Department of Computer and Information Science

TDT4165 - Programmeringsspråk

Introduction to Language Theory

Author:
Aksel Hjerpbakk

September, 2020

Table of Contents

1 Introduction

All code can be found on github here

2 Task 1

I already implement the list functions, so I am skipping this. See my previous hand-in for details.

3 Task 2 and 3

It should be sufficient to document my process in the code of my submission through execution output and comments. See README.md for instruction on running the code.

If you run my code, you should be seeing the following output:

```
$ ozengine main.ozf
------------Task 2a-----------
[[49] [50] [51] [43] [100]]

------------Task 2b-----------
[number(1) number(2) number(3) operator(plus) command(duplicate)]

------------Task 2c-----------
[number(2)]

------------Task 2d-----------
[number(1) number(2) number(3)]
[number(1) number(5)]

------------Task 2e-----------
[number(1) number(5) number(5)]

------------Task 2f-----------
[number(~4)]

------------Task 2g-----------
[number(1)]

------------Task 3a-----------
(3 + 3)

------------Task 3b-----------
((3 - (10 * 9)) + 0.3)
```

4 Task 4

4.1 Formally describe the regular grammar of the lexemes in task 2

Using my implementation of lexeme generation, all character sequences separated with space between are valid lexemes. There is no guarantee that generated lexemes are valid tokens and therefore a valid expression. We only know whether they are when we tokenize the lexemes. Having said that, we define **valid** lexemes as followed:

Using regex we can define lexemes as followed:

$$L = ([1-9] + ([\.1-9]+)?|[+\-*\/]|[pdi\hat{}])$$

Using BNF notation we can define lexemes as followed:

$$\begin{aligned} &< lexeme > ::= < operator > | < number > | < command > \\ &< operator > ::= " + "|" - "|" * "|/ \\ &< number > ::= < integer > | < integer > . < integer > \\ &< integer > ::= < digit > | < integer >< digit > \\ &< digit > ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9" \\ &< command > ::= "p"|"d"|"i"|"\hat{}" \end{aligned} \tag{1}$$

4.2 Describe the grammar of the infix notation in task 3 using (E)BNF. Beware of operator precedence. Is the grammar ambiguous? Explain why it is or is not ambiguous?

My current implementation can be defined using BNF as followed:

$$\begin{aligned} &< infix > ::= < number > \\ &\quad | \ (< infix > | < number >< operator >< infix > | < number >) \\ &< operator > ::= " + "|" - "|" * "|/ \\ &< number > ::= < integer > | < integer > "." < integer > \\ &< integer > ::= < digit > | < integer >< digit > \\ &< digit > ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9" \end{aligned} \tag{2}$$

This definition resolves ambiguity by using parenthesis. It does not account for operator precedence either, as it does not need to. Like discussed in the lecture, this is the most primitive solution, and comes with the drawback of requiring verbose expressions. In mathematics these parenthesis are not required because the grammar is more complex.

4.3 What is the difference between a context-sensitive and a context-free grammar?

Context sensitive grammar is a subset of context-free grammar. By looking at the definition from the lectures we see that Context-free is defined as followed:

$$v ::= \gamma$$

Where v is any variable from V and gamma is any sequence of variables from V and symbols from S. While context sensitive is defined as:

$$\alpha v \beta ::= \alpha \gamma \beta$$

Where v is any variable from V and alpha, beta and gamma is any sequence of variables from V and symbols from S

So there is a limitation to the composition of the expression in a context sensitive language compared to a context free one.

4.4 You may have gotten float-int errors in task 2. If you haven't, try running 1+1.0. Why does this happen? Why is this a useful error?

Mixing types i.e a *float* and a *integer* can be bad for many reasons. Implicit type mixing is a possible sign of programmer error and might lead to undefined behaviour.

Specifically, allowing 1 + 1.0 means allowing ambiguity as the compiler will have to assert the type for the sum, either it will have to guess (i.e always use float) or do runtime analysis to determine a suitable type which leads to overhead, heisenbugs ... etc.

Floats can lead to rounding issues and are more prone to overflows. Integers does not have a decimal component. One can claim that the compilers best way of handling such cases is to report this as a user error at *compile time* and let the programmer resolve the ambiguity explicitly by casting or using other methods.