generator

cmput415

Jun 11, 2025

CONTENTS

For this assignment you will be building a parser for a number generator. The generator will produce a series of numbers through an expression. Your task is to parse the generator statement, interpret it, and print the numbers that should be produced by the generator. You will be using *ANTLR4* to generate a **lexer** and **parser** for your interpreter. You will then implement the interpreter in *C++*. Documentation and tutorials for *ANTLR4* can be found here Antlr4 documentation.

CHAPTER

ONE

RESERVED KEYWORDS

The following keywords are reserved in *Generator*.

- `in`

CHAPTER

TWO

INTEGER LITERALS

In this assignment integer literals are defined as being a string that contains only the number characters 0-9 with no spaces.

Assertion: All integer literals will be ≥ 0. (*nonnegative-literals*)
Assertion: All integer literals will fit in 31 unsigned bits. (*literal-size*)

Examples of valid integers literals:

```
1
123
5234
01
10
```

Examples of invalid integers literals:

```
-1
1.0
one
1_1
1o
4294967296
```

CHAPTER

THREE

IDENTIFIERS

For the purpose of this assignment, identifiers are simple. They must start with an alphabetical character. This character may be followed by numbers or alphabetical characters. A keyword cannot be used as an identifier.

Examples of valid identifiers:

```
hello
h3llo
Hi
h3
```

Examples of invalid identifier:

```
in
3d
a-bad-variable-name
no@twitter
we.don't.like.punctuation
not_at_all
```

CHAPTER

FOUR

EXPRESSION

An expression is composed of integers, identifiers, and integer mathematical operations.

4.1 Operators

Operation	Symbol	Usage	Associativity
exponentiation	^	`expr ^ expr`	right
multiplication	*	`expr * expr`	left
division	/	`expr / expr`	left
remainder	%	`expr % expr`	left
addition	+	`expr + expr`	left
subtraction	-	`expr - expr`	left

Assertion: All exponents are ≥ 0. (*nonnegative-exp*)
Clarification: The `%` operator is remainder not modulus. (*rem-not-mod*)
Clarification: Division is integer division. (*int-div*)

4.2 Valid Expressions

Valid formats for expressions are

```
(<expr>)
<expr> <op> <expr>
<int>
<id>
```

- `expr` is an expression.
- `int` is an integer.
- `id` is the identifier of a variable.

Assertion: All expressions will result in a value that fits in a 32 bit signed integer. (*expression-size*)
Assertion: No expression will contain a division by 0. (*zero-divide*)

Examples of valid expressions are

```
i * 2 * 10 + 4
2 ^ 4 * 5
```

4.3 Precedence

Precedence determines what order operations are evaluated in. Precedence works as defined in the following table:

Precedence	Operations
HIGHER	^
	* / %
LOWER	+ -

The higher the precedence the sooner the value should be evaluated. For example, in the expression

```
1 + 2 * 3
```

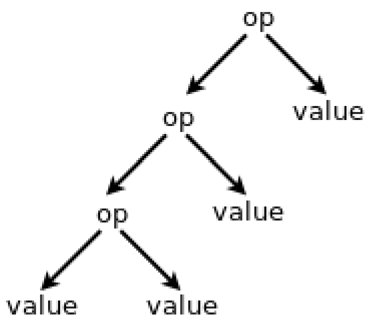
`2 * 3` should be evaluated before `1 + 2`. This is because multiplication, division, and remainder have higher precedence than addition and subtractions.

4.4 Associativity

When parsing expressions associativity determines in what order operators of the same precedence should be evaluated in. For example:

```
1 / 2 * 3
```

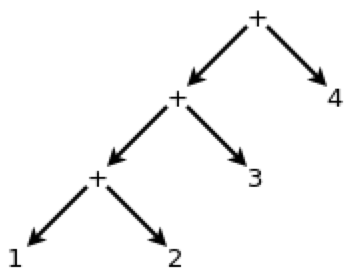
In this example both division and multiplication have the same precedence; associativity determines which operations are evaluated first. Left associative operations will form a parse tree like this:



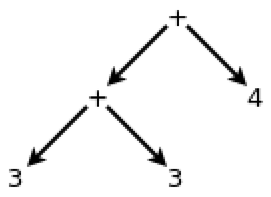
An example of one of these operations is addition. Lets say we have the following expression:

```
1 + 2 + 3 + 4
```

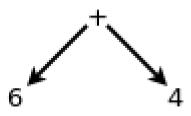
Because addition is left associative it will form the following parse tree:



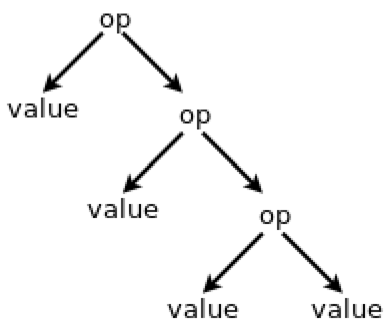
An operation in such a parse tree can only be evaluated when all the operands are leaves. Thus, in this parse tree, the expression `1 + 2` is evaluated first and then the result of this evaluation replaces the subtree for the expression `1 + 2` to create the following tree.



Next, the expression `3 + 3` is evaluated, making



Most operations used in this assignment are left associative, but there are also operations that are right associative and take this format:

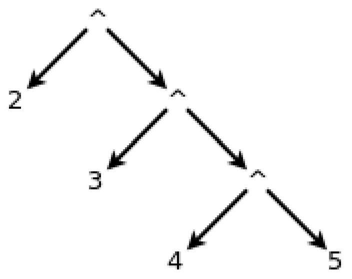


An example of a right-associative operation is the exponentiation operation represented by the symbol `^`. For example

```
2 ^ 3 ^ 4 ^ 5
```

should be evaluated as: $2^{3^{4^5}}$

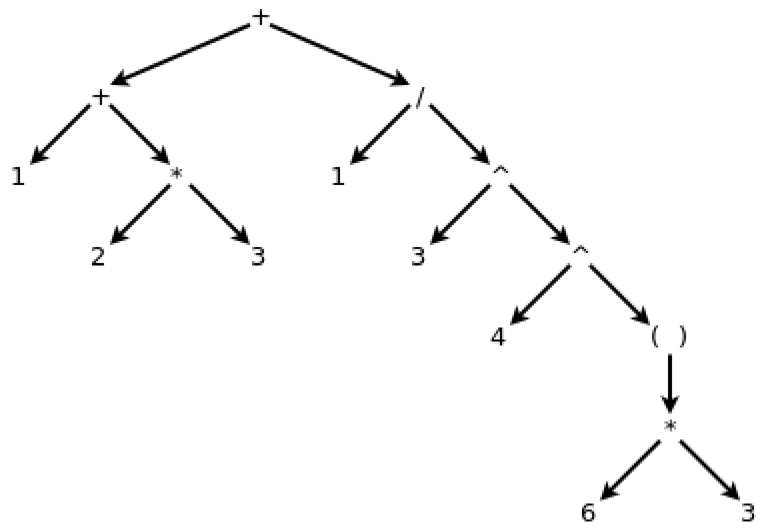
In order for this expression to be evaluated correctly the following parse tree must be generated



For a more complex example consider the expression:

```
1 + 2 * 3 + 1 / 3 ^ 4 ^ (6 * 3)
```

which generates the following parse tree:



CHAPTER

FIVE

GENERATOR

A generator creates a series of numbers by applying an expression to the value of an index. A generator is similar to a *C* style `for` loop. For this assignment, the index variable will always start at the lower bound and continue until it is equal to the upper bound (*the last value of the index will be the upper bound*). The index will always be incremented by the integer value 1. In *C*, this would be:

```
for(int i = <start>; i <= <end>; ++i)
```

5.1 Generator Format

A generator statement will always follow the same format:

```
[<id> in <int_1>..<int_2> | <expr>];
```

- `id` is the identifier of the generator's index.
- `int_1` is an integer representing the lower bound of the generator
- `int_2` is an integer representing the upper bound of the generator
- `expr` is an expression

Assertion: `int_1` and `int_2` will never be expressions, only integer literals. (*simple-bounds*)
Assertion: `int_1` will be never be greater than `int_2`. (*sane-bounds*)
Assertion: If an identifier is used in `expr` then it will match `id`. (*matching-id*)

For this assignment the value of the identifier variable <`id`>:

1. is initialized to the value of `int_1`.
2. is used to evaluate the expression.
3. is incremented by one.
4. stops when its value is greater than `int_2`.

For each value assumed by `id`, `expr` is used to generated the next number in the series.

Examples of valid generators:

```
[i in 1..10 | i * i];
[i in 0..10 | 2 ^ i];
```

In this assignment white space is not important so the following is valid:

```
[i
in
1
..
10
|
i*i];
[i in 1..10|2^i];
```

Assertion: Whitespace is guaranteed to be a space, a tab, a carriage return, or a new line. (*simple-whitespace*)

Because identifiers need white space to separate each other the following is invalid:

```
[iin1..10|i*i];
[i in1..10|2^i];
```

CHAPTER SIX

COMMENTS

Generator supports a subset of *C99* comments.

Single line comments are made using `//`. Anything on the line after the two adjacent forward slashes is ignored. For example:

```
// A comment on its own line
[i in 1..10 | i * i];  // This is ignored
```

CHAPTER SEVEN

INPUT

The input processed by your interpreter will be in a file specified on the command line. Your interpreter will be invoked with the following command:

```
generator <input_file_path> <output_file_path>
```

You should open the file `input_file_path` and parse it. The input file will be a valid generator file.

CHAPTER

EIGHT

OUTPUT

Output is to be written to a file specified on the command line. Your interpreter will be invoked with the following command:

```
generator <input_file_path> <output_file_path>
```

You should open the file `output_file_path` and write to it. The output file should be overwritten if it already exists.

Output content is standardized to ensure everyone can pass everyone's tests. Follow these specifications:

- All generated numbers should be printed on the same line with a single space separating the numbers.
- There *must* be a new line after each generator's output.
- There *must not* be any trailing space after the final number and before the newline.
- There *must* be an empty line at the end of your output (a result of #2).

Clarification: Empty input should result in empty output. (*empty-input*)

Example input (3 lines):

```
[i in 1..10| i];
[x in 0..3| x-1];
[x in 1..4| 10];
```

Expected Output (4 lines since each generator is newline terminated.):

```
1 2 3 4 5 6 7 8 9 10
-1 0 1 2
10 10 10 10
```

Depending on the text editor a file is viewed in, the final empty line may not be rendered (like the viewer above). To be sure your test cases adhere to the spec, make use of the 415 Tester which ensures that generated and expected output match to every single byte.

CHAPTER

NINE

ASSERTIONS

ALL input test cases will be valid. It can be a good idea to do error checking for your own testing and debugging, but it is *not necessary*. If you encounter what you think is undefined behaviour or think something is ambiguous then *do* make a forum post about it to clarify. While the generator is a relatively small spec, the latter assignments *will not be*.

What does it mean to be valid input? The input must adhere to the specification. The rules below give more in-depth explanation of specification particulars.

1. **undef-behaviour**:

   A test case *will not* take advantage of undefined behaviour. Undefined behaviour is functionality that does not have an outcome described explicitly by this specification.

2. **nonnegative-literals**:

   All integer literals will be ≥ 0. For example, the following tests would be considered invalid:

   ```
   [i in 0..1 | -1];
   [i in -2..-1 | i];
   ```

3. **literal-size**:

   All integer literals will fit in 31 unsigned bits. This means an integer literal can be anywhere in the range $[0, 2^{31} - 1]$ or $[0, 2147483647]$. For example, the following tests would be considered invalid:

   ```
   [i in 0..1 | -1];
   [i in 0..1 | 2147483648];
   [i in 0..2147483648 | 0];
   [i in -1..1 | 0];
   ```

4. **nonnegative-exp**:

   All exponents are ≥ 0. Exponents that are < 0 result in fractions (except when the base is 1). Given that the specification restricts generated numbers to be integers, it does not make sense to produce fractions. Therefore, you may assume that any exponent will be greater than or equal to zero, even if the base is 1. For example, the following test would be considered in invalid:

   ```
   [i in 0..1 | 2 ^ (0 - 2)];
   ```

5. **expression-size**:

   All expressions and their intermediate values will be representable in 32 signed bits. This means the result of an expression can be anywhere in the range $[-2^{31}, 2^{31} - 1]$ or $[-2147483648, 2147483647]$. Any expression which results in or produces in its intermediate computation an integer underflow or overflow is considered invalid. For example, the following testcase is invalid:

```
[i in 0..1 | 2147483647 + 1];
[i in 0..1 | 0 - 2147483647 - 2];
```

6. **zero-divide**:

   No expression will contain a division by 0. The result of a division by zero is indeterminate so we will not handle it. For example, the following tests would be considered invalid:

```
[i in 0..1 | 1 / 0];
[i in 0..1 | 1 / (1 - 1)];
[i in 0..1 | 1 % 0];
```

7. **simple-bounds**:

   The bounds on the index variable will always be integer literals and never an expression. For example, the following test would be considered invalid:

```
[i in (1-1)..1 | i];
```

8. **sane-bounds**:

   The bounds on the index variable will always be such that $\text{int}_1 \leq \text{int}_2$. For example, the following test would be considered invalid:

```
[i in 1..0 | i];
```

9. **matching-id**:

   Any variable used in an expression will match the variable defined in the generator. For example, the following test would be considered invalid:

```
[i in 0..1 | j];
```

10. **simple-whitespace**:

    Whitespace is guaranteed to be a space, a tab, a carriage return, or a new line. Any other whitespace characters will render the input invalid. Line comments skip all characters until the next newline or EOF. The following ANTLR rules will ensure you adhere to this:

    ::

    WS: [trn]+ -> skip; LINE_COMMENT: '//' .*? ('n' | EOF) -> skip;

CHAPTER

TEN

CLARIFICATIONS

These clarifications are meant to add more information to the specification without cluttering it.

1. **rem-not-mod**:

   The `%` operator is remainder not modulus. Some languages (e.g. Python) define `%` as the modulus operator while others (e.g. C++) define it as remainder. Using the `%` operator in C++ is sufficient for this assignment. For example, using the following test:

   ```
   [i in 0..2 | (i - 9) % 3];
   ```

   The following output would be considered incorrect because modulus was used:

   ```
   0 1 2
   ```

   The following output would be considered correct because remainder was used:

   ```
   0 -2 -1
   ```

2. **int-div**:

   Division is integer division. This means that any decimal portion of a division operation result is truncated (not rounded). No extra work is required: this is the default in C++. For example:

   ```
   [i in 0..6 | i / 3];
   [i in 0..6 | (0 - i) / 3];
   ```

   produces the following output:

   ```
   0 0 0 1 1 1 2
   0 0 0 -1 -1 -1 -2
   ```

3. **empty-input**:

   Empty input should result in empty output. This is in keeping with all of the output rules defined. There are no generators so there would be no numbers, spaces, newlines or output of any kind. All that you are left with is a single empty line, which matches “*should* be an empty line at the end of your output”.

CHAPTER ELEVEN

DELIVERABLES

Your submission will be **the latest commit before the deadline** to your github repository. Your submission will be automatically snapshotted by the GitHub classroom at the submission time.

Do no submit your binaries, they will be built just before being tested. The solutions will be built using the lab machines. You should make sure your solution builds in a lab environment prior to the submission time.

Your tests also should be committed to your github repository. We will pull both your submission and tests directly from your repository.

You do not need to submit anything on eclass or anywhere else.