Second Weekly Assignment for the IPS Course

This is the text of the second weekly assignment for the DIKU course "Implementation of Programming Languages", 2023.

Hand in your solution in the form of a short report in text or PDF format, named `report.txt` or `report.pdf`, along with a `code.zip` archive consisting of whatever extra files (such as source code) are mandated by specific subtasks.

Task 1

This task refers to lexical and syntax analysis.

1) For the alphabet Σ = {o,g}, write **regular expressions** for the following languages:

a. Strings of an even length
b. Strings of an even length, whose first character is 'o' (so, in particular, the length cannot be zero)
c. Strings of an even length, in which all the 'o's (if any) come before all the 'g's (if any)

In each case, *briefly* (~1 sentence) explain how your regular expression works.

2) For the alphabet Σ = {a,b,c}, write **context-free grammars** for the following languages:

a. $\{a^n c^m b^n \mid n \geq 0, m \geq 1\}$, i.e., the language in which words start and end with an equal (possibly 0) number of consecutive 'a' and 'b' characters, respectively, and in the middle they have one or more consecutive 'c' characters, e.g., `aaaccbbb`.
b. $\{a^n b^{2n} \mid n \geq 1\}$, i.e., one or more 'a's, followed by exactly twice as many 'b's, e.g., `aabbbb`.
c. $\{a^n b^m \mid n \geq 0, m \leq n\}$, i.e, any number of 'a's followed by as many, or fewer, 'b's., e.g., `aaaaabbb`.

Remember to clearly indicate the start symbol in each case (if you have more than one nonterminal).

Again, *briefly* explain your grammars.

3) Consider the parser-implementation snippet below (lightly adapted from Fasto's `Parser.fsp`):

```
%token <(int*int)> PLUS MINUS DEQ LTH EQ
%token <(int*int)> LET IN
%token <string*(int*int)> ID

%nonassoc letprec
%left DEQ LTH
%left PLUS MINUS

%type <AbSyn.UntypedExp> Exp

%%

Exp :   ...
      | LET ID EQ Exp IN Exp %prec letprec
          { Let (Dec (fst $2, $4, $3), $6, $1) }
```

a. Explain the use of `%nonassoc letprec`, i.e., what does it accomplish? What would happen if we were to omit it? If in doubt, try it! (Also pay attention to the output from FsYacc during the build.) **Hint:** Consider what is the expected parse of a `let`-expression whose body (second subexpression) contains an infix operator?
b. What would happen if we placed the `%nonassoc letprec` declaration *between* the two `%left …` ones, instead of *above* them? Illustrate by an example.
c. Explain how the code between `{…}` in the rule works. (What does the rule produce, and from what, i.e., what are each of the following: `$1`, `fst $2`, `$3`, `$4`, `$6`, `Dec`, and `Let`?)

Task 2

This task involves making a proper lexer and parser for last week's calculator.

Last week we asked you to implement an **interpreter** for the calculator, and gave you an ad hoc lexer and parser written in F#. This approach enabled you to test your interpreter with expressions like `4`, `4 + 3`, `let x = 3 in x - 1`, etc., but hand-coding such lexers and parsers for larger languages can get laborious and error-prone.

This week we ask you to implement a **lexer** in FsLex and a **parser** in FsYacc for the same small calculator language. You'll also be using FsLex and FsYacc for the group project, so this is a good warm-up.

We hand out a zip archive `calculator.zip` along this assignment containing a template for your lexer and parser. Run `make` (or just `dotnet build`) to build the calculator, and `./calculator.sh` to run it. The template can only lex and parse numbers and parentheses, and you must implement the rest in the `Parser.fsp` and `lexer.fsl` files. (Please follow the `FIXME` comments in these files.)

Running `./calculator.sh` should lex and parse the input line and print the resulting abstract-syntax tree. For instance, staring the program and entering the string `12` should print

```
Input an expression: 12
Parse result: CONSTANT (INT 12)
```

This works in the handed out template, but e.g. entering `3 + 9` will print `lexer error`, since the `+` character is not yet recognized.

You need to extend `Parser.fsp` and `Lexer.fsl` to also lex and parse:

- Variable names are missing only in the lexer. They are already declared with a token named `VAR` in the parser, and an associated parsing rule is provided (`VAR { AbSyn.VARIABLE $1 }`).
- Infix operators (`+`, `-`, `*`) are missing in both the lexer and the parser.
- Let-expressions, sum-expressions, etc. are also missing in both the lexer and the parser

The (lexing) rules for a variable name are:

- a variable name consists of ASCII letters (uppercase and lowercase), underscores `_`, and digits (`0`, `1`,..., `9`), **but**
- it must start with one or more (uppercase or lowercase) letters, and
- a digit must be (not necessarily immediately) preceded by an underscore `_`.

(Note that these differ from the simpler rules in the previous lexer.) For example, `a`, `Ab`, `aBBBacd`, `abAcd_3`, `abS_abfG33As4` are legal variable names. The following are **illegal** variable names:

- `_a123` because it does not start with a letter,
- `ab5_A1` because `5` is not preceded by `_`.

For the parser, you should use the "parsing directives" approach; that is, supplement the original, ambiguous grammar with associativities and precedence levels for the various operations, rather than rewriting the grammar itself (as was done in the handed-out parser last week). Like in that parser, the body of a `let`-, `sum`-, etc. expression should always extend as far to the right as syntactically possible; for example, `prod x=1 to 10 of x+x` should parse as `prod x=1 to 10 of (x+x)`, not as `(prod x=1 to 10 of x)+x`

The documentation of FsLex and FsYacc is pretty rudimentary, but the syntax is very close to that of `ocamllex` and `ocamlyacc` of the OCaml programming language (on which F# is inspired); see http://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html for a useful syntax documentation page. Note that you don't have to run the `fslex` and `fsyacc` tools manually; that is taken care of by the `Calculator.fsproj` build-configuration file.

In your `code.zip`, you only need to include your final `Lexer.fsl` and `Parser.fsp`; and you definitely should *not* include any binaries. In the report, you should explain if you did anything unusual/clever that you want to highlight, or if you know of any bugs or limitations in your code, but otherwise you don't need to write anything for this task.

Hint: you may use for inspiration the lexer and the parser handed out for the group project. However, note that the tokens there contain the position at which they appear in the program—this is not needed for the weekly assignment. For example the declaration of the variable-name token in Fasto is: `%token <string*Position> ID`, while for Weekly 2, it is `%token <string> VAR`.

Task 3

This task refers to type-checking implementation.

Assume you have to implement `filter` in Fasto. `Filter` receives as parameters a function identifier and an array. The denoted function needs to be a predicate, i.e., takes exactly one argument and returns either `true` or `false`. The result of `filter(p,arr)` is a new array that contains only the elements of the input array `arr` that succeed under the predicate `p`. (The elements that are kept should appear in the same order as in the input array.)

For example, assuming predicate `fun bool gth0(int a) = a > 0`, then `filter(gth0, {-1, 2, -3, 4, 5})` should result in array `{2, 4, 5}`.

To complete this task you need to:

- a.) Write the type of `filter` (with universal quantifiers).
- b.) Write the type checking for `filter(p, arr_exp)`, where `p` is a function identifier, and `arr_exp` is an (array-producing) expression. (That is to say, write the high-level pseudocode of how to compute the result type of `filter(p, arr_exp)` together with whatever other checks are necessary to ensure that the type constraints assumed by the intuitive definition of `filter` above are respected.) Try to stay close to the notation used in the textbook; `vtable` and `ftable` denote the variable and function symbol tables, respectively.

```
CheckExp(exp, vtable, ftable) = case exp of
  filter(p, arr_exp) =>
    ... write your pseudocode here ...
```

At this point we strongly encourage you to go and implement `filter`, `scan`, `replicate` in the type-checking phase of the Fasto compiler (group project).

Task 4

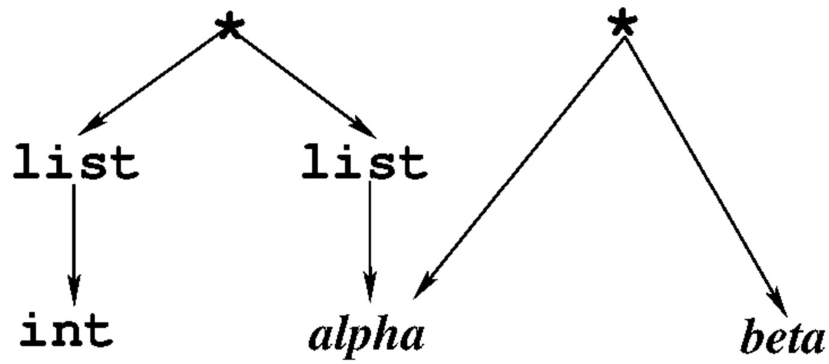
This task refers to type inference (type-expression unification), see Section "Advanced Concepts: Type Inference", from the Type Checking lecture notes.

Show how to apply the unification algorithm discussed in class to unify the type expressions

- `list(int) * list(alpha)` and
- `alpha * beta`

and explain who alpha, beta and the unified type are after unification.

The initial unification graph is shown below: note that both type expressions use the same alpha node, i.e., each type variable (Greek) appears only once in the unification graph, albeit it can be the destination of several edges. (This property is assumed by the unification algorithm!)


*
*
list
list
int
alpha
beta