KØBENHAVNS UNIVERSITET

Assignment 2

Adit (hjg708)

IPS

17. maj 2023

Task 1

1) For the alphabet $\Sigma = \{o, g\}$, write regular expressions for the following languages:

a)

Regular expression for strings of an even length:

$$(oo|og|go|gg)^*$$

The regular expression uses alternation (the | symbol) to match either 'oo', 'gg', 'go', or 'og'. The * operator allows for zero or more repetitions of this pattern, ensuring an even length for the string.

b)

Regular expression for string of an even length, whose first character is 'o':

$$(oo|og)(oo|go|og|gg)^*$$

The regular expression starts with 'oo' or 'og' followed by alternation of 'oo', 'go', 'og' or 'gg'. The * operator allows for zero or more, ensuring repetition an even length string.

c)

Regular expression where all 'o's (if any) come before all 'g's (if any):

$$(oo)^*(gg)^*|o(oo)^*g(gg)^*$$

The $(oo)^*(gg)^*$ part is for even number of 'o' and 'g'. The latter part is for odd number of 'o' and 'g'.

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

a)

$\{a^n\, c^m\, b^n \quad | \quad n \geq 0, m \geq 1\}$, **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.**

$$T \rightarrow aTb$$
$$T \rightarrow cT$$
$$T \rightarrow c$$

By stating that 'T' can directly become 'c', I am ensuring that c^m for $m \geq 1$ holds.

b)

$\{a^n\, b^{2n} \quad | \quad n \geq 1\}$, **i.e., one or more 'a's, followed by exactly twice as many 'b's, e.g., aabbbb.**

$$T \rightarrow aTbb$$
$$T \rightarrow abb$$

As $n \geq 1$ I need to make sure that 'T' does not return empty.

c)

$\{a^n\, b^m \quad | \quad n \geq 0, m \leq n\}$, **i.e, any number of 'a's followed by as many, or fewer, 'b's., e.g., aaaaabbb.**

$$T \rightarrow aTb$$
$$T \rightarrow aT$$
$$T \rightarrow \epsilon$$

I have to make sure that I can add 'a' without also adding 'b'. And now that $n \geq 0$ I am allowed to let 'T' return empty immediately.

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

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?

By introducing %nonassoc letprec, we ensure that the "let"construct has higher precedence than the infix operators, allowing it to be properly parsed and grouped together with its associated expressions.

Omitting %nonassoc letprec would make the "let"construct have the same precedence level as the infix operators, leading to incorrect parsing. If the body of a let-expression contains an infix operator, the parser would prioritize the infix operator over the "let"construct, altering the intended parsing.

b)

What would happen if we placed the %nonassoc letprec declaration between the two %left ... ones, instead of above them?

If we were to place the %nonassoc letprec declaration between the two %left declarations, as in:

```
%left DEQ LTH
%nonassoc letprec
%left PLUS MINUS
```

In the example of

$$\text{let}\ \ x = 0 \text{in}\ \ x == 1$$

the parser will not know how to correctly associate the == operator inside the body of the let-expression, it will instead put the let-expression inside the body of DEQ expression. This is of course wrong, as let-expression should not be inside its own body expression.

Wrong:

```
OPERATE (BDEQ, LET_IN ("x", CONSTANT (INT 0), VARIABLE "x"), CONSTANT (INT 1))
```

Correct:

```
LET_IN ("x", CONSTANT (INT 0), OPERATE (BDEQ, VARIABLE "x", CONSTANT (INT 1)))
```

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?)

The code constructs the abstract syntax tree for a let-expression. $1 is the position of the let token, fst $2 retrieves the identifier, $3 represents the equality token, $4 represents the left expression, $6 represents the right expression, Dec constructs a declaration operator, and Let constructs a let-expression operator.

Task 2

See the zipped file 'code.zip' for the implementation.

Task 3

a)

Write the type of filter (with universal quantifiers).

$$\forall \alpha : \ (\alpha \rightarrow \mathit{bool}) * [\alpha] \rightarrow [\alpha]$$

b)

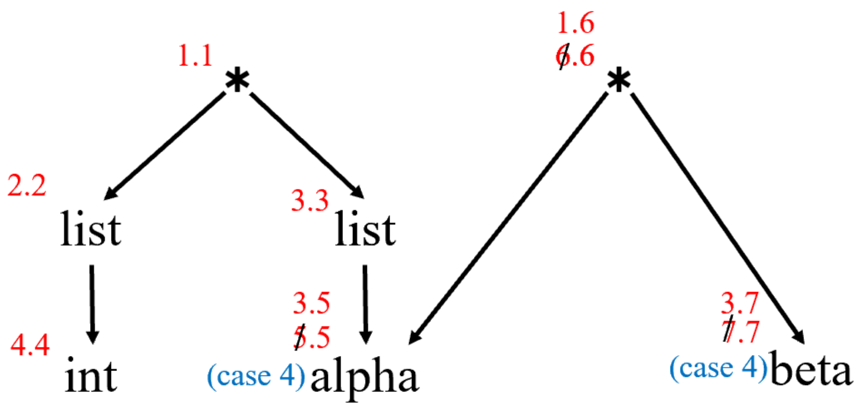
```
CheckExp(exp, vtable, ftable) = case exp of
  filter(p, arr_exp) =>
    # Type check the array expression
    arr_type = CheckExp(arr_exp, vtable, ftable)

    # Look up the type of the predicate function
    pred_type = lookup(ftable, name(p))

    # Check if the array expression is of type Array(tElem)
    case arr_type of
      Array(tElem) =>
        # Check if the predicate function has the form
        # arg_type -> result_type
        case pred_type of
          arg_type -> result_type =>
            # Check if the predicate argument type matches the
            # array element type and the predicate result type is bool
            if arg_type = elem_type && result_type = bool then
              # Return Array(elem_type) as the result
              # type of the filter expression
              Array(elem_type)
            else
              # Raise an error for type mismatch
              error()
          _ => error()
      _ => error()
```

Task 4

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

$\mathit{list}(\mathit{int}) * \mathit{list}(\mathit{alpha})$
$\mathit{alpha} * \mathit{beta}$

To begin to unify the two trees, we examine the two top nodes. Since they are not in the same set and both have 2 children, they are merged, and the algorithm is subsequently called on their left children and then on their right children. For the recursive call on the left children, it falls into case 4, as alpha is a variable. Consequently, they are merged, and the algorithm returns true. Consequently, alpha becomes a list(int). In the second recursive call, it also falls into case 4, as beta is a variable. Hence, they are merged as well, and the algorithm returns true. Consequently, beta becomes a list(alpha), where alpha is a list(int), therefore beta becomes a list(list(int)).


1.1
*
2.2
list
3.3
list
4.4
int
3.5
5.5
(case 4) alpha
1.6
6.6
*
3.7
7.7
(case 4) beta