IPS2

Rasmus Ladefoged DHT579

May 2023

Task 1

1

In the first grammar we use the definition, that it needs to be divisible by two, there a^2n fufills that.

For the second one we use the same definition as in the first one, but we also add the condition that it and a second condition, namely that the first first character should be zero must be met. $a^{2\cdot n}\ \ \&\ \ a_0 = 0$.

For the third one we again use the even number definiton. But we also use the raised * notation to indicate that there are zero or more o's coming before any g's. If we were to use raised $+$, then there would have to be one or more of the characters, but that is not the case. Therefore our grammar will look like this: $a^2no^*g^*$

2

For the first one we need to ensure that there is always an equal number of a an b characters, therefore they each have the same variable for amount. They also need to be on opposite sides of c, therefore we place c in the middle with it's own variable for amount. It will therefore look like this:

$$N_i - > (N_a N_n)(N_c N_m)(N_b N_n)$$
$$N_i - > N_c N_m$$

What the second line indicates is the event that there are no a or b, but we must always have at least one c, so there is no need for a condition with only a and b.

For the second one we need to have a followed by twice as many b's. Therefore we just set b's variable to n*2.

$$N_i - > (N_a N_n)(N_b N_n \cdot 2)$$

For the final one we know that n is at least 0 and m is always less than or equal n. Therefore we have three possible scenarioes, one where n and m are greater than 0, one where n is greater than zero and m is not and one where both are.

$$N_i - > (N_a N_n)(N_b N_m)$$
$$N_i - > N_a N_n$$
$$N_i - >$$

3

%nonassoc letprec is used to set the precedence level for let and allow it to go first. If we failed to include it then it would not properly assign the let variables first, which could result in errors. It is therefore used to correctly parse let expressions.
The reason why %nonassoc letprec is placed at the top, is that it takes precedence over everything else, so if you were to put it in the middle of the %left lines, then it would again not parse correctly. So something like let x = 2 in x+2 might result in error as the + can take precedence over let and therefore x will not be assigned a value.

Finally what the code between the curly brackets mean is basically a the same as the line above, so each $ value corresponds to one of the values above in the given order, so $1 is ID, then fst $2 is the position, $3 is the EQ, $4 is the expression being bound to the variable and $6 is the second Exp. Dec is then the entire variable and let denotes that Dec is a variable. So the function basically say let dec = Exp in.

Lexer and parser implementation

To start with we will discuss the parser, as that was the first one I implemented as the lexer requires the parser to test.

Firstly I implemented the different tokens, this includes types like INT, variables, the different operators, such as plus and minus, let, sum, max and so on. These tokens are essentially all of our keywords in the language, that we need handle.

Next we add the precedence rules, this is the order we need to operate in. So first the most important is LET, as we need to establish the variables first, then we do plus and minus next and finally we do times. This is standard math with the order of operations.

Next we do the operators, these are based on the tokens we listed earlier. The way they have been implemented is that I have used the AbSyn fucntions like RSUM and RMAX to implement the various functions. We have then give it the input that it requires, so for example for plus, we send an EXP followed by a PLUS followed by another exp. This is then parsed with AbSyn to generate the correct message.

This is then repeated for each step, based on last week's understanding of how the functions work.

In the lexer I started out with establishing all the keywords and inserting them in a match case in which we send the different keywords to the Parser in the correct form. This is why I started by implementing the Parser.

Next we establish all the different symbols and operations. Here the most noteworthy is for vaiable names, where we need to fulfill the rules stated in the assignemnt text. To do this we establish that the first character must be a letter and after that we have less restrictions. This is done by adding a second array after the first one.

In terms of testing, then I have reused the calculator test from the first assignment and checked if they return the expected result. I was not able to find any errors and it behaved as I would expect for every test.

Type for filter

$filter : (\forall -> bool) -> [\forall] -> [\forall]$
What this does is that it takes an input, which can be anything, char, bool, int, whatever. It also has a bool, which is a check for if the condition has been met.

If the depending on what the bool says, then it can then insert that element into and array of the type that the element is. The function then returns that type.

pseudocode for filter

```

filter (function p, arr_exp)
    if (type (p.EXP_3) != type(arr_exp[0])
        fail("Types don't match")
    if (arr_exp[0] == type(char))
        CharFilter(p, arr_exp)
    else if (arr_exp[0] == type(int))
        IntFilter(p, arr_exp)
    else
        fail("Type not supported")

int[] IntFilter(function p, int[] arr)
    int[] arrFilter
    int filterIndex = 0
    vtable function
    for int i = 0 to arr.length
        ftable.update(p(arr[i]))
        function = ftable.get(p)
        if function then
            arrFilter[filterIndex] = arr[i]
            filterIndex++
    return arrFilter

char[] CharFilter(function p, char[] arr)
    char[] arrFilter
    int filterIndex = 0
    vtable function
    for int i = 0 to arr.length
        ftable(p(EXP_1)) = arr[i]
        function = ftable.get(p)
        if function then
            arrFilter[filterIndex] = arr[i]
            filterIndex++
    return arrFilter
```

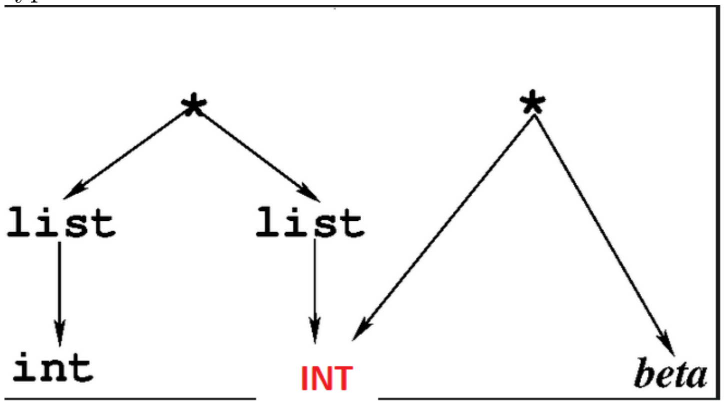
What this code does is that it first checks if both types (EXP_1 and EXP_3) match. If not it will fail. Then it determines the type of arr[0] as that should always be populated. Then it matches that type with the correct filter type.

These filters accepts a function identifier p and an array arrm which we pass down from the main filter. Then it creates a new array for the filtered elements and a filterindex for the filtered array.

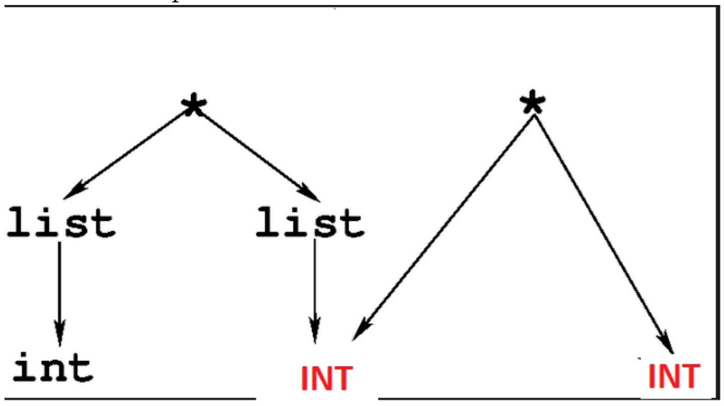
Then it enters a for loop that will run through the entire original array. Then for each iteration, we will update the ftable with arr[i] on p(EXP_1). This is the equivelant to a in gth0(int a) = a > 0. Then we load the result of p into a variable called function, which we then check if it's true. If it is, then we add arr[i] to our filtered array at the filterIndex and increment the filter index. At the end of the loop we return the filtered array.

Type inference

Alpha and beta are currently not bound by any type, but we know that the first list that we add with list alpha is of the type int, therefore we can infer that alpha must also be of type int as the types need to match



Next we can repeat the same step for Beta, which will also become int because of the same reasons as Alpha:


*
list
list
*
int
INT
INT