COL 226: Programming Languages
Assignment2: Lexing and Parsing

Submission Lifeline: Wed 10 Mar 2021, 11:59 PM
Submission Deadline with Late Penalty: Sun 14 Mar 2021, 11:59 PM

Problem Statement

Your task is to write lexer and parser for Boolean algebra using ML-Lex and ML-Yacc.

Problem Specification

Create lexer and parser files using ML-Lex and ML-Yacc for SML. Add a make-file to generate executable named `a2`.

Input: The executable should take the name of the file to be analyzed as a command-line argument. The syntax of the program, along with the names of token types for each token, is as follows:

1. The input file consists of a single input program. Define a non-terminal `program` for the entire program. Each file should end with a token type `EOF`.
2. A program is a set of statements. A statement is a boolean formula followed by a 'termination character semicolon (";"). Use non-terminal `statement` for a valid statement and token type `TERM` for semicolon.
3. Represent a formula by non-terminal `formula`. A formula may consists of:
   (a) Constants "TRUE" and "FALSE" representing bool 1 and 0 respectively. Use token type `CONST` for constants.
   (b) Right-to-left associative prefix unary operator "NOT" of a formula, having form "NOT formula". Use token type `NOT` for token "NOT".
   (c) Left-to-right associative infix binary operators over two formulas, having form "formula1 binop formula2", where "binop" can be "AND", "OR", "XOR" and "EQUALS". Represent these using token types `AND, OR, XOR` and `EQUALS` respectively.
   (d) Right-to-left associative implication operator of the form "formula1 IMPLIES formula2". Use token type `IMPLIES` for token "IMPLIES".
   (e) Right-to-left associative if-then-else operator of the form "IF formula1 THEN formula2 ELSE formula3". Use token types `IF, THEN` and `ELSE` for "IF", "THEN" and "ELSE".
   (f) Parenthesis to define order of evaluation over different operations "(formula)". Use token types `LPAREN` and `RPAREN` for left and right parenthesis respectively.

(g) For formulas and sub-formulas without parenthesis, the order of evaluation is decided according to associativity rules.

(h) Any other string containing only lower and upper case English alphabets is a variable. Use token type `ID` for variable identifiers.

(i) All the operations mentioned in any point above have the same precedence, and the precedence is decreasing from point 3b to point 3e. For example, NOT operation has higher precedence than AND, OR, XOR and EQUALS operations.

4. A formula or statement may be written in several lines.

Executing the Program: We will compile your submission by running `make` command and run the executable as `./a2` ⟨`filename`⟩

Output: The executable `a2` should produce the output of the lexer followed by a newline, then the parser's output. Lexer output should be a comma-separated list (enclosed in square brackets) of tokens in order of their appearance in the input file. Each token in output should be of the form "⟨`token type`⟩ `space` ⟨`actual token in the input file enclosed in double quotes`⟩". The output of the parser should be the preorder traversal of the generated parse tree. The preorder traversal should be a comma-separated list of each node's representation. Use the production rules to represent a non-terminal node, and "⟨`token type`⟩ `space` ⟨`actual token in the input file`⟩" to represent the terminal nodes in preorder traversal.

Error Reporting: Whenever an invalid token is encountered, lexer should generate `Unknown Token:`⟨`line no`⟩`:`⟨`column number`⟩`:`⟨`token`⟩ error. Here ⟨`line no`⟩ and ⟨`column number`⟩ start from 1, and ⟨`token`⟩ is the invalid token. If the input is not syntatically correct according to the specifications, the parser should generate `Syntax Error:`⟨`line no`⟩`:`⟨`column number`⟩`:`⟨`production rule`⟩ error. Here ⟨`production rule`⟩ is the production rule where syntax did not match.

Examples: Some examples of valid and invalid programs are as follows:

1. `(xyz IMPLIES FALSE) OR TRUE AND IF A THEN b ELSE c;` is a valid statement having identifiers `xyz, A, b` and `c`. The lexer output for this example should be as follows:

   [`LPAREN` "(", `ID` "xyz", `IMPLIES` "IMPLIES", `CONST` "FALSE", `RPAREN`, ")", `OR` "OR", `CONST` "TRUE", `AND` "AND", `IF` "IF", `ID` "A", `THEN` "THEN", `ID` "b", `ELSE` "c", `TERM` ";"]

2. If line 5 of input file is `a | b;` lexer should result in error `Unknown token:5:3:|`.

3. `IF x EQUALS y z THEN TRUE ELSE a XOR b;` is an invalid statement. Since all the tokens in this expression are valid, the lexer should produce the correct output. However, the parser should generate an error. If this statement is in line 5, the parser error should be

   `Syntax Error:5:15:‘‘concerned production rule’’`

Tasks to be done

Task1 (20 marks): Write EBNF for the language given in this document in a file named `ebnf.txt`. Take care of the order of evaluation specified here.
Task2 (30 marks): Create a file for lexical analysis using ML-Lex for the language given in this document. Follow the specifications given regarding input, output, and execution.
Task3 (50 marks): Create a file for parsing the language given in this document using ML-Yacc. Follow the specifications given regarding input, output, and execution.

Submission Instruction

Submit a zip file named ⟨EntryNumber⟩.zip. On unzipping the file, it should produce the lexer, parser, `ebnf.txt` and `makefile`. It should also contain any other source file used in your program. You may provide a README.md that contains suggestions for the evaluator.

Important Notes

1. Do not change any of the names given in this document. You are not even allowed to change upper-case letters to lower-case letters or vice-versa.

2. Follow the input/output specification as given. A part of this assignment will be auto-grade. In case of mismatch, you will be awarded zero marks.

3. Take care of extra whitespace characters.

4. You may create new token types if required. Make sure that all the terminals use the token types specified in the document.

5. We have created a piazza post titled "A2 Queries". If you have doubts, please post only in this thread. The queries outside this thread or over email *will not* be entertained.

COL 226: Programming Languages
Assignment3: Type Checking and Evaluation

Submission Lifeline: Sun 25 Apr 23:59
Submission Deadline (with Late Penalty): Fri 30 Apr 23:59

---

1. (easy) Extend the language for Boolean algebra from assignment 2 to support the following:

   - Integer arithmetic with operators `PLUS, MINUS, TIMES, NEGATE` (unary operator), `EQUALS, LESSTHAN`, `GREATERTHAN`. Suitably extend the `AST` type specification.
   - Extend the expression language with the following:
     - `if` exp `then` exp `else` exp `fi`. You must raise appropriate exceptions when the first expression does not evaluate to a boolean value type or the expression in the `then` and `else` evaluate to different types.
     - `let` var = exp `in` exp `end`. This is to create temporary identifier bindings.

     Here, exp is either a valid formula from assignment 2[1] or a valid integer expression.
   - You can use the usual precedence and associativity rules (refer to C programming language rules) for integer operations
   - Parenthesis can be used to define order the evaluation over all of the integer and boolean operations.
   - Use the rules given in assignment 2 for boolean algebra.

   You may refer to the demo code shown in the class (on April 1) to implement the above mentioned tasks.

2. (medium) Provide support for functions in your language implementation. That is, you should provide support for the following concrete syntax:

   - `fn` $(x : typ) : typ \Rightarrow$ `exp`
   - `fun` $x(y : typ) : typ \Rightarrow$ `exp`

   Some examples of concrete and abstract syntax pertaining to functions are given below:

Concrete syntax	Abstract syntax
`(f 3)`	`AppExp ( VarExp ("f"), NumExp (3))`
`fn (`x `: int): int` $\Rightarrow 1$	`Fn("x", INT, INT, NumExp(1))`
`fun g (f: int` $\rightarrow$ `int): int` $\Rightarrow$ `(f 1)`	`Fun("g", "f", ARROW(INT,INT), INT, AppExp ( VarExp ("f"), NumExp (1)))`

[1] except IF THEN ELSE

Recall the datatypes `value` and `typ` defined for AST specification in our evaluator demo. You may have to carefully think about the extensions to these datatypes to provide support for functions.

3. (medium) Implement a type checker for the extended language sought in this assignment. Each expression is either *well-typed* (*i.e.*, associated with at most one type), or an exception must be raised. For instance, `if e1 then e2 else e3 end` expression is well-typed when $e1$ is a boolean type, and $e2$ and $e3$ must have the same types. To implement the type checker, you must implement a type grammar. The type grammar provides rules and *syntax-directed* derivation that are required to conclude the type checking. HINT: You may have to define a type environment and domain and range types for each operator. Finally, if the type checking goes through without any hiccups, then evaluate an input program in the extended language using the evaluation strategy call-by-value.

Submission Instruction

Submit a zip file named ⟨EntryNumber⟩.zip. You may provide a README.md that contains suggestions for the evaluator.

Important Notes

1. Do not change any of the names given in this document. You are not even allowed to change upper-case letters to lower-case letters or vice-versa.

2. We have created a piazza post titled "A3 Queries". If you have doubts, please post only in this thread. The queries outside this thread or over email *will not* be entertained.

3. Instead of using `IF-THEN-ELSE` from assignment 2, use the syntax provided here (`if` exp `then` exp `else` exp `fi`).