Homework 1

Due: Tue, Feb 14, 2017

In this assignment, you will be creating an internal language representation for a small expression language.

1 Language

The language has the following kinds of types:

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

The type **bool** describes the values true ($\top$) and false ($\bot$). The type **int** describes integer values in the left-open range $[-2^{32-1}, 2^{32-1})$.

The language has the following expressions.

e ::=	**true**	
	false	
	n	integer values
	e_1 **and** e_2	and
	e_1 **or** e_2	inclusive or
	not e_1	logical negation
	if e_1 **then** e_2 **else** e_3	conditional
	$e_1 = e_2$	equal to
	$e_1 \neq e_2$	not equal to

$e_1 < e_2$	less than
$e_1 > e_2$	greater than
$e_1 \leq e_2$	less than or equal to
$e_1 \geq e_2$	greater than or equal to
$e_1 + e_2$	addition
$e_1 - e_2$	subtraction
$e_1 * e_2$	multiplication
e_1 **div** e_2	integer division
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.

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

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

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.

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.

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.

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.

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.

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.

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 with each line containing a single *expression*. Note that the source code may be provided directly through user input (i.e. `std::cin`).

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*.

2.1 Lexical conventions

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 type of a *boolean-literal* is **bool**.

The integer literals are:

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

The type of an *integer-literal* is **int**.

3 Syntactic conventions

The concrete syntax of the language is defined by the following grammar. Each production defines a sequence tokens defining a string of the language. The meaning of each production is defined in terms of a correspondence with the abstract syntax above. If the type requirements of corresponding abstract syntax are not satisfied, the program is ill-formed.

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

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.

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.

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.

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.

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.

TODO: Finish defining the translation.

The *boolean-literal*s `true` represent the expressions **true** and **false** respectively.

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

An expression `(e)` corresponds to the abstract syntax e.

4 Requirements

Implement a lexer for the grammar specified in the lexical conventions above. You do not need to implement a parser for this project.

Implement a calculator program that accepts lines of text from standard input, and performs lexical analysis on each line. Empty lines should be ignored. Print the names and and attributes of each token lexed from the input.

Note that the calculator should be able to accept multiple lines of input through redirection. This will help make testing easier.

Submission: Submit your printed homework on the due date. Send me an email with a link to your online source code.

Above and beyond:

Extend the language 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.