Module 2: Recursive descent, exceptions, function objects and more

In this module you will write a calculator which reads and calculates arithmetic expressions. You will get a very simple version to start with which should be built upon with several features.

Concepts covered: parsing with ”recursive descent”, exceptions, function objects.

The program comprises the second mandatory task.

Instructions

1. The zip-file belonging to the module contains the following files:
   (a) `MA2.py` which is the embryo of the program you are going to write,
   (b) `MA2tokenzer` which contains a help class that you just have to use and
   (c) `MA2test.txt` which contains test data that you can use as input to your calculator. You can provide it with more test cases.
2. When your solution is approved by a teacher or assistant:
   (a) Fill in the name, email and assistants / teachers who reviewed the assignment.
   (b) Upload the file `MA2.py` in Studium under MA2.

Note that you should upload only **one** file and it should be a `.py` file that is directly executable in Python! We do not accept other formats (word, pdf files, Jupyter notebook, . . .) and no submissions via email.

Note: You may collaborate with other students, but you must write and be able to explain your own code. You may not copy code neither from other students nor from the Internet except from the places explicitly pointed out in this lesson. Changing variable names and similar modifications does not count as writing your own code. Since the information is included as part of the examination, we are obliged to report failures to follow these rules.

Syntax analysis with recursive descent

Recursive methods are especially suitable when processing data that are recursively structured. An example of this is ordinary arithmetic expressions. Consider e.g. following expression

$$1 + (2.3 + 4) \cdot 3.42 + 0.34 \cdot (1.2 + 13 \cdot 0.7)$$

There are a number of rules for how this should be interpreted, for example ”multiplication before addition”, ”parentheses first” and ”from left to right when priority is equal”. It is not easy (but doable) to implement these rules in a program which reads and interprets an expression.

However, we can define expressions in a more structured way that in itself contains the priority rules:

An *expression*
: is a sequence of one or more *terms* with a plus or minus character in between.

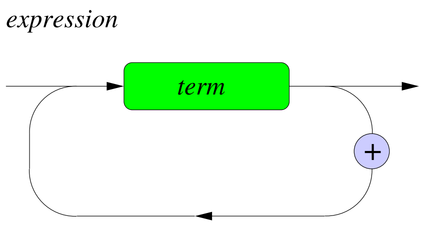
A *term*
: is a sequence of one or more *factors* with a multiplication or division character in between.

A *factor*
: is either a *number* or an *expression* surrounded by parentheses.

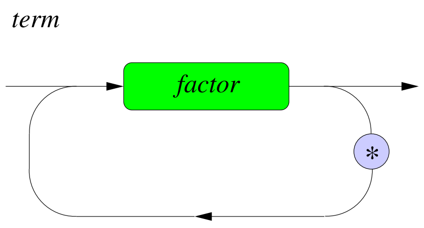
Note that this definition of is indirectly recursive.

For the sake of simplicity, we will limit ourselves for the time being to addition, multiplication and parentheses. This does not change the structure — only the details of the code.

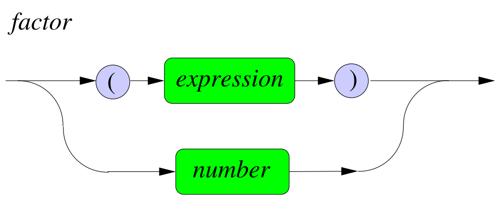
The definition can be illustrated graphically with the following *syntax charts*. There is *pseudocode* to the right of the charts describing how the chart can be translated into code where we mix programming language constructions such as `if` and `while` statements with commented running text for parts we wait to specify until later.



```
def expression():
    sum = term()
    while # next is '+':
        # Get passed '+'
        sum += term()
    return sum
```



```
def term():
    sum = factor()
    while # next is '*':
        # Get passed '*'
        sum += factor()
    return sum
```



```
def factor():
    if # next is '(':
        # Get passed '('
        result = expression()
        # Get passed ')'
    else:
        result = # read number
    return result
```

We have not drawn a chart for *numbers* but assume that someone else can find out how a number is defined using digits, decimal point etc.

Note the recursion again: an *expression* is defined by *terms* which are defined by *factors* which, maybe, are defined by an *expression*.

All charts contain ”junctions”. It is the next character (plus, times and parentheses) which decides which path we should choose. If, for example, in *expression* there is a plus character after the first term, we will go back and take a new term. If there is no plus-character, the expression is done **no matter what comes** — it’s someone else’s problem to handle!

For the time being, we assume that only syntactically correct expressions are entered.

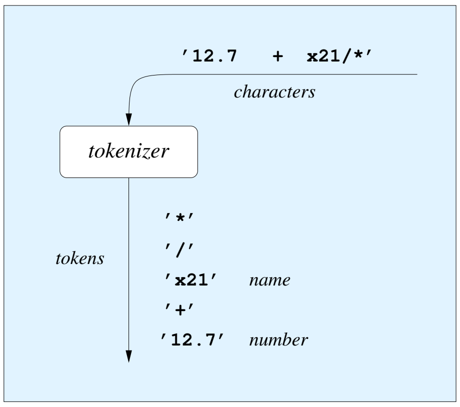
The input to the program is basically a sequence of characters (10 digits as well as `'.'`, `'+'`, `'*'`, `'('` and `')'`). We want to handle *numbers*, not individual digits, decimal point etc. For this we will use a so-called *tokenizer*.

Tokenizer

A *tokenizer* is used to group individual characters to larger units such as words and numbers, so-called *tokens*. Different applications have different rules for how the tokens are formed. Of course, a tokenizer for programming code has different rules than a tokenizer for natural language.

The Python module `tokenize` is useful for our purposes but to simplify the usage, we provide a wrapper class `TokenizeWrapper`. An object from the wrapper class is coupled to a character string (such as an input line) and produces a sequence of tokens. There is a *current token*, methods to get information about this and a method of stepping up to the next token.

This figure exemplifies how a tokenizer receives the string `'12 .7 + x21 / * '` with 15 characters and produces a sequence of 5 different string tokens.



The most important methods in the `TokenizeWrapper` are:

`TokenizeWrapper(line)`	Creates a tokenizer object and connects it to `line`
`next()`	Advances to the next token (number, character, word, ...)
`has_next()`	Returns `True` if there are more tokens on the line, otherwise `False`
`get_current()`	Returns the current token as a string
`get_previous()`	Returns the previous token as a string
`is_number()`	Returns `True` if the current token is a number, otherwise `False`.
`is_name()`	Returns `True` if the current token is a name, otherwise `False`.
`is_at_end()`	Returns `True` if end of line otherwise `False`.

Note that `next()` is the *only* method that advances the tokenizer. The other methods just provide information about the current token.

Here is a small demo code. The output is shown in the adjacent pane.

```python
def main():
    line = 'hej hopp + 234 * 26.4  ='
    wtok = TokenizeWrapper(line)
    while wtok.has_next():
        print(wtok.get_current(), end='\t')
        if wtok.is_name():
            print('NAME')
        elif wtok.is_number():
            print('NUMBER')
        elif wtok.is_newline():
            print('NEWLINE')
        else:
            print('OPERATOR')
        wtok.next()
    print(wtok.get_current())
```

```
hej     NAME
hopp    NAME
+       OPERATOR
234     NUMBER
*       OPERATOR
26.4    NUMBER
=       OPERATOR
        NEWLINE
```

Using these methods, we can express the commented running text of the pseudocode in Python code. We will then get following simple calculator:

```python
def expression(wtok):
    result = term(wtok)
    while wtok.get_current() == '+':
        wtok.next()                     # bypass +
        result = result + term(wtok)
    return result

def term(wtok):
    result = factor(wtok)
    while wtok.get_current() == '*':
        wtok.next()                     # bypass *
        result = result * factor(wtok)
    return result

def factor(wtok):
    if wtok.get_current() == '(':
        wtok.next()                     # bypass (
        result = expression(wtok)
        wtok.next()                     # bypass )
    else:                               # should be a number
        result = float(wtok.get_current())
        wtok.next()                     # bypass the number
    return result

def main():
    print("Very simple calculator")
    while True:
        line = input("Input : ")
        wtok = TokenizeWrapper(line)
        if wtok.get_current() == 'quit':
            break
        else:
            res = expression(wtok)
            print('Result: ', res)
    print("Bye!")

if __name__ == "__main__":
    main()
```

What can go wrong?

The program above can be downloaded from ou2_0.py. Try, in turn, these incorrect input lines:

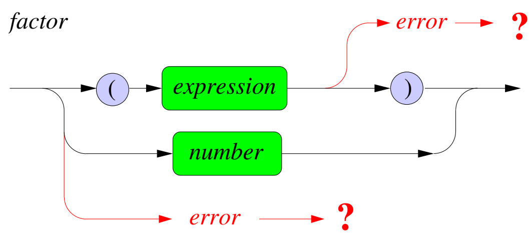
Try to understand the output!

```
3*2 ++ 1
2-3
-1
1 2 3
1**2
2*(1+3-+4
```

There are two places in the code where things can go wrong and both are when dealing with factors:

1) The factor begins neither with a left parenthesis nor with a number.

2) There will be no matching right parentheses after *expression*.



The errors can easily be detected by the code:

```
def factor(wtok):
    if wtok.get_current() == '(':
        wtok.next()            # bypass (
        result = expression(wtok)
        if wtok.get_current() == ')':
            wtok.next()        # bypass )
        else:
            pass # What shall we do here
    elif wtok.is_number():
        result = float(wtok.get_current())
        wtok.next()            # bypass the number
    else:
        pass  # What shall we do here?
    return result
```

It is possible to produce an error message here but how should the program continue? The function is expected to return a number that is to be used higher up in the code. Once an error has been detected, it is probably pointless to continue calculating the expression. However, we do not want the program to exit but to print error message, ignore the rest of the expression and continue with a new expression on a new line.

To handle such situation *exceptions* are well suited.

Exceptions

If you already know about exceptions i Python, you can skip to the next section.

Exceptions are a general mechanism that Python uses to handle errors. The terminology is that when Python detects an error then " *an exception is thrown* ".

This is what it looks like, for example, if you divide by 0 in the command window:

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

Division by 0 has thus created an exception with the name `ZeroDivisionError`.

Other examples of exceptions that Python has are `TypeError` which is created by `math.log ('a')` and `ValueError` created by `math.log (-1)`.

If you do not do anything else, the program will be interrupted with error prints. However, you can let the program *catch* the exceptions and decide for yourself what should happen. For this, constructions with the words `try` and `except` are used.

Example:
This code

```python
from math import sqrt

while True:
    x = input("Give a positive number: ")
    try:
        y = sqrt(int(x))
    except ValueError:
        print(f"Squarehead! {x} is not a positive number!")
    else:
        print(f"The square root of {x} is {y}")
```

can result
in this
conversation:

```
Give a positive number: 2
The square root of 2 is 1.4142135623730951
Give a positive number: -1
Squarehead! -1 is not a positive number!
Give a positive number: 4
The square root of 4 is 2.0
Give a positive number: a
Squarehead! a is not a positive number!
Give a positive number: 9
The square root of 9 is 3.0
Give a positive number:
```

If the code between `try` and `except` gives causes an exception of type `ValueError`, the statements in the `except`-block is executed.

If no error occurs, the statements in the `else`-block are executed.

Of course, this code could easily have been obtained with a `if` statement without exception. One point with the exception is that you can handle several errors in one place that also may be in a different function than where they occured.

Suppose, for example, that we call a function `compute` which in itself does many root calculations and which may also call other functions who do such. We then do not need to test on each individual operation, but we capture all of them in one place.

An exception is an *object* from a *exception class*. The code which creates the exception can enter more information about the error.

It is also possible to capture several types of errors.

In this code there can be three errors on line 4 (two `ValueError` and one `ZeroDivisionError`) and one on line 9 (also a `ValueError`)

```python
 1  from math import sqrt, log
 2
 3  def compute(x):
 4      return sqrt(x) + log(2-x) + 1/x
 5
 6  while True:
 7      x = input('Give a value: ')
 8      try:
 9          y = compute(int(x))
10      except (ValueError,
11              ZeroDivisionError) as e:
12          print('*** Error:', e)
13      else:
14          print('Result: ', y)
```

```
Give a value: 1
Result:  2.0
Give a value: 2
*** Error: math domain error
Give a value: -2
*** Error: math domain error
Give a value: a
*** Error: invalid literal for int() with base 10: 'a'
Give a value: 0
*** Error: division by zero
Give a value:
```

A common situation is that you want to handle different types of errors in different ways. You can then use several `except`-blocks.

```python
from math import sqrt, log

def compute(x):
    return sqrt(x) + log(2-x) + 1/x

while True:
    x = input('Give a value: ')
    try:
        y = compute(int(x))
    except ValueError as e:
        print('*** Illegal argument!', e)
    except ZeroDivisionError as e:
        print('*** Division by zero!', e)
    else:
        print('Result: ', y)
```

The variable `e` is a reference to the exception object and when printed its `__str__`-method will be called.

Read more about exceptions and error handling, for example here or find a YouTube channel to watch - there are several to choose from!

Error handling in the calculator

To handle syntax errors in the calculator, we will make our *own* exception class. It must be written as a subclass of the built-in class `Exception`. Exception classes are easy to write. In this case we only need:

```python
1 class SyntaxError(Exception):
2     def __init__(self, arg):
3         self.arg = arg
```

Thus, the class has only one constructor which receives and saves a message. The word `Exception` in parentheses in line 1 indicates that this is a subclass of the `Exception` class.

To cause an exception, we use the command raise. We use raise in the factor function and handle the exceptions in main:

```python
def factor(wtok):
    if wtok.get_current() == '(':
        wtok.next()                     # bypass (
        result = expression(wtok)
        if wtok.get_current() == ')':
            wtok.next()                 # bypass )
        else:
            raise SyntaxError("Expected ')'")
    elif wtok.is_number():
        result = float(wtok.get_current())
        wtok.next()                     # bypass the number
    else:
        raise SyntaxError('Expected number or (')
    return result

def main():
    print("Very simple calculator")
    while True:
        line = input("Input : ")
        wtok = TokenizeWrapper(line)
        try:
            if wtok.get_current() == 'quit':
                break
            else:
                result = expression(wtok)
                if wtok.is_at_end():
                    print('Result: ', result)
                else:
                    raise SyntaxError('Unexpected token')
        except SyntaxError as se:
            print("*** Syntax: ", se.arg)
            print(f"Error ocurred at '{wtok.get_current()}'" +
                  f" just after '{wtok.get_previous()}'")
        except TokenError:
            print('*** Syntax: Unbalanced parentheses')
    print('Bye!')
```

We have also added handling of a TokenError that can occur in case of unbalanced parentheses.

Now you can begin working with the downloaded file ou2.py

Specification of the calculator

The task is to continue to develop the program above to handle several operations.

We start by exemplifying with a run of the program:

```
Input : 3-1+3-1/2                         # Float type
Result: 4.5
Input : 1 - (5-2*2)/(1+1) - (-2 + 1)      # Float type
Result: 1.5
Input : sin(3.14159265)                   # Standard functions
Result: 3.5897930298416118e-09
Input : cos(PI)                           # Predefined constant
Result: -1.0
Input : log(exp(4*0.5 - 1))               # Standard functions
Result: 1.0
Input : 1 + 2 + 3 = x                     # Variable. Assignment goes RIGHT!
Result: 6.0
Input : x/2 + x
Result: 9.0
Input : (1=x) + sin(2=y)                  # Assignments
Result: 1.9092974268256817
Input : vars                              # Prints variable values
   E     : 2.718281828459045
   PI    : 3.141592653589793
   ans   : 1.9092974268256817
   x     : 1.0
   y     : 2.0
Input : 1+2
Result: 3.0
Input : 2*ans + 5                         # Result of last computation
Result: 11.0
Input : ans
Result: 11.0
Input : z                                 # Undefined variable
*** Evaluation error:  Undefined variable: 'z'
Input : mean(1,6,2,4,9,8)
Result: 5.0
Input : mean(1,6,2,4,9,8)
Result: 5.0
Input : 1 + max(sin(x+y), cos(1), log(0.5))
Result: 1.5403023058681398
Input : fib(3)                            # Fibonacci number. Note integer!
Result: 2
Input : fac(5)                           # Factorial. Note integer
Result: 120
Input : fib(-2)                           # Illegal argument
*** Evaluation error:  Argument to fib is -2.0. Must integer >= 0
Input : fac(2.5)                          # Illegal argument
*** Evaluation error:  Argument to fac is 2.5. Must integer >= 0
Input : fib(100)                          # Large integer
Result: 354224848179261915075
Input : fac(40)                           # Larger integer
Result: 815915283247897734345611269596115894272000000000
Input : 2 ++ 4*ans/0                      # Syntax error before evaluation error
*** Syntax error:  Expected number, word or '('
Error ocurred at '+' just after '+'
Input : ans/0 + * x                       # Evaluation error begfore syntax error
*** Evaluation error:  Division by zero
```

Comments:

1. The program can handle expressions with constants, variables, the arithmetic operators `+` , `-` , `*` and `/` as well as a number of functions, including `sin`, `cos`, `exp` och `log`.

2. The usual priority rules apply and parentheses can be used to change the calculation order.

3. Variable assignment is made *from left to right*.

   Example: The expression

   `1+2*3 = y`

   will give the value `7` to `y` .

4. Variable assignment can be used in subexpressions

   Example:

   ```
   (2=x) + (3=y=z) = a
   ```

   will give values to `x`, `y`, `z` och `a`.

5. The predefined variable `ans` contains the value of the last calculated complete expression. This variable can be used in the next expression. Example:

   ```
   Input : 1+1
   Result: 2.0
   Input : ans
   Result: 2.0
   Input : exp(2)
   Result: 7.38905609893065
   Input : ans
   Result: 7.38905609893065
   Input : ans + 3
   Result: 10.38905609893065
   Input : 3 + ans
   Result: 13.38905609893065
   ```

6. The program must detect and diagnose errors. Example:

   ```
   Input : 1++2
   *** Error. Expected number, name or '('
   *** The error occurred at token '+' just after token '+'
   Input : 1+-2
   Result: -1.0
   Input : 1--2
   Result: 3.0
   Input : 1**2
   *** Error: Unexpected token
   *** The error occurred at token '**'
   Input : 1/0
   *** Error. Division by zero
   Input : 1+2*y-4
   Result: 1.0
   Input : 1+2*k-4
   *** Error. Undefined variable: k
   Input : 1+2=3+4**x - 1/0
   *** Error. Expected variable after '='
   *** The error occurred at token '3.0' just after token '='
   Input : 1+2*(3-1 a
   *** Error: Unbalanced parentheses
   *** The error occurred at token 'a' just after token '1.0'
   Input : 1+2+3+
   *** Error: Expected number, name or '('
   *** The error occurred at token '*EOL*' just after token '+'
   Input :
   ```

7. The command `vars` shows all stored variables with values and the command `quit` ends the run. Example:

   ```
   Input : vars
    ans     : 13.38905609893065
    E       : 2.718281828459045
    PI      : 3.141592653589793
    x       : 1.0
    y       : 2.0
   Input : quit
   Bye!
   ```

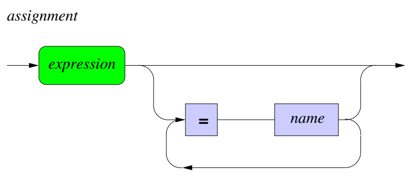
8. The `file` command causes the input data to the calculator to be read from a file with the specified name. When all lines from the file are read, the calculator continues to

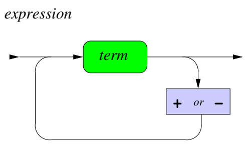
read from the terminal.

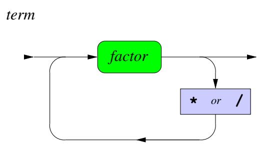
Hint: Implement this feature from the beginning and let it read from a ”default file” (code test.txt for example). This makes it very easy to collect and run test cases!

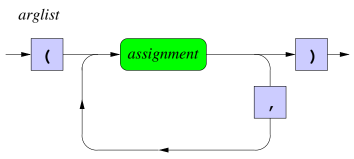
Syntax charts

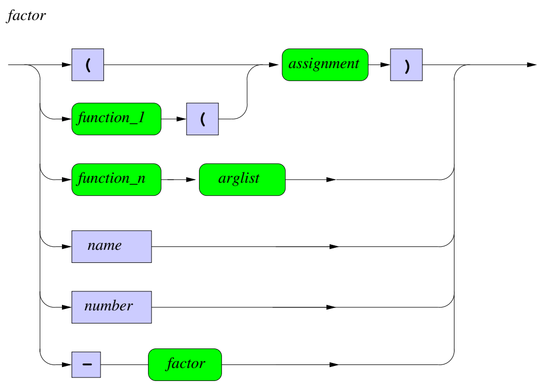
The syntax of the expressions is defined by the following charts:





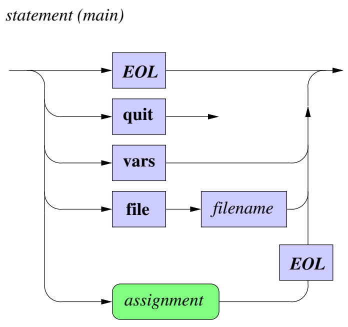






The commands `vars`, `file` och `quit` are handled in the `main`-function according to this figure:

This function also handles exceptions.



Functions

In addition to the functions `sin`, `cos`, `exp` och `log` the following functions must be available:

Function	Meaning	Example
fib(n)	The n:th Fibonacci number. Integer argument.	fib(0) = 0, fib(1) = 1, fib(3) = 2
fac(n)	n! Integer argument.	fac(0) = 1, fac(1) =1, fac(3) = 6
sum($a_1, a_2, \ldots$)	Sum of arguments	sum(1,2,3,4) = 10
max($a_1, a_2, \ldots$)	Largest argument	max(1,2,3,4) = 4
min($a_1, a_2, \ldots$)	Smallest argument	min(1,2,3,4) = 1
mean($a_1, a_2, \ldots$)	Mean of arguments	mean(1,2,3,4) = 2.5

As the syntax charts show, there are two different groups of functions:

1. Those with exactly one argument: `sin`, `cos`, `exp`, `log`, `fib` and `fac` which are called *function_1* in the charts.
2. Those with several arguments: `min`, `max`, `sum` and `mean`) which are called *function_n* in the charts.

Program design

Program parts

The program must have the following components (with specified names):

- The class `TokenizeWrapper`. This class is given and you may or may not amend it as you see fit.
- A number of functions which handle the parsing and calculations. There should be a function for each syntactic element (i.e. *assignment*, *expression*, *term*, *factor*, ...) These functions should exactly follow the syntax charts!
- Functions for the built in `fib`, `fac`, ... Of course, you can directly use functions built into Python (`min`, `max`, ...)

Variable values

To keep track of the values of variables, a dictionary should be used.

Generally speaking, one should avoid global variables but this is a clear candidate to be made global because it must be available to many of the functions. The alternative is to create it in `main` and send it as a parameter to all functions.

Also add the constants `PI` and `E` and the predefined variable `ans` to the dictionary! This makes the handling as uniform as possible. However, the variable `ans` needs to be updated in the `main` function.

Functions

Functions should be implemented using a dictionary with *function names* as keys and *function objects* as values. It is a good idea to use different dictionaries for *function_1* and *function_n*.

To add a new function, there must be *no changes* in the code needed other than adding a new key-value pair to the dictionary. The function definition itself must of course be included.

This example demonstrates how function objects can be stored and transferred.
Note that the example deals with *function objects*, not function values!

```
def demo(f, x):
    return f(x)

print(demo(sqrt, 4))
print(demo(log, 1))

foo = sqrt
print(foo(9))          # Prints the square root
                          of 9

d = {'f':sqrt, 'g':log}
print(d['f'](25))      # Prints the square root
                          of 25
```

More about handling functions as objects can be found at Corey Schafer's YouTube-lesson!

Error handling

The user of the program can make two types of errors: *syntax errors* and *calculation errors*. The expression `x + * y` is an example of incorrect syntax while the expression `log(2*x - x - x)` is example of a calculation error (because the argument to `log` will be 0).

In the first expression the error can be detected *before* the summation is to be done but in in the second case, the argument must be calculated before the error can be detected. In the latter case, information about the current token is usually not relevant.

Syntax errors

If the user writes an expression which does not match the syntax according to the syntax charts (e.g. `a ++ b` or `sin + 4` or `2 * 3) + 8`) it is a *syntax error*.

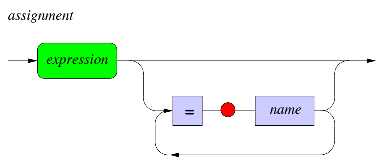
The program should detect such errors and deal with them with an exception of type `SyntaxError`.

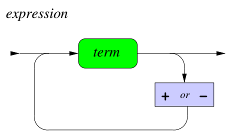
It can be tempting to look for errors everywhere but it only makes sense in a few places:

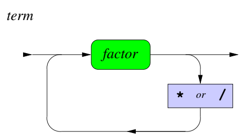
- In *assignment*: the equals sign is not followed by a variable name.
- In *factor*:
  1. The right parenthesis after *assignment* is missing.
  2. The left parenthesis after *function_1* is missing.
  3. None of the five cases, i.e. not a parenthesis, not a *function_1* or *function_n* name, not a number, not a minus sign and not a name.

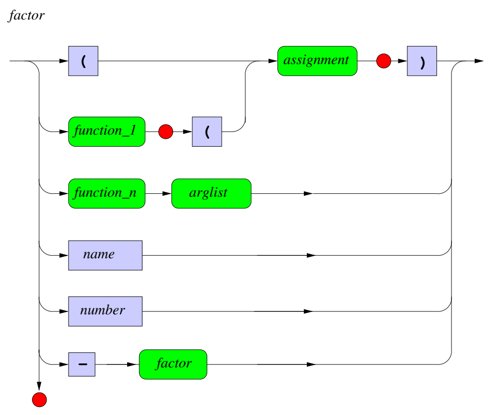
- In *arglist*:
  1. The argument list does not start with a left parenthesis
  2. An argument is not followed by a comma or a parenthesis.
- I *main*: There are more items on the line.

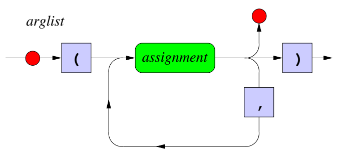
The red circles indicates places to check the syntax.

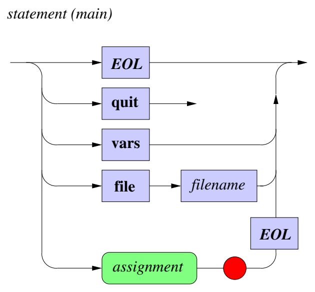












We must also check that the top level assignment is followed by an end of line:

In *expression* and *term* nothing can go wrong and these methods should therefore not look for errors.

When the code detects a syntax error, it should throw a `SyntaxError`.

Important coding hint: Include the error checks *from the beginning*! This greatly facilitates your own troubleshooting and testing of the program!

Evaluation errors

Even if an expression is syntactically correct, it may be incomputable. Examples of such errors:

- Division by 0
- Argument to `log` less than or equal to 0.
- Too many arguments (e.g. `sin(2,3,4)`. Depending on the implementation this could

also be considered a `SyntaxError`.

- Incorrect type of argument (e.g. `fib (2.5)`)

These errors should be handled with the exception class `EvaluationError`. If it is an incorrect argument (i.e. all examples except the first) then the function name and the argument should be specified in the error message.

Generally

Errors are caught in the `main` function. When an error occurs, the rest of the line is ignored and a new expression is begun.

The `file` command

It is easiest to load the lines from the file into a list and let the `main` function retrieve lines from the list until it is empty.

If you want to allow filenames with type extension"(such as `test.txt`, `tast.xyz`, . . .) then you should write your own function to parse these because the tokenizer leaves names such as those as *three* tokens. However, this is optional.

As stated before, it is good to implement this feature in an early stage. You can then successively add test cases for each feature you implement instead of manually entering test data each time.