Seminarium 3 – Pythons syntax

I detta seminarium ska vi titta närmare på syntaxen för Python. Vi studerar dess formella syntax beskriven med Backus-Naur-Form (BNF). Detta är intressant för att verkligen förstå vad man kan skriva och inte skriva rent syntaktiskt. Det är också intressant att se hur Python är konstruerat i detalj för att själv reflektera över hur man tillverkar ett programmeringsspråk som Python och kunna utnyttja detta i egna konstruerade datorspråk senare (t ex i kursen Konstruktion av datorspråk).

Inför seminarium 3 ska du lösa ett antal uppgifter och skicka in till kursledningen. Under seminariet kommer vi gemensamt arbeta på tavlan med att rita upp BNF-träd för uppgifterna nedan.

Förberedelser

Läs PL kap 3.1-3.3

Studera BNF-grammatiken för Python.

http://www.python.org/doc/2.5.2/ref/grammar.txt

Not om BNF-syntax: Lexem anges i grammatiken med dubbelcitat runt tecknen. T ex betyder ”(” lexemet/tecknet vänsterparentes. Det som i boken beskrivs som → skrivs här som ::=. | betyder eller precis som i boken. * betyder noll eller fler gånger. (...) * att sekvensen inom parentesen ska upprepas noll eller fler gånger. [...] valfritt, kan anges eller ej.

Titta även översiktligt i (och se hur BNF används i en sådan kontext):

http://www.python.org/doc/2.5.2/ref/ref.html Python Reference Manual

Gör lösningar på nedanstående uppgifter. Du förutsätts ha gjort seriösa försök, lägg cirka två timmar på detta, som du skickat in på dessa BNF-träd, och vara beredda att redogöra för dina försök på vita tavlan.

Exempel: en grammatik för plus av heltal

Här följer ett exempel som visar hur BNF-reglerna ovan definierar ett litet språk av uttryck. Exemplet visar hur vi läser BNF-regler och även hur du kan skriva ner BNF-träd (eng. BNF parse trees) och BNF-härledningar (eng. BNF derivations). Observera att exemplet är **inte** hämtat ur Pythons syntax utan bara är tänkt som ett extra introduktionsexempel och ett exempel på notation du kan använda när du skickar in dina lösningar.

Notation: gör t ex dina lösningar i vanlig textfil i Emacs (eller om du föredrar i OpenOffice/ Word). Du kan t ex använda skrivsättet för parse-träd så som visas i detta exempel.

Vi ska här göra en mini-grammatik för uttryck som accepterar ental och operatorn +. Exempel på uttryck som är syntaktiskt korrekta i detta språk är:

2 + 5

3 + 5 + 4

Exempel på uttryck som inte är korrekta i detta språk är:

23 + 4

(3 + 4) + 5

3 * 7

Följande BNF-regler beskriver detta språk (ibland kallas första regeln en lexem/token-regel men vi ska inte skilja på detta här, eftersom åtskillnad inte görs i grammatiken för Pythons referensmanual):

digit → "0"..."9"

expr → digit | digit + expr

expr och digit kallas en icke-terminala symboler i grammatiken. ”0” ... ”9” är exempel på terminala symboler. Icke-terminala symboler förekommer till vänster i någon regel men icke-terminala gör inte det. En av symbolerna är grammatikens startsymbol, i detta fall expr.

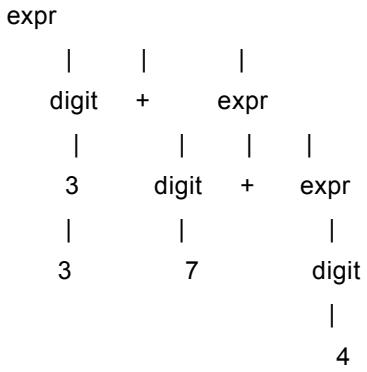
Här ges ett exempel på hur vi kan härleda uttrycket 3 + 7 + 4 har korrekt syntax enligt grammatiken (se LP 3.3.1.5 för detaljer runt detta):

expr => digit + expr => 3 + expr => 3 + digit + expr => 3 + 7 + expr

=> 3 + 7 + digit => 3 + 7 + 4

Detta kan vi utläsa som att vi applicerar reglerna i grammatiken stegvis tills vi får ett uttryck som bara innehåller terminaler. Alla sådana uttryck ”tillhör språket” för grammatiken. I varje steg applicerar vi exakt en regel och vi gör det på den icke-terminal som står längst till vänster (man kan också ha en högerregel, jmf LP-boken).

Ett annat sätt att uttrycka samma sak är genom att rita upp ett så kallat parse-träd. Här är motsvarande parse-träd för uttrycket 3 + 7 + 4 (se LP 3.3.1.6):



En sista notering är att regeln vi skrev för expr innehåller denna icke-terminal både i vänsterled (huvudet) och högerled (kroppen). Detta kallas att regeln är **rekursiv**. Effekten av detta blir att språket är oändligt, eftersom vi kan göra godtyckligt stora uttryck som uppfylls av denna regel, t ex har vi följande uttryck med 1or enbart i språket:

1, 1+1, 1+1+1, 1+1+1+1, 1+1+1+1+1, ...

Rekursiva regler är vanligt just för alla delar av en BNF-grammatik som ska tillåta godtyckligt stora programblock/uttryck.

Uppgifter

Gör ett parse-träd enligt Pythons BNF-grammatik för var och en av följande kodsegment. Vi börjar trädet på en icke-terminal symbol som är tillräckligt omfattande för kodsegmentet. Eftersom Python är ett omfattande språk så blir antalet icke-terminaler stort vilket betyder stora parse-träd. Exemplen nedan är därför på relativt små källkodssegment. Men de visar ändå hur grammatiken är strukturerad i olika segment. Prova gärna ”följa reglerna” för ett större exempel (även om du inte skriver ner trädet explicit). Hur skapas exempelvis ett parentiserat Booleskt uttryck?

Uppgift 1: När vi skapar ett program i en fil är startsymbolen file_input. Utgå ifrån denna icke-terminal och skapa parse-trädet för följande program:

```
pass
```

```
print
```

Uppgift 2: Uttryck är det mest komplicerade delen av en syntax så vi har många olika val på olika strukturer. Detta syns i att träden får djupa grenar med många olika nivåer/val. Uttryck utgår ifrån icke-terminalen expression. Symbolen + är en variant av * men betyder ”1 eller fler förekomster”. För detta exempel räcker det dock med att utgå ifrån icke-terminalen a_expr (a står för addition, och m för multiplication i m_expr). Skapa parseträd för följande aritmetiskt uttryck:

```
x + 3 * -y
```

Uppgift 3: Här följer ett exempel på en tilldelningssats. Utgå ifrån icke-terminalen assignment_stmt. Vi använder samma uttryck som i uppgift 2 så du behöver inte rita ut det delträdet under icke-terminalen expression . Rita ut a parseträdet för följande tilldelning:

```
myint = x + 3 * -y
```

Uppgift 4: Här ska vi titta på ett or-uttryck. Booleska uttryck har många nivåer så grenarna blir djupa i detta fall (även för enkla exempel). Liknande grenar kan förkortas i skrivsättet (”...”) och enbart skriva till det som skiljer. För detta exempel räcker det att börja på nivån or_test. Skapa parse-trädet för följande uttryck:

```
x < 0 or  x >  y
```

Uppgift 5: If-satsen utgår från icke-terminalen if_stmt. Exemplet återanvänder koden ovan så du behöver inte rita ut de delar av trädet som finns under icketerminalerna statement och expression. Skapa parseträdet för följande if-sats (print-satserna är tomma här för att begränsa trädets storlek men i princip skulle det kunna vara uttryck här också):

```
if x < 0 or  x >  y :
   myint = x + 3 * -y
   print
else:
   print
```

Uppgift 6: for-satsen utgår från icke-terminalen for_stmt. Du gör detta i två steg för enkelhetens skull. Först skapar du parseträdet för följande lista, börja från icke-terminalen list_display (använd gärna ”...” för identiska delar av de två uttrycksgrenarna):

```
[7,13]
```

Återigen återanvänder vi kod från ovan så du behöver inte rita upp de delar av trädet som finns under suite, target_list och expression_list. Skapa parse-trädet för följande for-sats:

```
for x in [7,13]:
   myint = x + 3 * -y
   print
```