Compilation

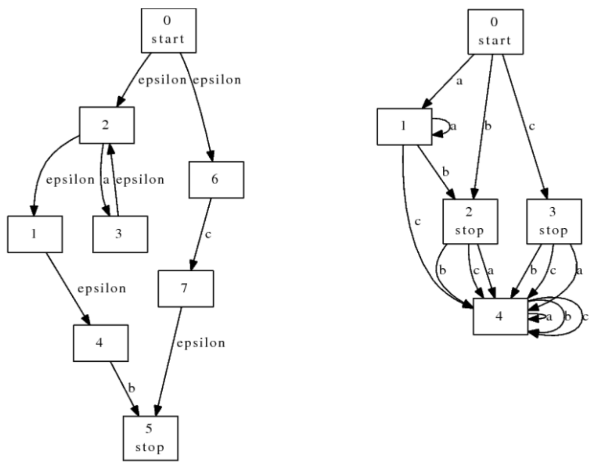
TP 1 : Orthography

C. Alias & G. Iooss

The first step of a compiler is to divide the program text into lexical units (called *token*, or *lexeme*). Writing a lexical analyzer is fastidious, and it is out of question to do everything by hand... Thus, we will build a compiler (or compiler) which generates automatically the C code from a lexical analyzer, given a lexical description.

Exercise 0. *Warming up*

Let us consider the regular expression $a^*b \mid c$ over the alphabet $\Sigma = \{a, b, c\}$. The Thompson construction and its determined version are given in the following figure:



Questions.

- **Check** that the Thompson construction and its deterministic version are the results of the algorithm given during the course.
- By applying the course algorithm, **minimize** the deterministic automaton. Notice that the deterministic automaton is also complete. This hypothesis is essential for the minimization. **Check this** by minimizing the deterministic automaton without state 4.

Exercise 1. *Regular expressions*

Download the file `lexer_V1.tar.gz` and decompress it. Our lexical analyzer generator is formed of the following files:

- **Regexp.***: Regular expressions
- **Automaton.***: Automatons: Thompson construction, determinization, minimization.
- **Lexer.***: Lexical analyser generator (will be given later)

Manipulations:

- **Open the file** `Regexp.h`. A regular expression is a type sum which can be naturally described by boolean fields (`is_epsilon`, `is_letter`, etc). Notice the *pretty-constructors*. Have a look at the implementation in `Regexp.cc`.
- In the file `main.cc`, write the code to **build and print-out** $a^*.b \mid c$.

Exercise 2. *Automatons*

The first step is to build a deterministic, complete and minimal automaton which represents the lexical description. These steps are considered point by point in this exercise.

Manip.

- **Open the file** `Automaton.h`. An automaton includes a set of states (line 34), a relation of transition (line 35), an initial state and a final state. Two auxiliary classes are needed: `State` (lines 15–18) where every instance represents a different state, and `Label` (lines 20–28) which represent either a letter, or ε.
- **Open the file** `Automaton.cc`. Complete Thompson construction to manage the union (line 83), by using the figure on the course slides.
- **Analyze** the method `determinize()` (after the lines 169–171) and `minimize()` (after the lines 273–275), that implements directly the algorithms we saw during the course.
- **Open the file** `main.cc`. Add the Thompson construction, the determinization and the minimization for the expression $a^*b \mid c$. Use the method `print_dot()` to **print-out the automaton graphically**. To do this, place the generated script on a file `test.dot`, and generate the automaton by using the command `dot -Tps test.dot > test.ps`.
- Create and print-out the automaton for the expression (`while\1 | b\2 | ␣\3 | ([w|h|i|l|e|b])*\4`)* where the characters `\1`, `\2`, `\3` et `\4` are *markers*. What is remaining to be done to build a lexical analyzer?

Exercise 3. *Lexical analyzers*

One the automaton is build, we need to (i) generate the tables (transition tables and table of the recognized lexemes for each states), and (ii) generate the C code of the analyzer.

A) Table production

In the file `Automaton.cc`, the method `transition_table` returns three data:

- The **transition table** of the automaton, where the lines are the states and the columns the letters of the alphabet.
- A **mapping letter → column** to know which column is corresponding to what.
- A **mapping state → recognized token**, which returns the code of the recognized token in the considered state. The lexeme code is the ASCII code of the associated marker. In exercise 2, the code of `while` is 1, `b` is 2, etc. If no lexeme is recognized in the considered state, we associate to this state the code 0.

Manip.

- **Open the file** `main.cc`. Apply `get_transition` to the minimized automaton. Print-out the computed data (the table and the two mappings). How does the ambiguous states are treated (states on which we can recognized several lexemes)?
- **On paper**, write down the main loop of the lexical analyzer.

B) Code generation for the lexical analyzer

Download the file `lexer_V2.tar.gz` in your repository and decompress it. The file `Lexer.cc` implements the code generation of the lexical analyzer. More precisely:

- The **constructor** takes as arguments an alphabet (without the markers) and a lexical description. The lexical description is implemented by a vector of regular expressions `Regexp`, the *rank* of a regular expression indicates *its priority*. The construction creates a minimal automaton for the given lexical description. The markers we use are the ASCII characters of code 1, 2, etc.

- The method **print_code** generates the lexical analyzer C code. It first calls `transition_table`, then generates the table (static array), and the necessary functions. The generated code supposes the existence of the following functions:
  - **reset_input_flow()** initializes the input stream (opening of a file, for example).
  - **read_next()** reads the next character on the input stream.
  - **unread()** go back of one character in the input stream.
  - **Eof** is the end of stream character (\0 for a char stream, EOF for a file).
  - **accept(token,string)** is called by the analyzer each time a token is recognized. Typically, the implementation of "accept" could print it out.
  - **error(car_number)** is called by the analyzer as soon as we encounter an error (non-valid character which is not in the input alphabet; or unrecognized token).

Manip.

- **Open the file** `main.cc`. Add the code that produces the lexical analyzer for the lexical description we have seen in exercise 2. Redirect the code produced by the lexical analyzer for the lexical description we have seen in exercise 2. Redirect the output of your executable toward the file `../src-gen/lex.c`. Notice that a file `automaton.dot` is also generated with the minimal automaton.
- **Open the file** `lex.c` and compare with the algorithm you had at the end of exercise 3A.
- **Open the file** `main.c` (in C) and implement the missing functions for the lexical analyzer. To simplify, we will read the characters in a chain of characters (char*). We will add an integer that will indicate the current position in this chain. Test your implementation.
- Push the analyzer to its limits... What part takes the most of the time? What could we optimized?

Exercise 4. *Flex*

`flex` is a very effective lexical analyzer generator (which is open-source) and is broadly used in the community of compiler developers.

A) Basic functionalities

Download the file `flex.tar.gz` and decompress it.

The file `lexer.l` contains the lexical description of our C compiler. If you have some time, you could try to generate it using the analyzer from exercise 3. A `flex` specification file is composed of 3 disjoints parts, separated by %%.

- The first part contains some **C/C++ code** which will be directly copy/paste at the beginning of the analyzer code (between {% et %}). Typically, we have inclusions, such that #include "parser.h" (line 32) which imports the token symbols to be recognized.
- The second part contains the **lexical description** under the form of a serrie of pair "regular expression"-"action to be done when we encounter it". typically, we transmit the recognized token to the syntactic analyzer *through* a return.
- The third part contains some **C/C++ code** which will be directly copy/paste at the end of the analyzer code. Typically, a `main()` function when we just want to try the lexical analyzer, as it is the case in this exercise.

The generated analyzer by `flex` export a function `yylex()` that returns the **first recognized token** in the file `FILE* yyin` (which needs to be opened). The following invocation of `yylex()` will return the following token, etc. When there is no more token to be read, `yylex()` returns 0.

Manip.

- **Open the file** `lexer.l`. Add a function `main()` which opens the file given as an argument (in `argv[1]`) and which will print-out all the recognized tokens. Try it on the file `bd.c`.

- Modify the lexical description to **print-out the token** when it is an identificator (TK_ID et TK_TYPE_ID). We will use the variable `char* yytext` (containing the value of the string corresponding to the token). Test it.

B) Start conditions

`flex` allows us to build several automatons and to switch between each others. Each automaton is called a *start condition.*

- An automaton (start condition) **is declared** in the first part by using `%x name_of_the_automaton`.
- Then, the automaton **lexical description** is declared by prefixing each regular expressions (which is corresponding with this automaton) by `%x name_of_the_automaton`.
- We ask the analyzer to **switch the automaton** by calling the macro `BEGIN(name_of_the_automaton)`.
- The default automaton is called `INITIAL`. So, **we can come back to the default automaton** by calling `BEGIN(INITIAL)`.

Manip.

- Add the comments management // et /* ... */

Exercise 5. *Bonus: Transition table compacting*

The generated transition tables might be huge. To save some memory state, it can be useful to compress them. **Invent a compacting algorithm**. We could use the transition tables generated by the lexical analyzer generator to test its performance.
Remark: Often, the regular expressions describe tokens that have only a few letter in common. They do not use (or almost) the same column in the transition table.