Hur har det gått?

Kompilatorer

Del 1: Tokens, parsing och syntaxträd

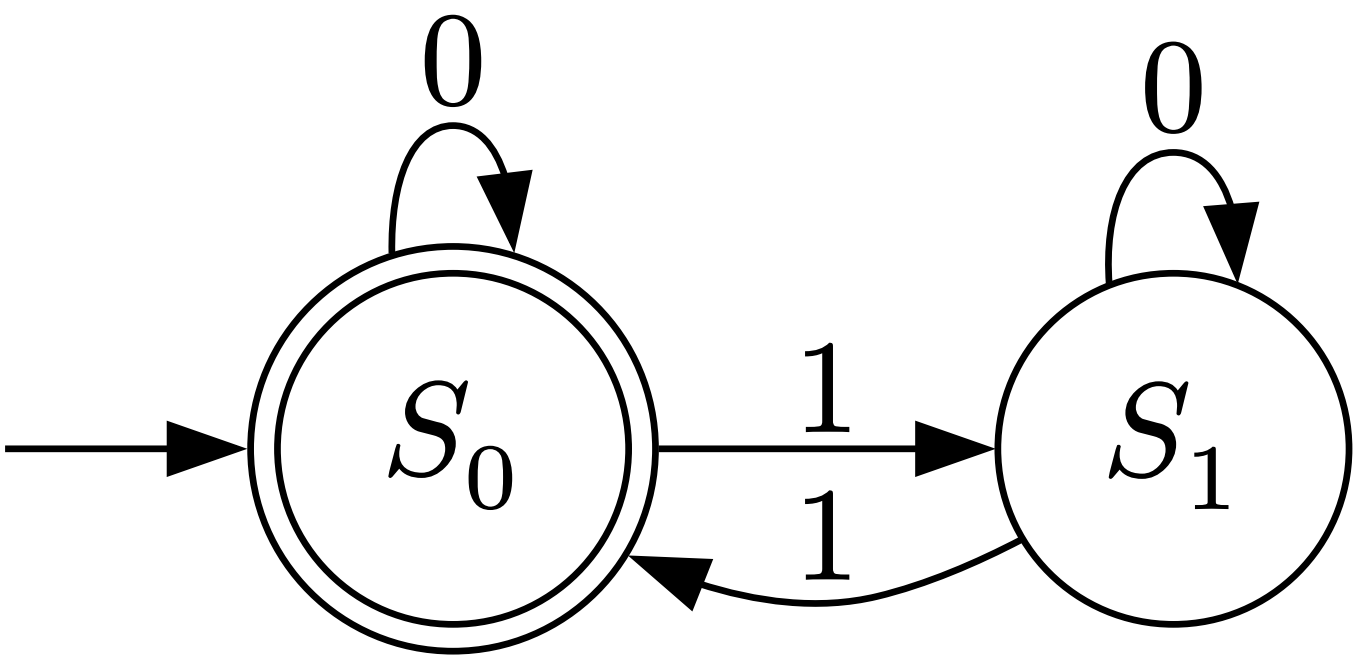
Formella språk

- En uppsättning symboler $\Sigma \neq \emptyset$
- En uppsättning strängar $L \subseteq \Sigma^*$ är ett **språk**
- Vanligt problem: tillhör en given sträng ett visst språk?

Formella språk: Exempel

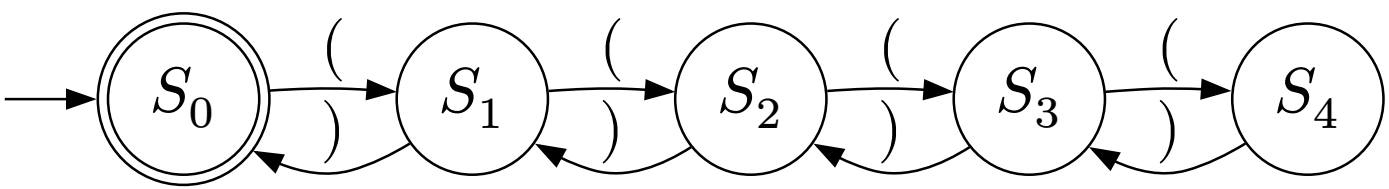
- $\Sigma = \{0, 1\}$
- L är alla strängar med ett jämnt antal ettor
- T.ex. `001101001`

Ändliga automater beskriver *reguljära språk*



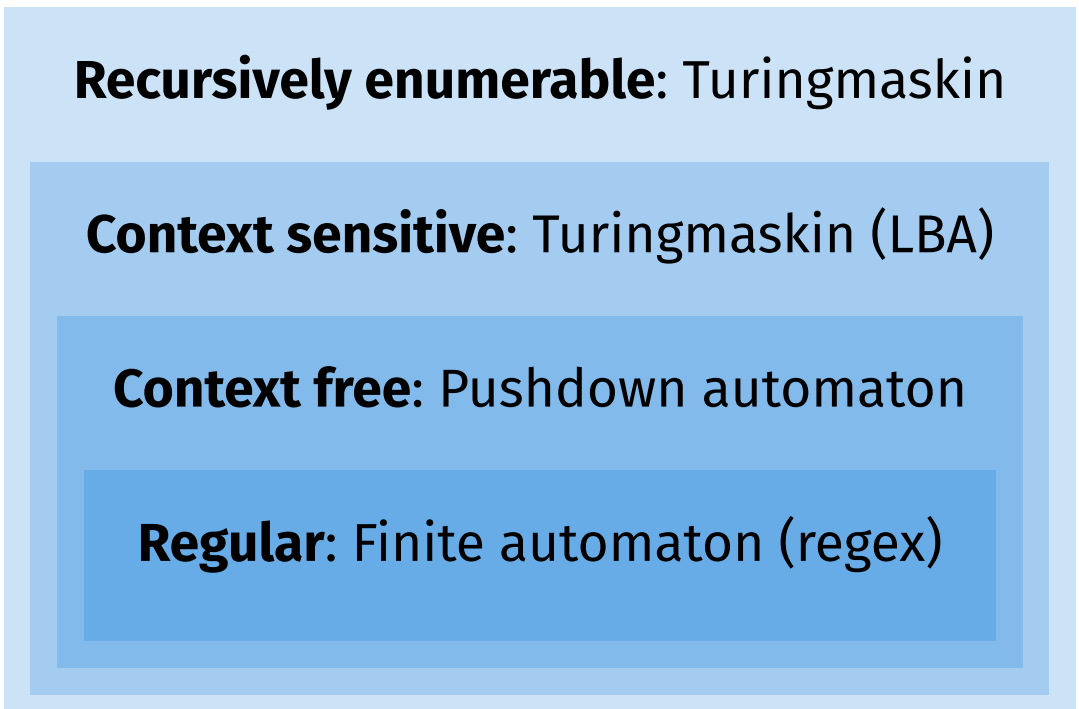
Ändliga automater (eng. Finite automata)

Exempel: Balanserade parenteser, t.ex. `(()()())()`



Går ej att hantera godtycklig nästning med en ändlig automat!

Chomskyhierarkin definierar en ordning för språk:



Kontextfria språk

- Ett kontextfritt språk kan beskrivas med en *kontextfri grammatik*
- *Produktionsregler*, *slutsymboler*, *icke-slutsymboler*, och en *startsymbol*

$$S \to E$$

$$E \to (E)$$

$$E \to EE$$

$$E \to \varepsilon$$

Backus-Naur form

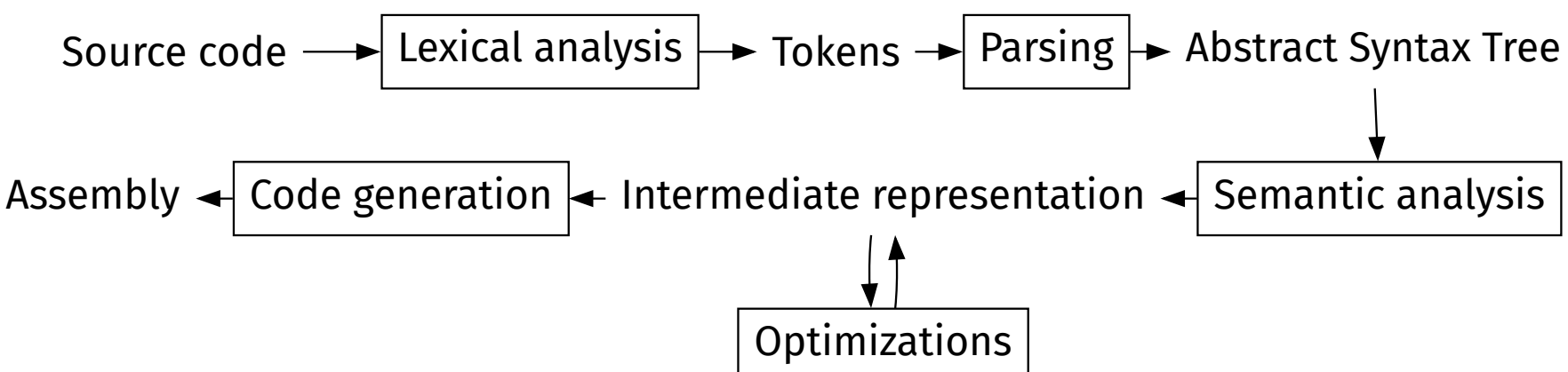
- Används ofta för programmeringsspråk
- BNF för C
- BNF för BNF

```
<start> ::= <expression>
<expression> ::= "(" <expression> ")" | <expression> <expression> | ""
```

Kompilatorer

Kompilering sker i flera separata steg.

Vi fokuserar på den övre delen denna vecka.



Dela upp källkoden i tokens

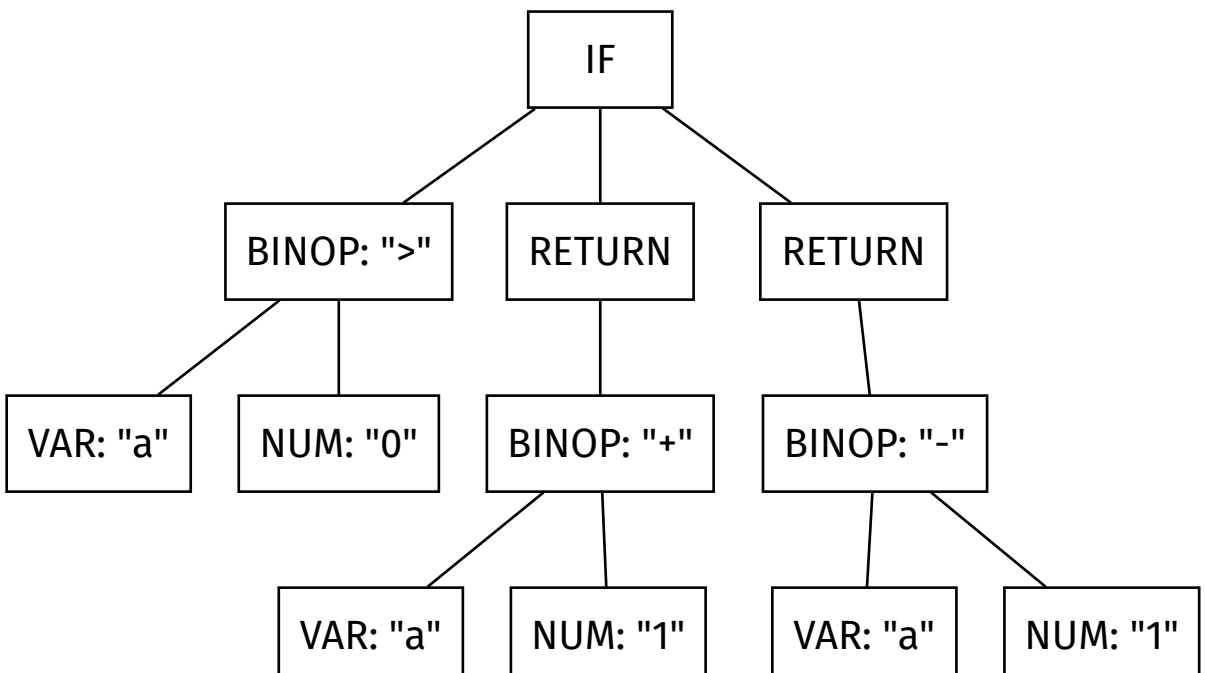
- Görs med en scanner/lexer/tokenizer.
- Reguljära uttryck lämpar sig för detta
- Parsern arbetar med tokens snarare än karaktärer (varför?)

```
int f(int c) {
  return c + 1;
}
```

Sträng	Typ
`int`	*primitive type*
`f`	*identifier*
`(`	*left parenthesis*
`int`	*primitive type*
`c`	*identifier*
`)`	*right parenthesis*
`{`	*left brace*
`return`	*keyword*
...	...

Syntaxträd (eng. Abstract Syntax Tree)

Beskriver strukturen för ett program




```
if a > 0:
  return a + 1
else:
  return a - 1
```

Parsing

Omvandla en lista med tokens till ett syntaxträd

Exempel: Rekursiv medåkning

Rekursiv medåkning (eng. recursive descent)

Exempel-grammatik:

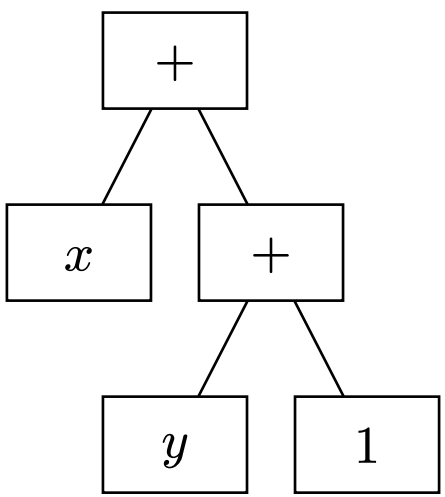
```
<start> ::= <addition>
<addition> ::= <term> | <term> "+" <addition>
<term> ::= <number> | <variable>
```

Vi definierar tokens med reguljära uttryck (inte standard men OK i läxan):

```
<number> ::= [1-9][0-9]*
<variable> ::= [A-Za-z_]+
```

Rekursiv medåkning (eng. recursive descent)

Givet strängen `x + y + 1` vill vi få följande syntaxträd (AST):



Rekursiv medåkning (eng. recursive descent)

Pseudokod för regeln `<addition> ::= <term> | <term> "+" <addition>`:

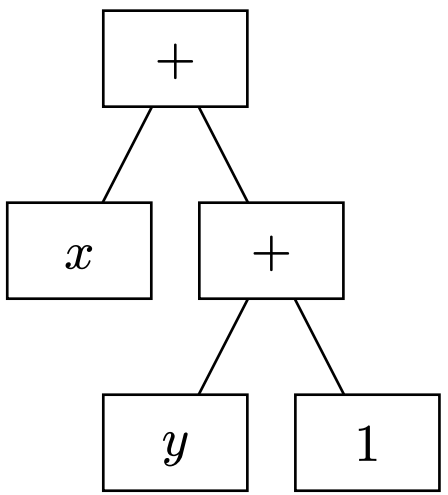
```
function parse_addition()

  -- Parsa en term
  term = parse_term()

  expect("+") or return term

  -- Parsa resten av additionen
  rest = parse_addition()

  -- Skapa en nod i syntaxträdet
  return ast_addition_node(term, rest)
end
```



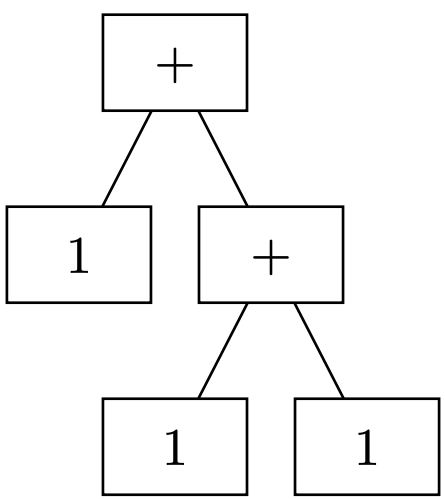
$$x + y + 1$$

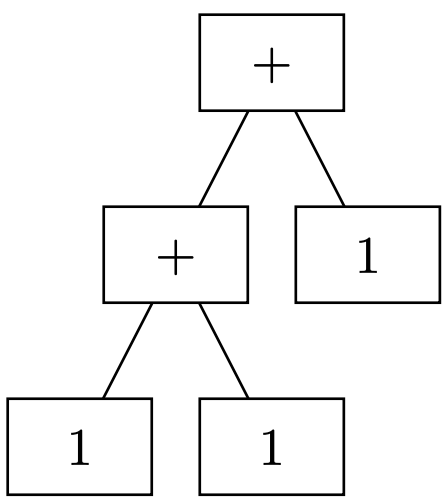
Tvetydiga grammatiker

- För vissa grammatiker kan vi härleda flera olika syntaxträd
- Dessa kallas *tvetydiga* (eng. ambiguous grammar)
- Denna grammatik innehåller även *vänsterrekursion* (eng. left recursion)

$$
\begin{aligned}
S &\to A \\
A &\to A + A \\
A &\to 1
\end{aligned}
$$

$$1 + 1 + 1$$





Omvänd polsk notation (eng. reverse polish notation, RPN)

Omvänd polsk notation (*Reverse Polish Notation*, RPN), efter Jan Łukasiewicz.
Eller bara *postfix notation*.

Exempeluttryck: $x - y \times (3 + z)$

`Infix`	$x - y \times (3 + z)$
`Prefix`	$- \times + z\ 3\ y\ x$
`Postfix`	$x\ y\ 3\ z + \times -$

Omvänd polsk notation (eng. reverse polish notation, RPN)

Varför RPN? Inga parenteser! Implicit precedence! Lätt att parsa!

```
function parse_rpn(tokens)
  -- Stack av AST-noder som vi skapat hittills
  stk = stack()
  -- Läs tokens
  while not tokens.empty() do
    token = tokens.next()
    if is_leaf(token) then
      -- Pusha en ny lövnod
      stk.push(ast_leaf_node(token))
    else if is_binary_op(token) then
      -- Sätt ihop de två översta noderna i en
      -- ny nod som är en binär operation
      right = stk.pop()
      left = stk.pop()
      stk.push(ast_binary_op_node(token, left, right))
    end
  end
  -- Här finns bara en nod kvar på stacken (om syntaxen är korrekt), returnera den
  return stk.pop()
end
```

Shunting yard algorithm

Algoritm för att skapa översätta infix-notation till postfix (RPN). Se gärna Wikipedia.

```
function shunting_yard(tokens)
  out = vector() -- Utdata
  stk = stack() -- Operator-stack
  while not tokens.empty() do
    token = tokens.next()
    if is_leaf(token) then
      out.append(token) -- Lägg till i output
    else if token == '(' then
      stk.push(token) -- Pusha till stack
    else if token == ')' then
      shunt(out, stk, nil) -- Poppa stack
      stk.pop() -- Poppa vänterparentes
    else if is_operator(token) then
      shunt(out, stk, token) -- Poppa & pusha
    end
  end
  shunt(out, stk, nil) -- Poppa resterande ops
  return out
end
```

```
function shunt(out, stk, op)
  while not stk.empty()
  and stk.top() ~= '(' do
    -- Poppa ut operatorer från stacken, fram
    -- tills någon med lägre (eller samma för
    -- left-associative) precedence som op
    if op == nil
    or prec(stk.top) > prec(op)
    or (prec(stk.top) == prec(op)
        and is_left_associative(op))
    then
      out.append(stk.pop())
    else
      break
    end
  end
  -- Pusha ny operator (om ej nil)
  if op ~= nil then stk.append(op) end
end
```

Läxa

Läxa

- Hitta på eller låna ett enkelt programmeringsspråk
- Skriv en kompilator för språket
- Det ska gå att beräkna Fibonacci-tal i språket

Till nästa vecka

- Skriv en grammatik för språket i BNF
- Implementera en lexer och parsning
- Se till att det går att printa ut syntaxträdet i något läsligt format
- Dokumentera hur programmet används

Nästa läxa

- Under tentaperioden och veckan därefter har ni tid att implementera resten
- För att generera assembly är det OK att använda t.ex. QBE (enklast) eller LLVM.
- Det går även bra att kompilera till ett annat språk, men...
- **...språkets syntax måste vara väsentligt annorlunda!**