TDDC10 Perspektiv på datateknik/datavetenskap
TDDC79 Perspektiv på informationsteknologi
TDP001 Handhavande av datormiljö
(D, IT, C, IP)

Breddgivande föreläsning

formella språk
grammatik
parsing

Att läsa mer:
Brookshear, *Computer Science an overview*, edition 7, Pearson/Addison Wesley, avsnitt 5.4

Sebesta, *Concepts of Programming Languages*, kapitel 3

Kozen, *Automata and Computability* (kursbok i Formella språk och automatateori)

Aho, Sethi, Ullman, *Compiler Principles, Techniques and Tools* (kursbok i Kompilatorkonstruktion)

Anders Haraldsson, 2007-09-25

Repetition från föregående breddföreläsning om Datorspråk

Datorspråk

språk med **textuell** representation

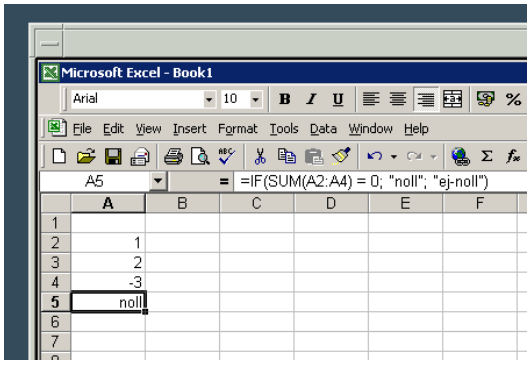
språk för att beskriva algoritmer
= **programmeringsspråk**

språk för att beskriva data eller layout
= **beskrivningsspråk**

=> dessa kallas ***formella språk***

Repetition från föregående breddföreläsning om Datorspråk

EXCEL



Repetition från föregående breddföreläsning om Datorspråk

Java

```
public class TestThing {

  public static void main(String[] args) {
    Thing t1 = new Thing(4);
    Thing t2;

    t1.setInfo(5);
    t2 = new Thing(6);
    doSomething(t2);
    System.out.println("t1: " + t1.getInfo() +
                       " (" + t1.getChanges() + ")");
    System.out.println("t2: " + t2.getInfo() +
                       " (" + t2.getChanges() + ")");
    t1 = null;
  }

  public static void doSomething(Thing t) {
    for (int i=0; i < 10; i++)
      t.setInfo(2*i);
  }
}
```

Repetition från föregående breddföreläsning om Datorspråk

Lisp

```
;;; Mönstermatchningsfunktionen från avsnitt

(defun patmatch (l pat)
  (cond
   ((endp pat) (endp l))                    ; 1.
   ((eq (first pat) '--)                    ; 2.
    (cond
     ((patmatch l (rest pat)) t)            ; 2a
     ((endp l) nil)                         ; 2b
     (t (patmatch (rest l) pat))))          ; 2c
                                            ;
   ((endp l) nil)                           ; 3.
   ((eq (first pat) '&)                     ; 4.
    (patmatch (rest l) (rest pat)))
   ((eql (first l) (first pat))             ; 5.
    (patmatch (rest l) (rest pat)))
   (t nil)))                                ; 6.

;;; Exempel på bokdatabas för uppgift 3B*

(setq *small-library*
  '(((author (anders haraldsson))
     (title (programmering i lisp))
     (year 1993))
    ((author (anders haraldsson))
     (title (programmering i pascal))
     (year 1979))
    ((author (nils nilsson))
     (title (artificial intelligence))
     (year 1999))))
```


Python

```
numbers = []
for number in range(100):
    numbers.append(str(number) + "\n")

open("filnamn", "w").writelines(numbers)
```

Indenteringen styr hur satser hänger samman (block), som i andra språk kan ha begin ... end eller måsvingar { }

Repetition från föregående breddföreläsning om Datorspråk

Formella språk (CS edition 7, avs 5.4)

Hur är språket uppbyggt?

Vilka program / beskrivningar är korrekta i det formella språket.

Språkets ***syntax*** beskriver hur de olika konstruktionerna får skrivas (som text).

Syntaxen brukar skrivas i någon typ av ***grammatik***.

Exempel: Aritmetiska uttryck

3 - (4.5 * -2) / 1.56

Men vad är egentligen tillåtet? Får vi skriva följande uttryck?

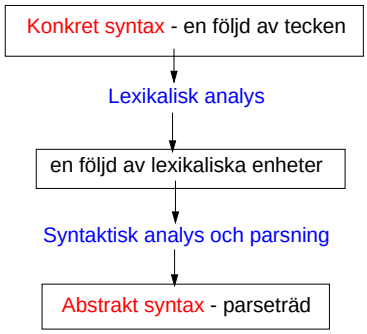
-(2+3)	+3-2
3+(2)	3-+2
4*()-1	((((2))))

Grundproblem.

Program är en följd av tecken, som man skriver t ex med texteditorn emacs.

Frågan är vilka teckenföljder bildar ett program, en sats, ett uttryck etc?

x = y + -2 +(radie+sin 2 * pi) ;



sedan kan vi använda parseträdet för beräkning eller som utgångspunkt för att generera maskinkod "kompilerad kod"

Lexikalisk analys

int i = -3+x-sin(-1.5);
består i Java av de lexikaliska enheterna (*lexemes*):

int
i
=
-3
+
x
-
sin
(
-1.5
)
;

Man har grundläggande enheter som kallas *token*: heltal, flyttal, sträng, identifierare, symbol etc

Syntax bör utformas så att triviala fel ej kan får oväntade tolkningar:

Exempel på gammalt känt fel:

```
    DO 10 I = 1.5
    sats
10 sats
```


För att beskriva tokens använder man ofta *reguljära uttryck (regular expressions)*.

Ett sådant uttryck sätts samman med

sekvens

val |

upprepning * (ingen eller flera gånger)
+ (en eller flera gånger)

Vi vill i ett språk ange att en identifierare (namn på olika storheter) skall börja på en bokstav och sedan följas ev antingen bokstäver eller siffror.

identifierare = letter (letter | digit)*

letter = a | b | c .... | z

digit = 0 | 1 | 2 .... | 9

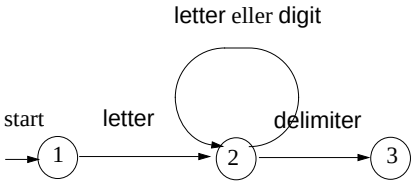
a a1 abc1 a12345 är alla identifierare

1 2ab är ej identifierare

identifierare = letter (letter | digit)*
letter = a | b | c .... | z
digit = 0 | 1 | 2 .... | 9

a a1 abc1 är alla identifierare
1 23+ är ej identifierare

Reguljära uttryck kan överföras i "*transition diagrams*"



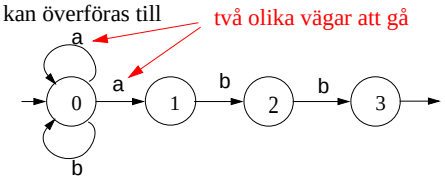
Ett språk som beskrivs av reguljära uttryck kan överföras till en *ändlig automat*.

Reguljära uttryck kan överföras till en *ickedeterministisk ändlig automat* (nondeterministic finite automata)

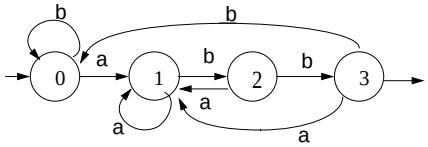
Reguljära uttrycket (a|b)*abb tolkar av alla strängar som har a och b i sig och som avslutas med exakt abb, t ex

ababbbaabb

kan överföras till



men i denna automat finns alternativ så att man ibland måste hoppa tillbaka. Det ger ineffektivitiet. En sådan automat kan omformas till en *(deterministisk) ändlig automat* (finite automata).



här finns alltid bara ett alternativ

I Unix och i många programspråk använder man regul-jära uttryck för att ange mönster vid t ex sökning, t ex i Unix med grep.

Exempel på kommandon på mönster:

foo	ordet foo
.	vilket tecken som helst
a*	noll eller flera av bokstaven a
.*	noll eller flera av vilket tecken som helst
a+	ett eller flera av bokstaven a
a?	noll eller ett av bokstaven a
(foo)+	en eller flera förekomster av strängen "foo"
foo|bar	strängen "foo" eller "bar"
[ab]	bokstaven a eller bokstaven b
[a-e]	någon av bokstäverna a, b, c, d eller e
[^a]	något tecken som inte är bokstaven a
[-a]	något av tecknet minus eller bokstaven a
^a	bokstaven a först på en rad
a$	bokstaven a sist på en rad
\[	tecknet vänsterklammer

Automatisering

LEX är verktyg i UNIX som tar ett språk beskrivet som reguljära uttryck och skapar automatiskt ett program som utför den lexikaliska analysen, dvs delar upp teckenströmmen i lexems.

Från Google om Lex och Yacc
http://dinosaur.compilertools.net/lex/index.html (27 oct 05)

Regular expressions in Lex use the following operators:

x	the character "x"
"x"	an "x", even if x is an operator.
\x	an "x", even if x is an operator.
[xy]	the character x or y.
[x-z]	the characters x, y or z.
[^x]	any character but x.
.	any character but newline.
^x	an x at the beginning of a line.
<y>x	an x when Lex is in start condition y.
x$	an x at the end of a line.
x?	an optional x.
x*	0,1,2, ... instances of x.
x+	1,2,3, ... instances of x.
x|y	an x or a y.
(x)	an x.
x/y	an x but only if followed by y.
{xx}	the translation of xx from the definitions section.
x{m,n}	m through n occurrences of x

Generellt lexikaliskt analysproblem.

Man behöver specialsymboler för att markera t ex början och slut på en lexikalisk enhet, t ex sträng
"detta är en sträng"

Hur gör man om man vill att " skall ingå i strängen. Jag vill skriva strängen
"i början och slutet skall det stå ett "-tecken"

I html-kod:
För att göra ett ord halvfet skriver du före ordet och efter ordet

Man har ofta någon typ av *escape*-tecken.
"i början och slutet skall det stå ett \"-tecken"

Om escape-tecknet skall stå måsta man sätta escape för detta, t ex \\

Gäller nästling av sådana specialsymboler

Här skall vara halvfet och mer halvfet , men är även detta halvfet?

Skiljetecken (*delimiters*, *breakcharacters*)

Hur skiljer man lexems åt.

Skiljetecken - blanka, tabtecken, ny rad

Specialtecken, t ex ;
Olika syn: Ibland ser man ; för att avsluta något, ibland för åt skilja något.

a; b; c;
a; b; c

Bryttecken, men tecknet är även ett lexem
12+34 delas upp i 12 + och 34

Syntaxanalys

Nu har vi en sekvens med lexikaliska eneheter. Bildar dessa ett korrekt program?

För att beskriva syntax för ett språk brukar man använda en *kontextfri grammatik* eller *BNF (Backus Naur Form) -grammatik*.

Detta utvecklades runt Algol som kom i början av 60-talet.

Syntaxanalys - grammatik

En grammtik beskrivs i ett metaspråk med *produktionsregler* (rules, productions). Vi har *terminal* symboler och *icke-terminala* symboler.

Terminal symboler skall återfinnas i programmet och markeras här med halvfet stil.

<program> ::=[1] **begin** <stmt_list> **end**
<stm_list> ::= <stmt>
| <stmt> **;** <stmt_list>
<stmt> ::= <var> **:=** <expr>
<var> ::= **A** | **B** | **C**
<expr> ::= <expr> **+** <expr>
| <expr> ***** <expr>
| (<expr>)
| <var>

Med en grammatik kan vi generera alla tänkbara program som finns.

Med en grammatik kan vi avgöra om ett program uppfyller grammatiken. Vi önskar samtidigt få ett "*parsningsträd* (parse-tree).

1. Även -> användas för att skriva produktionsregler

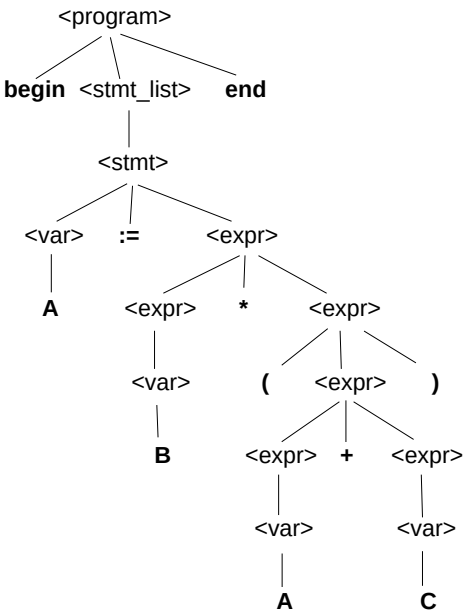
<program> ::= **begin** <stmt_list> **end**
<stm_list> ::= <stmt>
| <stmt> **;** <stmt_list>
<stmt> ::= <var> **:=** <expr>
<var> ::= **A** | **B** | **C**
<expr> ::= <expr> **+** <expr>
| <expr> ***** <expr>
| (<expr>)
| <var>

begin
A := B * (A + C)
end

Att generera en *mening* (sentence) kallas "*derivation*"

<program>
=> **begin** <stm_list> **end**
=> **begin** <stmt> **end**
=> **begin** <var> **:=** <expr> **end**
=> **begin A :=** <expr> **end**
=> **begin A :=** <expr> ***** <expr> **end**
=> **begin A :=** <var> ***** <expr> **end**
=> **begin A := B *** <expr> **end**
=> **begin A := B * (** <expr> **) end**
=> **begin A := B * (** <expr> **+** <expr> **) end**
=> **begin A := B * (** <var> **+** <expr> **) end**
=> **begin A := B * (A +** <expr> **) end**
=> **begin A := B * (A +** <var> **) end**
=> **begin A := B * (A + C) end**

Resultatet kan var ett parsningsträd, som beskriver den abstrakta syntaxen,

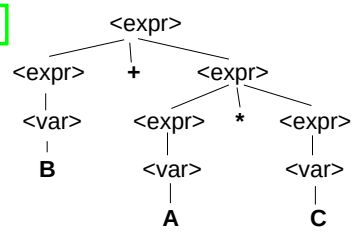


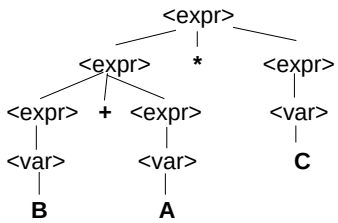
Problem

Är grammatiken entydig? Kan flera olika parsningsträd genereras?

```
<expr> ::= <expr> + <expr>
         | <expr> * <expr>
         | (<expr>)
         | <var>
```

B + A * C





Grammatiken kan skrivas om:

```
<expr> ::= <expr> + <term>
         | <term>

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

<factor> ::= ( <expr> )
         | <var>
```

B + A * C

<expr>
=> <expr> + <term>
=> <term> + <term>
=> <factor> + <term>
=> <var> + <term>
=> **B** + <term>
=> **B** + <term> * <factor>
=> **B** + <factor> * <factor>
=> **B** + <var> * <factor>
=> **B + C** * <factor>
=> **B + C** * <var>
=> **B + C * A**

Annat exempel på satskostruktion som ger flertydlighet. Hängande **else**-gren. I Pascal har vi

if *villkor* **then** *sann-sats*
if *villkor* **then** *sann-sats* **else** *falsk-sats*

Om vi nästlar dessa satser kan vi få ?

if villkor **then if** villkor **then** sann-sats **else** falsk-sats

Många språk löste detta med avslutande nyckelord

if *villkor* **then** *sann-sats* **endif**
if *villkor* **then** *sann-sats* **else** *falsk-sats* **endif**

if villkor **then if** villkor **then** sann-sats **endif**
else falsk-sats **endif**

if villkor
then if villkor **then** sann-sats **else** falsk-sats **endif**
endif

Det stora problemet är hitta effektiva algoritmer som tar ett "program" i teckenformat

- genomför lexikalisk analys

- genomför syntax analys med uppbyggnad av ett parsningsträd

=> detta gör en parser

I samband med kompilering vill man att parserna skall rapportera alla syntaktiska fel som påträffas.

Frågeställning: (*error recovery*) Var återstartar man parsningen efter felet. Hur undviker man att samma fel ger upprepade felmeddelanden.

För vissa klasser av grammatiker och metoder att genomföra parsningen kan en parser automatiskt genereras. Ett exempel är för Unix YACC (Yet Another Compiler Compiler).

YACC tar en grammatik i BNF (liknande) och genererar en parser i C.

Top down parser - Recursive descent

Skriver ett program som följer grammatiken med en funktion för varje produktionsregel.

<program> ::= **begin** <stmt_list> **end**
<stm_list> := <stmt>
| <stmt> **;** <stmt_list>
<stmt> ::= <var> **:=** <expr>
<var> ::= **A** | **B** | **C**
<expr> ::= <expr> **+** <term>
| <term>

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

<factor> ::= **(** <expr> **)**
| <var>

parse-program
1) kolla om första lexemet är **begin**
2) bearbeta alla satserna <stmt_list>
3) kolla att sista lexemet är **end**

parse-stmt_list
1) bearbeta nästa <stmt>
2) följer **; ?**
3) om ja, anropa parse-stmt_list

Man klarar inte alla BNF-grammatiker utan en grammatik måste ibland göras om på en lämplig form.

Man klarar inte produktionsregler med sk *vänster-rekursivitet*

<expr> ::= <expr> **+** <term>
| <term>

parse-expr:
1) försök hitta första uttrycket <expr> genom att anropa parse-expr
2) om det fanns, är nästa lexem +, fortsätt sedan med ..
3) om det inte fanns hitta en <term>

Man kan skriva om ovanstående regel som:

<expr> ::= <term> <expr'>

<expr'> ::= **+** <term> <expr'> | "empty"

Bottom-up parser - LR(n) parsers

Idag finns en mycket utvecklad teori om parser där man i princip tar ett lexem i taget från vänster till höger.

Från grammatiken bygger man tabeller och använder sig att "stack" och "tillstånd".

Den är en relativt komplicerad process men man kan automatiskt generera parsern!

Mer om detta får ni i en Kompilatorkurs.

Hantering av operatorer och prioriteter

I matematiken och i de flesta språk använder man operatorer, där varje operator har en prioritet.

1+2*3 = 7
(1+2)*3 = 9

* har högre prioritet än +.

Hur är prioriteten här ?
x + y > 4 and z = 5 + v

Vi tycker det är självklart
((x + y) > 4) and (z = (5 + v))

men ej säkert att detta gäller i programmeringsspråket.

Associativitet
1 - 2 - 3 = ((1 - 2) - 3) = ? vänster associativitet
1 - 2 - 3 = (1 - (2 - 3)) = ? höger associativitet

Prefix, infix, postfix notationer

För att undvika dessa problem med prioriteter och associativitet så överför man ofta uttryck från infix till *prefix* eller *postfix* (*polish notation*).

Exempel om varje operator tar exakt två operander:
3+4*5 -> +3*45 *prefix* tex i Lisp (+ 3 (* 4 5))
(3+4)*5 -> *+345 *prefix* tex i Lisp (* (+ 3 4) 5)

3+4*5 -> 345*+ *postfix*
(3+4)*5 -> 34+5* *postfix*

Man kan då med en stack beräkna uttrycket (mer mekaniskt) sekvensiellt.

Översättning av infix-uttryck till prefix-uttryck kan göras med t ex Järnvägsalgoritmen.

Utdrag ur A Byte on Python
Operator precedence

(http://www.ibiblio.org/g2swap/byteofpython/read/operator-precedence.html 2007-10-10)

(lägst till höst prioritet)

Operator	Description
lambda	Lambda Expression
or	Boolean OR
and	Boolean AND
not x	Boolean NOT
in, not in	Membership tests
is, is not	Identity tests
<, <=, >, >=, !=, ==	Comparisons
|	Bitwise OR
^	Bitwise XOR
&	Bitwise AND
<<, >>	Shifts
+, -	Addition and subtraction
*, /, %	Multiplication, Division and Remainder
+x, -x	Positive, Negative
~x	Bitwise NOT
**	Exponentiation
x.attribute	Attribute reference
x[index]	Subscription
x[index:index]	Slicing
f(arguments ...)	Function call
(expressions, ...)	Binding or tuple display
[expressions, ...]	List display
{key:datum, ...}	Dictionary display
‘expressions, ...‘	String conversion

Semantik

3 olika modeller
operationell semantik
axiomatisk semantik
denotationssemantik

Operationell semantik genom en interpretator

Funktionen logikvärde (från Lisp/Scheme-laborationerna) och interpretatorn för KALKYL (Lisp/Scheme-uppgiften) kan ses som en operationell semantisk beskrivning av det implementerade logikspråket.

Vi har skrivit en interpretator för språket.

Genom att studera denna interpretator kan vi härleda viktiga egenskaper från hur logikspråket beräknas.

Hur beräknas de logiska funktionen or och and ?

När testas slutvillkoret i en repetition, först eller sist.

Variablers synbarhet, räckvidd (scope)
f(x) = x + y vilket y avses?

Alternativa sätt att beskriva språk är med *syntaxdiagram*.

Syntaxdiagram för Pascal.

Java språket beskrivs på SUN's sida på ca 10 sidor.

The Java Language Specification, Second Edition

http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html (27/10-07)

Den lexikaliska grammatiken tar ca 10 sidor.
Själva syntaktiska grammatiken tar 5 sidor.

Utdrag ur Java grammatiken för kommentarer

.7 Comments

There are two kinds of comments:

/* text */	A traditional comment: all the text from the ASCII characters /* to the ASCII characters */ is ignored (as in C and C++).
// text	A end-of-line comment: all the text from the ASCII characters // to the end of the line is ignored (as in C++).

These comments are formally specified by the following productions:

```
Comment:
        TraditionalComment
        EndOfLineComment

TraditionalComment:
        / * NotStar CommentTail

EndOfLineComment:
        / / CharactersInLineopt LineTerminator

CommentTail:
        * CommentTailStar
        NotStar CommentTail

CommentTailStar:
        /
        * CommentTailStar
        NotStarNotSlash CommentTail

NotStar:
        InputCharacter but not *
        LineTerminator

NotStarNotSlash:
        InputCharacter but not * or /
        LineTerminator

CharactersInLine:
        InputCharacter
        CharactersInLine InputCharacter
```

These productions imply all of the following properties:

* Comments do not nest.
* /* and */ have no special meaning in comments that begin with //.
* // has no special meaning in comments that begin with /* or /**.