Imperative Interpreter with Functional Interpreter

ART-C Interpreter

Description

You must write an interpreter for a toy imperative language named ART-C (which is shorthand for Another Rhetorical Toy-C). The interpreter must be written in the functional language of Scheme. A code-base is provided that you will complete. The code can be found at ART_C codebase(artc-baseline.rkt). You must download this code base and add features in order to satisfy this assignments requirements.

Syntax

ART-C is an imperative language that supports only the boolean and int data types along with basic arithmetic and logical operators. There are no function calls, arrays, or complex data types. While strings can occur in an ART-C program in the context of an sprint, the string data type is not supported.

The syntax of ART-C is given by the EBNF grammar below where the last three productions are informally defined and the starting non-terminal is `<PROGRAM>`. Literal elements (terminal strings) are colored in blue while non-literals (items that are either part of EBNF or non-terminals) are in black.

```
<PROGRAM> ::= ( program <BLOCK> )
<BLOCK> ::= ( block <DECLARATIONS> <BODY> )
<DECLARATIONS> ::= { <DECL> }
<BODY> ::= { <STATMENT> }
<DECL> ::= ( declare <TYPE> <VARIABLE> )
<STATEMENT> ::= <BLOCK> | <ASSIGN> | <IF> | <WHILE> | <SPRINT>
<ASSIGN> ::= ( := <VARIABLE> <EXPRESION> )
<IF> ::= ( if <EXPRESSION> <STATEMENT> [ <STATEMENT> ] )
<WHILE> ::= ( while <EXPRESSION> <STATEMENT> )
<SPRINT> ::= ( sprint <STRING> [ <EXPRESSION> ] )
<VARIABLE> ::= <LETTER> { ( <LETTER> | <DIGIT> ) }
<EXPRESSION> ::= <INT> | <BOOLEAN> | <VARIABLE> | <BINARY> | <UNARY>
<BINARY> ::= ( <OP> <EXPRESSION> <EXPRESSION> )
<OP> ::= + | - | * | / | @ | # | < | > | = | <= | >= | & | %
<UNARY> ::= ([~|-] <EXPRESSION> )
<INT> ::= <DIGIT> { <DIGIT> } | -<DIGIT> {<DIGIT>}
<TYPE> ::= boolean | int
<BOOLEAN> ::= true | false
<STRING> ::= A Scheme String
<LETTER> ::= Any of A-Z or a-z
<DIGIT> ::= Any of 0-9
```

Semantics

This section informally defines the semantics of each program element. Since this specification is not formal, it is likely to be ambiguous and/or incomplete but should nonetheless convey a reasonable specification of the expected meaning of each program element. Seek clarification of any part of the specification that you believe is unclear. The language supports int and boolean types long with the expected imperative control structures.

PROGRAM
Execution of a program means execute the block. The meaning of the program is the meaning of the block when executed in the context of the state imposed by the declarations.

BLOCK
Executing a block means execute each statement in the body of the block in the contextd of the type-map imposed by the `DECLARATIONS`. Each `STATEMENT` is executed in the order it occurs and the meaning of the `BLOCK` is the state produced by the final statement in the `BLOCK`.

DECLARATIONS
A list of `DECL`s. Since a `DECLARATIONS` does not effect the state and meaning is defined in terms of state, there is a sense in which a `DECLARATIONS` does not have any meaning. However, a `DECLARATIONS` will always affect the type map by inserting each declaration into the map. Note, that the scope of a single declaration extends only to the `BODY` of the associated `BLOCK`.

ASSIGN
Binds the value the expression to the named variable. The type of the variable and the type of the expression must be identical; otherwise there is an error.

IF
The meaning is the meaning of the first statement if the conditional expression evaluates to true. Otherwise, the meaning is the meaning of the second statement if it is present. Otherwise, the meaning is the state in which the `IF` is executed.

WHILE

Execution of a while first evaluates the expression and then executes the associated statement if the expression is true after which this process repeats.

EXPRESSION
Evaluation of an expression produces the value of the expression.

BINARY
A binary expression represents either a logical or arithmetic operation. See Figure 1 for details. Each operator is defined as that of the corresponding Java operator.

UNARY
There are two unary expressions. See Figure 1 for details. Each operator is defined as that of the corresponding Java operator.

SPRINT
Prints the string (this is for labeling an output) followed by the value of the expression to the interpreter window and then prints a newline. If the expression is not provided, this function will print the state (a list of all variables that are presently in scope and their corresponding values).

Operator	Arity	Meaning	Operand Type	Expression Type
+	2	addition	int	int
-	2	subtraction	int	int
*	2	multiplication	int	int
/	2	division	int	int
@	2	exponentiation	int	int
?	2	remainder	int	int
<	2	less than	int	boolean
>	2	greater than	int	boolean
=	2	equal to	int	boolean
<=	2	less than or equal to	int	boolean
>=	2	greater than or equal to	int	boolean
&	2	logical and	boolean	boolean
%	2	logical or	boolean	boolean
~	1	logical negation	boolean	boolean
-	1	arithmetic negation	int	int

Figure 1. Operators

Validity

You must write a function `(is-program-valid? pgm)` that accepts a program and returns true if the program is valid and false otherwise. The intent of this function is to ensure that the program is strongly and statically typed. Note that the input program is assumed to be syntactically correct. Each of the following rules must be checked such that if any one of them is violated, the program is not valid.

1. Reserved words are not valid variable names. For example, variable names “true”, “while” and “int” are not allowed.
2. Block scoping rules are enforced.
    a. Duplicate variables declared in the same scope (block) are not allowed.
    b. Variables of inner scopes (blocks) hide variables of outer scopes (blocks).
    c. All variables that are referenced have been declared and are in scope.
3. Enforce constraints in the literals such that every `INT` literal must be a valid Java int literal.
4. Ensure that the expression of an `IF` is a boolean
5. Ensure that the expression of a `WHILE` is a boolean
6. Ensure that the types of the `VARIABLE` and `EXPRESSION` of each assignment are identical.
7. Ensure that the operands of all operators are of the correct type.

Additional Rule

One other rule must also be checked: the rule that all variables must be initialized prior to use. You have the option of performing this check statically or dynamically.

Interpreter

You must complete the `interpret` *function*. This implies completing all of the statement-specific interpret functions that are referenced in the skeletal code along with any other supporting functions that you need. Ensure that your interpreter is strongly (and necessarily dynamically) typed.

The `interpret` function will execute a valid program and return the resulting state. The `interpret` method will also accept programs for which `(is-program-valid? pgm)` is false and will execute them correctly until the very moment that an invalid action is invoked. At that moment, rather than performing an invalid action, the interpreter will throw an error and will be called. A valid program is given below for your consideration.

```
(define pgm '(program
              (block
               (declare int n)
               (declare boolean error)
               (declare int result)
               (:= error false)
               (:= result 1)
               (block
                (declare int local)
                (:= n 5)
                (:= local n)
                (while (> local 0)
                       (block
                        (:= result (* result local))
                        (:= local (- local 1)))))
              (sprint "result: " result)
              (if (~ error) (sprint "a") (sprint "b")))))
```

Parallelism (521 MSE Students Only)

Extend the ART-C language by adding a parallel statement. A parallel construct is a tagged list of statements as given by example below. Semantically (not necessarily in real-time), the statements in the parallel construct will be executed simultaneously; each starting from the same state. The resulting state will be the union of the states generated by each statement. If there are conflicting values in any of the resulting states, then an error occurs. In order to complete this excercise you must

1. Modify the EBNF syntax wherever necessary to include a parallel construct.
2. Extend the interpreter by writing an interpret-parallel function.
3. Extend the static type-checking function to include type-checking a parallel element.

An example of a valid `parallel` construct is shown below. Of course, variables x, y, and q must already exist int the enclosing typemap for this to be valid.

```
(parallel
  (:= n 5)
  (while (> x 0)
     (block
       (declare int z)
       (:= z 3)
       (:= x (- x 1))
       (:= y (+ y x))))
   (:= q 2))
```

Submission

1. You must submit all your work using GitLab(https://cs.uwlax.edu:17443) using a project named `cs421` with a folder named **hw3**. Here is a GitLab tutorial(http://charity.cs.uwlax.edu/shared/gitlab-tutorial/tutorial.html).
2. The scheme code must be placed in a single file named **artc.scm**.