INFO0004-2: Parsing a painting language

Pr Laurent MATHY, Cyril SOLDANI

April 12, 2019

1 Introduction

In this assignment, you will design and implement a parser for a language describing painting with geometric figures. Your program will verify that the given input file respects the constraints of the language, and output some information about its content. If the input file contains (at least) one error, your program will abort with an error message pointing to the (first) error.

2 The painting language

Note that the following description of the painting language is not meant to fully explain the *meaning* of the language, *i.e.* how a file should be rendered to an image. It is just to explain the *syntax* of the language, so that your program can verify that the input file follows the language syntax (and constraints).

Rendering painting scripts to images will be the topic of the next assignment, and more information about the painting instructions will be given in the brief of the third project.

2.1 Overall structure

A *paint* file is a text (ASCII) file. White-space is generally ignored, but separates tokens so that `123␣456` represents the two numbers 123 and 456, and not 123456.

A `#` character introduces a comment. All characters from the `#` to the end of the line where it appears should be ignored.

A paint file begins with a `size WIDTH HEIGHT` command. The file can then contain any number of shape definitions, color definitions and paint commands, which are described below. In EBNF[1] notation, this gives:

```
paint_file = size_command, {shape_def | color_def | paint_command};
size_command = "size", number, number;
```

2.2 Numbers

Numbers in the paint language are floating-point numbers (except for the `size` command, but you can ignore that for now). They can be given as floating-point or integer literals, or by taking the abscissa or ordinate of a point (see section 2.6).

```
number = ["+" | "-"], [".", digit, {digit}
       | ["+" | "-"], digit, {digit}, ".", {digit}
       | point, ".", "x" | "y";
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
```

[1] Extended Backus-Naur form: https://en.wikipedia.org/wiki/Ebnf.

All of the following are valid numbers: `123` (= 123.0), `.5` (= 0.5), `+42.` (= 42.0), `-3.14`.

2.3 Names

The *paint* language allows to name shapes and colors. A `name` begins with a letter, and can contain any number of letters, digits and underscores.

```
letter = ? any char c such that isalpha(c) is true ?;
name = letter, {letter | digit | "_"};
```

All definitions and commands can only refer to names that were already defined above in the file. It is an error to use a name that is not defined, or is defined after the location where it is used. You should report such an error where the undefined name is used.

It is also a error to try to give a new shape the name of an existing shape, or a color the name of an existing color. You should report such an error at the location where the name is redefined. It is however fine to use the same name for a shape and a color, those lie in different *namespaces*.

2.4 Colors

A color is given by a triplet of numbers between braces. Each number represent a color component in the RGB color space, and should be between 0.0 and 1.0. However, you don't (yet) need to report an error if the component is out of range.

One can name a color with the `color` instruction, which takes a name and a color. A color name can be used wherever a color is expected.

```
color = "{" number, number, number, "}"
      | name;
color_def = "color", name, color;
```

2.5 Shapes

Each shape instruction is directly followed by a (new) `name`, which will be bound to the created shape. It is then followed by arguments which are specific to the type of shape being defined.

There are 3 primitive shapes: circles, rectangles and triangles.

Circle `circ` followed by the circle name, its center point and its radius.

Rectangle `rect` followed by the rectangle name, its center point, its width and its height.

Triangle `tri` followed by the triangle name, and its three vertices.

There are 4 derived shapes: shifts (translations) of a shape, rotations of a shape, unions of shapes and differences of shapes.

Shift `shift` followed by the name of the created shape, the point to which the origin of the original shape is to be translated (*i.e.* the translation vector), and the original shape to translate.

Rotation `rot` followed by the name or the created shape, the angle of rotation in degrees, the point around which to rotate the original shape, and the original shape to rotate.

Union `union` followed by the name of the union, then a non-empty list of shape names, between braces.

Difference `diff` followed by the name of the created shape, the name of the shape of which to subtract from, and the name of the shape to be subtracted.

```
shape_def = "circ", name, point, number
          | "rect", name, point, number, number
          | "tri", name, point, point, point
          | "shift", name, point, name
          | "rot", name, number, point, name
          | "union", name, "{", name, {name}, "}"
          | "diff", name, name, name;
```

2.6 Points

A point is defined by a pair of coordinates between braces, by referring to a point inside a shape, or by an arithmetic expression.

```
point = "{", number, number, "}"
      | named_point
      | point_expr;
named_point = name, ".", name;
```

A named point is given by `SHAPE_NAME.POINT_NAME` where `SHAPE_NAME` is an existing shape name, and `POINT_NAME` is the name of a specific point inside that shape. *E.g.* `a_circle.c` will return the center of the circle `a_circle` (assuming `a_circle` is a valid circle). Valid point names depend on the shape type. However, we have not described the valid point names for each shape type, and you cannot verify their validity as of now. Simply assume all point names are valid for now (but still check that the shape name refers to an existing shape).

A point can be subscripted with `.x` or `.y` to obtain respectively its abscissa and its ordinate. *E.g.* `a_circle.c.x` returns the abscissa of the center of circle `a_circle` (if it is defined). `{4 3}.y` returns 3.

Point expressions allow to do arithmetic operations on points. They use a syntax reminiscent of scheme, where the operator precedes its operands, and the order of operations is explicit. Addition and subtraction accept between one and an arbitrary number of points. *E.g.* `(- {12 0} {2 0} {3 1})` returns the point `{7.0 -1.0}` (as $12 - 2 - 3 = 7$ and $0 - 0 - 1 = -1$).

The multiplication and division take exactly one point and one number. They apply the operation to the coordinates of the point. *E.g.* `(/ {10 5} 2)` returns point `{5.0 2.5}`.

```
point_expr = "(", "+" | "-", point, {point}, ")"
           | "(", "*" | "/", point, number, ")";
```

2.7 Painting commands

There is a single painting command, `fill SHAPE_NAME COLOR`, which fills shape named `SHAPE_NAME` with the given `COLOR`.

```
paint_command = "fill", name, color;
```

3 Interface

Your program should be called `painter-check`, and will be called as `./painter-check INPUT_FILE`, where `INPUT_FILE` is the path to a file containing painting instructions.

It should output some info about the input file on `stdout`, namely the number of defined shapes, the number of named colors, and the number of fill operations. The format is as in the following example:

```
$ ./painter-check my_figure.paint
Number of shapes: 44
Number of colors: 7
Number of fills: 43
$
```

If there is an error in the input file, your program should abort with an error message on `stderr`, formatted as *FILE:LINE:COL: error: MSG* where *FILE* is the input file name, *LINE* and *COL* the line and column number where the error appears at (both starting at 1), and *MSG* is a message describing the error to the user. The definition of *MSG* is up to you, but try to be as informative as possible. *E.g.*

```
$ ./painter-check tests/wrong.paint
tests/wrong.paint:17:6: error: redefinition of shape 'appears_twice', already
    defined at 8:6.
$
```

If the input file contains multiple errors, you should only report the first one.

4 Remarks

4.1 Respect of the interface

You **must scrupulously follow the given interface**. Your archive should respect the format given in section 5, your code should build `painter-check` with a simple `make` in your folder, and its output should be exactly as described in section 3.

4.2 Good error messages

Your error messages should point to the exact location of the error, and be as descriptive as possible.

4.3 Readability

Your code must be readable:

- Make the organisation of your code as obvious as possible. Remember you can create as many auxiliary classes and functions as you see fit.
- Use descriptive names.
- Complement your self-documenting code with comments, where appropriate.
- Choose a coding convention, and stick to it. *Consistency* is key.

4.4 Robustness

Your code must be robust. The `const` keyword must be used correctly, sensitive variables must be correctly protected, memory must be managed appropriately, the program must run to completion **without crash** (even if the user provides invalid input).

4.5 Warnings

Your code must compile **without error or warning** with `g++ -std=c++11 -Wall` on `ms8??` machines. However, we advise you to check your code also with `g++ -std=c++11 -Wall -Wextra` or, even better, `clang++ -Weverything -Wno-c++98-compat`.

4.6 Evaluation

Your code will be evaluated based on all the above criteria. Failure to comply with any of the points mentioned in **bold face** can (and most often will) result in a mark of zero!

4.7 Advice

You are free to implement the parser as you see fit, as long as it works and its code is decently readable. However, you might find the following advise useful.

First, we advise you to extract characters from the input stream yourself, rather than using the STL I/O stream operators. It will make keeping track of the positions (line and column) much easier, and give you more control over what is accepted or not (*e.g.* `314e-2` will happily be converted to `double` by `operator>>(std::istream&, double)`, but is not a valid number in our painting language.

Second, we advise you to first split the input into *tokens* (*i.e.* sequence of characters that belong to the same logical entity, like a name, a number or an operator), rather than trying to manipulate characters directly from your parsing routines. This is called *lexing*, and will reduce the total amount of code, as your parser won't have to keep track of positions and potential white-space in every parsing routine. It will also make the parsing code higher-level and easier to follow.

Finally, note that our painting language is such that you can always tell what is expected next just based on the language rules and the next unprocessed token. Such a language lends itself well to a *recursive descent parser*[2], *i.e.* a parser written as a set of mutually recursive functions (or methods). Moreover, as our language does not support forward references, it can be parsed in just one pass.

As an example of how to implement a recursive-descent parser with a lexer in C++, you can have a look at the first two pages of the Kaleidoscope tutorial[3], which implements a recursive-descent parser for a simple arithmetic language (you can ignore sections 1.1, 2.2 and 2.5 which are mostly irrelevant to our simpler language).

5 Submission

Projects must be submitted through the submission platform before **April the 26th, 23:59 CET**. Late submissions will be accepted, but will receive a penalty of $2^n - 1$ points (/20), where n is the number of days after the deadline (each day start counting as a full day).

You will submit a `s<ID>.tar.xz` archive of a `s<ID>` folder containing your C++ source code and a `Makefile` to compile it (with a simple `make`), where `s<ID>` is your ULiège student ID.

The submission platform will do basic checks on your submission, and you can submit multiple times, so check your submission early!

[2] https://en.wikipedia.org/wiki/Recursive_descent_parser.
[3] https://llvm.org/docs/tutorial/LangImpl01.html and https://llvm.org/docs/tutorial/LangImpl02.html.