Modul 2: Recursive descent, undantag, funktionsobjekt mm.

I den här modulen ska du skriva en kalkylator som läser in och beräknar aritmetiska uttryck. Du kommer få ett en mycket enkel version som ska byggas ut med flera faciliteter.

Begrepp som tas upp: parsning med ”recursive descent”, undantag, funktionsobjekt.

Programmet utgör den andra obligatoriska uppgiften.

Instruktioner

1. Den zip-fil som hör till modulen innehåller filerna:
   (a) `MA2.py` : Ett embryo till det program du ska skriva och ladda upp.
   (b) `MA2tokenzer` : En hjälpklass som du inte behöver göra något med.
   (c) `MA2init.txt` : En fil som med testdata som läses automatiskt när kalkylatorn startas. Förse den med flera testfall allteftersom du implementerar de olika faciliteterna.
   (d) `MA2_test.py`: Unit-tester. Kör den när du tror att du är klar med alla uppgifter!
2. Du får inte använda andra paket än de som redan är importerade.
3. Du måste ha de funktioner som finns med i den nedladdade filen och de som nämns i dokumentet. De måste ha exakt samma namn och samma parametrar!
4. Implementera och testa en sak i taget! Lägg dina tester i den bifogade startfilen (`MA2init.txt`). Tag inte bort dessa tester när du tycker att du är klar med en sak eftersom fortsatt arbete kan introducera nya fel.
5. Filen med unit-tester kan du köra när du tycker att du är klar. Du kan använda den tidigare genom att kommentera bort det som testar saker du inte har implementerat.
6. Skriv feltester med riktiga utskrifter från början! Det underlättar din egen felsökning av programmet.
7. När din lösning är godkänd av en lärare eller assistent:
   (a) Fyll i namn, mail och assistenter/lärare som granskat uppgiften
   (b) Ladda upp filen `MA2.py` i Studium under MA2. Ladda *inte* upp någon av de andra filerna

Observera att du bara ska ladda upp **en** fil och det ska vara en `.py`-fil som är direkt körbar i Python! Vi accepterar inga andra format (word, pdf-filer, Jupyter notebook, . . .) och inga inlämningar via mail.

Observera: Du får samarbeta med andra studenter, men du måste skriva och kunna förklara din egen kod. Du får inte kopiera eller skriva av kod vara sig från andra studenter eller från nätet förutom från de ställen som explicit pekas ut i denna lektion. Att byta variabelnamn och liknande modifikationer räknas inte som att skriva sin egen kod.
Du får inte sprida dina lösningar varken direkt till andra studenter eller allmänt på nätet — även medhjälp till fusk räknas som fusk.
Eftersom uppgifterna ingår som moment i examinationen är vi skyldiga att anmäla brott mot dessa regler.
Allt material som vi tillhandahåller är upphovsrättsligt skyddat och får inte spridas.

Syntaxanalys med ”recursive descent”

Rekursiva metoder är speciellt lämpliga när de data man skall behandla i sig är rekursivt uppbyggda. Ett exempel på detta är vanliga aritmetiska uttryck. Betrakta t.ex. följande uttryck

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

Det finns ett antal regler för hur detta skall tolkas av typ ”multiplikation före addition”, ”parenteser först” och ”från vänster till höger vid lika prioritet”. Det är inte alldeles enkelt (men naturligtvis väl genomförbart) att realisera dessa regler i ett program som läser och tolkar ett uttryck.

Vi kan emellertid definiera uttryck på ett mer strukturerat sätt som i sig innehåller prioritetsreglerna:

Ett *uttryck* (eng. *expression*)
: är en sekvens av en eller flera *termer* med plus- eller minustecken mellan.

En *term*
: är en sekvens av en eller flera *faktorer* med gånger- eller divisionstecken mellan.

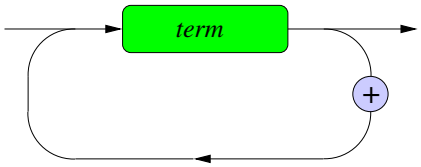
En *faktor* (eng. *factor*)
: är antingen ett *tal* eller ett *uttryck* omgivet av parenteser.

Observera att denna definition av är indirekt rekursiv.

För enkelhetens skulle begränsar vi oss tills vidare till addition, multiplikation och parenteser. Detta ändrar inget på strukturen — bara på detaljer i koden.

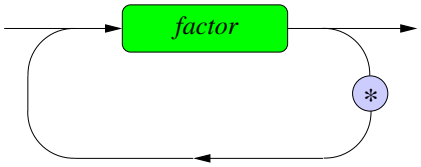
Definitionen kan illustreras grafiskt med nedanstående *syntaxdiagram*. Bredvid diagrammen finns *pseudokod* som visar hur diagrammen kan omsättas i kod. I pseudokoden blandar vi typiska programspråkskonstruktioner såsom `if`- och `while`-satser med naturligt språk för delar som vi kan vänta med att exakt specificera hur de ska uttryckas. Här har vi skrivit dessa delar som kommentarer.

expression



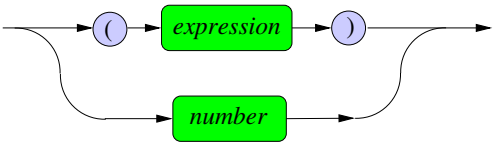

```
def expression():
    sum = term()
    while # next is '+':
        # Get passed '+'
        sum += term()
    return sum
```

term




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

factor




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

Vi har inte ritat någon figur för *number* (tal) utan förutsätter att någon annan håller reda på hur ett tal definieras med hjälp av siffror, decimalpunkt mm.

Observera igen rekursionen: *expression* definieras med *term* som defineras med *factor* som definieras med *expression*.

Alla figurerna innehåller ”vägskäl”. Det är det närmast kommande tecknet (plus-, gånger- och parentestecken) som avgör vilken väg vi skall välja. Om det t ex i *expression* kommer ett plustecken när en term har behandlats så skall vi gå tillbaka och ta en ny term. Om det *inte* kommer ett plustecken är uttrycket klart **oavsett vad som kommer** — det är någon annans problem att hantera det!

Vi förutsätter tills vidare att bara syntaktiskt korrekta uttryck matas in.

Indata till programmet är i grunden en följd av tecken (10 siffror samt `'.'`, `'+'`, `'*'`, `'('` och `')'`). I strukturerna vill vi hantera *tal*, inte enskilda siffror, decimalpunkt etc. För detta ska vi använda en så kallad *tokenizer*.

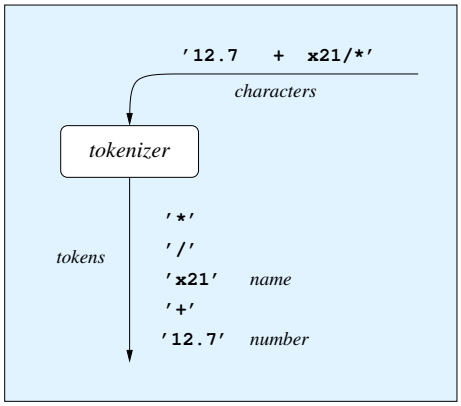
Tokenizer

En *tokenizer* används för att gruppera enskilda tecken till större enheter som ord och tal, så kallade *token*. Olika tillämpningar har olika regler för hur token ska sättas ihop. En tokenizer för programkod har naturligtvis andra regler än en tokenizer för naturligt språk.

Python-modulen `tokenize` är användbar för våra ändamål men för att förenkla användandet tillhandahåller vi en omslagsklass `TokenizeWrapper`. Ett objekt ur omslagsklassen kopplas

till en teckensträng (t.ex. en inläst rad) och lämnar ifrån sig en följd av tokens. Det finns en *aktuell token*, metoder att få information om denna och en metod att stega fram till nästa.

Vidstående figur exemplifierar hur en tokenizer tar emot strängen `'12.7  +  x21/*'` med 15 tecken och lämnar ifrån sig en följd av 5 strängar med olika ”tokens”.



De mest väsentliga metoderna i klassen är:

`TokenizeWrapper(line)`	Skapar en tokenizerobjekt och kopplar den till `line`
`next()`	Stegar fram till nästa token (tal, tecken, ord, . . .)
`has_next()`	Returnerar `True` om det finns fler token på raden, annars `False`
`get_current()`	Returnerar aktuell token som en sträng
`get_previous()`	Returnerar föregående token som en sträng
`is_number()`	Returnerar `True` om aktuell token är ett tal, annars `False`.
`is_name()`	Returnerar `True` om aktuell token är ett namn, annars `False`.
`is_at_end()`	Returnerar `True` om slut på raden annars `False`.

Observera att det är *bara* är metoden `next()` som flyttar fram. De övriga metoderna returnerar bara information om den aktuella token. Upprepade anrop till övriga metoder ger alltså samma resultat — för att komma vidare måste `next` anropas.

Här är ett liten demokörning som resulterar i vidstående utskrifter

```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
```

Med hjälp av dessa primitiver kan vi uttrycka pseudokodens naturliga språk i Python-kod. Det ger då följande enkla kalkylator:

```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()
```

Vad kan gå fel?

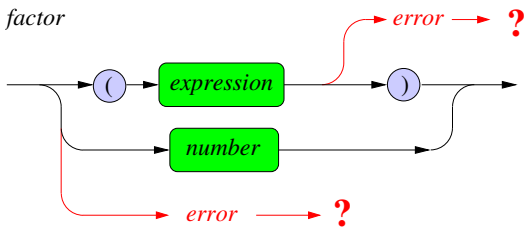
Ovanstående program finns på filen MA2micro.py bland de filer som du laddat ned. Prova i tur och ordning följande felaktiga till programmet och se vad som händer och förstå varför.

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

Det finns två ställen i koden där det kan gå fel och båda är vid hantering av faktorer:

1) Faktorn inleds varken med en vänsterparentes eller med ett tal.

2) Det kommer inte en matchande högerparentes efter *expression*.



Det är lätt låta koden upptäcka felen:

```
1  def factor(wtok):
2      if wtok.get_current() == '(':
3          wtok.next()            # bypass (
4          result = expression(wtok)
5          if wtok.get_current() == ')':
6              wtok.next()        # bypass )
7          else:
8              pass # What shall we do here
9      elif wtok.is_number():
10         result = float(wtok.get_current())
11         wtok.next()            # bypass the number
12     else:
13         pass  # What shall we do here?
14     return result
```

Det går ju bra att göra en felutskrift på dessa ställen men vad ska ska programmet fortsätta? Funktionen förväntas ju returnera ett tal som kommer användas högre upp i uttrycket.

När ett fel har upptäckts så är det förmodligen meningslöst att fortsätta att beräkna uttrycket. Vi vill inte att programmet avslutas utan vi vill ge ett felmeddelande, ignorera resten av uttrycket och fortsätta med ett nytt uttryck på en ny rad.

För att hantera detta passar *undantag* eller, på engelska, *exception* bra.

Felhantering i kalkylatorn

För att hantera syntaxfel i kalkylatorn gör vi en *egen* undantagsklass. Den ska skrivas som en underklass till den inbyggda klassen `Exception`. Undantagsklasser är enkla att skriva.

Detta räcker för våra behov:

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

Klassen är given i den nedladded koden. Den har alltså bara en konstruktor som tar emot och sparar ett meddelande. Ordet `Exception` inom parentes på rad 1 anger att detta är en subklass till klassen `Exception`.

(I någon av de inspelade lektioner sägs att det behövs ytterligare en parameter till `init`-funktionen men det gäller inte längre.)

För att ”framkalla” ett undantag används kommandot `raise`. Vi lägger in det i `factor`-funktionen samt hantering av dessa i `main`:

```
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!')
```

Vi har också lagt till hantering av ett `TokenError` som kan uppstå vid obalanserade parenteser.

Specifikation av kalkylatorn

Uppgiften går ut på att fortsätta att utveckla det nedladded programmet (MA2.py) för att klara flera operationer.

Vi börjar med att exemplifiera med en körning av programmet:

```
Input : 3-1+3-1/2                          # Flyttalsaritmetik
Result: 4.5
Input : 1 - (5-2*2)/(1+1) - (-2 + 1)       # Flyttalsaritmetik
Result: 1.5
Input : sin(3.14159265)                    # Standardfunktioner
Result: 3.5897930298416118e-09
Input : cos(PI)                            # Fördefinierad konstant
Result: -1.0
Input : log(exp(4*0.5 - 1))                # Standardfunktioner
Result: 1.0
Input : 1 + 2 + 3 = x                      # Variabler. Tilldelning åt HÖGER!
Result: 6.0
Input : x/2 + x
Result: 9.0
Input : (1=x) + sin(2=y)                   # Tilldelningar
Result: 1.9092974268256817
Input : vars                               # Skriver variabelvärden
   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                          # Resultatet av senaste beräkningen
Result: 11.0
Input : ans
Result: 11.0
Input : z                                  # Odefinierad variabel
*** 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)                             # Fibonaccital. Obs heltal!
Result: 2
Input : fac(5)                             # Facultet. Obs heltal
Result: 120
Input : fib(-2)                            # Illegalt argument
*** Evaluation error:  Argument to fib is -2.0. Must integer >= 0
Input : fac(2.5)                           # Illegalt argument
*** Evaluation error:  Argument to fac is 2.5. Must integer >= 0
Input : fib(100)                           # Stort heltal
Result: 354224848179261915075
Input : fac(40)                            # Större heltal
Result: 815915283247897734345611269596115894272000000000
Input : 2 ++ 4*ans/0                       # Syntaxfel före beräkningsfelet
*** Syntax error:  Expected number, word or '('
Error ocurred at '+' just after '+'
Input : ans/0 + * x                        # Beräkningsfel före syntaxfelet
*** Evaluation error:  Division by zero
```

Kommentarer:

1. Programmet skall hantera uttryck med konstanter, variabler, de aritmetiska operatorerna `+`, `-`, `*` och `/` samt ett antal funktioner, bl. a. `sin`, `cos`, `exp` och `log`.

2. De vanliga prioritetsreglerna skall gälla och parenteser skall kunna användas för att ändra beräkningsordning på vanligt sätt.

3. Variabeltilldelning görs *från vänster till höger*.

   Exempel: Uttrycket

   ```
   1+2*3 = y
   ```

   skall tilldela `y` värdet `7`.

4. Variabeltilldelningar skall kunna göras i deluttryck.

   Exempel:

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

   skall ge värden till `x`, `y`, `z` och `a`.

5. Den fördefinierade variabeln `ans` skall innehålla värdet av det senast beräknade fullständiga uttrycket. Denna variabel kan användas i nästa uttryck. Exempel:

   ```
   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. Programmet skall upptäcka och diagnostisera fel. Exempel:

```
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 :
```

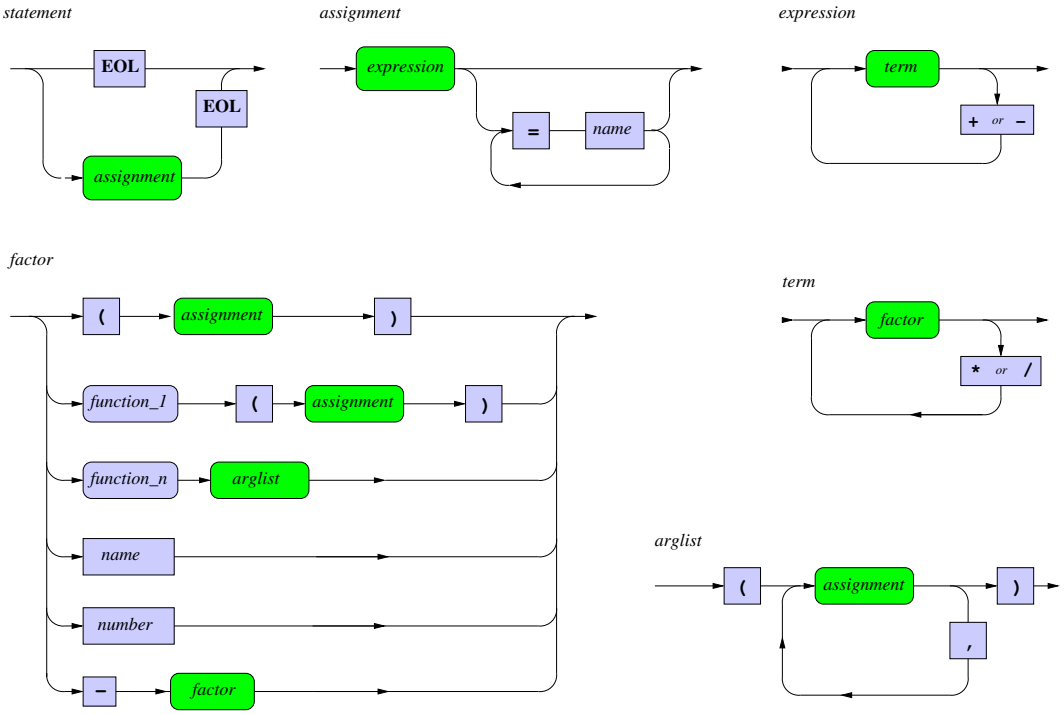
7. Kommandot `vars` visar alla lagrade variabler med värden och kommandot `quit` avslutar körningen.

   Exempel:

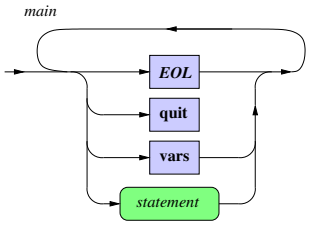
```
Input : vars
 ans    : 13.38905609893065
 E      : 2.718281828459045
 PI     : 3.141592653589793
 x      : 1.0
 y      : 2.0
Input : quit
Bye!
```

Syntaxdiagram

Syntaxen för uttrycken definieras av följande diagram:



Kommandona `vars` och `quit` hanteras i `main`-funktionen enligt denna figur:



Denna funktion hanterar också undantag.

Funktioner

Förutom funktionerna `sin`, `cos`, `exp` och `log` ska följande funktioner finnas:

Funktion	Betydelse	Exempel på värden
fib(n)	Det n:te Fibonaccitalet. Heltalsargument.	fib(0) = 0, fib(1) = 1, fib(3) = 2
fac(n)	$n!$ Heltalsargument.	fac(0) = 1, fac(1) =1, fac(3) = 6
sum$(a_1, a_2, \ldots)$	Summan av argumenten	sum(1,2,3,4) = 10
max$(a_1, a_2, \ldots)$	Största argumentet	max(1,2,3,4) = 4
min$(a_1, a_2, \ldots)$	Minsta argumentet	min(1,2,3,4) = 1
mean$(a_1, a_2, \ldots)$	Medelvärdet av argumenten	mean(1,2,3,4) = 2.5

Se också de inledande exemplen!

Som syntaxdiagrammen visar finns det två grupper av funktioner:

de med exakt ett argument: `sin`, `cos`, `exp`, `log`, `fib` och `fac`

och de som kan ha flera argument: `min`, `max`, `sum` och `mean`.

Programdesign

Programdelar

Programmet skall ha följande komponenter (med angivna namn):

- Klassen `TokenizeWrapper`. Klassen är given men du får komplettera den om du tycker att det behövs.
- Ett antal funktioner som hanterar parsningen och beräkningarna. Det ska finnas en funktion för varje varje syntaktiskt element (alla gröna rutor dvs *statement*, *assignment*, *expression*, *term*, *factor* *arglist*) Dessa funktioner skall fungera som syntaxdiagrammen beskriver.
- Funktionsdefinitioner för de inbyggda funktionerna `fib`, `fac`, ... Du kan naturligtvis utnyttja alla i Python inbyggda funktioner (`min`, `max`, . . .)

Variabelvärden

För att hålla reda på variablers värden skall ett lexikon användas. Detta lexikon ska skapas i `main` och skickas som parameter till alla parser-funktioner.

Lägg in konstanterna `PI` och `E` samt den fördefinierade variabeln `ans` lexikonet! Det gör hanteringen av så uniform som möjligt. Variabeln `ans` måste dock uppdateras i `main`-funktionen.

Funktioner

Funktioner ska implementeras med hjälp av lexikon med *funktionsnamn* som nycklar och *funktionsobjekt* som värden. Använd ett lexikon *function_ 1* för funktioner med en parameter och ett annat lexikon *function_ n* för funktioner som kan ha flera parametrar.

För att lägga till en ny funktion ska alltså inga *inga ändringar* i koden behöva göras förutom att lägga till ett nytt nyckel-värdepar i lexikonet. Själva funktionsdefinitionen måste naturligtvis vara med.

Detta exempel demonstrerar hur funktionsobjekt kan lagras och skickas. Observera att exemplet bollar med *funktionsobjektet*, inte funktionsvärden!

```
1  def demo(f, x):
2      return f(x)
3
4  print(demo(sqrt, 4))
5  print(demo(log, 1))
6
7  foo = sqrt
8  print(foo(9))          # Skriver roten ur 9
9
10 d = {'f':sqrt, 'g':log}
11 print(d['f'](25))      # Skriver roten ur 25
```

Mer om hantering om funktioner som obekt finns på Corey Schafers YouTube-lektion!

Felhantering

Användaren av programmet kan göra två typer av fel: *syntaxfel* och *beräkningsfel*. Uttrycket `x + * y` är ett exempel på felaktig syntax medan uttrycket `log(2*x - x - x)` är exempel på ett beräkningsfel (eftersom argumentet till `log` kommer bli 0).

I det första uttrycket kan felet upptäckas *innan* man ska göra summationen men i det andra fallet måste man beräkna argumentet innan felet kan upptäckas.

Syntaxfel

Om användaren skriver ett uttryck som inte stämmer med syntaxen enligt syntaxdiagrammen (t ex `a + +b` eller `sin + 4` eller `2*3) + 8`) är det ett *syntaxfel*.

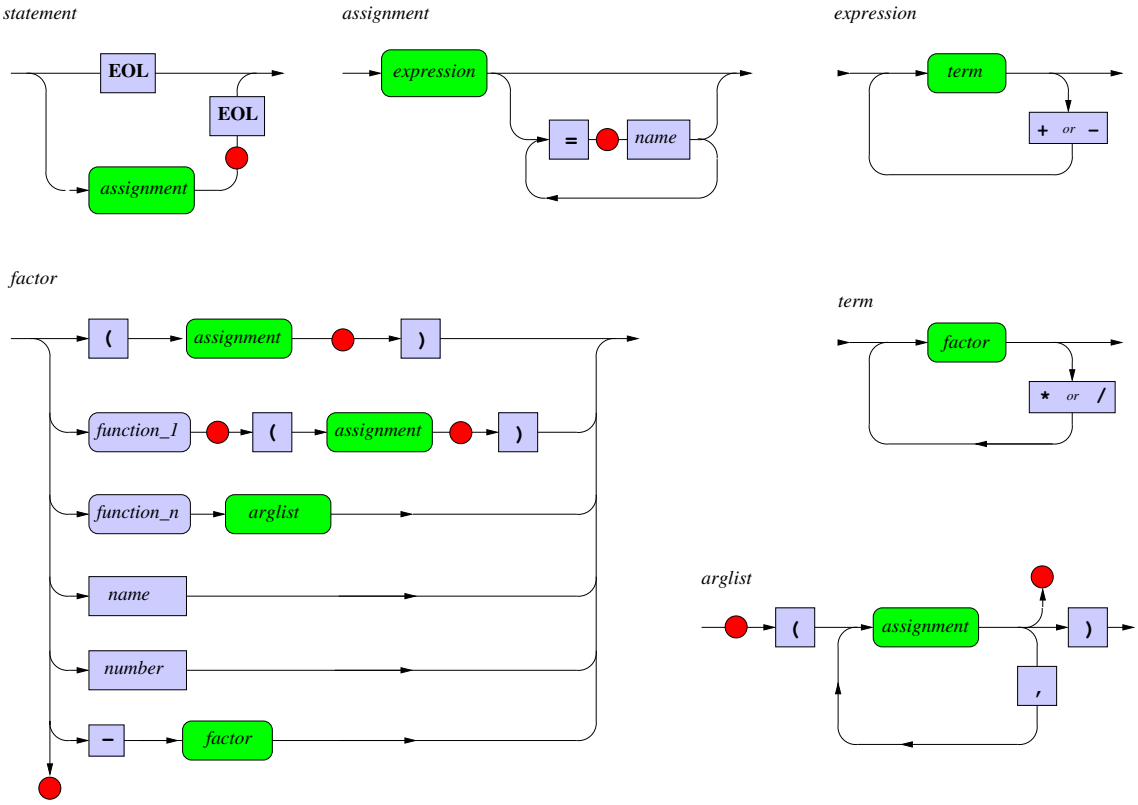
Programmet ska upptäcka sådan fel och hantera dem med undantaget `SyntaxError`.

Det kan vara frestande att leta efter fel överallt men det är bara meningsfullt på några ställen:

- I *assignment*: likhetstecknet följs inte av ett variabelnamn.
- I *factor*:
  1. Högerparentesen efter *assignment* saknas.
  2. Vänsterparentes efter *function_1* saknas.
  3. Inget av de fem fallen stämmer dvs inte vänsterparentes, inte funktionsnamn, inte tal, inte minustecken och inte ett namn.
- I *arglist*:
  1. Argumentlistan börjar inte med vänsterparentes
  2. Ett argument följs ej av ett kommatecken eller en högerparentes.
- I *statement*: Parsern har inte kommit till radslut.

De röda cirklarna i nedanstående figur visar var det är meningsfullt att se om det är fel.

Tänk inte på vad som kan komma *efter* ett syntaktiskt element (*statement*, *assignment*, *term*, ...) utan på vad som kan ingå *i* elementet!



I uttrycken för *expression* och *term* finns det inget som kan gå fel och dessa metoder ska alltså inte leta efter fel. När koden upptäcker ett syntaxfel ska den kasta ett `SyntaxError`.

Viktigt kodningstips: Lägg in felkontrollerna *från början* med tydliga diagnoser! Detta underlättar i hög grad er egen felsökning och testning av programmet!!

Beräkningsfel

Även om ett uttryck är syntaktiskt korrekt är det inte säkert att det går att beräkna. Exempel på beräkningsfel:

- Division med 0
- Argument till `log` mindre än eller lika med 0.
- Felaktig typ på argument (t.ex. `fib(2.5)`)

Dessa fel ska hanteras med klassen `EvaluationError` som du själv ska skriva. Om det är ett felaktigt argument (dvs alla exempel utom det första) så ska funktionens namn och det aktuella argumentet anges i felmeddelandet.

Information om var parsern står är oftast inte relevant vid evalueringsfel och bör därför inte skrivas ut.

Generellt

Felen fångas i `main`-metoden. När ett fel inträffat ignoreras resten av raden och ett nytt utttryck påbörjas.

Innan du redovisar

- Kontrollera att hela unittesten går igenom! Se till att tester som du eventuellt har kommenterat bort kommer med igen!
- Kom ihåg att parsern ska använda sig av lexikon med namn och funktionsobjekt! För att lägga till en ny funktion ska det, förutom att skriva funktionsdefinitionen, räcka med att lägga till en rad i lexikonet *function_1* respektive *function_n*. Räkna med att du får en sådan uppgift under redovisningen!
- Se till att du förstår hur undantagshantering fungerar!