr1> <op> <expr2>` where `<op>` is one of `==`, `!=`:

- `<expr1>` and `<expr2>` can be of any type (they must be the same type)
- The resulting expression has type `bool`

Unary Operations

Given a unary minus expression `- <expression>`

- `<expression>` must be either of type `int` or of type `float`
- The resulting expression has the same type as `<expression>`

Given a unary not expression `! <expression>`

- `<expression>` must be of type `bool`
- The resulting expression has type `bool`

Bonus (3 points)

- **Read** values into a boolean variable. Booleans are read as *strings* and must be either "TRUE" or "FALSE" (case-sensitive) and converted to the appropriate `TRUE` or `FALSE` value. You must produce a runtime error (i.e. an error produced by your *generated* code) if the string does not conform to the specification
- **Print** a boolean expression: "TRUE" and "FALSE" for `TRUE` and `FALSE` respectively.
- **Assignment** into a variable. Support the additional assignment compatibility for string/boolean conversion

  ```
  string := boolean
  boolean := string
  ```

  Use the following relations for converting between booleans and strings

  - `true`: "TRUE"
  - `false`: "FALSE"

  If the conversion from string to boolean fails (i.e. an invalid string), then a runtime error should be produced.