COMP 520 Compiler Design
Individual Assignment #2
Language Specifications

(Updated February 4th)

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 interwoven variable declarations and statements.

Declarations

Variable declarations must follow the following rules or emit a compile time error "`Error: <description>`".

- 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 = 0; # Error: (line 2) "a" is already declared
```

- Variable identifiers follow typical static nested scoping rules. Identifiers may be re-declared in nested scopes and shadow parent declarations, but are not visible in sibling or parent scopes.

```
var a : int = 0;
if (true) {
    var a : float = 0.0;
    a = 5.0; # Uses the inner-most declaration (float)
    var b : int = 0;
}
b = 5; # Error: (line 7) "b" is not declared
```

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

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

Variable Declarations

Variable initialization must follow the compatibility rules described in the section above. If an initial type is not specified, the variable type is inferred after typechecking the expression.

```
var x = True; # Type of ‘x’ inferred as bool
var y = 0 + 1.0; # Type of ‘y’ inferred as float
```

Statements

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

  ```
  read(x);
  ```

  Note that booleans must be read in as either “True” or “False” (case-sensitive) and converted to the appropriate `True` or `False` value. Anything else should produce a runtime error (i.e. an error produced by your *generated* code).

- **Print** an expression according to `C printf` semantics for integers, floats, and strings. For booleans, print “True” or “False”. The output is terminated by a newline.

  ```
  print(x * x);
  ```

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

  ```
  int := int
  float := float, int
  string := string
  bool := bool
  ```

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

- **If statement**, with zero or more else if and optional else branch. The conditions (`<expression>`) for all branches must be of type `bool`

  ```
  if (<expression>) {
      <stmts>
  }
  [else if (<expression>) {
      <stmts>
  }]*
  [else {
      <stmts>
  }]
  ```

- **While loop**. The condition `<expression>` must be of type `bool`

  ```
  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.

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

- `<expr1>` and `<expr2>` must be of type `bool`
- The resulting expression has type `bool`

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

- *Updated Feb 4*: `<expr1>` and `<expr2>` must either be the same type (valid for any type), or some combination of `int`/`float`
- The resulting expression has type `bool`

Unary Operations

Given a unary minus expression `-<expression>`

- `<expression>` must be either of type `int` or `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`