Homework 3

Due: Tue, Apr 4, 2017

In this assignment, you will be implementing a parser for your calculator. The program will accept input, one line at a time, parse each line of input as an expression (see below), and then evaluate it, printing the result.

The following sections define the abstract and concrete syntax of the calculator's language.

1 Abstract syntax

This section defines the abstract syntax of the calculator. This defines the internal representation of mathematical expressions, the rules for constructing well-formed expressions, and their meaning in terms of their mathematical values.

The calculator defines two types, which are defined as follows:

$$\begin{array}{lll} t & ::= & \textbf{bool} \\ & & \textbf{int} \end{array}$$

The type **bool** describes the values true ($\top$) and false ($\bot$). The type **int** describes integer values in the left-open range $[-2^{31-1}, 2^{31-1})$.

The language defines a number of expressions. These are:

e ::=	**true**		$e_1 < e_2$	less than
	false		$e_1 > e_2$	greater than
	n	integer values	$e_1 \leq e_2$	less than or equal to
	e_1 **and** e_2	and	$e_1 \geq e_2$	greater than or equal to
	e_1 **or** e_2	inclusive or	$e_1 + e_2$	addition
	not e_1	logical negation	$e_1 - e_2$	subtraction
	if e_1 **then** e_2 **else** e_3	conditional	$e_1 * e_2$	multiplication
	$e_1 = e_2$	equal to	e_1 **div** e_2	integer division
	$e_1 \neq e_2$	not equal to	e_1 **rem** e_2	remainder of division
			$-e_1$	arithmetic negation

An expression is a sequence of operands and operators that specifies a computation. The evaluation of an expression results in a value. The type of an expression determines how expressions can be combined to produce complex computations and the kind of value it produces. The following paragraphs define the requirements on operands and the result types of each expression as well the values they produce.

The order in which an expression's operands are evaluated is unspecified unless otherwise noted.

1. The expressions **true** and **false** have type **bool** and the values $\top$ and $\bot$, respectively.

2. Integer values have type **int**. The value of an integer literal is the one indicated by the expression.

3. The operands of the logical expressions e_1 **and** e_2 and e_1 **or** e_2 shall have type **bool**. The result type of each is **bool**. The result of e_1 **and** e_2 is $\top$ if both operands are $\top$ and $\bot$ otherwise. If e_1 is $\bot$, then e_2 is not evaluated. Note that this is equivalent to **if** e_1 **then** e_2 **else false**. The result of e_1 **or** e_2 is $\bot$ if both operands are $\bot$ and $\top$ otherwise. If e_1 is $\top$, then e_2 is not evaluated. Note that this is equivalent to **if** e_1 **then true else** e_2.

4. The operand of **not** e_1 shall have type **bool**, and the type of the expression is **bool**. The result of the expression is $\top$ when the e_1 is $\bot$ and $\bot$ otherwise.

5. In the expression **if** e_1 **then** e_2 **else** e_3, the type of e_1 shall be **bool**, and e_2 and e_3 shall have the same type. The type of the expression is the type of e_2 and e_3. The result of expression is determined by first evaluating e_1. If that value is $\top$ then the result of the expression is the value of e_2, otherwise, it is the value of e_3. Only one of e_2 or e_3 is evaluated.

6. The operands of the expressions $e_1 = e_2$ and $e_1 \neq e_2$ shall have the same type. The result type is **bool**. The result of $e_1 = e_2$ is $\top$ if e_1 and e_2 are equal and $\bot$ otherwise. The result of $e_1 \neq e_2$ is $\top$ if e_1 and e_2 are different and $\bot$ otherwise.

7. The operands of the expressions $e_1 < e_2$, $e_1 > e_2$, $e_1 \leq e_2$, and $e_1 \geq e_2$ shall have type **int**. The result type is **bool**. The result of $e_1 < e_2$ is $\top$ if e_1 is less than e_2 and $\bot$ otherwise. The result of $e_1 > e_2$ is $\top$ if e_1 is greater than e_2 and $\bot$ otherwise. The result of $e_1 \leq e_2$ is $\top$ if e_1 is less than or equal to e_2 and $\bot$ otherwise. The result of $e_1 \geq e_2$ is $\top$ if e_1 is greater than or equal to e_2 and $\bot$ otherwise.

8. The operands of the expressions $e_1 + e_2$, $e_1 - e_2$, $e_1 * e_2$, e_1 **div** e_2, and e_1 **rem** e_2 shall have type **int**. The result type is **int**. The result of $e_1 + e_2$ is the sum of the operands. If the sum is greater than the maximum value of **int**, the result is undefined. The result of $e_1 + e_2$ is the difference resulting from the subtraction of the e_2 from e_1. If the difference is less than the minimum value of **int**, the result is undefined. The result of $e_1 * e_2$ is the product of the operands. If the product is greater than the maximum value of **int**, the result is undefined. The results of e_1 **div** e_2 and e_1 **rem** e_2 are the quotient and remainder of dividing e_1 by e_2, respectively. In either case, if e_2 is 0, the result is undefined. If e_2 is the minimum value of **int**, the result is undefined. For division, the fractional part of the value is discarded (the value is truncated towards zero). If the expression a **div** b is defined, (a **div** b) $* b + a$ **rem** b is equal to a.

9. The operand of $-e_1$ shall have type **int**, and the type of the expression is **int**. The result of the expression is the additive inverse of the value of e_1. Note that $-e_1$ is equivalent to $0 - e_1$.

2 Program execution

This section describes the structure and execution semantics of the program. An implementation uses the rules described in this section to produce and evaluate expressions from the previous section.

A *program* is a sequence of *expression*s to be evaluated. The *source code* of a program is a sequence of lines where each line contains a single *expression*.

The program is executed by evaluating the *expressions* in the source file in the order in which they appear. The result of each expression is written to an *output* file in the following format:

- If the type of the expression evaluated is **bool**, then the printed value is `true` or `false`.
- Otherwise, the value printed is the integer value of the expression.

Note that the output file may be standard output (i.e., `std::cout`).

The value of the last expression shall be the exit code of the program.

If the evaluation of any expression results in undefined behavior, the program *aborts*.

Note that the source code may be provided directly through user input (i.e., `std::cin`). This also allows for input redirection. The program should read a single line of input, lex, parse, and evaluate that line, and then print the result. For example, if the input is:

```
3 + 4 * 5;
true == false
```

Then the output will be:

```
23
false
```

3 Lexical conventions

This section describes the lexical conventions of the language. That is, it determines the set of symbols used in the concrete syntax of the language. The information in this section can be used to derive a specification of a lexical analyzer and the tokens that it produces.

The lexical conventions describe the symbols (tokens) of the grammar's language. The tokens are punctuators, operators, or literals. The language defines the following tokens representing punctuators and operators:

```
+   -   *  /  %
&&  ||  !
==  !=  <  >  <=  >=
?   :
(   )
```

The boolean literals are:

$$\textit{boolean-literal} \;\rightarrow\; \texttt{true} \mid \texttt{false}$$

The integer literals are:

$$\begin{array}{lcl} \textit{integer-literal} & \rightarrow & \textit{digit digit}* \\ \textit{digit} & \rightarrow & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \end{array}$$

Note that this language effectively has only 3 kinds of tokens:

- punctuators and operators are tokens that have no information beyond their name,
- boolean literals have an associated boolean value as an attribute, and
- integer literals have an associated integer value as an attribute.

4 Concrete syntax

This section gives the concrete syntax of the language as a context-free grammar in Backus-Naur form (i.e., a set of productions that define the valid strings of the language. This specification can be used to derive a parser that matches a sequence of tokens to produce a corresponding abstract syntax tree.

A program is well-formed only if its tokens form a string in the language. Said otherwise, if the parser fails to match a sequence of tokens to a particular production, the input contains a syntax error. In such cases, the program is not evaluated, and calculator should diagnose the specific source of the error.

Most productions have associated semantic actions which include additional rules that determine whether or not a program is well-formed. These are defined below.

expression	→	*conditional-expression*
conditional-expression	→	*logical-or-expression* `?` *expression* `:` *expression*
		logical-or-expression
logical-or-expression	→	*logical-or-expression* `||` *logical-and-expression*
		logical-and-expression
logical-and-expression	→	*logical-and-expression* `&&` *equality-expression*
		equality-expression
equality-expression	→	*equality-expression* `==` *ordering-expression*
		equality-expression `!=` *ordering-expression*
		ordering-expression
ordering-expression	→	*ordering-expression* `<` *additive-expression*
		ordering-expression `>` *additive-expression*
		ordering-expression `<=` *additive-expression*
		ordering-expression `>=` *additive-expression*
		additive-expression
additive-expression	→	*additive-expression* `+` *multiplicative-expression*
		additive-expression `-` *multiplicative-expression*
		multiplicative-expression
multiplicative-expression	→	*multiplicative-expression* `*` *unary-expression*
		multiplicative-expression `/` *unary-expression*
		multiplicative-expression `%` *unary-expression*
		unary-expression
unary-expression	→	`!` *unary-expression*
		`-` *unary-expression*
		primary-expression
primary-expression	→	*boolean-literal*
		integer-literal
		`(` *expression* `)`

Note that this grammar can be easily implemented as a top-down, recursive descent parser. However, you must remove all left-recursive productions from the grammar by refactoring them as right-recursive productions.

Semantically, when a production of the grammar matches a sequence of tokens, it produces a corresponding abstract syntax tree. The semantic rules in this section define that correspondence. If the types of operands do not conform to the requirements specified in the abstract syntax, the program is ill-formed, and the calculator must diagnose the error.

1. A *conditional-expression* `e1 ? e2 : e3` corresponds to the abstract syntax **if** e_1 **then** e_2 **else** e_3 where each subexpression is represented by its corresponding abstract syntax.

2. A *logical-or-expression* `e1 || e2` corresponds to the abstract syntax e_1 **or** e_2 where each subexpression is represented by its corresponding abstract syntax.

3. A *logical-and-expression* `e1 && e2` corresponds to the abstract syntax e_1 **and** e_2 where each subexpression is represented by its corresponding abstract syntax.

4. An *equality-expression* `e1 == e2` corresponds to the abstract syntax $e_1 = e_2$ where each subexpression is represented by its corresponding abstract syntax. An *equality-expression* `e1`

`!= e2` corresponds to the abstract syntax $e_1 \neq e_2$ where each subexpression is represented by its corresponding abstract syntax.

5. A *relational-expression* `e1 < e2` corresponds to the abstract syntax $e_1 < e_2$ where each subexpression is represented by its corresponding abstract syntax. A *relational-expression* `e1 > e2` corresponds to the abstract syntax $e_1 > e_2$ where each subexpression is represented by its corresponding abstract syntax. A *relational-expression* `e1 <= e2` corresponds to the abstract syntax $e_1 \leq e_2$ where each subexpression is represented by its corresponding abstract syntax. A *relational-expression* `e1 >= e2` corresponds to the abstract syntax $e_1 \geq e_2$ where each subexpression is represented by its corresponding abstract syntax.

6. An *additive-expression* `e1 + e2` corresponds to the abstract syntax $e_1 + e_2$ where each subexpression is represented by its corresponding abstract syntax. An *additive-expression* `e1 - e2` corresponds to the abstract syntax $e_1 - e_2$ where each subexpression is represented by its corresponding abstract syntax.

7. An *multiplicative-expression* `e1 * e2` corresponds to the abstract syntax $e_1 * e_2$ where each subexpression is represented by its corresponding abstract syntax. An *multiplicative-expression* `e1 / e2` corresponds to the abstract syntax e_1 **div** e_2 where each subexpression is represented by its corresponding abstract syntax. An *multiplicative-expression* `e1 % e2` corresponds to the abstract syntax e_1 **rem** e_2 where each subexpression is represented by its corresponding abstract syntax.

8. A *unary-expression* `!e1` corresponds to the abstract syntax **not** e_1 where the subexpression is represented by its corresponding abstract syntax. A *unary-expression* `-e1` corresponds to the abstract syntax $-e_1$ where the subexpression is represented by its corresponding abstract syntax.

9. A *boolean-literal* `true` represent the expressions **true** and **false** respectively.

10. An *integer-literal* represents the integer value determined by its spelling. For example, the literal `42` represents the integer value forty-two.

11. The expression `(e)` corresponds to the abstract syntax e. That is, applying parentheses to a well-formed expression affects neither the type nor its value.

In your parser, define a function for each non-terminal. Each production should return its corresponding abstract syntax tree. If the production does not match a sequence of tokens, or if a corresponding abstract syntax tree cannot be formed, the input is ill-formed, and you should diagnose the error. Note that throwing an uncaught exception is perfectly reasonable for now.

5 Requirements

Finish the implementation of your calculator by implementing a parser for the grammar above. The parser should accept a sequence of tokens produced by the lexer from your previous assignment, and produce and evaluate the abstract syntax trees you implemented in your first assignment.

The required behavior of your calculator is described above. It is strongly suggested that you read input one line at a time for e.g., `std::cin`. This will allow you to use input redirection to evaluate entire files.

In addition to the implementation, write a brief overview of your implementation. This should include a description of the major components (classes) in your implementation and their responsibilities. Include instructions for building and running the calculator, and document any known bugs.

Submission: Submit your printed homework on the due date. Send me an email with a link to your online source code. If you did not create a new repository, I will use your old repository.

Above and beyond: Extend the language and grammar to include the bitwise operators and, inclusive or, exclusive or, and complement for both boolean and integer values.

Extend the lexical structure of the language to include binary and hexadecimal integer literals. Allow the user to modify the output of the calculator to print results in binary, decimal, or hexadecimal.

Modify the lexer to handle comments. For this language, a comment would start with a `#` character and consist of all characters up to the end of the line. Comments are typically removed from the input text prior to lexical analysis.

Support simple variables. For example:

```
x = 2 + 4 * 5
T = true
(x > 20) == T
```

The output for a line containing a variable should be its initializer (the value computed from the expression after the '=' token. When a variable is used in an expression (as a *primary-expression*), its type and value are determined by the expression and initializer. For example, in `(x > 20) == T`, `x` has type **int** and value 22, and `T` has type **bool** and value $\top$.