generator

cmput415

Aug 17, 2020

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
```

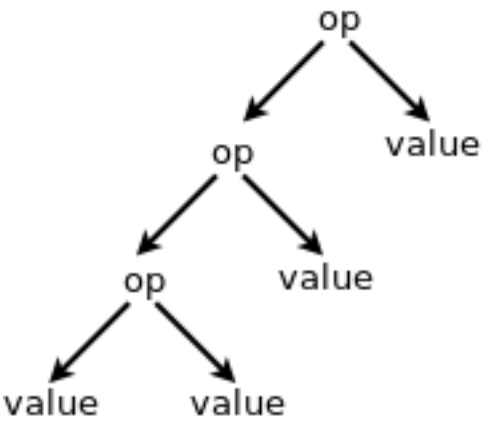
`2 * 3` should be evaluated before `1 + 2`. This is because multiplication, division, and remainder have higher precedence than addition and subtractions.

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
```

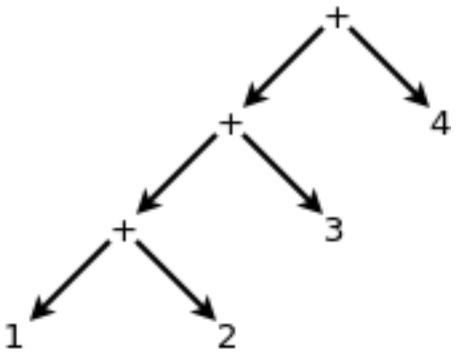
In this example both division and multiplication have the same precedence; associativity determines which operations are evaluated first. Left associative operations will form a parse tree like this:



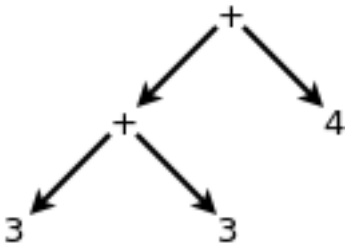
An example of one of these operations is addition. Lets say we have the following expression:

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

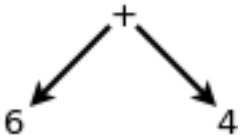
Because addition is left associative it will form the following parse tree:



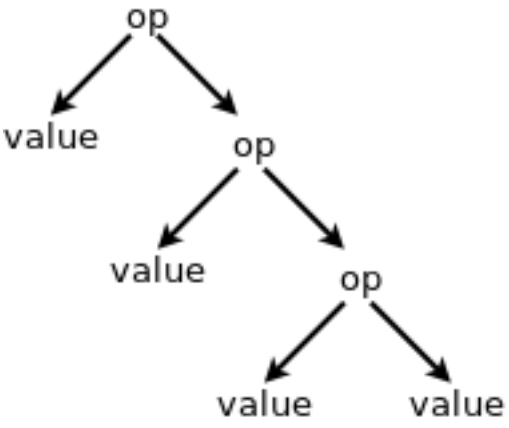
An operation in such a parse tree can only be evaluated when all the operands are leaves. Thus, in this parse tree, the expression `1 + 2` is evaluated first and then the result of this evaluation replaces the subtree for the expression `1 + 2` to create the following tree.



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:

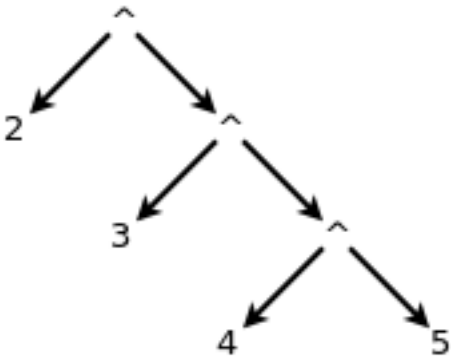


An example of a right-associative operation is the exponentiation operation represented by the symbol `^`. For example

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

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

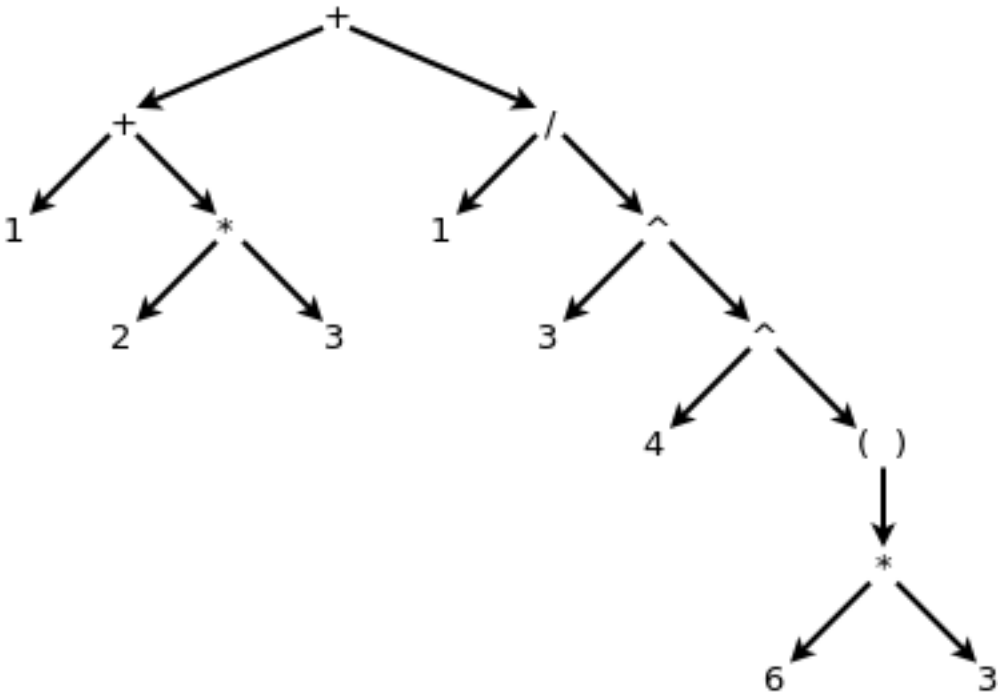
In order for this expression to be evaluated correctly the following parse tree must be generated



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

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

SEVEN

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.

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

Example input:

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

Expected Output:

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

The above output may be rendered incorrectly on your platform, therefore, the above input and its output are available to test for yourself. Remember that some editors (e.g. vim) hide a final empty line because they assume everyone will want one. Do not include this test in your submission.

CHAPTER

EIGHT

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 will result in a value that will fit 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 operation that results in underflow or overflow will render the input invalid. For example, the following tests would be considered 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)];
```

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. The following ANTLR rule will ensure you adhere to this:

```
WS: [ \t\r\n]+ -> skip;
```

CHAPTER

NINE

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

TEN

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.

CHAPTER

ELEVEN

TIPS AND HINTS

- Write tests *before* you implement the things they will test. The testing script provided is designed to handle failed test cases. You can reduce output in the testing tool by passing the `-q` flag.
- Make sure you've *read the documentation for ANTLR4*. Here is the link again Antlr4 documentation.
- *Antlr4* has two ways of navigating the parse tree, visitors and listeners. For this assignment, visitors can prove more effective.
- Be careful with your lexer rule ordering. Remember that the lexer matches tokens according to definition order and not according to any parser rules.
- In particular, *rule element labels* can be supremely useful. More info can be found here.
- Style and design are part of your mark. Here are some easy ways to ensure you remain stylish:
  - *ANTLR4* has no standardized naming convention or style guide. This means you *could* do whatever you want for style. With that in mind I highly recommend you follow something similar to the one on the *ANTLR4* documentation pages. What ever you choose make sure it is easy to read and consistent.
  - Keep your source organised. That means separating declarations (classes, class member functions, file scope functions) into headers and their definitions into source files. A good rule of thumb: if you're writing three or more lines of function code in a header, it's probably better in a source file. There are of course exceptions (template classes/functions, etc.) but their definitions can often be separated to below the class declaration to keep a "clean" declaration. If you can't follow this style for some reason, leave a comment for yourself (and the marker) to understand why. Consider reading this answer for more info.
  - Now that you've separated all of these files, you should make sure they're organised. The project is already set up to let you separate your headers from your source. A good idea is to have identical directory structures under `src` and `include` with headers and source files mirroring each other (e.g. `include/dir/dir2/class.h` and `src/dir/dir2/class.cpp`). You will be able to include the header via `#include "dir/dir2/class.h"`.
  - This is your work and you should own that. Add your name to the README and License as well. Your project is not "GeneratorBase". Change the base `CMakeLists.txt`'s project name to something more appropriate. "Generator", "<ccid>Generator" or something equally descriptive should be more appropriate. Just don't change the executable name!
  - You will also need to add your source files in the src/CMakeLists.txt file (or new subdirectory `CMakeLists.txts` that you make yourself). You should be explicit about your file paths (use `CMAKE_CURRENT_SOURCE_DIR` to make them absolute). These variables may be helpful.
  - This assignment is not as demanding as later assignments so a lack of documentation and information might be understandable. However, in the future, I expect to see appropriate comments in source and headers, changes to README where applicable, using the issue tracker, etc.
- The `antlrcpp::Any` type returned by `visit` can be very fickle if you don't understand its internals exactly (if you're curious). Instead of spending hours in that file, consider these best practices:

- Ensure the type you're returning is what you want to receive. You may *think* you're returning a certain type, but unless there is an explicit cast in the return statement or you're returning a typed variable you may get inexpicable type errors from `Any`. In other words, don't return temporaries. For example, this will break if the receiving side wants a pointer to the parent class even though there's an available typesafe cast:

```
{
  ...
  return std::make_shared<MyChildClass>(...);
}
```

- Ensure the type you're receiving from a call to `visit` is what you expected. The problem is a two way street. See the above reasons.

These may not seem relevant now, but it's better to get in the habit now than have it bite you later and not understand why.

CHAPTER

TWELVE

GETTING YOUR PROJECT

We will be using GitHub to manage our assignments.

1. Follow the assignment link on eclass to receive your repository.
2. Clone your repository to your working directory.

```
git clone <repository_clone_link>
```

12.1 Project Layout

For the tools provided to work your project should be in the specified layout.

```
+-- cmake
|   +-- antlr_generate.cmake
|   +-- get_antlr.cmake
|   +-- get_antlr_manual.cmake
|   +-- symlink_to_bin.cmake
+-- CMakeLists.txt
+-- grammar
|   +-- Generator.g4
+-- include
|   +-- placeholder.h
+-- LICENSE.md
+-- README.md
+-- src
    +-- CMakeLists.txt
    +-- main.cpp
+-- tests
    +-- GeneratorConfig.json
    +-- input
        +-- ...
    +-- output
        +-- ...
```

CHAPTER

THIRTEEN

SETTING UP CLION

CLion requires a little bit of setup.

1. Open up CLion. If you've been using CLion and it opens to a previous project, select `Close project`. Now that you're at the welcome screen, select `Open or Import` on the right. Navigate to where your project is located and select the folder that contains it. Select `OK` to open the project.
2. CLion doesn't make use of your command line environment, it has its own storage place. Therefore we need to add `ANTLR_INS` to CLion's environment.
    1. Open your settings. On Linux this is `File` → `Settings...`, while on MacOS this is `CLion` → `Preferences...`.
    2. From the left menu, expand `Build, Execution, Deployment`.
    3. Select `CMake` from the newly expanded options.
    4. In the right pane, select the `...` to the right of the empty text field to the right of `Environment`.
    5. In the new pane, select the + symbol to add a new entry in the environment. On Linux this is in the top right of the pane, while on MacOS this is in the bottom left of the pane.
    6. In the new text field under `Name` enter `ANTLR_INS`. In the field under `Value` enter the path to your `antlr_install` directory. If you've forgotten it but your terminal is set up correctly, or if you're using the lab machines, then you can enter the following command to print it:

        ```
        $ echo $ANTLR_INS
        ```

    7. Select `OK` to save your changes to the environment variables, select `OK` again to close the
3. We want to build all targets, not just the generator target. To do this we must add a new build configuration. If you have an older version of CLion this may already be present as a target called `Build All`.
    1. From the build configuration drop-down menu on the toolbar (which likely says `generator | Debug`) select `Edit configurations`.
    2. In the new pane, select the + to add a new configuration and select `CMake application`.
    3. Change the `Name` field to `all` and the target field to `All targets`.
    4. Click `OK` to save the new target.
    5. From the build configuration drop-down menu, select your new target called `all`.
    6. The build configuration menu should now show `all | Debug`.
4. CLion may not automatically pick up ANTLR's generated sources as your project's. We can fix this by telling CLion where the files are. Build once to have the `gen` directory appear in the project manager pane. Right click on the directory, near the bottom of the menu find `Mark directory as`, within that menu select `Project Sources and Headers`.

CHAPTER

FOURTEEN

TESTING

14.1 Testing Tool

Inside the `tests` directory, a testing configuration file, `GeneratorConfig.json`, is provided. You need to edit `inDir` with the absolute path of `.../tests/input`, `outDir` with the absolute path of `.../tests/output`, and finally `testedExecutablePaths` with your ccid and the absolute path of `.../bin/generator`.

Running the tester should now run your tests with your solution. Note that the output files will not be cleaned up so it might be wise run your tests in a directory that is not your repository. If you would still like to run them in the same directory, just know that the output files do not need to be tracked by git and can be safely deleted.

```
tester <path_to_config>
```

For more information about the testing tool and how it works, see the Tester README.

14.2 Generating Test Cases

A Python script `fuzzer.py` is available for automatic generation of random test cases.

More information is available in the README file included with the script, but an example usage of the script is as follows:

```
python fuzzer.py config.json test 10
```

The above command will generate two files `test.in` and `test.out`, where `test.in` will contain 10 generators and `test.out` will contain 10 lines each containing the expected output for each of the 10 generators. These files can be placed in your `tests` directory for use with the *Testing Tool*.