Executing algorithms on a computer

Needed: **System software:**

- **Operating system**
  - makes it unnecessary for the user to be aware of awkward features of the computer hardware
  - provides facilities for the manipulation of programs and data
  - allows resource sharing among several simultaneous users.

- **Language translators**
  - **Interpreters**
    - translate and execute a high-level program statement by statement.
  - **Compilers**
    - translate the entire program before execution commences.
Algorithm for compilation

\begin{algorithm}
\textbf{start at the beginning of the high-level program;}
\textbf{repeat}
\quad \textbf{translate the next high-level statement}
\textbf{until} \textbf{the end of the high-level program} \textbf{endrepeat ;}
\textbf{arrange for the entire translated program to be executed}
\end{algorithm}
Source program vs. object program

- Source program: original high-level program
- Object program: result of the compilation
- The object program can be executed an arbitrary number of times without recompilation of the source program.
Program translation is crucially dependent on the syntax of the language concerned.

**Grammars** are used to define the syntax of programming languages.

They are expressed in so-called Backus-Naur Form (BNF).
Recall that a program consists of a sequence of elementary symbols, such as the names of data items, certain punctuation characters, and various special words like if and repeat.

These elementary symbols are called the terminal symbols of the language.

The terminal symbols can be combined in various ways to form more complex constructs, such as conditions, statements, and the programs themselves.

Each kind of construct which can be built in this way is called a syntactic category or nonterminal symbol of the language.

There is a similarity with natural language, whose terminal symbols are words and punctuation marks, and whose syntactic categories include phrases, clauses and sentences.
The basic idea behind syntax definition is to specify precisely how the syntactic categories of a language are constructed.

This is done by formulating a set of rules, or productions, each of which defines how a particular syntactic category can be built from

1. members of other syntactic categories, and/or
2. terminal symbols of the language.

The set of terminal symbols, the set of syntactic categories, and the set of productions together define the syntax of a language.
Example: telephone directory

- Each entry details a subscriber’s name, address and numbers.
- A typical directory entry is
  
  Smith John B. 97 Main 51374

- The syntax of an entry can be defined by the BNF production

  \[ \text{entry} \rightarrow \text{person-name address number} \]

- Syntactic categories are shown in italics.
Syntactic conventions

- More productions:

  \[
  \text{person-name} \rightarrow \text{surname forename \{forename\}} \\
  \text{forename} \rightarrow \text{proper-forename} | \text{initial}
  \]

- Curly brackets mean that the enclosed category of symbol is repeated zero or more times.

- The vertical bar means “or”.

Complete grammar for telephone directory

```
entry        \rightarrow person-name address number
person-name \rightarrow surname forename \{forename\}
surname     \rightarrow name
forename     \rightarrow proper-forename | initial
proper-forename \rightarrow name
name         \rightarrow letter | letter name
initial      \rightarrow letter.
address      \rightarrow number street-name
street-name  \rightarrow name
letter       \rightarrow A | ... | Z | a | ... | z
number       \rightarrow digit | digit number
digit        \rightarrow 0 | 1 | ... | 9
```
Remarks

- Every syntactic category is ultimately defined in terms of terminal symbols (letters, digits, period).
- The terminal symbols themselves need no further definition, and therefore do not appear on the left-hand side of any production.
- The above definition does not include any definition of format (e.g. that various symbols should be separated by spaces).
- If format specification is required then 'space' characters must be added to the set of terminal symbols and included in the appropriate productions (exercise).
Syntax of a programming language fragment

\[
\begin{align*}
\text{statement} & \rightarrow \text{conditional} \mid \text{loop} \mid \text{assignment} \\
\text{conditional} & \rightarrow \text{if condition then statement endif} \\
\text{loop} & \rightarrow \text{while condition do statement endwhile} \\
\text{assignment} & \rightarrow \text{set name to expression} \\
\text{expression} & \rightarrow \text{name operator name} \\
\text{name} & \rightarrow \text{letter} \mid \text{letter name} \\
\text{letter} & \rightarrow \text{A} \mid \ldots \mid \text{Z} \mid \text{a} \mid \ldots \mid \text{z} \\
\text{operator} & \rightarrow + \mid - \\
\text{condition} & \rightarrow \text{name relation number} \\
\text{relation} & \rightarrow = \mid \neq \\
\text{number} & \rightarrow \text{digit} \mid \text{digit number} \\
\text{digit} & \rightarrow 0 \mid 1 \mid \ldots \mid 9
\end{align*}
\]
Examples

- **assignment**: set $X$ to $Y + Z$, set $Net$ to $Gross - Tax$

- **condition**: $X \neq 5$, $Gross = 0$

- **conditional**: if $X \neq 5$ then set $X$ to $Y + Z$ endif

- **loop**: while $X \neq 0$ do set $Y$ to $Y - X$ endwhile

- **nested**: if $X \neq 0$

  then while $Y \neq 0$ do

  set $Y$ to $Y - X$ endwhile

endif

- Note that recursive productions allow arbitrarily complex programs to be constructed from a grammar with relatively few productions.
The step “translate the next high level statement” of the compilation algorithm must be refined.

There are three major phases in the process of compilation:

- **Lexical analysis**
- **Syntax analysis**
- **Code generation**

Note the phases can be interleaved; e.g., the module for syntax analysis may call the lexical analysis to obtain the next token.
Lexical analysis

- The string of characters comprising the source program is broken up into a sequence of separate symbols, or tokens, much as the characters comprising a piece of English text can be separated into a sequence of words and punctuation marks.

- The symbols in the source program are typically
  - the names of data items manipulated by the program
  - operators such as + and -
  - special reserved words such as if and while

- Example: the statement
  ```plaintext
  if X ≠ 1 then set Y to X + Y endif
  ```
  is transformed into the token sequence
  ```plaintext
  if name ≠ number then set name to name + name endif
  ```
The program is parsed, i.e., its syntactic structure is determined by using the grammatical rules of the language concerned to analyze the sequence of tokens produced during lexical analysis.

The syntactic structure of any valid sequence of terminal symbols can be represented by its parse tree, which has the terminal symbols at its leaves and the syntactic category to which the sequence belongs to as its root.

The major task of the syntax analyzer is to construct the parse tree from the program text.

There are different parsing algorithms, which may depend on properties of the grammar.
Example of a parse tree (1)
Example of a parse tree (2)
Bottom-up parsing starts with the leaves of the parse tree and works up towards the root.

Combine neighboring tokens in the source program to form elements of simple syntactic categories.

These elements are then combined to form elements of more complex categories, and so on.

Syntax analysis is complete when successive combinations result in the entire source text being identified as a member of the category *program*.

At any stage several combinations may be possible, but if the syntax is unambiguous only one will lead to a complete parse tree. This process of controlled trial and error is an instance of **backtracking** – the exploration of a particular avenue and the return to explore other avenues if the first proves unsuccessful.
Top-down parsing

- **Top-down** parsing starts with the root of the parse tree and works down towards the leaves.
- Start with the hypothesis that the source text is a valid program.
- Verify the hypothesis by looking for subunits of the text which are members of syntactic categories occurring at the first level of the parse tree.
- Recognizing first level subunits involves looking for second level subunits which can possibly form them.
- This in turn involves looking for third level subunits, and so on.
- Some backtracking may be required.
Top-down parsing using module calls

- Algorithms for top-down parsing can often be derived from the productions of a language.
- Such an algorithm consists of modules, each of which corresponds to a certain syntactic category.
- The function of such a module is to parse character sequences which are members of that category.
- The body of each module consists of calls to other modules which perform parsing of members of simpler categories.
- The modules called are those which correspond to the categories on the right-hand side of the defining production.
Algorithm for parse-statement

Consider again:

\[
\text{statement} \rightarrow \text{conditional} \mid \text{loop} \mid \text{assignment}
\]

\begin{verbatim}
module parse-statement
{Outline of module to parse a statement}
pmove-conditional;
if not successful
    then parse-loop
        if not successful
            then parse-assignment
                if not successful
                    then report failure
                    endif
                endif
            endif
    endif
endif
endmodule
\end{verbatim}
Algorithm for parse-assignment

module parse-assignment

{Outline of module to parse an assignment}
check that the type of the next token is 'set';
check that the type of the next token is 'name';
check that the type of the next token is 'to';
parse-expression
endmodule

Remark: these kinds of modules reflect the structure of the defining production.
Eliminating backtracking

For rapid parsing, backtracking should be eliminated. In practice, this is achieved with a *lookahead* of at most one symbol.

```plaintext
module parse-statement
{Outline of module to parse a statement using single symbol lookahead}
if the type of the next token is 'if'
   then parse-conditional
else if the type of the next token is 'while'
   then parse-loop
   else if the type of the next token is 'set'
      then parse-assignment
      else report failure
endif endif endif endmodule
```
Dealing with syntax errors

Detection of syntax errors is easy, because the construction of the parse tree will fail at some stage.

However, the reason of a failure is not obvious, since the error may be hidden in part of the tree which has already been built. E.g the directory entry

Jones Mary 42 High 67431

is erroneously written

Jones Mary High 67431
Example of a parse tree of an erroneous directory entry

```
Entry
  |___________________________|
  |                           |
  | Person-name               |
  |   ________________________|
  |   |                      |
  |   | Surname              |
  |   |                     |
  |   | Forename            |
  |   |   ____________________|
  |   |   |                    |
  |   |   | Proper-forename   |
  |   |   |                   |
  |   |   | Name (Jones)      |
  |   |   |                   |
  |   | Forename           |
  |   |   ____________________|
  |   |   |                    |
  |   |   | Proper-forename   |
  |   |   |                   |
  |   |   | Name (Mary)       |
  |   |   |                   |
  |   | Forename           |
  |   |   ____________________|
  |   |   |                    |
  |   |   | Proper-forename   |
  |   |   |                   |
  |   |   | Name (High)       |
  |   |   |                   |
  |   | Address            |
  |   |   ____________________|
  |   |   | Street-name      |
  |   |   |                   |
  |   |   | Number (67431)    |
  |   |   |                   |
  |   |   | Name ??           |
```

Name (Jones)
Name (Mary)
Name (High)
Preventing certain kinds of syntax errors

Problem: relatively late detection of syntax errors

Solution: use certain terminal symbols to separate pieces of text which belong to different syntactic categories.

Semicolons and commas are popular choices here.

Modified production:

\[ \text{entry} \rightarrow \text{person-name, address, number} \]

Reconsider the erroneous entry

Jones Mary, High, 67431

The error can now be detected at the point it occurs.
Code generation

- Appropriate machine language statements are generated for each syntactic element of the program.
- Memory space is allocated for the data items manipulated by the program.
- Example: the language expression $N_1 + N_2$ is translated to
  LOAD A1
  ADD A2
where A1 and A2 are the addresses of the data items $N_1$ and $N_2$.
(The result is left in the accumulator, which is a special register. Registers in turn are special storage locations.)
Compilers translate *source programs* into *object programs*.

The syntax of the source programming language must be defined by a *grammar*.

A grammar consists of *terminal symbols*, *nonterminal symbols*, and *productions*.

The compilation process consists of the phases *lexical analysis*, *syntax analysis*, and *code generation*.

Lexical analysis transforms the program into a *token sequence*.
Syntax analysis generates a **parse tree**.

The main parsing principles are **bottom-up** and **top-down** parsing.

Algorithms for top-down parsing can often be derived from the productions of a language.

For rapid parsing, **backtracking** should be eliminated.

Code generation generates machine language statements and allocates memory for data items.