Description of a programming language

• Syntax
  – describes the structure of a language
  – given as grammatical rules
  – which streams of symbols (characters) form a legal program

• Syntax checking (syntax analysis, parsing)
  – construction of a parse tree
  – does the input program follow the grammatical rules
  – checking requires program transformation from a character string into a stream of tokens (lexical analysis, scanning)

• Semantics
  – what is the meaning of a given legal program
  – which kind of computation does a legal program produce
Phases of compilation

• Compilation is usually divided to separate phases: easier, simpler, clearer
• Output of a previous phase is the input of the next one
• Symbol table collects information on user-defined constructs (variables, functions, types, …)
Analyses (check-ups)

• Natural language levels:
  – lexical: ”It iß seven o’clock.”
  – syntactic: ”It seven o’clock.”
  – semantic: ”It is thirty o’clock.”

• Programming language levels:

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>lexical</td>
<td>scanning</td>
</tr>
<tr>
<td>syntactic</td>
<td>parsing</td>
</tr>
<tr>
<td>contextual</td>
<td>e.g. type checking</td>
</tr>
<tr>
<td>semantic</td>
<td>code generation</td>
</tr>
</tbody>
</table>
Compilation process

Source program (characters) → Lexical analyzer (tokens) → Syntax analyzer → Semantic analyzer → Intermediate code generator → Optimization → Intermediate code

Symbol table → Parse tree → maybe the same

Intermediate code → Code generator → Target language (assembly) → Executable
Interpretation process

Source program

Input data

Interpreter

Results
Hybrid process

Source program

Lexical analyzer

Syntax analyzer

Intermediate code generator

Interpreter

Input data

Results

tokens

parse tree

intermediate code
Preprocessor

An example of macros:

C-code:
```c
#define MAX_LOOP 100
#define INCR (a) (a)++
#define FOR_LOOP (var, from, to) \for (var = from; var <= to; INCR(var)) {
#define END_FOR
#define NULL

FOR_LOOP (n, 1, MAX_LOOP)
NULL;
END_FOR;
```
Lexical analysis

characters (source code)

Lexer
- grammar: regular
- format: regular expressions
- implementation: finite state machine

→ list of tokens
Examples of regular expressions

**Digits and letters**

\[
<\text{digit}> \rightarrow 0 \mid 1 \mid \ldots \mid 9 \\
<\text{letter}> \rightarrow a \mid \ldots \mid z \mid A \mid \ldots \mid Z
\]

**Numbers**

\[
<\text{unsigned int}> \rightarrow <\text{digit}>^*
\]

**Identifiers**

\[
<\text{id}> \rightarrow <\text{letter}> \mid <\text{id}> <\text{letter}> \mid <\text{id}> <\text{digit}>
\]
Lexical analysis

• Grouping of input characters
  – lexeme
    • a unit that can be detected from a program text
  – token
    • classification of lexemes
    • a name given to a lexeme

• Lexeme
  – terminal symbol

index = 2 * count;

<table>
<thead>
<tr>
<th>Lexeme</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>index</td>
<td>identifier</td>
</tr>
<tr>
<td>=</td>
<td>equal_sign</td>
</tr>
<tr>
<td>2</td>
<td>int_literal</td>
</tr>
<tr>
<td>*</td>
<td>mult_op</td>
</tr>
<tr>
<td>count</td>
<td>identifier</td>
</tr>
<tr>
<td>;</td>
<td>semicolon</td>
</tr>
</tbody>
</table>
Lexemes/tokens

- **Keywords**
  - reserved words

- **Identifiers**
  - names chosen by the programmer

- **Literals**
  - constant values

- **Operators**
  - acronyms for (e.g. arithmetic) functions

- **Separators**
  - characters and strings between language constructs

- **Other things to be considered in lexical analysis:**
  - comments
  - white spaces
  - indentations
Lexical analysis

Identifying the lexemes (pattern matching)

<table>
<thead>
<tr>
<th>Program</th>
<th>gcd ( input, output ) ;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Var</td>
<td>i, j : integer ;</td>
</tr>
<tr>
<td>Begin</td>
<td>read ( i, j ) ;</td>
</tr>
<tr>
<td>While</td>
<td>while i &lt;&gt; j do</td>
</tr>
<tr>
<td></td>
<td>if i &gt; j then i := i – j</td>
</tr>
<tr>
<td></td>
<td>else j := j – i ;</td>
</tr>
<tr>
<td></td>
<td>writeln ( i ) ;</td>
</tr>
<tr>
<td>End</td>
<td></td>
</tr>
</tbody>
</table>

Pascal code:

```pascal
program gcd ( input, output );
var i, j: integer;
begin
  read ( i, j );
  while i <> j do
    if i > j then i := i – j
    else j := j – i;
  writeln ( i )
end.
```
Syntactic analysis

- list of tokens
- Parser
  - grammar: context free
  - format: BNF
  - implementation: push-down (stack) automaton
- symbol table
- parse tree
Describing syntax

Context-free grammar:

\[ G = (N, T, P, S) \]

- \( N \): set of nonterminals
- \( T \): set of terminals
- \( P \): set of productions (rules)
- \( S \): start symbol, \( S \in N \)

The rules do not depend on the context in which they appear.
Examples of grammar rules (BNF)

Variable definition

\[ \text{<variable def> ::= <identifier> <identifier> = <expr>} \]

\[
\begin{align*}
\text{(type)} & \quad \text{(name)} \\
\end{align*}
\]

while-loop

\[ \text{<iteration stmt> ::= while ( <expr> ) <stmt>} \]

Statements

\[ \text{<stmt> ::= <iteration stmt>} \]
\[ \text{<stmt> ::= <compound stmt>} \]
\[ \text{<compound stmt> ::= \{ <statement seq> \}} \]
\[ \text{<statement seq> ::= <stmt>} \]
\[ \text{<statement seq> ::= <stmt> <statement seq>} \]
### Example of a grammar

<table>
<thead>
<tr>
<th>Grammar rules in BNF</th>
<th>Nonterminals</th>
<th>Terminals</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;expr&gt; ::= &lt;expr&gt; + &lt;term&gt;</td>
<td>expr, term, factor, integer</td>
<td>(, ), +, -, *, / (and integer instances)</td>
</tr>
<tr>
<td>&lt;expr&gt; ::= &lt;expr&gt; - &lt;term&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;expr&gt; ::= &lt;term&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;term&gt; ::= &lt;term&gt; * &lt;factor&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;term&gt; ::= &lt;term&gt; / &lt;factor&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;term&gt; ::= &lt;factor&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;factor&gt; ::= &lt;integer&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;factor&gt; ::= ( &lt;expr&gt; )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Start symbol**: `expr`
Derivation of
$2 \times (12 + 3)$

$$
<\text{expr}> = <\text{term}>
= <\text{term}> \times <\text{factor}>
= <\text{factor} > \times <\text{factor} >
= <\text{integer} > \times <\text{factor} >
= 2 \times <\text{factor} >
= 2 \times ( <\text{expr} > )
= 2 \times ( <\text{expr} > + <\text{term} > )
= 2 \times ( <\text{term} > + <\text{term} > )
= 2 \times ( <\text{factor} > + <\text{factor} > )
= 2 \times ( <\text{integer} > + <\text{integer} > )
= 2 \times ( 12 + 3 )
$$
Derivation as a parse tree

$E = expr$
$T = term$
$F = factor$
$I = integer$
Ambiguous grammar

• There exist several parse trees for one input
  – input can be derived in several ways

<expr> ::= <expr> + <expr>
<expr> ::= <expr> - <expr>
<expr> ::= <expr> * <expr>
<expr> ::= <expr> / <expr>
<expr> ::= ( <expr> )
<expr> ::= <integer>
A real example of ambiguity

dangling else:

**Pascal-code:**

```
if E1 then
  if E2 then S1
else S2
```

```
if E1 then
  if E2 then S1
else S2
```

**Corresponding grammar:**

```
<stmt> ::= <if_stmt> | ...
<if_stmt> ::= if <expr> then <stmt>
<if_stmt> ::= if <expr> then <stmt> else <stmt>
```
"Dangling else" in parse trees

\[
\text{<stmt> ::= <if_stmt> | ...}
\]

\[
\text{<if_stmt> ::= if <expr> then <stmt>}
\]

\[
\text{<if_stmt> ::= if <expr> then <stmt> else <stmt>}
\]
Solutions for dangling else problem

• (The same problems exists in C)
• Semantic rule:
  – else branch belongs to the latest condition that
    not yet has an else branch
• Programmer can use compound statements
Solutions in other languages

Comb-like structures
e.g. Ada and Modula-2

if <expr> then
  <stmt-list>
elsif <expr> then
  <stmt-list>
  ...
else
  <stmt-list>
end if

Ada:

if E1 then
  if E2 then
    S1;
  else
    S2;
  end if;
end if;

if E1 then
  if E2 then
    S1;
  else
    S2;
  end if;
end if;
Extended BNF (EBNF)

parenthesis: grouping
curly brackets: repetition (0, 1, … times)
brackets: optionality

\[
\begin{align*}
\langle expr \rangle & ::= \langle term \rangle \{ ( '+' | '-' ) \langle term \rangle \} \\
\langle term \rangle & ::= \langle factor \rangle \{ ( '*' | '/' ) \langle factor \rangle \} \\
\langle factor \rangle & ::= '(\langle expr \rangle ')' | \langle integer \rangle
\end{align*}
\]

repetition:
*:: 0, 1,.. times
+: 1, 2, ... times

\[
\begin{align*}
\langle expr \rangle & ::= \langle term \rangle \{ ( '+' | '-' ) \langle term \rangle \}^* \\
& \ldots
\end{align*}
\]
Syntax diagram
Parsing

- **Top-down – parsing (LL)**
  - produces left derivation
  - parse tree is constructed in depth-first order
  - recursive-descent parsing
  - cannot handle left recursion

- **Bottom-up – parsing (LR)**
  - parse tree is constructed from leaves to root
  - produces right derivation

left recursion:

\[
A \rightarrow A + B \\
A \rightarrow B \ a \ A \\
B \rightarrow A \ b
\]
Grammar example for LL-parsing

<Decl> ::= <VarDecl> | <TypeDecl>
<VarDecl> ::= VAR <Variable> :: <Type> <VarInit>;
>TypeDecl> ::= TYPE <OwnType> = <Type> ;
<Variable> ::= identifier
<Type> ::= identifier
<OwnType> ::= identifier
<VarInit> ::= := <Value> | ε
<Value> ::= number

Lexemes and tokens
VAR reservedVar
TYPE reservedType
identifier instance identifier
number instance number
:: varSep
= typeSep
:= initSep
; semicolon
Scanning as state machine

VAR reservedVar
TYPE reservedType
identifier instance identifier
number instance number
:: varSep
:= typeSep
:= initSep
; semicolon

start ;
digit
number
; letter

= reservedVar

reservedType
identifier

varSep

initSep

number

typeSep
Parse code
(in pseudo code)

**procedure** ParseDecl ( )
if LookUp ( reservedVar ) then ParseVarDecl ( );
else if LookUp ( reservedType ) then ParseTypeDecl ( );
else Error ( );
end if;

**procedure** ParseVarDecl ( )
Scan ( reservedVar );
ParseVariable ( );
Scan ( varSep );
ParseType ( );
ParseVarInit ( );
Scan ( semicolon );

**procedure** ParseVariable ( )
Scan ( identifier );
// identifier processing

**Grammar:**

```
<Decl> ::= <VarDecl> | <TypeDecl>
<VarDecl> ::= VAR <Variable> :: <Type> <VarInit> ;
<TypeDecl> ::= TYPE <OwnType> = <Type> ;
<Variable> ::= identifier
```
More parser code (in pseudo code)

```
<TypeDecl> ::= TYPE <OwnType> = <Type> ;
<Type> ::= identifier
<OwnType> ::= identifier
<VarInit> ::= := <Value> | ε
<Value> ::= number

procedure ParseTypeDecl ( )
    Scan ( reservedType );
    ParseOwnType ( );
    Scan ( typeSep );
    ParseType ( );
    Scan ( semicolon );

procedure ParseVarInit ( )
    if LookUp ( initSep ) then
        Scan ( initSep );
        ParseValue ( );
    end if;

procedure ParseType ( )
    Scan ( identifier );
    // identifier processing

procedure ParseOwnType ( )
    Scan ( identifier );
    // identifier processing

procedure ParseValue ( )
    Scan ( number );
    // number processing
```
top_down_parsing is
push the start symbol onto an empty stack;
while the stack is not empty do
    /* let X be the top stack symbol */
    /* let a be the current input token */
    if X is a nonterminal symbol then
        pop X from the stack;
        push the components of X onto the stack in reverse order;
    else if X is a terminal symbol and X = a then
        pop X from the stack;
        scan a;
    else
        /* syntax error */
    end if;
end while;
end top_down_parsing;
An example on LL-parsing

Example program

VAR x :: Integer := 10;

Tokens

reservedVar
identifier
varSep
identifier
initSep
number
semicolon
LR-parsing

• Operations in LR-parsing
  – shift
    • add input symbol (lexeme) onto the stack
    • move forward in the input text (move cursor)
  – reduce
    • find the right side of a grammar rule from the top of the stack (it may consist of several stack items) and replace (reduce) it with the left side of the rule
An example on LR-parsing

Grammar:

1. \( E \rightarrow E + T \)
2. \( E \rightarrow T \)
3. \( T \rightarrow T * F \)
4. \( T \rightarrow F \)
5. \( F \rightarrow ( E ) \)
6. \( F \rightarrow id \)

Input:

id + id * id
## LR-parsing table

<table>
<thead>
<tr>
<th>State</th>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S5</td>
<td>S4</td>
</tr>
<tr>
<td>1</td>
<td>S6</td>
<td>accept</td>
</tr>
<tr>
<td>2</td>
<td>R2</td>
<td>S7</td>
</tr>
<tr>
<td>3</td>
<td>R4</td>
<td>R4</td>
</tr>
<tr>
<td>4</td>
<td>S5</td>
<td>S4</td>
</tr>
<tr>
<td>5</td>
<td>R6</td>
<td>R6</td>
</tr>
<tr>
<td>6</td>
<td>S5</td>
<td>S4</td>
</tr>
<tr>
<td>7</td>
<td>S5</td>
<td>S4</td>
</tr>
<tr>
<td>8</td>
<td>S6</td>
<td>S11</td>
</tr>
<tr>
<td>9</td>
<td>R1</td>
<td>S7</td>
</tr>
<tr>
<td>10</td>
<td>R3</td>
<td>R3</td>
</tr>
<tr>
<td>11</td>
<td>R5</td>
<td>R5</td>
</tr>
</tbody>
</table>

1. \( E \rightarrow E + T \)
2. \( E \rightarrow T \)
3. \( T \rightarrow T * F \)
4. \( T \rightarrow F \)
5. \( F \rightarrow ( E ) \)
6. \( F \rightarrow id \)
<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>id + id * id $</td>
<td>shift 5</td>
</tr>
<tr>
<td>0, id5</td>
<td>+ id * id $</td>
<td>reduce 6 [ 0, F ]</td>
</tr>
<tr>
<td>0, F3</td>
<td>+ id * id $</td>
<td>reduce 4 [ 0, T ]</td>
</tr>
<tr>
<td>0, T2</td>
<td>+ id * id $</td>
<td>reduce 2 [ 0, E ]</td>
</tr>
<tr>
<td>0, E1</td>
<td>+ id * id $</td>
<td>shift 6</td>
</tr>
<tr>
<td>0, E1, +6</td>
<td>id * id $</td>
<td>shift 5</td>
</tr>
<tr>
<td>0, E1, +6, id5</td>
<td>* id $</td>
<td>reduce 6 [6, F ]</td>
</tr>
<tr>
<td>0, E1, +6, F3</td>
<td>* id $</td>
<td>reduce 4 [6, T ]</td>
</tr>
<tr>
<td>0, E1, +6, T9</td>
<td>* id $</td>
<td>shift 7</td>
</tr>
<tr>
<td>0, E1, +6, T9, *7</td>
<td>id $</td>
<td>shift 5</td>
</tr>
<tr>
<td>0, E1, +6, T9, *7, id5</td>
<td>$</td>
<td>reduce 6 [ 7, F ]</td>
</tr>
<tr>
<td>0, E1, +6, T9, *7, F10</td>
<td>$</td>
<td>reduce 3 [ 6, T ]</td>
</tr>
<tr>
<td>0, E1, +6, T9</td>
<td>$</td>
<td>reduce 1 [ 0, E ]</td>
</tr>
<tr>
<td>0, E1</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>
Semantic analysis

parse tree  symbol table

Semantic analyzer
- contextual check / semantics
- type definitions, type compatibility
- function definitions, parameter compatibility
- etc.
Often combined with intermediate code generation
Describing semantics

- Semantics
  - What does the program do?
  - Does the program do what it is supposed to do?
- Semantics is usually described informally
- Static semantics
  - can be checked at compilation-time
    - type compatibility (in assignments, in parameter passing)
    - are variables declared before using them
- Dynamic (run-time) semantics
  - checked at run-time
Formal descriptions  
(for dynamic semantics)

• Operational semantics  
  – describes the operation of the program  
  – meaning of a statement = change in the state of the machine (memory, registers)

• Denotational semantics  
  – models the program functionality in recursive functions  
  – meaning of a statement = the value of the function associated with the statement

• Axiomatic semantics  
  – goal is to prove correctness  
  – meaning of a statement = transformation in logical formulas that describe the state of computation
An example on operational semantics

<table>
<thead>
<tr>
<th>for statement in C:</th>
<th>Operational semantics:</th>
</tr>
</thead>
<tbody>
<tr>
<td>for ( expr1; expr2; expr3 )</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td>expr1;</td>
</tr>
<tr>
<td>...</td>
<td>loop: if expr2 = 0 goto out</td>
</tr>
<tr>
<td>}</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>expr3;</td>
</tr>
<tr>
<td></td>
<td>goto loop</td>
</tr>
<tr>
<td></td>
<td>out:</td>
</tr>
</tbody>
</table>
An example on denotational semantics

Grammar rules for binary numbers:

\[
\begin{align*}
<BinNum> & ::= 0 \\
<BinNum> & ::= 1 \\
<BinNum> & ::= <BinNum> 0 \\
<BinNum> & ::= <BinNum> 1
\end{align*}
\]

Evaluation of binary numbers:

\[
\begin{align*}
M(\text{'0'}) &= 0 \\
M(\text{'1'}) &= 1 \\
M(<BinNum>\text{'0'}) &= 2 \times M(<BinNum>) \\
M(<BinNum>\text{'1'}) &= 2 \times M(<BinNum>) + 1
\end{align*}
\]

$M$ is a semantic function
Final compilation phases

- Intermediate code generation
- Code optimization
- Code generation

see:
https://www.tutorialspoint.com/compiler_design/index.htm
Code generation / optimization

```
a = b + c;
d = a + e;
MOV b R0
ADD c R0
MOV R0 a
MOV a R0
ADD e R0
MOV R0 d
MOV a R0
ADD #1 R0
MOV R0 a
INC a
```

```
a = a + 1;
```

MOV y R0 -- load y into R0
ADD z R0 -- add z to R0
MOV R0 x -- store R0 into x