yz709-compiler-sup1

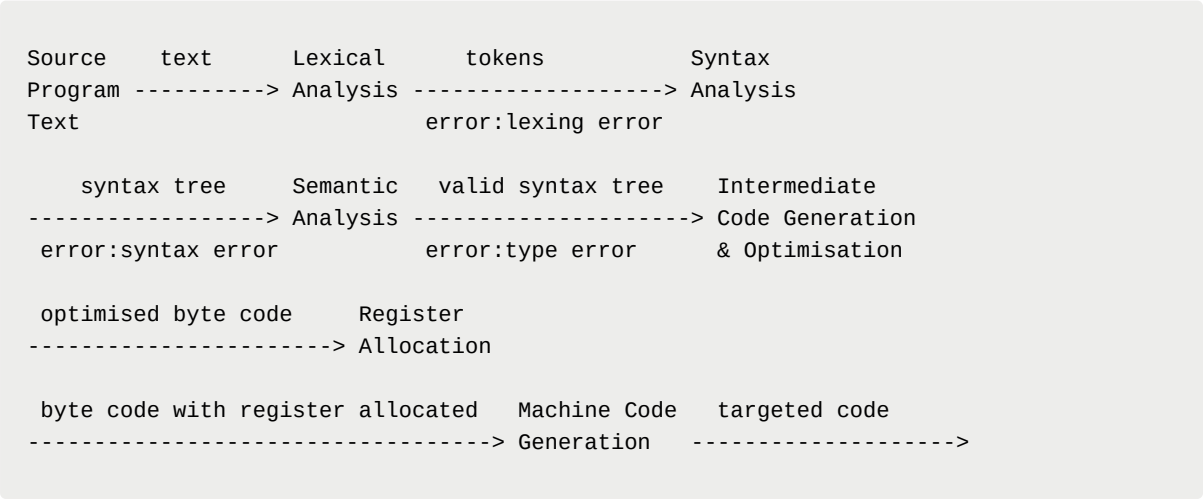
Question 2

Write down a *pipeline* of steps involved in compilation paying special attention to what goes into each step and what goes out. You can be as vague or specific as you want, but be prepared to discuss compiler related concepts and how it fits to your pipeline.

- The compiler phases can be split into frontend, middle, and backend.
- Frontend:
    - (1) Lexical analysis: reading and analysing program text into tokens, each of which corresponds to a symbol in the programming language (e.g., a variable name, keyword or number)
    - (2) Syntax analysis (parsing): take the list of tokens into a syntax tree that reflects the structure of the program
    - (3)Semantic analysis(e.g., type checking, def/use rules): analyses the syntax tree to determine if the program violates certain consistency requirements (e.g., the undeclared variable used); this phase ensures the absence of syntax errors
- Middle: (4) Intermediate code generation and optimisations & transformations on the intermediate code: program translates into a simple machine-independent

language

- Backend, needs to understand the ISA of the targeted machine & OS interface:
  - (5) Register allocation: the symbolic variable names used in the intermediate code are translated to numbers, each of which corresponds to a register in the target machine code
  - (6) Machine code generation: the intermediate language is translated to assembly language (a textural representation of machine code) for a specific machine architecture; this phase ensures the absence of type errors.



Comments:

Question 3

How is LLVM approach to compiler construction is different than that is presented in this course?

- LLVM is a framework to generate object code from any kind of source code.
- LLVM follows five basic phases similar to the phases presented in this course. However, the Assembly & linking phase and execution phase are not phases of the compiler model, but LLVM supports these two phases. It provides an optimising linker to combine LLVM object files and additional runtime and offline optimisers added to the execution phase.
- After the machine code generation phase, the `.o` file generated by LLVM does not contain any codes but a compressed format of LLVM codes.

💡 Comments:

Question 4

Why are functional languages are commonly used to prototype compilers?

- Functional programming languages have powerful pattern matching techniques embedded and good support for recursion. Since the majority of the work of a compiler is to manipulate the syntax tree (e.g., convert tokens into a syntax tree, check the validity of a syntax tree and optimise it), pattern matching will be useful when dealing with the tree structure.
- A compiler aims to transform from one language representation to another (e.g., source code to tokens, tokens to syntax trees, syntax trees to byte codes, byte codes to machine codes), so functions with no side-effects are desirable for this type of transformation tasks.

💡 Comments:

Question 5

Why does `past.ml` in first Slang compiler have `loc` parameter for every constructor, while the corresponding constructors in `ast.ml` don't have them?

- The file `ast.ml` simplifies the codes by removing all syntactic sugar, file location information and most type information. Hence maybe the locations are used in `past.ml` to provide file location information when errors are generated or give users more information when debugging.

💡 Comments:

Question 6

According to the lexical analysis algorithm described in the slides, what are the two rules that disambiguate multiple possible matches out of the same character stream?

- Find the longest possible match: if we have a word `thenx`, we will not stop when we encounter `then`, but continue exploring until we reach the termination of the word.
- Choose the token with the highest priority among all possible matches: if we have a word `if`, the keyword `if` has a higher priority than the variable `if`, so we could choose it to be a keyword.

💡 Comments:

Question 7

Starting with regular expressions that match individual tokens, how do we generate a single lexing program?

- In the single lexing program, it has an input of a list of regular expressions that match individual tokens and a string w.
- Set up an ordered list of regular expressions $(r_1, r_2, ..., r_n)$, ordered by priority to resolve ambiguity (e.g., if matches keyword RE before identifier RE).
- Then construct an NFA for the union of the regular expressions: $r = r_1|r_2|...|r_n$, where the accepting states indicate the token types, the dead state indicates the termination of the word.
- We can convert an NFA into a DFA using subset construction. At this stage, for each accepting state, there might have one or more NFA states, we erase all but choose the highest priority one.
- Finally, we could (1) start from the initial state of the DFA, (2) repeatedly read characters from w until we reach a dead state, emit token associated with the last accepting state, (3) then reset to the initial state for parsing the rest of w.

💡 Comments:

Question 8

What is the difference between concrete and abstract syntax trees (CST vs AST)? What stages of compilation involve these?

- A concrete syntax tree is a representation of grammar in a tree structure. It only conveys syntactic information and thus is simple to create from grammar but difficult to analyse.
- The abstract syntax tree is a simplified syntactic representation of the source code. It drops all the syntactic details and focuses on the structure of the input, and thus is more useful for analysis and translation
- During the syntax analysis phase, the parser will first construct a CST, then convert it into an AST

Expression : 2 * 7 + 3

Grammar :

exp → exp + term

exp → term

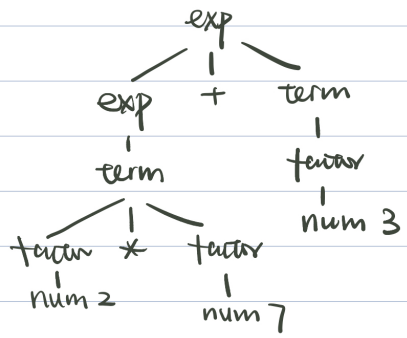
term → term * factor

term → factor

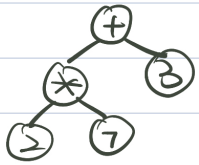
factor → num

factor → (exp)

concrete syntax tree / Parsed Tree



abstract syntax tree



💡 Comments:

Question 9

What does it mean for a grammar to be ambiguous? How can this ambiguity be resolved?

- When a grammar permits several different syntax trees for some strings. Given that left derivation and right derivation give the same syntax tree, permitting different syntax trees for the same string makes the parser non-deterministic, which is not acceptable.
- Solving ambiguity - for example use a set of operators $T = \{+, -, *, \div, (,), num\}$:
  - Group the operators with the same precedence, they must have the same associativity to make the grammar unambiguous: $\{+, -\}$, $\{*, \div\}$, $\{(,), num\}$
  - Handle operators with different precedences, by using a nonterminal for each precedence level (e.g., T, F are added as new nonterminal symbols for operators with different precedence)

```
(* for {+,-} *)
E -> E + T (* T must have a higher precedence than E *)
E -> E - T
E -> T

(* for {*,÷} *)
T -> T * F (* F must have a higher precedence than T *)
T -> T ÷ F
T -> F

(* for {(,),num} *)
F -> (E)
F -> num
```

💡 Comments:

Question 10

How does a recursive descent parser work? What class of languages can it be used to describe? What are the problems with it?

- Recursive descent parser works by translating the grammar structure directly into the program structure.
- Process as follows: (1) Each non-terminal in the grammar is implemented as a function; (2) each such function looks at the next input symbol to choose one of the productions for the nonterminal; (3) either match the terminal, report an error, or recursively calling the next nonterminal function; (4) functions return once no input symbols left.
- The class of languages that are (1) unambiguous, and we can eliminate the ambiguity in the grammars for those languages; (2) contains no left recursion (e.g., $E \rightarrow E + T$) and left factorisation.
- The problems:
  - (1) it does not work for all context-free grammars. For instance, those ambiguous context-free grammars cannot be accepted by a recursive descent parser because it may need to choose a production among several possible ones. Hence, with such non-deterministic property, the ambiguous language cannot be parsed.
  - (2) Most grammars need extensive rewriting to get them into a form that allows the unique choice of production, hence not very efficient in terms of performance.

Comments:

Question 11

How does a shift-reduce parser work? What class of languages can it be used to describe? What problems of recursive descent parser does it address?

- LR parsing (shift-reduce parser) are table-driven bottom-up parsers and use actions (shift & reduce) involving the input stream and a stack:
  - Shift: a symbol is read from the input and pushed on the stack
  - Reduce: (1) the top N elements of the stack hold symbols identical to the N symbols on the RHS of a production, (2) reduce these N symbols by replacing them with a nonterminal at the left-hand side of the production

- Can accept context-free grammars that will not cause shift-reduce conflicts or reduce-reduce conflicts in the SLR parse-table. Precedence rules can solve some of the shift-reduce conflicts caused by ambiguous grammar. But there are still some unambiguous grammars that a shift-reduce parser cannot accept.
- Addresses the problems:
  - Accept a much larger class of grammars than recursive descent parser, though still not all the grammars.
  - Less rewriting is required to get a grammar in an acceptable form for the shift-reduce parser to work on.
  - Allow external declaration of operator precedences to resolve ambiguity instead of requiring the grammars themselves to be unambiguous.

> 💡 Comments:

Question 12 (y2012p3q4(a)(b))

(a) Define the following terms used when discussing a grammar:

(i) a non-terminal symbol [1 mark]

(ii) an ambiguous grammar [1 mark]

(iii) a production [1 mark]

(iv) a context-free grammar [2 marks]

(v) a regular grammar [2 marks]

- (i): a non-terminal symbol is one which can be further derived into a string of terminals or ε using a set of productions.
- (ii): an ambiguous grammar is one that allows multiple syntax trees for the same strings.
- (iii): a production is a rule for replacing a specific nonterminal symbol, it is written as a nonterminal $N \rightarrow X_1 ... X_n$, where X_i are symbols (either terminal or

nonterminal), and the set denoted by N contains strings that are obtained by concatenating strings from the sets denoted by $X_1 ... X_n$

- (iv):
  - A context-free grammar is a recursive notation for describing sets of strings and imposing a structure on each such string. It can capture both regular and irregular languages.
  - For each CFG G, there is a push-down automaton M such that the languages described by each are the same. $L(G) = L(M)$.
- (v):
  - A regular grammar, like context-free grammar, it is also a recursive notation for describing sets of strings and imposing a structure on each such string, but it can only accept regular languages.
  - For each regular grammar G, there is a Non-deterministic finite automaton N and a deterministic finite automaton D, such that the languages described by each are the same. $L(G) = L(N) = L(D)$.

(b) The following grammar defines a language where expressions are strings or integers. A type error arises when an integer is added to a string.

```
Var -> x | y
Exp -> Var | 0 | 1  | "cat" | "dog" | Exp + Exp
S   -> Var := Exp | S S
```

(i) Give a syntactically-correct sentence of the language that contains a type error. [1 mark]

(ii) What phase (or pass) of a typical, simple compiler would detect such a type error? Sketch the fragment of code that actually spots the error. [3 marks]

(iii) Provide a modified grammar such that type errors are also syntax errors. [3 marks]

(iv) Why is such a modified grammar generally impractical? [1 mark]

- (i):

```
"cat" + 1
```

- (ii):
  - During the semantic analysis stage, type checking would be carried out.
  - In the following code fragment, the error would be spotted in `do_oper(op, v1, v2)`, when the interpreter is trying to do a `+` operation on two values with different types.

```
let rec interpret(e, env, store) =
  match e with
  |Op(e1, op, e2) -> let (v1, store1) = interpret(e1, env, store) in
                     let (v2, store2) = interpret(e2, env, store) in
                     do_oper(op, v1, v2)
  |...
```

- (iii):

```
(* original grammar *)
Var -> x | y
Exp -> Var | 0 | 1 | "cat" | "dog" | Exp + Exp
S -> Var := Exp | S S

(* modified grammar *)
Var ->  x | y
Str -> "cat" | "dog"
Num -> 0 | 1
Exp -> Var | Str + Str | Num + Num | Str | Num
S -> Var := Exp | S S
```

- (iv):
  - It is similar to operator overloading, for n primitive value types, we have to add an overloaded version of each operator for such primitive type. Hence the number of rules increases exponentially for real programming languages.

Comments:

Question 13 (y2015p3q3)

Consider the following context-free grammar of expressions

$$E ::= n \mid (E,\ E)$$

where n ranges over integers.

(a) Present a right-most derivation of the expression ((21, 18), 17). [2 marks]

$stack, input$
$, ((21, 18), 17)$
$(, (21, 18), 17)$
$((, 21,18), 17)$
$((21, ,18), 17)$
$((E, ,18), 17)$
$((E,, 18), 17)$
$((E, 18,), 17)$
$((E, E,), 17)$
$((E, E), , 17)$
$(E, , 17)$
$(E,, 17)$
$(E, 17,)$
$(E, E,)$
$(E, E), $
$E, $

(b) List the LR(0) items for this grammar. [2 marks]

```
E → . n
E → n .
E → . (E, E)
E → ( . E, E)
E → (E . , E)
E → (E, . E)
E → (E, E . )
E → (E, E) .
```

(c) Describe the states of the deterministic finite automata associated with an LR(0) parser for the grammar presented above. Explain your method of constructing these states. [4 marks]

- Add a new production $E' \to E$ where E is the original start symbol, hence the NFA start state $q_0 = E' \to \cdot E$, thus the DFA start state would be ε-closure $(\{E' \to \cdot E\})$ as follows

```
E' → .E
E → .n
E → .(E, E)
```

- For this DFA, we have $\delta(I, X) = \varepsilon\text{- closure}(\{A \to \alpha X \cdot \beta | A \to \alpha \cdot X\beta \in I\})$

(d) Describe the calculation of the *goto* function associated with an LR(0) parser for the grammar above. How is the goto function used by the parser? [4 marks]

- If $\delta(I_i, A) = I_j$, then `GOTO[i, A] = j`, where A is a nonterminal, the action `goto` is for transitions on nonterminals.
- In the parser, a `reduce` in the current state is followed by a `goto` in the state found when the RHS of the production is popped off. This is done because when we reduced a string of terminals into a nonterminal using a particular production, we arrive at the state of the new stack top (the string of terminals have been popped off), so the following transition using the nonterminal happens at that new state.

```
reduce p:
  n := the LHS of production p
  r := len(RHS of p)
  pop r elements from the stack (* replace the r symbols with nonterminal n *)
  push(s,stack)
    where table[top(stack), n] = goto s
    (* destination of goto depends on new stack top *)
```

(e) Carefully describe the LR(0) parsing associated with your derivation in (a). That is, show each transition of the parser and how it performs *shift* and *reduce* operations. [8 marks]

$stack, input$	action	reason
$, ((21, 18), 17)$	shift	E → . (E, E)
$(, (21, 18), 17)$	shift	E → . (E, E)
$((, 21,18), 17)$	shift	E → . n
$((21, ,18), 17)$	reduce E → n	, ∈ FW(E)
$((E, ,18), 17)$	shift	E → (E . , E)
$((E,, 18), 17)$	shift	E → . n
$((E, 18,), 17)$	reduce E → n	) ∈ FW(E)
$((E, E,), 17)$	shift	E → (E, E .)
$((E, E), , 17)$	reduce E → (E, E)	, ∈ FW(E)
$(E, , 17)$	shift	E → (E . , E)
$(E,, 17)$	shift	E → . n
$(E, 17,)$	reduce E → n	) ∈ FW(E)
$(E, E,)$	shift	E → (E, E .)
$(E, E), $	reduce E → (E, E)	$ ∈ FW(E)
$E, $	accept!!!	

💡 Comments:

Question 14

Consider a language of regular expressions consisting of

- characters (e.g., `a` matching the string `a`),
- concatenation operation (e.g., `ab` matching `a` then `b`),
- alternative operator (e.g., `a|b` matching `a` or `b`),
- the Kleene star (e.g., `a*` matching zero or more `a`s),
- a restricted form of character classes with ranges (e.g., `[a-c]` to mean matching `a` or `b` or `c`) as well as lists (e.g., `[abc]` to mean matching `a` or `b` or `c`),
- and parentheses (e.g., `a(b|c)` meaning matching `a` followed by matching `b` or `c`).

For this language,

a. Design a grammar for this language taking operator precedence into account (concatenation binds tighter than the alternative). Write it down using the EBNF notation;

```
L -> [E] | E
E -> E|F | F
F -> FT| T
T -> (E) | T* | S
S -> a | b | c | ε
(* assume we only have characters a, b, c *)
```

b. Write a hand-coded lexer and a recursive descent parser for this grammar in OCaml. Clearly explain any transformations you made to your original grammar to accomplish this;

- Recursive descent parser:
  - Need to eliminate grammar ambiguity (already removed), and left recursion, otherwise, we will have an infinite loop $(e.g., F \to FT)$

    ```
    L -> [E] | E
    E -> FE'
    E' -> |FE' | ε
    F -> TF'
    F' -> TF' | ε
    T -> (E)T' | ST'
    T' -> *T' | ε
    S -> a | b | c | ε
    ```

  - Add a production $L' \to L\$$, calculates all FIRST and FOLLOW sets.

    ```
    FIRST(L') = {[, (, a, b, c}
    FIRST(L) = {[, (, a, b, c}
    FIRST(E) = {(, a, b, c}
    FIRST(E') = {|}
    FIRST(F) = {(, a, b, c}
    FIRST(F') = {(, a, b, c}
    FIRST(T) = {(, a, b, c}
    FIRST(T') = {*}
    FIRST(S) = {a, b, c}

    FOLLOW(L) = {$}
    FOLLOW(E) = {], ), $}
    FOLLOW(E') = {], ), $}
    FOLLOW(F) = {|}
    FOLLOW(F') = {|}
    ```

```
FOLLOW(T) = {(, a, b, c}
FOLLOW(T') = {(, a, b, c}
FOLLOW(S) = {*}
```

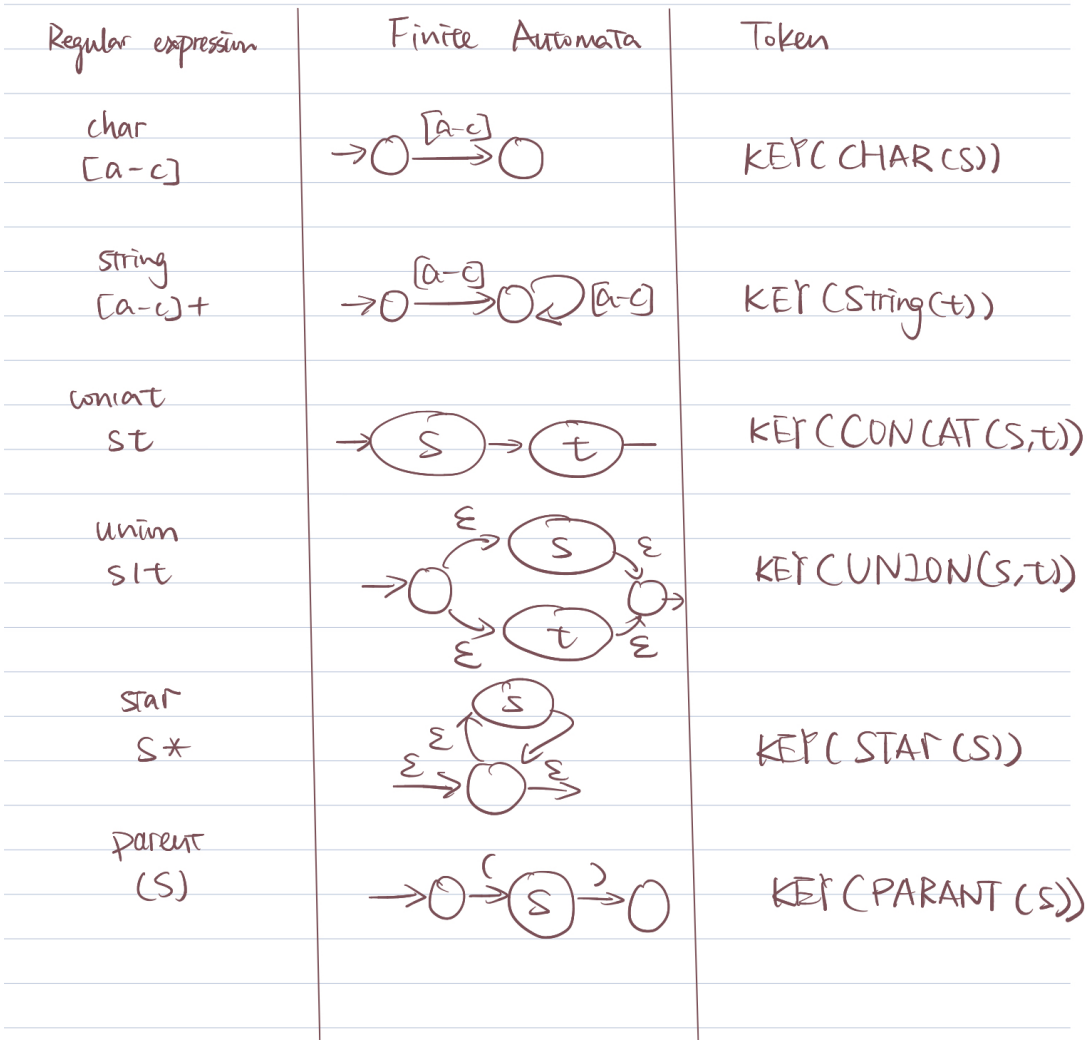
- Each non-terminal in the grammar is implemented as a function; each such function looks at the next symbol to choose one of the productions for the nonterminal.
- Either match the terminal, or report an error, or recursively call the following nonterminal function.
- Functions return once there are no input symbols left.

```
(* assume we have a function inset(set, symbol)
   to test whether a symbol ∈ set
   and a function next N = (hd, tl)
   gives the head element in N and the rest as tl
*)

exception Err

(* production L' -> L, use L_ to represent L' *)
let parseL_ L_ =
  let (hd, tl) = next L_ in
    if inset(FIRST(L'), hd) then parseL tl; match('$');
    else raise Err;

(* productions L -> [E] | E *)
let rec parseL L =
  let (hd, tl) = next L in
    if  hd = '[' then match('['); parseE tl; match(']');
    else if inset(FIRST(E), hd) || inset(FOLLOW(L), hd) then parseE L;
    else raise Err;

(* productions E -> FE' *)
let rec parseE E =
  let (hd, tl) = next E in
    if inset(FIRST(F), hd) then parseF E;
    else raise Err;

(* productions E' -> |FE' | ε *)
let rec parseE_ E_ =
  let (hd, tl) = next E_ in
    if hd = '|' then match('|'); parseF tl;
    else if inset(FOLLOW(E'), hd) = false then raise Err;

(* productions F -> TF' *)
let rec parseF F =
  let (hd, tl) = next F in
    if inset(FIRST(T), hd) || inset(FOLLOW(F), hd) then parseT F;
    else raise Err;

(* productions F' -> TF' | ε *)
```

```
let rec parseF_ F_ =
  let (hd, tl) = next F_ in
    if inset(FIRST(T), hd) then parseT F_;
    else if inset(FOLLOW(F'), hd) = false then raise Err;

(* productions T -> (E)T' | ST' *)
let rec parseT T =
  let (hd, tl) = next T in
    if hd = '(' then match('('); parseE tl; match(')');
    else if inset(FIRST(S), hd) || inset(FOLLOW(T), hd) then parseS T;
    else raise Err;

(* productions T' -> *T' | ε *)
let rec parseT_ T_ =
  let (hd, tl) = next T_ in
    if hd = '*' then match('*'); parseT_ tl;
    else if inset(FOLLOW(T'), hd) = false then raise Err;

(* productions S -> a | b | c | ε *)
let rec parseS S =
  let (hd, tl) = next S in
    match hd with
    |a -> match('a');
    |b -> match('b');
    |c -> match('c');
    |hd -> if inset(FOLLOW(S), hd) = false then raise Err;
```

- Hand-coded lexer:

```
type var = string

type token =
  | CHAR of var
  | CONCAT of token * token
  | UNION of token * token
  | STAR of token
  | PARENT of token
```



c. Write a computer-generated lexer and parser using `ocamllex` and `menhir` OCaml packages. You might like to consult this tutorial to learn more about the tools.

```
{
open Lexing
open Parser

exception SyntaxError of string

let next_line lexbuf =
  let pos = lexbuf.lexcurr_p in
  lexbuf.lex_curr_p <-
    {pos with pos_bol = lexbuf.lex_curr_pos;
              pos_lnum = pos.pos_lnum + 1
    }
}

(*
L -> [E] | E
E -> E|F | F
F -> F.T| T
```

```
T -> (E) | T* | S
S -> a | b | c | ε
*)

(* define helper regexes *)
let char = ['a'-'c']
let string = ['a'-'c']*
let newline = ('\010' | "\013\010" )

rule token =
  parse
  | "(" {LPAREN}
  | ")" {RPAREN}
  | "[" {LSPAREN}
  | "]" {RSPAREN}
  | "|" {UNION}
  | "." {CONCAT}
  | char {CHAR (Lexing.lexeme lexbuf)}
  | string {STRING (Lexing.lexeme lexbuf)}
  | newline {next_line lexbuf; token lexbuf}
  | eof {EOF}
  | _ {Errors.complain
  ("Lexer : Illegal character" ^ (Char.escaped(Lexing.lexeme_char lexbuf 0)))
```

💡 Comments: Sorry, I had some problems answering this question, and I don't think I did well; I may need more time to grasp the concepts of implementing a lexer using `ocamllex` and `menhir` OCaml packages. Apologise for the sketchy answers :(

Question 1(Discuss in the supervision)

Build and play with the SLANG compiler. Experiment with the frontend (syntax/type checker) and be prepared to discuss what you have done in the supervision. It is OK if the changes you make do not work.

💡 Comments: