CS 415 Project 2

Spring 2020

You may implement this using any programming language that runs on the ilab machines.

Let's write a compiler for an imperative language, adding features as we go. For each part, the target language is ILOC code (see Appendix A of EaC). You can use the provided ILOC simulator on ilab machines to run your ILOC code. The simulator assumes a memory size of 20,000 bytes, starting at 0, and 256 registers.

The simulator also supports an `output a` instruction, which prints the contents of memory at location `a`.

For example, this ILOC program should output 42 when run through the simulator:

```
loadI 1024 => r0
loadI 42   => r1
storeAI r1 => r0, 4
output 1028
```

You can run the simulator by piping in the ILOC code on stdin (here assuming it's in file `test.i`):

```
$ ./sim < test.i
42
Executed 4 instructions and 4 operations in 8 cycles.
```

(You can ignore the last line.)

Your compiler should read from stdin and output to stdout.

If you have a test source file in `test1`, for example, you should be able to compile and run the resulting ILOC code with the one-liner:

```
cat test1 | ./compiler | ./sim
```

Or to just print the compiler's ILOC output:

```
cat test1 | ./compiler
```

or

```
./compiler < test1
```

Please submit a tarball on Canvas with all your source code, as well as a README file that describes which features are working, and instructions on how to build and run your program.

Since the parts below build on each other, you don't have to submit separate files for each part, just the files for your final compiler.

Part 1: getting started (30 points)

Let's start with this imperative language:

$$\begin{aligned}
\langle \mathit{Program} \rangle &::= \langle \mathit{Main} \rangle \\
\langle \mathit{Main} \rangle &::= \texttt{def main}\ \langle \mathit{VarDecls} \rangle\ \langle \mathit{Stmts} \rangle\ \texttt{end} \\
\langle \mathit{VarDecls} \rangle &::= \langle \mathit{VarDecl} \rangle \texttt{;}\ \langle \mathit{VarDecls} \rangle \mid \epsilon \\
\langle \mathit{VarDecl} \rangle &::= \langle \mathit{Type} \rangle\ \langle \mathit{Id} \rangle \\
\langle \mathit{Type} \rangle &::= \texttt{int} \mid \texttt{bool} \\
\langle \mathit{Stmts} \rangle &::= \langle \mathit{Stmt} \rangle \texttt{;}\ \langle \mathit{Stmts} \rangle \mid \langle \mathit{Stmt} \rangle \texttt{;} \\
\langle \mathit{Stmt} \rangle &::= \langle \mathit{Assign} \rangle \mid \langle \mathit{Print} \rangle \\
\langle \mathit{Assign} \rangle &::= \langle \mathit{Id} \rangle\ \texttt{=}\ \langle \mathit{Expr} \rangle \\
\langle \mathit{Print} \rangle &::= \texttt{print (}\langle \mathit{Id} \rangle\texttt{)} \\
\langle \mathit{Expr} \rangle &::= \langle \mathit{Expr} \rangle\ \langle \mathit{BinOp} \rangle\ \langle \mathit{Expr} \rangle \\
&\quad \mid\ \texttt{!}\ \langle \mathit{Expr} \rangle \\
&\quad \mid\ \langle \mathit{Id} \rangle \\
&\quad \mid\ \langle \mathit{Const} \rangle \\
&\quad \mid\ \texttt{(}\langle \mathit{Expr} \rangle\texttt{)} \\
\langle \mathit{BinOp} \rangle &::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{\^{}} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{==} \mid \texttt{!=} \mid \texttt{>=} \mid \texttt{>}
\end{aligned}$$

The semantics are, roughly speaking, the same as what you'd expect from C or Java, but the language is restricted to types of int and boolean, so the `/` operator does integer division. Note that the `^` operator is xor, not exponentiation.

Identifiers are any sequence of alphanumeric characters, but they cannot start with a number.

Constants are either a non-negative integer constant (10, 3, 0, 123, etc.) or a boolean constant (true or false).

Your compiler should do typechecking to make sure that int and bool variables only store values of the appropriate type. There is no implicit type conversion. Print "`error: type mismatch`" for any type error. Print "`error: variable undeclared`" if a variable is used without being declared. For example, these should be detected as compile-time errors:

- `int x; x = true;`
- `bool b; b = 1;`
- `bool b; int x; x = b;`

- `int x; x = y;`

For example, here's a program that does some arithmetic:

```
def main
    int n;
    n = (10 * (10 + 1)) / 2;
    print(n);
end
```

Part 2: control flow (30 points)

Now let's modify the set of statements to include if statements and two kinds of loops:

$\langle Stmt \rangle$ ::= $\langle Assign \rangle$ | $\langle Print \rangle$ | $\langle While \rangle$ | $\langle If \rangle$ | $\langle For \rangle$

$\langle While \rangle$ ::= `while` `(`$\langle Expr \rangle$`)` `do` $\langle Stmts \rangle$ `end`

$\langle If \rangle$ ::= `if` `(`$\langle Expr \rangle$`)` `do` $\langle Stmts \rangle$ `end`
  | `if` `(`$\langle Expr \rangle$`)` `do` $\langle Stmts \rangle$ `else` $\langle Stmts \rangle$ `end`

$\langle For \rangle$ ::= `for` `(`$\langle Stmt \rangle$`;` $\langle Expr \rangle$`;` $\langle Stmt \rangle$`)` `do` $\langle Stmts \rangle$ `end`

Note that the conditional expressions in each statement should have type boolean.

For example, here is a program to compute the sum of 1 through 10:

```
def main
    int sum;
    int i;
    sum = 0;
    for (i = 1; i <= 10; i = i+1) do
        sum = sum + i;
    end;
    print(sum);
end
```

Part 3: functions (30 points)

Let's modify the grammar to allow for functions besides main:

$\langle Program \rangle$ ::= $\langle Functions \rangle$ $\langle Main \rangle$

$\langle Functions \rangle$ ::= $\langle Function \rangle$ $\langle Functions \rangle$ | ϵ

$\langle Function \rangle$ ::= `def` $\langle Id \rangle$ `(`$\langle Params \rangle$`)` `:` $\langle Type \rangle$ $\langle VarDecls \rangle$ $\langle Stmts \rangle$ `end`

$$
\begin{array}{lcl}
\langle \mathit{Params} \rangle & ::= & \epsilon \mid \langle \mathit{SomeParams} \rangle \\[1ex]
\langle \mathit{SomeParams} \rangle & ::= & \langle \mathit{Param} \rangle \ \langle \mathit{MoreParams} \rangle \\[1ex]
\langle \mathit{Param} \rangle & ::= & \langle \mathit{Type} \rangle \ \langle \mathit{Id} \rangle \\[1ex]
\langle \mathit{MoreParams} \rangle & ::= & \texttt{,} \ \langle \mathit{SomeParams} \rangle \mid \epsilon \\[1ex]
\langle \mathit{Stmt} \rangle & ::= & \langle \mathit{Assign} \rangle \mid \langle \mathit{Print} \rangle \mid \langle \mathit{While} \rangle \mid \langle \mathit{If} \rangle \mid \langle \mathit{For} \rangle \mid \langle \mathit{Return} \rangle \\[1ex]
\langle \mathit{Expr} \rangle & ::= & \langle \mathit{Expr} \rangle \ \langle \mathit{BinOp} \rangle \ \langle \mathit{Expr} \rangle \\
 & \mid & \texttt{!} \ \langle \mathit{Expr} \rangle \\
 & \mid & \langle \mathit{Id} \rangle \ \texttt{(}\langle \mathit{Args} \rangle\texttt{)} \\
 & \mid & \langle \mathit{Id} \rangle \\
 & \mid & \langle \mathit{Const} \rangle \\
 & \mid & \texttt{(}\langle \mathit{Expr} \rangle\texttt{)} \\[1ex]
\langle \mathit{Args} \rangle & ::= & \epsilon \mid \langle \mathit{SomeArgs} \rangle \\[1ex]
\langle \mathit{SomeArgs} \rangle & ::= & \langle \mathit{Expr} \rangle \ \langle \mathit{MoreArgs} \rangle \\[1ex]
\langle \mathit{MoreArgs} \rangle & ::= & \texttt{,} \ \langle \mathit{SomeArgs} \rangle \mid \epsilon \\[1ex]
\langle \mathit{Return} \rangle & ::= & \texttt{return} \ \langle \mathit{Expr} \rangle
\end{array}
$$

The return type of the function is written after the colon in the function definition. All functions return either an int or boolean value. For a function without a return statement, print "`error: no return value`".

The ⟨Expr⟩ nonterminal has a new third production for function calls, and that ⟨Id⟩ must be a function name.

Variables of the same name in different functions should be independent. All parameters are passed by value.

For example, we can now modify our sum example:

```
def sumUpTo(int n) : int
    int sum;
    int i;
    sum = 0;
    for (i = 1; i <= n; i = i+1) do
        sum = sum + i;
    end;
    return sum;
end

def main
    int n;
    int sum;
    n = 10;
    sum = sumUpTo(n);
    print(sum);
end
```

Part 4: optimization (10 points)

Implement one of the following optimizations:

- constant propagation – expressions involving only constants should be computed at compile time and replaced with their result. For example, the expression `2 * (3 + 4)` should be replaced with the value 14.
- common subexpression elimination – the same subexpression appearing multiple times in an expression should only be evaluated once. For example, `(x + y) * (x + y)` should only do one addition and one multiplication.