lab sessions 3 + 4: compiler construction

The third and fourth lab sessions of the course *Compiler Construction* focus on constructing a lexer (scanner) from a specification file. Of course, you will reuse the code of the previous two lab sessions, and do not start from scratch.

Format of the specification file

A specification file for the scanner generator may consist of 3 sections, of which two are optional. The following example specification file is given in the file `example.lex` which is available from Nestor.

```
section options
  lexer yylex;
  lexeme yytext;
  positioning on;
    where line yylineno;
          column yycolumn;
  default action defaultAction;
end section options;

section defines
  define digit  = [’0’-’9’];
  define letter = [’a’,’b’,’c’-’z’,’A’-’Y’, ’Z’];
end section defines;

section regexps
  regexp ’i’.’n’.’p’.’u’.’t’;
    token INPUTTOKEN;
  regexp ’o’.’u’.’t’.’p’.’u’.’t’;
    token OUTPUTTOKEN;
  regexp ’w’.’h’.’i’.’l’.’e’;
    token  WHILETOKEN;
    action foundWhile;
  regexp ’d’.’o’;
    token DOTOKEN;
  regexp ’i’.’f’;
    token IFTOKEN;
  regexp ’t’.’h’.’e’.’n’;
    token THENTOKEN;
  regexp ’e’.’l’.’s’.’e’;
    token ELSETOKEN;
  regexp ’e’.’n’.’d’;
    token ENDTOKEN;
  regexp ’:’.’=’;
    token BECOMESTOKEN;
  regexp ’;’;
    token SEMICOLON;
  regexp #39;
    token QUOTE;
  regexp { ’<’; ’<’.’=’; ’<’.’>’; ’>’.’=’; ’>’ };
    token RELOP;
  regexp { ’+’; ’-’ };
```

```
    token ADDOP;
  regexp { ’*’; ’*’ };
    token MULOP;
  regexp letter.(letter|digit)*;
    token IDENTIFIER;
  regexp digit+.(’.’.digit*)?.(’e’.(’+’|’-’|epsilon).digit+)?;
    token NUMBER;
  regexp eof;
    token EOFTOKEN;
    no action;
  regexp anychar;
    no token;
    action actionAnyChar;
end section regexps;
```

The first (optional) section is the **options** section. In this section, the user can set some parameters of the generated scanner. This section may contain the following fields (all of them are optional):

- **lexer** followed by the name of the lexer routine. If this field is not specified, then the generated scanner is named `yylex()` (which is compatible with the `(f)lex` generator).
- **lexeme** followed by the name of the lexeme string. If this field is not specified, then the lexeme is named `yytext` (which is compatible with the `(f)lex` generator).
- **positioning** followed by **on** or **off**. If this option is turned **on**, then the generated scanner will keep track of line numbers and column numbers. Only when this option is turned on, the user can specify in which (global) variables to store the line number and the column number by the use of a **where** clause (see example). The default is to turn this option **off**.
- **default action** followed by the name of a (void) C-routine that is executed after acceptance of a regular expression that is specified in the section **regexps**. If this optioned is omitted, then there is no such default action.

If the user omits the section **options**, then all fields are considered to be chosen as the default options.

The second (optional) section is the section named **defines**. In this section the user can define some classes of input characters, which helps to make easier regular expressions in the **regexps** section.

The third (required) section is named **regexps** and is used to define the regular expressions. Each regular expression starts with the keyword **regexp** followed by a regular expression. In a regexp specification, a character is enclosed by single quotes or it is specified by `#` followed by its ASCII number (for example, a single quote itself can be specified as `#39`). Concatenation of two regexps is denoted by a dot (`.`), choice is represented by a bar (`|`), and the Kleene closures are represented by `*` (zero or more times) or `+` (at least once). An optional part is followed by a question mark (`?`), but can also be implemented as a choice with the keyword **epsilon**.

The definition of a regexp is followed by some settings. The first is the token that the scanner should return on acceptance, specified via the keyword **token**. A regexp can be ignored by specifying **no token**.

If for some regexp, we do not want to perform an action after its recognition, then we specify this using the keywords **no action**. Also, it is possible to perform a specific action for a regexp (overriding the default action). This can be specified with the keyword **action**.

Sometimes, we want several regexps to return the same token. For this situation, the keyword **regexp** may also be followed by a set of regular expressions (see `RELOP` in the example).

Two special predefined regular expressions can be used (not in combination with a user defined compound regexp): **eof** and **anychar**. The first matches simply END-OF-FILE, and the latter accepts any character. Note that using `anychar` implies that the generated scanner accepts any input. Still, this is not in conflict with the other grammar rules, since the generated scanner will try to match the *longest* possible match, and in case two regexps yield both a longest match with the same length, then the one specified first in the specification file is chosen (which is the same behaviour as the one used in `(f)lex`).

Exercise 1: parsing the specification file

The above description of a specification file is deliberately given in text, and not specified by an (E)BNF grammar. Produce a parser and scanner combination that can accept lexer specification files. Use the example in the file `example.lex` to test your solution. Use the parser generator `LLnextgen` (so, you need to design a LL(1) grammar for specification files), and the scanner generator `(f)lex`. Your parser should produce reasonable (but not extensive) error reporting on wrong input, and should abort as soon as an error is found. Do not include actions in your parser/scanner yet. Once this exercise is solved, you completed lab session 3. Submit your code to Justitia (at `http://justitia.housing.rug.nl`). Note that justitia will always accept your code, even if it is incorrect! The teaching assistans will manually test your solution. You do not have to write a report (yet) for this part of the lab session.

Exercise 2: adding actions and produce a real scanner

Extend the LLnextgen grammar of exercise 1 with appropriate actions. Once the parser accepted the input specification, your program should construct a (super) NFA, convert it into a DFA, and produce a table driven scanner (that satisifies all specified options). Submit your final code to Justitia. Again, the teaching assistans will manually test your scanner generator. Also write a (short) report, in which you explain all steps (including previous lab sessions) and design decisions that you made to build the final product.

Deadlines:
session 3: Tuesday December 8th, beginning of lab session 4.
session 4: Tuesday December 15th, beginning of lab session 5.