Süntaks, semantika ja pragmaatika

Suhtlemiskeelel on kaks omadust:
- keele vorm (**süntaks** - *syntax*)
- vormiga seotud tähendus (**semantika** – *semantics*).

> **Keele süntaks** on reeglite hulk, mille abil määratakse grammatiliselt õiged keele vormid.

Süntaktiliselt õigel lausel on kaks tähendust:
- mida pidas silmas lause autor (semantika)
- kuidas sai aru lause vastuvõtja (pragmaatika)

> **Keele semantika** - lõplik hulk reegeleid, mis peab ära kirjeldama lõpmatu hulga erinevate erinevate programmide tähenduse.
>
> **Keele pragmaatika** - kuidas (kui kergesti) inimene programmeerimiskeelest aru saab e kas ja kuidas on keeles võimalik koostada loetavaid programme.

Süntaks on tavaliselt seotud sellega, kas arvuti (keele kompilaator) tunnistab toodud stringi (sümbolite jada) sobivaks keele lauseks või mitte. Keele grammatika abil tuleb kõrvaldada kõik süntaktilised mitmetähenduslikkused.
Kuid **sematilised** ja **pragmaatilised mitmetähenduslikkused** on siiski võimalikud ja sellest tulebki, et alati ei ole sama see, mida programmeerija mõtles ja see, kuidas arvuti aru sai.

Võime enda jaoks mõelda ka nii:
- **Süntaks** – õigekiri – millised stringid on keeles lubatud
- **Semantika** – loogika – mida tähendavad lubatud avaldised/stringid, mida programmi kokku kirjutada tohib. Näiteks, mida tähendab `a += b`
- **Pragmaatika** - mis on hea ja halb antud programmeerimiskeeles. Kuidas keele lauseid koostada praktikas. Kui mugav on programmi kirjutada. Kui kerge on omandada tehnilisi võtteid programmi kirja panemiseks. Kui palju keel aitab vältida raskel leitavaid vigu.

Semantilised mudelid:
- **kompilaatori mudel** *(compiler model)* - keele tähenduseks on vastavad sihtkeele stringid, mida kompilaator tekitab, semantika kirjelduseks on täidetavad käsud (vajalik kompilaatorile)
- **interpreteerimismudel** *(interpreter model e operational model)* - milline on seos sisendi ja täidetavate käskude vahel, selle mudeli kirjeldamiseks peab olema süsteem programmi struktuuride ülesmärkimiseks (vajalik programmeerijale)
- **matemaatiline mudel** *(mathematical model)* - milline on seos sisendi ja väljundi vahel (vajalik kasutajale)

Backus-Nauri normaalkuju (*Backus Naur Form* (BNF, algselt *Backus Normal Form*))
John Backus'e ja Peter Naur'i koostatud kirjaviis programmeerimiskeelte formaalseks kirjeldamiseks (ALGOL 60 jaoks 1959, 1960).

> **BNF** on **metakeel** kontekstivabade grammatikate üleskirjutamiseks. St see on viis, kuidas formaalseid keeli üles kirjutada.

BNF-i abil kirjeldatakse programmeerimiskeele grammatikat selliselt, et pole mingit kahtepidi arusaamist. Teooriad selliste grammatikate kohta ütlevad, et tema abil peab olema võimalik

genereerida automaatselt parser BNF-i abil kirjeldatud keele jaoks. Sellist süsteemi kutsutakse kompilaatori kompilaator ja ilmselt kõige kuulsam neist on YACC (UNIX-s) Alustades ühest lähtesümbolist hakatakse vastavalt produktsioonidele (grammatikale ehk tuletusreeglitele) neid asendama kuni jõutakse terminalideni. Terminalid ehk terminaalsed sümbolid on kõik seed märgid ja sõnad, mis programmeerimiskeeles on lubatud kasutada (`if = jne`). Kõikvõimalikud terminalide jadad, mis on saadavad reegleid kasutades ongi (süntaktiliselt õiged) programmid vastavas keeles. BNF-i kasutatakse kontekstivabade grammatikate üleskirjutamiseks, samuti käskude ja kommunikatsiooni protokollide jaoks.

BNF on metakeel (keel keele jaoks) ja temal on olemas sümbolid ja omad reeglid.

BNF-i meta-sümbolid:
::= *"is defined as"* (on kirjeldatud, kui ...)
| *"or"* (või)
< > *"category names"* (süntaksireeglite, ka mitteterminaalsete sümbolite nimed, kasutatakse mitteterminaalsete sümbolite eristamiseks terminaalsetest)

Erinevate autorite poolt on BNF-i juurde toodud uusi metasümboleid eesmärgiga teda kergemini loetavaks muuta. Näiteks **EBNF**-i nime all laiendas BNF-i Niklaus Wirth. Selle kohta kehtib ka standard ISO-14977.

Täiendavad metasümbolid (näiteid):
Fakultatiivsed *(optional)* osad pannakse [ja] vahele:

```
<if_statement> ::= if <boolean_expression> then
                     <statement_sequence>
                   [ else
                         <statement_sequence> ]
                   end if ;
```

Korduvad elemendid - 0 või enam (repetitive items) pannakse metasümbolite { ja } vahele:

```
<identifier> ::= <letter> { <letter> | <digit> }
```

Reegel võib olla ka rekursiivne (järgnev ja eelnev näide on samad)

```
<identifier> ::= <letter> | <identifier> [ <letter> | <digit> ]
```

Ühesümbolilised terminalid pannakse " ja " vahele eristamaks neid metasümbolitest. Samas kasutatakse " ja " ka pikema terminali eristamiseks “if”. Samuti kasutatakse ülakomasid:

```
<statement_sequence> ::= <statement> { ";" <statement> }
```

Trükitud materjalides tehakse terminalsümbolitel ja mitteterminalidel vahet ka nii, et terminalid trükitakse **poolpaksus** kirjas. Mitteterminalidel jäetakse sulud <> ümbert ära.

Näide Pythoni BNF-kirjelduse algusest:

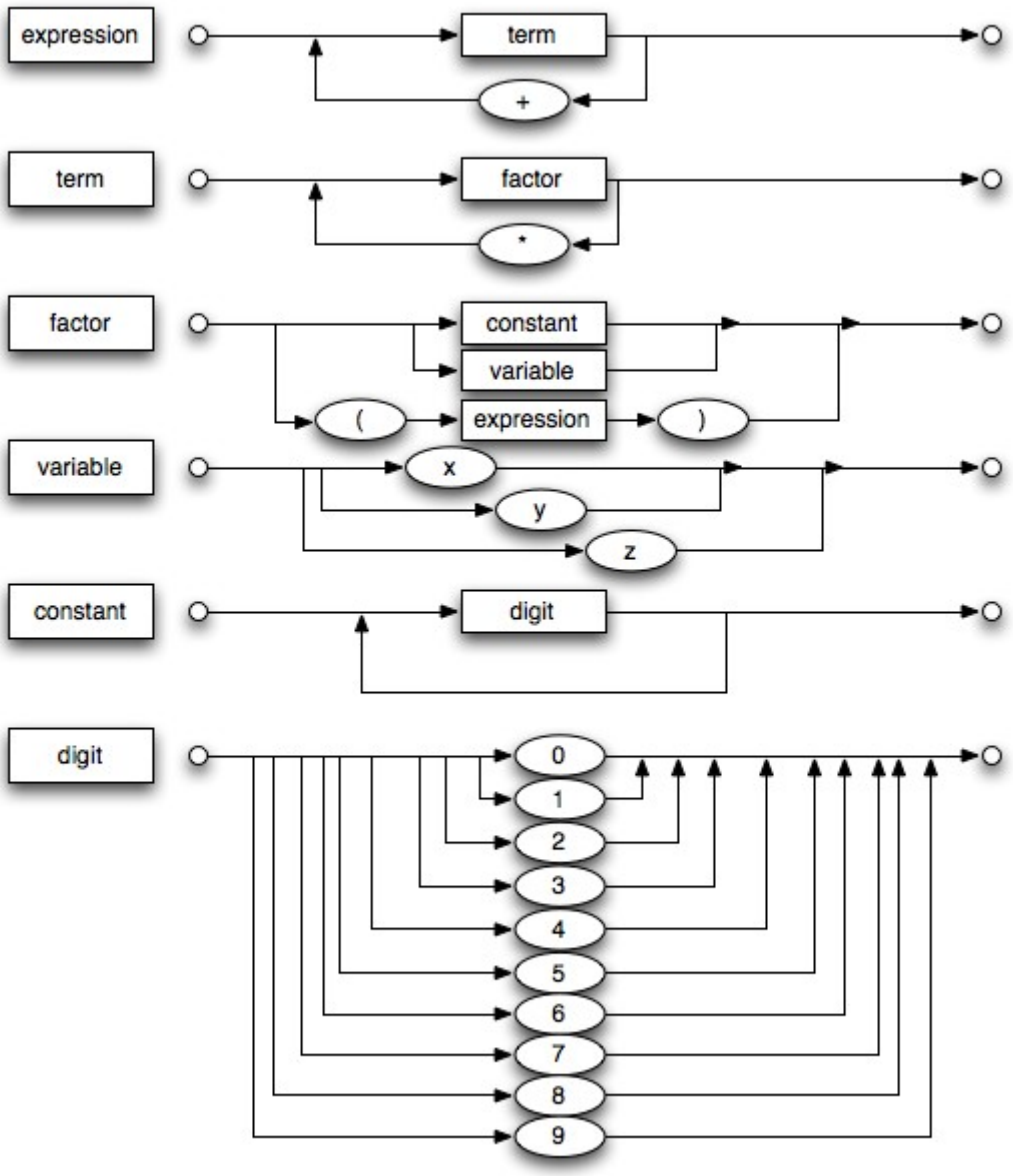
```
name ::= lc_letter (lc_letter | "_")*
lc_letter ::= "a"..."z"
identifier ::= (letter|"_") (letter | digit | "_")*
letter ::= lowercase | uppercase
lowercase ::= "a"..."z"
uppercase ::= "A"..."Z"
digit ::= "0"..."9"
```

Süntaksidiagrammid
On graafiline alternatiiv BNF-ile. Inimese jaoks keele lausete õigekirja mõistmisel paremini arusaadav. Keele kirjeldus koosneb paljudest diagrammidest, kus iga diagramm esitab mingit mitteterminaalset sümbolit.

Diagrammil on sisend- ja väljundpunkt. Terminale ja mitteterminale seovad omavahel nooled. Iga jada, mida noolte suunad läbida saab, moodustab süntaktiliselt õige lause.
Diagrammi joonistamiseks peaksid olema sobivad tehnilised vahendid.
Näide Wikipeedia lehelt (http://en.wikipedia.org/wiki/Syntax_diagram):

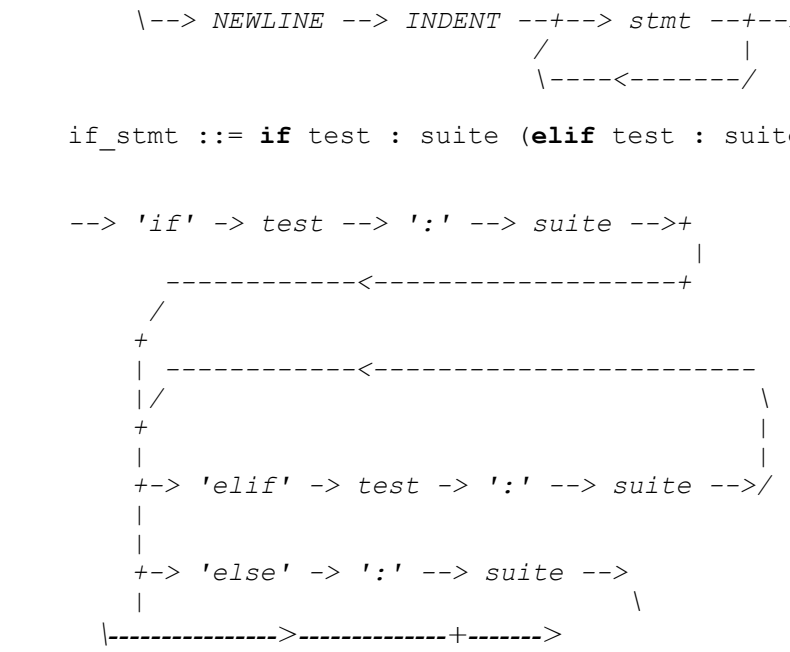
<expression> ::= <term> | <term> "+" <expression>

```
<term> ::= <factor> | <factor> "*" <term>

<factor> ::= <constant> | <variable> | "(" <expression> ")"
<variable> ::= "x" | "y" | "z"
<constant> ::= <digit> | <digit> <constant>
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```



Näide **Pythoni süntaksidiagrammist**, mis on esitatud ASCII-graafikas ning vastav BNF-laadne esitus:

```
suite ::= simple_stmt | NEWLINE INDENT stmt+ DEDENT

----+--> simple_stmt ---------------------------------->\
    |                                                    |
```



Protseduursed programmeerimiskeeled (PPK) *(Procedure-oriented languages)*
PPK on kunstlik kõrgtaseme keel, mida kasutatakse arvutile programmide kirjutamiseks inimesele arusaadavas vormis.

PPK-s kirjeldatakse probleemi lahenduse algoritm diskreetsete lausete e sammudena, kusjuures iga lause on reeglina keerukam, kui üks masinkoodis käsk ja abstraktsiooni tase on kõrgem.

PPK laseb käsud kombineerida suurematesse programmi moodulitesse, nn protseduuridesse e alamprogrammidesse.

Spetsifikatsioonid, kirjeldused
PPK säästab programmeerija seoste loomisest muutuja ja mälupesa vahel, piisab muutuja deklareerimisest ja nime teadmisest.

Keeled, kus nõutakse (ja kontrollitakse) iga muutuja deklareerimist, nimetatakse **rangelt tüpiseeritud keelteks** *(strongly typed languages)*.

Deklaratsioonide **näiteid**:

`REAL X, Y, Z`	(Fortran, pole kohustuslik)
`float x, y, z;`	(C)
`var x, y, z: real;`	(Pascal)

Programmi komponent on **lause** *(statement)*.

Omistuslaused *(assignment statement)*
PPK oluline koostisosa. Nad näevad erinevates keeltes välja enam-vähem sarnased, vahelduvad omistussümbolid:
muutuja <- avaldis

Omistuslausete **näited**:

`LET C = A + Y*B`	(Basic, LET kirjutati ette varem)

```
C = A + Y*B          (Fortran)
C = A + Y*B;         (C)
COMPUTE C = A + Y*B (Cobol)
c := a + y*b;        (Pascal)
C<- a + y*b          (APL)
```

Avaldis koosneb operaatoritest (tehtemärgid) ja operandidest (muutujad, konstandid, funktsioonide väljakutsed).

Kitsendused avaldisele (võrreldes aritmeetikaga)
1. Iga aritmeetikaoperatsioon peab olema määratud, ühtegi tehtemärki ei tohi ära jätta. Näiteks A(B +C) on arusaadav algebras, kuid programmi lauses peab ta välja nägema A*(B+C). Muidu võib tekkida mitmetähenduslikkus ja kompilaatorit on väga raske, kui mitte võimatu kirjutada.
2. Avaldis peab olema kirjapandud lineaarselt (ühel real). (A+B)/(C-D). Põhjuseks arvuti sisend/väljundseadmete piirid.

Tehete järjekord on tavaliselt sarnane tavapärastele aritmeetikatehetele. Loogikatehete puhul tuleb kindlasti kontrollida antud keele ja kompilaatori käitumist.

Tavaliselt saab arvutada ka stringidega – konkatenatsioon e liitmine.

Standardfunktsioonid *(Built-in Functions)*
Avardavad kasutamisvõimalusi, kirjeldavad keerulisemat protseduuri lihtsamalt.

```
C=SQRT(A**2+B**2)              (Fortran)
c:=sqrt(sqr(a)+sqr(b))         (Pascal)
c=sqrt(pow(a,2)+pow(b,2))      (C)
```

Valikud *(decisions)*
Oluliseks osaks on loogikatingimus võimalike väärustega *true* ja *false*, millele vastavalt toimub programmis hargnemine.

Valiku omadused:
1. Lihtne formuleering, kajastades üsna täpselt plokkskeemil esitatut. Kui test on tõene täida tegevus A ja väldi tegevust B, kui test on väär, täida tegevus B ja väldi tegevust A. Jätka lausega, mis on valikulausest sõltumatu.
2. Tingimus võib olla keeruline.
3. Kumbki alternatiivne tegevus võib olla pikk, kuid võib ka puududa.

Valikulausete **näited**:
C:
```
if ((x*y)<z)
   x=x+11;
else
   x=x-11;
```

Fortran:
```
IF (X*Y .LE. Z) THEN X = X+11
                ELSE X = X-11
END IF
```

Tsüklid e silmused *(Programming Loops)*
Kasutatakse tegevuste kordamiseks.

Tsüklite puhul leidma vastused küsimustele:
- Kuidas määrata korratavate lausete piirkond?
- Kuidas lugeda kokku vajalik korduste arv?
- Kuidas otsustada tsükli jätkamise või lõpetamise üle?

Kindla kordustearvuga tsüklid:
Basic:
```
10 DIM A1(14), A2(14)
15 FOR I = 1 TO 14
20 LET A1(I)=I
25 LET A2(I)=2*I+1
30 NEXT I
```

Fortran:
```
INTEGER A1(14), a2(14)
DO 30 I = 1, 4
A1(I) = I
A2(I)=2*I+1
30 CONTINUE
```

C:
```
float a1[14], a2[14];
int i;
for (i = 1; i<=14; i++)
{ a1[i] = i ;
   a2[i] = 2*i + 1;
}
```

Kui otsused tsükli jätkamise kohta tehakse tsükli sees, siis on vajalik teistsugune struktuur (nn WHILE-tsükkel)
Fortranis aeti asju järgmiselt:
```
      TOTAL = 0
1     IF (TOTAL .GE. Y) GO TO 30
      READ *, X
      TOTAL = TOTAL +X
      GO TO  1
30    töötlus jätkub
```

Sisend/väljund operatsioonid
Kuidas tulla toime erinevate sisend/väljundseadmetega?
Pascal:
`readln(x,y,z)` (standardsisendist)

Kõrgtaseme keeltes eriti protseduursetes keeltes on võimalus kirjeldada ja kasutusele võtta **alamprogramme** *(subroutine, procedure)*. Tihti kasutatavad protseduurid vormistamistatakse alamprogrammidena ja hoitakse alles edaspidiseks – **korduvkasutus** *(reuse)*

Massiivid

Lubatud tehete skaala võib olla väga lai. Kui Pascalis tohib terve massiiviga ainult omistada (kui on kaks ühte tüüpi massiivi), siis mõnes teises keeles võib leida massiividega opereerimiseks võimsamaid tehteid.
Näiteks keelest APL:

`A<-2 4 6 7 3 2`	paneb vektorisse A 6 väärtust.
`B<-(10100)/A`	tehakse uus vektor, kuhu saavad kuuluma 1. ja 3. element A-st
`B<-(A<5)/A`	massiivi B pannakse 5 väiksemad elemendid A-st