COMP2011 PA2 Polish Notation Interpreter

Introduction

In this assignment, we are going to make a simple interpreter for arithmetic expressions in Polish notation (https://en.wikipedia.org/wiki/Polish_notation), which reads the arithmetic expressions and computes the results. Besides the five common arithmetic operators, we also include an additional let-binding construct.

About Academic Integrity

We highly value academic integrity. Please read the Honor Code (../../web/code.html) section on our course webpage to make sure you understand what is considered as plagiarism and what the penalties are. The following are some of the highlights:

- **DO NOT try your "LUCK"** - we use sophisticated plagiarism detection software to find cheaters. We also review codes for potential cases manually.
- The penalty (for BOTH the copier and the copiee) is not just getting a zero in your assignment. Please read the Honor Code thoroughly.
- Serious offenders will fail the course immediately, and there may be additional disciplinary actions from the department and university including expulsion.

This programming assignment is challenging, so you are highly recommended to start early. If you need clarification on the requirements, please post your questions on Piazza after reading more questions?. However, to avoid cluttering the forum with repeated/trivial questions, please carefully read the given code, the description, the sample output, and the latest FAQ (refresh this page regularly) in this page before you post your questions. In addition, please be reminded that we won't help any student debug their assignment for the sake of fairness.

Terminology

Throughout the description, the terms "expression" and "program" are used repeatedly. The two terms always refers to the constructs in our Polish notation language. "Program" refers to the full expression given as input to the interpreter.

Do not confuse them with C++ expressions and C++ programs. The description has been worded to avoid references to C++ expressions and C++ programs outside of this paragraph.

Polish notation

The usual notation used for arithmetic expressions in mathematics is known as the infix notation, where the operator is placed in between the two operands. The infix notation only accommodates binary operators and requires the use of parenthesis for specifying the order for parsing. For example, the expression `2 + 3 * 5` is usually taken to mean `2 + (3 * 5)` by the conventional precedence of the operators. If we want to express the operation of multiplying the sum of 2 and 3 with 5, we must insert parentheses as `(2 + 3) * 5`.

The *Polish notation* solves this problem by placing the operator before its operands. The resulting expression can only correspond to a single parse tree, as long as the operators have fixed arities, i.e. fixed number of operands. For example, the infix expression `2 + (3 * 5)` is expressed in Polish notation as `+ 2 * 3 5`, which is equivalent to the parenthesized version `+ 2 (* 3 5)`; and `(2 + 3) * 5` as `* + 2 3 5`, which is equivalent to `* (+ 2 3) 5`.

To help you understand expressions in Polish notation, we can visualize them as their parse trees. A parse tree is a hierarchical representation of the structure of an expression. It shows the operands and the operators, and the relationships between them. For example, the expressions `+ 2 * 3 5` and `* + 2 3 5` correspond to the following parse trees respectively:

```
+               *
├─ 2            ├─ +
└─ *            │  ├─ 2
   ├─ 3         │  └─ 3
   └─ 5         └─ 5
```

The parse tree of `+ 2 * 3 5` has root node `+`, which has 2 child nodes, `2` and `* 3 5`. `*` is the root node of the subexpression `* 3 5`, which is a subtree. Similarly, the parse tree of `* + 2 3 5` has root node `*`, which has two child nodes, `+ 2 3` and `5`.

Notice that every subtree corresponds to a subexpression, and a subtree is rooted at every node, so we can refer to subexpressions with the nodes at which they are rooted. For example, here are a few subexpressions at different positions in the expression `* + 2 3 5`:

```
* + 2 3 5
↑ ↑ ↑
| | └ expr at pos 2: 2
| └── expr at pos 1: + 2 3
└──── expr at pos 0: * + 2 3 5
```

For simplicity, we represent numbers in our language as integers, and use `ADD`, `SUB`, `MUL`, `DIV`, and `MOD` instead of their symbols to denote the operators in Polish notation.

A reference implementation for the evaluation of our system is provided below. You can copy-and-paste any examples in the rest of this section or type in your own program.

Input your program:

Evaluate

Result:

Numbers evaluate to themselves. In order to evaluate an operator expression, evaluate the two expressions following the operator, then perform the corresponding operation on the results.

As an example, let's look at how the program above would evaluate. `MUL ADD 2 3 5`, which is equivalent to `MUL (ADD 2 3) 5`, has two expressions, namely `ADD 2 3` and `5`, following the operator `MUL`. The first expression, `ADD 2 3`, has two expressions, namely `2` and `3`, following the operator `ADD`. The expressions `2` and `3`, are evaluated to `2` and `3` respectively. Therefore, `ADD 2 3` is evaluated to `5`. Similarly, the second expression, `5`, is evaluated to `5`. Finally, `MUL ADD 2 3 5` is evaluated as `MUL 5 5` which is evaluated to `25`. The process of evaluation can be represented as a diagram called a stack trace (*optional*, see FAQ for more information).

```
[] MUL ADD 2 3 5
|  ^
├─ [] MUL ADD 2 3 5
|  |       ^
|  ├─ [] MUL ADD 2 3 5
|  |  |           ^
|  |  => 2
|  ├─ [] MUL ADD 2 3 5
|  |  |             ^
|  |  => 3
|  => 5
├─ [] MUL ADD 2 3 5
|  |               ^
|  => 5
=> 25
```

In the above stack trace, each branch (├─) represents a recursive call to the evaluation function, and the corresponding return value is denoted by an arrow (=>).

Besides numbers and the usual arithmetic operators, we also add to our language two additional constructs: variables and let-bindings. A variable must begin with a lower-case alphabet, followed by zero or more (upper or lower-case) alphabets, digits, or underscores. The *let-binding* construct has syntax `LET Var Expr₁ Expr₂`. When evaluating a let-binding, the expression `Expr₁` is first evaluated. Then the result is bound to the variable `Var`, and the expression `Expr₂` is evaluated. That is, any occurrence of the variable `Var` within `Expr₂` would evaluate to the result of `Expr₁`. The full let-binding then evaluates to the result of the evaluating `Expr₂`.

`Expr₁` and `Expr₂` are refered to as the *value expression* and the *body expression* respectively.

For example, the following expression, which is equivalent to `LET a 42 (ADD a 1)`, evaluates to 43:

```
[] LET a 42 ADD a 1
|  ^
├─ [] LET a 42 ADD a 1
|  |        ^
|  => 42
├─ [a↦42] LET a 42 ADD a 1
|  |               ^
|  ├─ [a↦42] LET a 42 ADD a 1
|  |  |                  ^
|  |  => 42
|  ├─ [a↦42] LET a 42 ADD a 1
|  |  |                    ^
|  |  => 1
|  => 43
=> 43
```

For the expression `LET a 42 ADD a 1`, `Expr₁` is `42` and is evaluated as `42` itself. `Expr₂` is `ADD a 1`, where the subexpressions, namely `a` and `1`, are evaluated to `42` and `1` respectively. Therefore, `ADD a 1` is evaluated to 43, which is the result of the evaluation of `LET a 42 ADD a 1`.

Let-bindings can be nested in their body expressions:

```
  LET a 2 LET a2 MUL a a LET a4 MUL a2 a2 MUL a2 a4
≡ LET a 2 (LET a2 (MUL a a) (LET a4 (MUL a2 a2) (MUL a2 a4)))
```

▶ Click here to see the stack trace

Or nested in the value expressions:

```
  LET a4 LET a2 LET a 2 MUL a a MUL a2 a2 MUL a4 a4
≡ LET a4 (LET a2 (LET a 2 (MUL a a)) (MUL a2 a2)) (MUL a4 a4)
```

▶ Click here to see the stack trace

Or both:

```
  LET a2 LET a 2 MUL a a LET a4 MUL a2 a2 MUL a2 a4
≡ LET a2 (LET a 2 (MUL a a)) (LET a4 (MUL a2 a2) (MUL a2 a4))
```

▶ Click here to see the stack trace

The let-binding construct is an expression, so it can be used as an operand to an arithmetic operator:

```
  ADD LET a 2 MUL a a 2
≡ ADD (LET a 2 (MUL a a)) 2
```

▶ Click here to see the stack trace

Note that let-bindings are scoped, so the following would result in an error:

```
  ADD LET a 2 MUL a a a
≡ ADD (LET a 2 (MUL a a)) a
           ↑              ↑
           └─ different scope
```

▶ Click here to see the stack trace

Let-bindings are shadowing, so the following evaluates to 15:

```
  LET a 2 LET b 3 LET a 5 MUL a b
≡ LET a 2 (LET b 3 (LET a 5 (MUL a b)))
```

▶ Click here to see the stack trace

The grammar of our language can be summarized as follows:

- A number is a sequence of one or more digits, optionally preceded by a minus sign
- A variable is a lower-case alphabet followed by zero or more alphabets, digits, or underscores
- An expression is either one of the following:
  - A number

- A variable
- A binary operator (`ADD` , `SUB` , `MUL` , `DIV` , `MOD`) followed by two expressions,
- `LET` followed by a variable, then by two expressions

While `LET` is not strictly an operator, for consistency, any reference to operators in the context of our language will hereon include `LET` .

Skeleton code

Download the skeleton code here: pa2_skeleton.zip (pa2_skeleton.zip)

The skeleton code contains the following files:

- **`interpreter.cpp`** : This is the only file you will submit. You should not write or modify any other file.
- **`raise.cpp`** : Utility function for error handling
- **`num.cpp`** : Utility function to convert C strings to integers
- **`interpreter.h`** , **`raise.h`** , **`num.h`** : Constant and data type definitions and function declarations for the corresponding sourse files
- **`main.cpp`** : Driver for the interpreter, handles input and output
- **`test.cpp`** : Given test cases for this assignment
- **`test.log`** : Sample output for the given test cases

For the source files, you only need to read `interpreter.cpp` , `interpreter.h` , `main.cpp` , and `test.cpp` . You do not have to read or understand `raise.cpp` , `raise.h` , `num.cpp` , and `num.h` .

Implementation

Restrictions

The following restrictions apply to your submission:

- The use of include directives other than the ones already included in the skeleton is not allowed
- The use of loops is not allowed
- The use of goto statements is not allowed
- Any use of the `static` keyword is not allowed
- The use of the `auto` placeholder type specifier is not allowed
- The use of global variables is not allowed

All of the above restrictions, except for the one on the use of global variables, will be checked on ZINC before compilation. ZINC will refuse to compile your submission if any of them is violated. The use of global variables will be checked after submission has been closed, and marks will be deducted if they are found.

On the other hand, the following are allowed:

- You are free to define any helper functions (and likely need to define some). Note that while functions defined in other translation units (e.g. raise.cpp, num.cpp) can be called provided that the appropriate headers are included (e.g. raise.h, num.h). Any helper functions you define are only visible within the same translation unit (i.e. interpreter.cpp).
- You are free to define any global constants. Similar to helper function, they are only visible within the same translation unit.
- While you are free to use dynamic memory, you are advised to avoid doing so as it is not necessary for completing this assignment. The extra tools used for grading checks for memory leaks as a kind of memory error, so those who use dynamic memory allocation may run the risk of point deduction.

Error handling

The header file `raise.h` defines an enumeration `Err`, which represents different types of errors that can arise during program evaluation. Their names are self-explanatory. The types of errors can be roughly divided into three kinds:

- Input errors: `tok_len_exceeded`, `prog_len_exceeded`
- Parsing errors: `invalid_op`, `invalid_num`, `invalid_ident`, `invalid_tok`
- Runtime errors: `ctx_overflow`, `num_overflow`, `num_div_by_0`, `unknown_ident`, `eof_reached`, `prog_not_consumed`

The function `void raise(Err)` takes a type error, outputs a statement, then immediately terminates execution. For example, you can use `raise(invalid_op);` to raise the error `invalid_op`. You will be told which error to raise under what circumstances.

Task 1: string manipulation

Since implementing the interpreter involves dealing with strings, we are first going to implement a few utility functions for the manipulation of C strings (i.e. null-terminated byte strings). Your task is to implement the following functions in `interpreter.cpp`.

eq

```
bool eq(const char lhs[], const char rhs[]);
```

Compares two C strings for equality.

Returns `true` if `lhs` and `rhs` compare equal, i.e. they have the same length and each character in `lhs` is the same as the character in `rhs` at the same position, `false` otherwise.

You should implement the following recursive helper function, which will not be directly tested. Your implementation of `eq`, which does not have to be itself recursive, should call the helper function with the appropriate arguments.

```
bool eq_impl(const char lhs[], const char rhs[], unsigned int idx);
```

`copy`

```
void copy(const char input[], char output[]);
```

Copies the C string `input`, including the null terminator, to the character array `output`. The behavior is undefined (i.e. you do not need to consider this case) if the `output` array is not large enough.

You should implement the following recursive helper function, which will not be directly tested. Your implementation of `copy`, which does not have to be itself recursive, should call the helper function with the appropriate arguments.

```
void copy_impl(const char input[], char output[], unsigned int idx);
```

Task 2: context manipulation

In order to evaluate let-binding constructs, we choose to store the association from variables to values in a *context*. The context has two modifying operations: append and remove. The *append* operation appends a new association from a variable to a value at the end of the context. The *remove* operation removes the last association from the context. The non-modifying operation *lookup* looks up the value associated with a particular variable in the context.

The context is represented as two arrays: an array of C strings `ctx_k` for the variables, and an array of numbers `ctx_v` for the values. Each value in `ctx_v` is then associated with the variable at the same position of `ctx_k`. Similar to how a C string is terminated by a null character, the context is terminated by an entry with the empty string in place of the variable. The terminaton of `ctx_v` does not have to be marked since it can be determined from `ctx_k`.

Your task is to implement the following functions in `interpreter.cpp`.

ctx_append

```
void ctx_append(const char ident[], int val, char ctx_k[][MAX_IDENT_LEN], int c
```

Binds the variable `ident` to the value `val` in the context represented by `ctx_k` and `ctx_v`, which have sizes `size`. Raises `ctx_overflow` if the arrays `ctx_k` and `ctx_v` are not large enough.

Do this by placing `ident` and `val` at the next available empty slot of `ctx_k` and `ctx_v` respectively.

You should implement the following recursive helper function, which will not be directly tested. Your implementation of `ctx_append`, which does not have to be itself recursive, should call the helper function with the appropriate arguments.

```
void ctx_append_impl(const char ident[], int val, char ctx_k[][MAX_IDENT_LEN],
```

ctx_remove

```
void ctx_remove(char ctx_k[][MAX_IDENT_LEN], int ctx_v[]);
```

Unbind the latest binding in the context represented by `ctx_k` and `ctx_v`. The behavior is undefined if the context is empty.

Do this by removing the last association from `ctx_k`.

You should implement the following recursive helper function, which will not be directly tested. Your implementation of `ctx_remove`, which does not have to be itself recursive, should call the helper function with the appropriate arguments.

```
void ctx_remove_impl(char ctx_k[][MAX_IDENT_LEN], int ctx_v[], unsigned int idx
```

ctx_lookup

```
int ctx_lookup(const char ident[], const char ctx_k[][MAX_IDENT_LEN], const int
```

Looks up and returns the value associated with the latest binding of the variable `ident` from the context represented by `ctx_k` and `ctx_v`. Raises `unknown_ident` if the variable is not found in the context.

Do this by returning the value in `ctx_v` at the same position as the last occurrence of `ident` in `ctx_k`.

You should implement the following recursive helper function, which will not be directly tested. Your implementation of `ctx_lookup`, which does not have to be itself recursive, should call the helper function with the appropriate arguments.

```
int ctx_lookup_impl(const char ident[], const char ctx_k[][MAX_IDENT_LEN], cons
```

Examples

We will illustrate the operations on contexts with a few examples. We only show the meaningful array elements at the beginning of `ctx_k` and `ctx_v`. The array elements after the termination marker are irrelevant.

The following represents an empty context:

```
ctx_k: { "" }
ctx_v: { }
```

Suppose the context originally had bindings `a↦42` and `b↦43`, appending a new binding `c↦44` modifies the arrays as follows:

```
Before:     ctx_k: { "a", "b", "" }
            ctx_v: { 42,  43 }

Operation:  ctx_append("c", 44, ctx_k, ctx_v, 20);

After:      ctx_k: { "a", "b", "c", "" }
            ctx_v: { 42,  43,  44 }
```

Suppose the context originally had bindings `a↦42`, `b↦43`, and `c↦44`, removing a binding modifies the arrays as follows:

```
Before:     ctx_k: { "a", "b", "c", "" }
            ctx_v: { 42,  43,  44 }

Operation:  ctx_remove(ctx_k, ctx_v);

After:      ctx_k: { "a", "b", "" }
            ctx_v: { 42,  43 }
```

Suppose the context has bindings `a↦42`, `b↦43`, `a↦44`, and `c↦45`, where the earlier binding `a↦42` is shadowed by the later binding `a↦44`, looking up `a` would return `44`, which is the value associated with the last occurrence of `a`.

```
Before:       ctx_k: { "a", "b", "a", "c", "" }
              ctx_v: { 42,  43,  44,  45 }

Operation:    ctx_lookup("a", ctx_k, ctx_v);
Return value: 44
```

Task 3: interpreter

Now we are ready to put the pieces together and implement the evaluation function of our interpreter. *Tokens* are the space-delimited "words" that make up a program, i.e. numbers, variables, and operators. The input program to the evaluation function is represented as an array of C strings, terminated by an empty string.

Your task is to implement the following functions in `interpreter.cpp`.

`eval_impl`

```
int eval_impl(char prog[][MAX_TOKEN_LEN], unsigned int& cur, char ctx_k[][MAX_I
```

Recursively evaluates the subexpression of the program `prog` at position `cur` under the context `ctx_k` and `ctx_v` and returns the result. Advances `cur` to the position just after the end of the subexpression. Assume `ctx_k` and `ctx_v` always have size `MAX_CTX_LEN`.

For example, if `prog` is `MUL ADD 2 3 5` and the value of `cur` is 1, then the value of `cur` after the call to `eval_impl` will be `4`, since the subexpression at position 1 is `ADD 2 3`.

Implement this by branching on the first character of the token at `cur`:

- If the first character is a digit or the minus sign, treat this token as a number and return the value. Use the function `to_num` to interpret the string representation as a number. For example, if we have `const char s[] = "42";`, `to_num(s)` will return the integer 42. `to_num` will raise `invalid_num` if the input is not a valid number.
- If the first character is a lower-case alphabet, treat this token as a variable and return the value associated with it in its latest binding. Raise `invalid_ident` if the token is not a valid variable, and `unknown_ident` if the variable is not found in the context.
- If the first character is an upper-case alphabet, treat this token as an operator, evaluate the subexpression at this position and return the result. Raise `invalid_op` if the token is not a valid operator.
  - If the operator is `DIV` or `MOD`, raise `num_div_by_0` if the divisor is zero.
  - If the operator is `LET`, the next token is not evaluated. Raise `invalid_ident` if the next token is not a valid variable. Raise `eof_reached` if there is no next token. If the next token is a valid variable, evaluate the second argument, bind the result to the variable in the context using `ctx_append`, evaluate the third argument,

unbind the binding using `ctx_remove`, and finally return the result of the evaluation of the third argument.

- If the first character is anything else, raise `invalid_tok`.
- If the end of the program has been reached, raise `eof_reached`.

`eval`

```
int eval(char prog[][MAX_TOKEN_LEN]);
```

Evaluates the program `prog` in an empty context and returns the result. Raises `prog_not_consumed` if the program is not fully consumed.

Your implementation of `eval` should call `eval_impl` with the appropriate arguments. `eval` itself does not have to be recursive.

Grading

Your submission will be graded on ZINC, using 20 given test cases and 37 hidden test cases.

While the test cases are themselves independent, your implementation of the later tasks will very likely call functions implemented in the the earlier tasks, thus coupling the test cases for the later tasks with the earlier tasks. We will not provide a reference implementation of the earlier tasks while grading the later tasks, but use the implementation in your submission.

The following table summarizes the test cases for each task:

Focus function	Given test cases	Hidden test cases	Marks for each test	Total marks
`eq`	2	5	2 marks	14 marks
`copy`	1	3	2 marks	8 marks
`ctx_append`	2	3	2 marks	10 marks
`ctx_remove`	1	3	2 marks	8 marks
`ctx_lookup`	3	3	2 marks	12 marks
`eval_impl`	9	19	1.5 marks	42 marks
`eval`	2	1	2 marks	6 marks
Total	**20**	**37**		**100 marks**

Before compiling your submission, ZINC will perform the checks for the restrictions mentioned above.

Note that after submission due date, we will also perform the checks for memory errors. If there are uses of uninitialized values, out-of-bounds array accesses, or if dynamic memory allocation is used and it results in memory errors, marks will be deducted.

Hidden test cases

The complete set of test cases can be downloaded here: test.cpp (test.cpp)

The focus area of the tests are separated by comments in the source file.

In addition to running the test cases normally, your submission will be graded using two additional tools: valgrind and sanitizer. Valgrind catches memory errors and uses of uninitialized values, while sanitizer catches memory errors and undefined behaviors.

Producing the correct output would give you 50% of the score for each of the test cases. Passing the checks by valgrind or sanitizer while producing the correct output would give you 25% respectively. For example, if a test case carries two marks, and you produce the correct output, pass the sanitizer checks, but fail the valgrind checks, you would be given 1.5 marks. If you pass the checks but produce the wrong output, you would not be given any marks.

Test cases with sanitizer and valgrind enabled are numbered 101–157 and 201–257 respectively on ZINC. They are otherwise identical to the normal tests numbered 1–57.

Solution

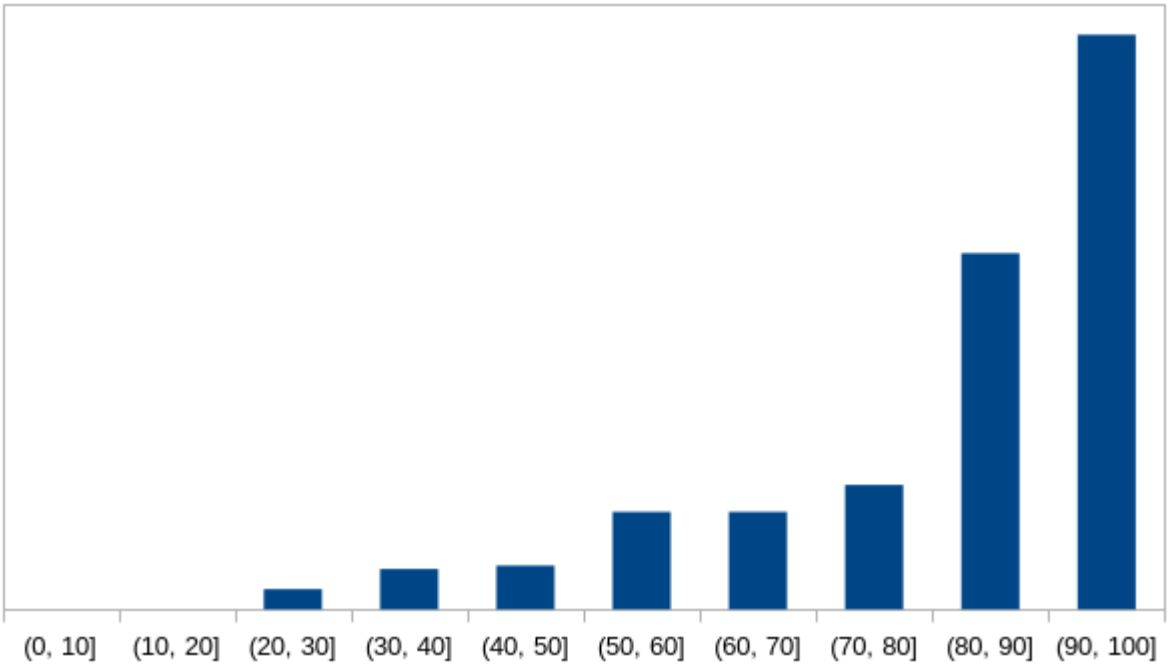
The solution can be downloaded here: interpreter.cpp (interpreter.cpp)

Note that the solution is merely one way to implement the functions. There are many alternative approaches for completing the tasks, especially for `ctx_lookup`.

Statistics

The following are statistics before applying any penalties, excluding zeros:

- Mean: 81.21
- Median: 88
- SD: 17.56



Appeal

To appeal for the grading, send an email to cychauab@cse.ust.hk (mailto:cychauab@cse.ust.hk?subject=COMP2011%20PA2%20Appeal), with subject **COMP2011 PA2 Appeal**. The appeal deadline is **3rd May 2024 23:59**.

Compilation

You can compile your code using the following commands. Note that the checks for the restrictions mentioned above are not performed. ZINC may refuse to compile your submission even if it compiles successfully on your environment.

```
g++ -std=c++11 -pedantic-errors -Wall -Wextra -Wno-parentheses -g -o main raise
g++ -std=c++11 -pedantic-errors -Wall -Wextra -Wno-parentheses -g -o test raise
```

In addition to the `-std=c++11` option for specifying the C++11 standard, the above commands pass a few extra options to g++:

- `-pedantic-errors` tells g++ to conform to the ISO standard and issue an error when a forbidden construct in present in the code, and in some cases when the code does not follow the ISO standard
- `-Wall -Wextra` enables all the warnings about constructions that some users consider questionable and likely correspond to bugs
- `-Wno-parentheses` disables the warning that suggests parentheses around `&&` within `||`
- `-g` tells g++ to produce debugging information

If you prefer to ignore the warning messages generated by g++, you are free to remove the extra options. However, you are advised to keep the `-pedantic-errors` option since it will be enabled on ZINC.

```
g++ -std=c++11 -pedantic-errors -g -o main raise.cpp num.cpp interpreter.cpp ma
g++ -std=c++11 -pedantic-errors -g -o test raise.cpp num.cpp interpreter.cpp te
```

After successful compilation, you can run the executable `main`, input a program, and hit enter. The result after evaluation will then be output.

To test your code using the given test cases, run the executable `test`. The sample output is given in `test.log`.

Submission

Deadline: ~~13th April 2024 23:59~~ 14th April 2024 23:59

Submit your solution to ZINC (http://zinc.cse.ust.hk/). Submit your `interpreter.cpp` as a single file. Ensure the filename is `interpreter.cpp`. Do not put the file in an archive.

Filename checking is enabled in ZINC. ZINC will refuse to compile your submission if the filename does not match.

Note:

- You may submit your file multiple times, but **only the last submission will be graded**. You do not get to choose which version we grade. If you submit after the deadline, late penalty will be applied according to the submission time of your last submission.
- Submit early to avoid any last-minute problem. Only ZINC submissions will be accepted.
- The ZINC server will be very busy on the last day especially in the last few hours. However, as long as your submission is successful, we would grade your latest submission with all test cases after the deadline.
- Make sure you submit the correct file yourself. You can download your own file back from ZINC to verify. Again, we only grade what you uploaded last to ZINC.

Compilation Requirements

It is required that your submissions can be compiled and run successfully on our online auto-grader ZINC. If we cannot even compile your work, it won't be graded. Therefore, for parts that you cannot finish, you are advised to put in dummy/empty implementation so that your whole submission can be compiled for ZINC to grade the other parts that you have done.

Late Submission Policy

There will be a penalty of -1 point (out of a maximum 100 points) for every minute you are late.

If you believe that you have special reasons to justify an extension, contact your instructor (not one of the TAs) as soon as possible and before the deadline. Requests raised after the deadline will typically not be granted.

FAQ

This section will be updated after the release

- **My code doesn't work / there is an error, here is the code, can you help me fix it?**

  As the assignment is a major course assessment, to be fair, we would not finish the tasks for you. We might provide you some hints, but we won't debug for you.

- **My submission gives the correct output on my computer, but it gives a different one on ZINC. What may be the cause?**

  Usually inconsistent strange result (on different machines/platforms, or even different runs on the same machine) is due to relying on uninitialized hence garbage values, missing return statements, accessing out-of-bound array elements, improper use of dynamic memory, or relying on library functions that might be implemented differently on different platforms (such as pow() in cmath).
  In this particular assignment, it is probably related to accessing out-of-bound array elements.

- **I don't understand the stack trace in Polish notation!**

  The stack traces are given as supplementary materials. They provide a visualization of the evaluation of some examples to help you understand the system and validate your understanding. All the information is already included in the text. You do not need to understand them to complete the required tasks. If they end up confusing you more, you can forget about them.

- **I know how to look up the first occurrence but I have no idea on how to look up the last, can I get some help?**

  Coming up with a way to perform this task is part of the assessment for this assignment. When restricted to recursion only, many problems can be solved by writing more helper functions.
  Please refrain from asking for the whole algorithm on Piazza, and please do not answer such questions. See more questions? below.

If you cannot come up with a solution for the last occurrence, you can put down the solution for the first occurrence. Shadowing is only present in a limited number of test cases.

- **Which error should I raise when there are multiple errors?** (@257 (https://piazza.com/class/lru9p0kwzvb7ee/post/257))

  You should raise the first error you come across, and shouldn't try to read ahead to try to find other errors. For example, when evaluating the expression `LET x' 42`, the first error the interpreter comes across is `invalid_ident`. Even though LET is missing an argument, `eof_reached` would not be raised because another error has been raised before.

- **What is the difference between `invalid_ident` and `unknown_ident`?** (@259 (https://piazza.com/class/lru9p0kwzvb7ee/post/259))

  `invalid_ident` should be raised when a variable is expected but the token does not satisfy the conditions in Polish notation.

  > A variable must begin with a lower-case alphabet, followed by zero or more (upper or lower-case) alphabets, digits, or underscores

  `unknown_ident` should be raised when the variable is not found in the context.

- **How should I remove the last element from `ctx_v` in `ctx_remove`?** (@229 (https://piazza.com/class/lru9p0kwzvb7ee/post/229))

  You cannot "remove" an element from an array. If you read carefully the description above the examples:

  > We only show the meaningful array elements at the beginning of `ctx_k` and `ctx_v`. The array elements after the termination marker are irrelevant.

  There is always an array element left in `ctx_v` after removal. The description of `ctx_remove` only requires you to remove the *association*, not the actual array element. As long as `ctx_lookup` does not think there is an association from some variable to the value in `ctx_v`, it can safely remain unchanged.

- **How can I identify the end of the arrays `ctx_k` and `ctx_v` when the context is full?** (@331 (https://piazza.com/class/lru9p0kwzvb7ee/post/331))

  Using the word "full" may lead to confusion. You may have misunderstood how many associations can fit in the context. `ctx_k` should always be terminated by the termination marker before and after the functions `ctx_append`, `ctx_remove`, and `ctx_lookup` are called.

- **My ZINC tests always fail with non-zero exit code: 139(SIGSEGV). What does it mean?** (@287 (https://piazza.com/class/lru9p0kwzvb7ee/post/287))

  Most likely in PA2, this is caused by attempting to access an element at an index that is outside the bounds of the array.

  Sometimes this kind of illegal access operation would not cause the program to crash — it just evaluates to the "garbage value" at that address (similar to accessing an uninitialized variable) — but sometimes, the operating system would detect this error, terminating your program and result in a non-zero exit code `SIGSEGV` (stands for Signal Segmentation Violation).

  So, next time if you see `SIGSEGV` on Zinc, you shall know that you accessed an illegal memory address.

More questions?

We anticipate a high volume of questions to be posted on Piazza for this assignment. In order to ensure that we can effectively handle your questions, you are advised to follow the following guidelines when posting your questions:

- Be polite
  Treat each other with courtesy even when interacting online. Questions like @121 (https://piazza.com/class/lru9p0kwzvb7ee/post/121) will not be answered.
- Be specific
  State clearly and precisely your question, what you have trouble understanding, and what you expect to be answered. Do not expect us to write an essay to explain everything by just quoting a large amount of material, followed by a question with no focus like "need some help on this" or "can someone what's going on here". It is impossible to provide a helpful answer without understanding what exact trouble you are having. If you do not put in the effort to ask a question, I will not put in the effort to answer especially when the effort is likely to be wasted anyways.
- Make sure your question is grammatically correct, or at least unambiguious
  While we understand that not everyone is a native speaker or is proficient in English, we expect the meaning of your communications to not be obscured by grammatical mistakes. It is sometimes impossible to decipher the meaning from statements with excessive grammatical mistakes.
- Post code properly
  Post lines of code using the "code block" feature in the editor, and inline code using the "teletype text" feature. Posting code as plain text makes reading difficult. Posting large amounts of code as images also makes it difficult for others to compile and test your code, or to post corrections to your code by copy-and-pasting.
- Avoid posting images
  In the vast majority of situations, posting an image is not necessary. If you need to

show some code or console output, use the “code block” feature in the editor. If you need to quote something from this description, use the “blockquote” feature in the editor. Posting images would very often result in bad formatting. In addition, images are sometimes posted as files instead of images, making reading difficult.

- Do not post photos
  Unless your computer is in a state where it cannot capture screenshots, for example when running your code results in a freeze, bluescreen, or kernel panic, do not post a photo. Posting photos of your screen makes reading extremely difficult, especially when the moire effect is present, and the photo is taken at an angle in an attempt to counter it.
- Do not quote ChatGPT, online tutorials, or online forums as your source
  ChatGPT can give wrong answers. Online tutorials are written by random people on the internet, and are wrong more often than not. Notable offenders include GeeksforGeeks and W3Schools. Online forums like Stack Overflow, while sometimes having correct and well-explained answers, also have no guarantee for their quality. Even with the voting system, the wrong answer can still be upvoted to the top. The online reference at cppreference.com (https://cppreference.com) has been found to be reliable.
- Do not expect us to debug your code
  As this assignment is a major course assessment, you are supposed to finish most of the tasks on your own. Expect significantly less help than what has been provided in the labs even if you come to the office hours in person.
- Do not ask directly for the whole algorithm
  Coming up with an algorithm to perform the desired task is a significant part of programming and learning, and it is also part of the assessment for this assignment. Given a description of the algorithm in natural language, ChatGPT would be able to translate it into code, and your remaining task will be nothing more than what a generative AI can perform. Any questions asking for the very detailed steps of how to perform a task in this assignment will be removed. Answering such questions by giving very detailed steps or very detailed codes will be considered plagiarism.
- **Do not post any code you have written for this assignment**
  Posting any code or code fragments for the tasks in this assignment is considered plagiarism, no matter how small or insignificant it may seem, even if it is a wrong solution. Any such postings will be removed, and serious offenders will be penalized in accordance to the Honor Code (../../web/code.html). This has occured already occured in this semester.
  This may not have been stated very clearly previously. However, we will enforce this very strictly as the assignment is a major course assessment.
  If you have questions about specific parts of your code, construct (https://stackoverflow.com/help/minimal-reproducible-example) a minimal reproducible example (https://en.wikipedia.org/wiki/Minimal_reproducible_example) with generic variable and function names and post that in the question instead.

If you believe that your question necessitates showing us the exact code, post it as a private question for instructors only.