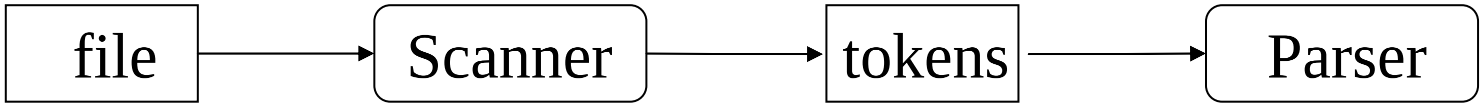
Syntax/semantics - II

- Syntax analysis
  - Scanning
  - Parsing
    ”top-down”
    ”bottom-up”
  - LL(1)-parsing
    Recursive descent

Scanner



- The scanner groups characters into *tokens*
- Example:

BEGIN	IDENT	LPAR	TEXT	RPAR	END
begin	OutText	(	”Hallo”	)	end

- A scanner can be constructed as a deterministic automaton

Parsing

- To check that a sentence (or a program) is syntactically correct, that is to construct the corresponding syntax tree.
- In general we would like to construct the tree by reading the sentence/program once, from left to right!

- Example grammar

<uttrykk> → <uttrykk> + <term>
<uttrykk> → <term>
<term> → <term> * **navn**
<term> → **navn**

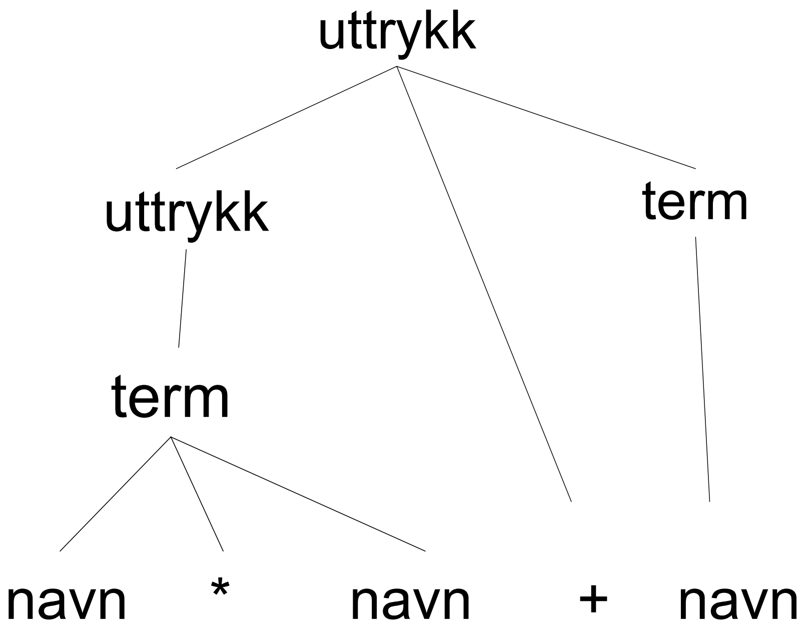
Given this grammar, we shall loook at the parsing of the sentence:

navn * navn + navn

Top-down parsing

The tree is constructed from the top and down, that is we start with the start symbol as the root, and try to **derive** the sentence from there:

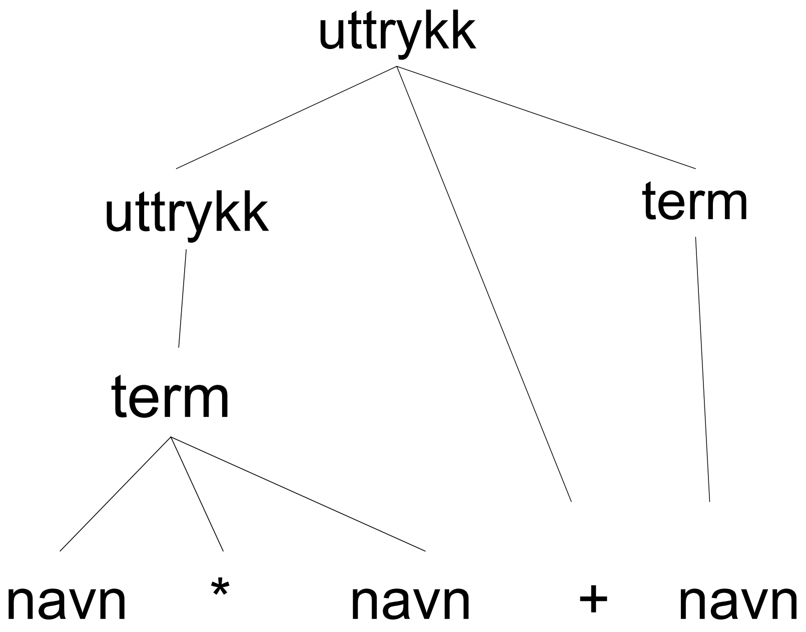
<uttrykk> → <uttrykk> + <term>
<uttrykk> → <term>
<term> → <term> * **navn**
<term> → **navn**



Bottom-up parsing

The tree is constructed from the bottom and up. We start by finding something in the sentence that matches a right-hand side in a production, and **reduces** this part of the sentence to the corresponding non-terminal on the left-hand side. The goal is to reduce so that we finally reduce to the start symbol:

<uttrykk> → <uttrykk> + <term>
<uttrykk> → <term>
<term> → <term> * **navn**
<term> → **navn**



LL(1)-parsering

- LL(1)-parsing is a top-down strategy where we do a *left derivation* from the start symbol.
- Recursive descent
  - To each non-terminal there is a method.
  - The method takes care of the terminals of the right hand side, and calls methods corresponding to the non-terminals:
    - For every terminal in the right-hand side, check that the next symbol in the sentence is this terminal.
    - For every *non-terminal* in the right-hand side, call the method corresponding to the non-terminal.
  - When the method is called, the first symbol in the text shall be the first symbol in the corresponding production, in order for the sentence to be syntactically correct.
  - When the method is finished, the scanner will have the next symbol after the sentence.

Example

<uttrykk> → <uttrykk> + <term> | <term>

<term> → <term> * **navn** | **navn**

```
static void uttrykk() {
    uttrykk();
    readSymbol(’+');
    term();
}
```

Example

<program> → <stmtList>
<stmtList> → <stmt> +
<stmt> → <input> | <output> | <assignment>
<input> → ? <variable>
<output> → ! <variable>
<assignment> → <variable> = <variable> <operator> <operand>
<operator> → + | -
<operand> → <variable> | <number>
<variable> → v <digit>
<digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<number> → <digit> +

```
static void assignment() {
    variable();
    readSymbol('=');
    variable();
    operator();
    operand();
}
```

Example

<program> → <stmtList>
<stmtList> → <stmt> +
<stmt> → <input> | <output> | <assignment>
<input> → ? <variable>
<output> → ! <variable>
<assignment> → <variable> = <variable> <operator> <operand>
<operator> → + | -
<operand> → <variable> | <number>
<variable> → v <digit>
<digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<number> → <digit> +

```
static void stmt() {
    if (checkSymbol('v')) {
        assignment();
    } else if (checkSymbol('?')) {
        input();
    } else if (checkSymbol('!')) {
        output();
    }
```

LL(1)-grammars

- We cannot make a ”recursive descent”-parser for all grammars; they have to fulfill the following requirements:
  - The grammar must be context free, i.e. only one non-terminal at the left-hand side.
  - During parsing we must know which of the alternatives on the right-hand side to choose, i.e. that the sets of start symbols for each alternative must be disjoint.
- A LL(1)-grammar is a grammar with these properties, and so that one just has to look 1 (but not more) symbol ahead in order to choose alternative.

Example

<uttrykk> → <uttrykk> + <term> | <term>
<term> → <term> * **navn** | **navn**

```
static void uttrykk() {
    uttrykk();
    readSymbol(’+');
    term();
}
```

What if a grammar is not LL(1)?

Two techniques to turn a grammar into a LL(1) grammar.

- Removal of left recursion
- Left factorization (making the sets of start symbols for each alternative disjoint)

Removal of left recursion I

<uttrykk> → <uttrykk> + <term> | <term>

- Strategy:

1. Translate to Extended BNF:

<uttrykk> → <term> {+ <term> }*

2. Back to BNF, but with right recursion (and possibly extra non-terminals):

<uttrykk> → <term> <xterm>
<xterm> → + <term> <xterm> | ε

```
static void uttrykk() {
    term();
    xterm();
}
```

```
static void xterm() {
    if not end then {
       readSymbol(’+');
       term();
       xterm()
    }
}
```

Removal of left recursion II

<term> → <term> * **navn** | **navn**

- Strategy:
  1. Translate to Extended BNF:

<term> → **navn** {* **navn** }*

  2. Back to BNF:

<term> → **navn** <xnavn>
<xnavn> → * **navn** <xnavn> | ε

```
static void term() {
    read(navn);
    xnavn();
}
```

```
static void xnavn() {
    if not end then {
        readSymbol('*');
        read(navn);
        xnavn();
    }
}
```

Left factorization

Often two alternatives may begin the same way (non-disjoint sets of start symbols), but have different endings, such as e.g.:

<setning> → <uttrykk> + <term> | <uttrykk> * <term>

The trick is here to introduce a new non-terminal for the part that may vary:

<setning> → <uttrykk> <xsetning>
<xsetning> → + <term> | * <term>