Interpret Mini-C with Java and Optimize with Scheme

Minor corrections to "Hints" and submission details. Changed Nov. 10th

Optimization

Overview

*This problem builds upon simplified expressions. For this problem, we assume that only arithmetic expressions involving the operators +, -, * and / along with single-character variable names and integer numbers. We also assume that simplification of these expressions conforms with the rules of the previous homework assignment.*

Once an expression has been simplified *syntatically*, it is then possible to optimize *evaluation* of the expression. For example, consider the expression `(* (+ a 3) (+ 3 a))`. This expression simplifies to `(* (+ a 3) (+ a 3))`. Notice, however, that while the expression has been simplified, evaluation is inefficient since the sub-expression `(+ a 3)` is evaluated twice. We can optimize this expression by extracting the common sub-expression, evaluating it once, and then refering to that result as required. If we optimize the simplified expression, we might obtain `(let* ((g15488 (+ a 3))) (* g15488 g15488))`.

Function

Write a scheme function named `optimize` that takes a scheme expression (as defined in homework 2), simplifies that expression (as defined in homework 2), and then optimizes that expression by extracting all common-subexpressions into a `let*` form. Follow the process outlined below when writing your solution.

1. Find all common-subexpressions within the expression EXP. Atoms (numbers and variables) do not count as common subexpressions. Refer to this list of subexpressions as EXPS.
2. Find the ***smallest*** element of EXPS. Call it S-EXP. We here define *smallest* as the expression having the fewest elements where each operator, variable and number are considered an element.
3. Generate a unique symbol via `(gensym)` refered to as V. Create a list of the form `(V S-EXP)`.
4. Replace all occurrences of S-EXP in the remaining elements of EXPS with V.
5. Replace all occurrences of S-EXP in EXP with V.
6. Repeat the prior steps until there are no more sub-expressions to process.
7. Finally, create a `(let* ((V S-EXP) ..) EXP')` expression such all `(V S-EXP)` constructs form the bindings of the `let*` and the expression EXP' is the result of having replaced all occurrences of all sub-expressins in EXP with their associated V values.

Examples

```
(optimize '(* a a)) ==> (* a a)
(optimize '(* (+ a 1) (+ 1 a))) ==> (let* ((g128572 (+ a 1))) (* g128572 g128572))
(optimize '(* (+ a (+ b 1)) (+ (+ 1 b) a))) ==> (let* ((g128574 (+ b 1)) (g128575 (+ a g128574))) (* g128575 g128575))
```

Hints

You can find the common sub-expressions of an expression EXP by first performing a walk of the expression (a tree) and accumulating all trees in the EXP. Second, sort this list-of-trees from **smallest** to **largest** (see above) while breaking ties in a way that ensures that all "equal" expressions are grouped together. If any expression occurs more than once in this list, it is a common sub-expression.

Mini-C Interpreter

Description

You must write a type-checker and interpreter for a toy imperative language named MINI-C, a C-Like language with block scoping. The type-checker and interpreter must be written in Java and incorporate a pre-provided parser and code base. The codebase can be downloaded as a zip file from minic.zip.

After downloading and extracting this project, you must complete the files named `interpreter.Interpreter.java` and `typing.TypeChecker.java`. No other code should be modified. The only classes that you should edit are the Interpreter and TypeChecker classes.

Mini-C Concrete Syntax (The Grammar)

Mini-C is an imperative language that supports only the `boolean` and `int` data types along with basic arithmetic, logic, and relational operators. There are no function calls, arrays, strings, or complex data types.

The syntax of ART-C is given by the implied EBNF grammar below where several productions are informally defined (LETTER and INTEGER for example) and the starting non-terminal is <PROGRAM>. Terminal symbols are colored in blue while non-terminal-symbols (items that are either part of EBNF or non-terminals) are rendered in black. **For this project, the concrete syntax is not as significant as the abstract syntax that follows since a parser (the code dealing with the concrete syntax) has already been provided.**

```
<PROGRAM> ::= int main() { <DECLARATIONS> <BLOCK> }
<BLOCK> ::= { <DECLARATIONS> {<STATEMENT>} }
<DECLARATIONS> ::= { <DECL> }
<DECL> ::= <TYPE> <VARIABLE> ;
<STATEMENT> ::= <BLOCK> | <ASSIGN> | <IF> | <WHILE> | <SKIP>
<ASSIGN> ::= <VARIABLE> = <EXPRESION> ;
<IF> ::= if(<EXPRESSION> ) <STATEMENT> [ else <STATEMENT> ] ;
<WHILE> ::= while(<EXPRESSION> ) <STATEMENT>
<IDENTIFIER> ::= <LETTER> { ( <LETTER> | <DIGIT> ) }
<SKIP> ::= ;
<EXPRESSION> ::= <CONJUNCTION> { || <CONJUCTION> }
<CONJUNCTION> ::= <EQUALITY> { && <EQUALITY> }
<EQUALITY> ::= <RELATION> { <EQ-OP> <RELATION> }
<RELATION> ::= <ADDITION> { <REL-OP> <ADDITION> }
<ADDITION> ::= <TERM> { <ADD-OP> <TERM> }
<TERM> ::= <FACTOR> { <MUL-OP> <FACTOR> }
<FACTOR> ::= [ <UNARY-OP> ] <PRIMARY>
<PRIMARY> ::= <IDENTIFIER> | <LITERAL> | ( <EXPRESSION> )
<LITERAL> ::= <INTEGER> | <BOOLEAN-LITERAL>
<EQ-OP> ::= = | !=
<REL-OP> ::= < | > | <= | >=
<ADD-OP> ::= + | -
<MUL-OP> ::= * | /
<UNARY-OP> ::= - | !
<TYPE> ::= boolean | int
<BOOLEAN-LITERAL> ::= true | false
```

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.

PROGRAM
Execution of a program means execute the body in the context of the state imposed by the `DECLARATIONS`. The meaning of the program is the state that results from executing the body.

BLOCK
Executing a block means execute each statement in the body of the block in the context of the state imposed by combining the enclosing state with modifications 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`. Note that blocks introduce a new variable scope such that variables from external scopes can be hidden (i.e. new variables with the same name can be declared in nested blocks).

DECLARATIONS
Executing a `DECLARATIONS` means execute each `DECLARATION` in the order that they occur. The meaning of the `DECLARATIONS` is the state that results after execution of the last `DECLARATION`.

DECLARATION
The meaning of a `DECLARATION` is the state that results from adding the declared variable to the state and assigning it a value that denotes the notion of *not initialized*. Note, that the scope of a single declaration extends only to the `BODY` of the associated `BLOCK` or `PROGRAM`.

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.

Operators

Each of the operators below takes on a conventional meaning. Note that *logical or* and *logical and* must be short circuited and that the types of the left-and-right operands of either of the assignments must be identical. This table is **not** related to the precedence of the operators.

Operator	Arity	Meaning	Operand Type	Expression Type
+	2	addition	int	int
-	2	subtraction	int	int
*	2	multiplication	int	int
/	2	division	int	int
<	2	less than	int	boolean
>	2	greater than	int	boolean
==	2	equal to	int or boolean	boolean
!=	2	not equal to	int or boolean	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

Mini-C Abstract Syntax

The abstract syntax of ART-C is given below. The abstract syntax defines the objects that you must use when writing your type checker and interpreter. These grammatical classes correspond to actual classes in the provided code base.

```
Program = Declarations decPart; Block body;
Declarations = Declaration*;
Declaration = Type t, Variable v;
Statment = Skip | Block | Assignment | Conditional | Loop
Skip = empty;
Block = Declarations* declarations; Statement* statements;
Assignment = Variable target; Expression source;
Conditional = Expression test; Statement thenbranch; Statement elsebranch;
Loop = Expression test; Statement body;
Expression = Variable | Value | Binary | Unary;
Binary = Operator op; Expression term2; Expression term2;
Unary = Operator op; Expression term;
```

Validity Function (40 Points)

You must complete the static function `TypeChecker.isValid( Program p )` that accepts a program and returns `true` if the program is valid and `false` otherise. The intent of this function is to ensure that the program is strongly and statically typed. **Note that the input program must be syntactically correct in order to obtain a Program object on which to operate**. Each of the following rules must be checked such that if any one of them is violated, the program is not valid.

Since this function returns only a boolean, but we would like to know why a program is not valid if the returned value is false, the function should print a message when any type-error is encountered. Ensure that you walk the entire tree and print all errors you detect.

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 must have been previously declared and must be in scope.
3. Enforce constraints in the literals such that every `INT` literal must be a in $[-2147483648 \ldots 2147483647]$.
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.

Meaning Functions (60 Points)

You must complete the static function `Interpreter.meaning(Program p)` that accepts a single valid program object and returns the resulting state. Recall that the meaning of a program is given by the state that it produces.

Test Files

Here are several test files for your interpreter and the expected output. The **c** and **txt** versions of the file are identical apart from naming since the web browser won't show a **c** file but only download it. The **txt** file is provided as a convenience. Note that the ordering of the elements in the output is not semantically meaningful. Also note that my test suite will be more expansive when grading your submissions as these files don't provide full coverage of all requirements.

test1.c | test1.txt

```
{factorial=6, i=3, n=3}
```

test2.c | test2.txt

```
{baseRaisedToExp=8, exp=0, base=2}
```

test3.c | test3.txt

```
{result=64, exp=0, base=2}
```

test4.c | test4.txt

```
{divisor=5, isInputDivisibleByDivisor=true, input=40}
```

test5.c | test5.txt

```
{output=1, x=3, y=4, z=false}
```

test6.c | test6.txt

```
{divisor=5, high=100, low=1, sum=1050}
```

Submission

1. You must submit your work using GitLab using a project named `cs421` within a folder named **hw3**. Submite files named: `optimize.rkt`, `Interpreter.java` and `TypeChecker.java`. Ensure that the Java files work in the context of the provided code base.