Foundations of Information Processing

Language implementation

Considerations about language implementation

Consideration 1:

How to utilize
the definition of a language,
e.g., a grammar,
when converting
from one language to another?

This process of converting is called translation.

Consideration 2:

What are the steps in translation from a high-level language program to a machine-language program?

Consideration 3:

How to simplify the translation

in language implementation?

A language can be defined either
enumerating words/structures involved with it,
or describing words/structures based on
properties, approved by an automaton,
or generated by a grammar.

The well-formed language enables
to convert structured information automatically
to another form,
and thus, to compile or to interpret
the program of a chosen language.

Language implementation

- Converting one language to another (translation)
  - From a high-level language to a machine-language.
  - Compilers.
  - Interpreters.
- Syntax and grammars.
- The steps of the translation process.
  - Lexical analyzer.
  - Parser.
  - Code generator.
    - Memory allocation.
    - Construction of machine-language instructions.
    - Optimization of the code.

Source (partly/modified): Brookshear, J.G., Brylow, D., *Computer Science - An overview*, 13th ed. Addison Wesley, 2019.

Translation: compilers and interpreters

- An algorithm written by a well-defined, natural language alike *high-level* programming *language* is converted to the form of the *machine-language* which can be executed by a computer.
- Each *instruction* is *translated* to one or more *machine-language* instructions.
- Interpreters:
  - The program is executed instruction by instruction.
  - The interpreter itself is a machine-language program.
  - Interactive (finding programming errors, debugging), but slow (instruction by instruction).
  - For example, Matlab.
- Compilers:
  - First the whole program is translated to the machine-language, and only then executed.
  - Faster to execute (important especially with repetition structures).
  - For example, the C language.

Syntax of a language

- An algorithm is a sequence of characters which belong to some set of characters (the alphabet).
- Syntactic grammar rules define what kinds of strings and their structures form the language.
- The syntax of the language is a set of these syntactic rules, and can be done using
  - an automaton or
  - a grammar.
- Syntax does not consider the content of a language (semantics):
  - A monkey ate a banana.
  - A banana ate a monkey: syntax correct, semantics incorrect.

Grammars

- A grammar defines syntax: *a set on rules* how to use strings correctly.
- A program is written by a language following the grammar rules.
- The syntax of *a programming language* is usually defined by *context-free grammar (CFG)*: each symbol is always interpreted in the same way regardless of the surrounding context.
- CFG is defined as follows (Chomsky Grammar *G*):
  - Nonterminals *N*.
  - Terminals Σ.
  - Production rules *P*.
  - The start symbol *S*.
- Prof. Noam Chomsky.
  - MIT, UCB, Columbia U.
  - An American linguist, philosopher, cognitive scientist, historian, social critic, and political activist.

λ denotes the empty string

$S \rightarrow aSb \mid \lambda$

Production rules

$P = \{S \rightarrow aSb, S \rightarrow \lambda\}$

$G = (\mathrm{N}, \Sigma, P, S)$

$\mathrm{N} = \{S\}$

Nonterminals

$\Sigma = \{a, b\}$

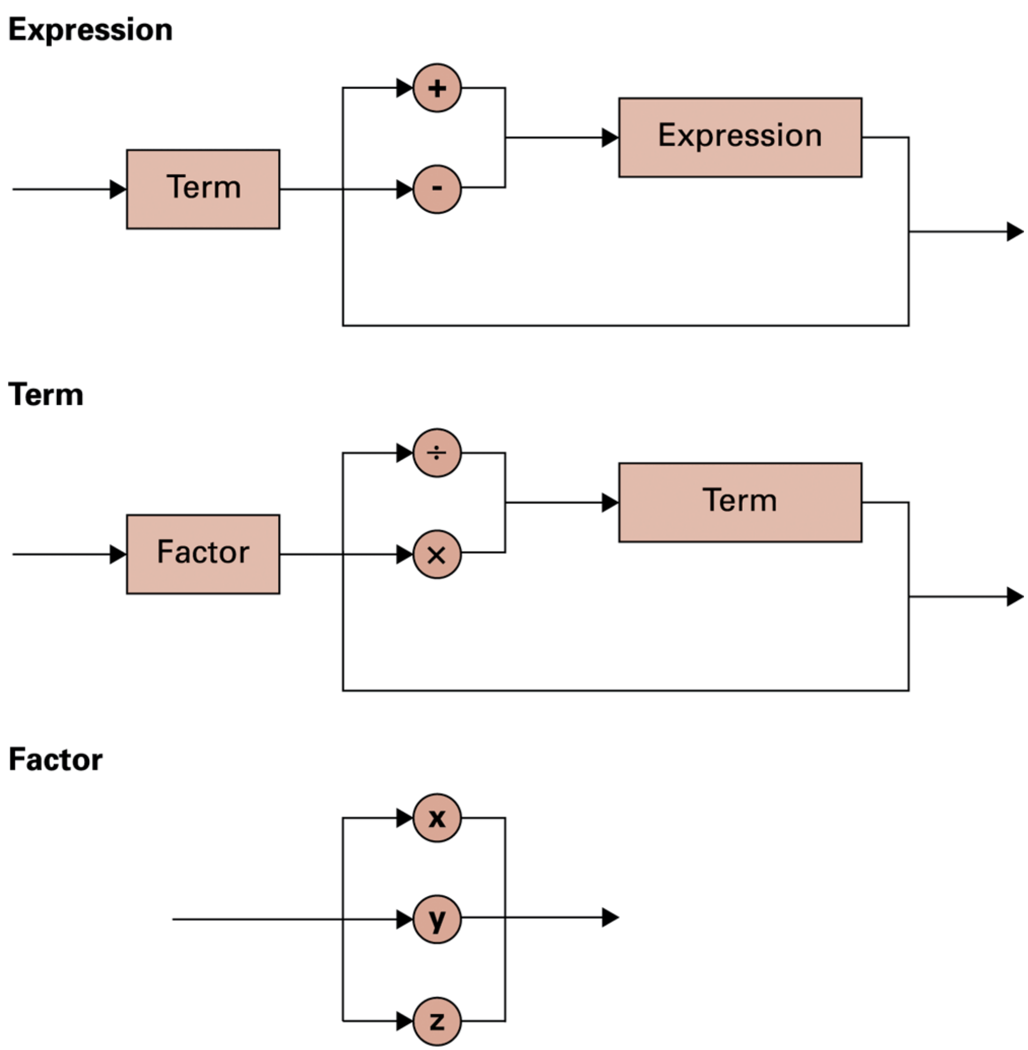
Terminals

Start symbol S

Possible productions: *ab* *aabb* *aaabbb*

Not possible: a aab abab bba b

Syntax diagrams

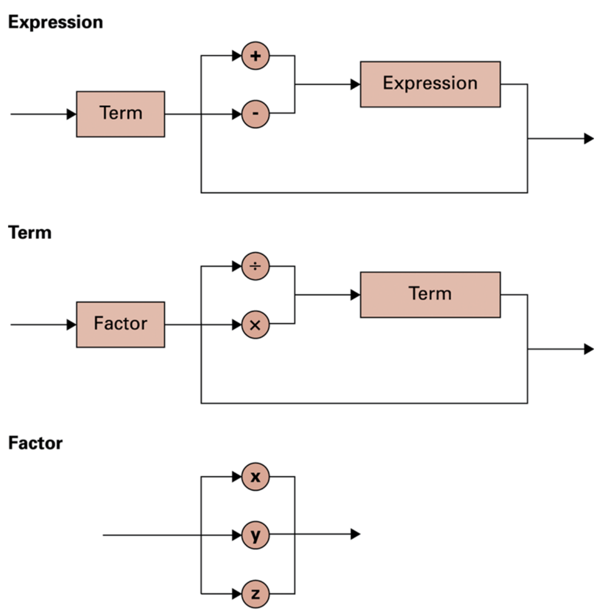


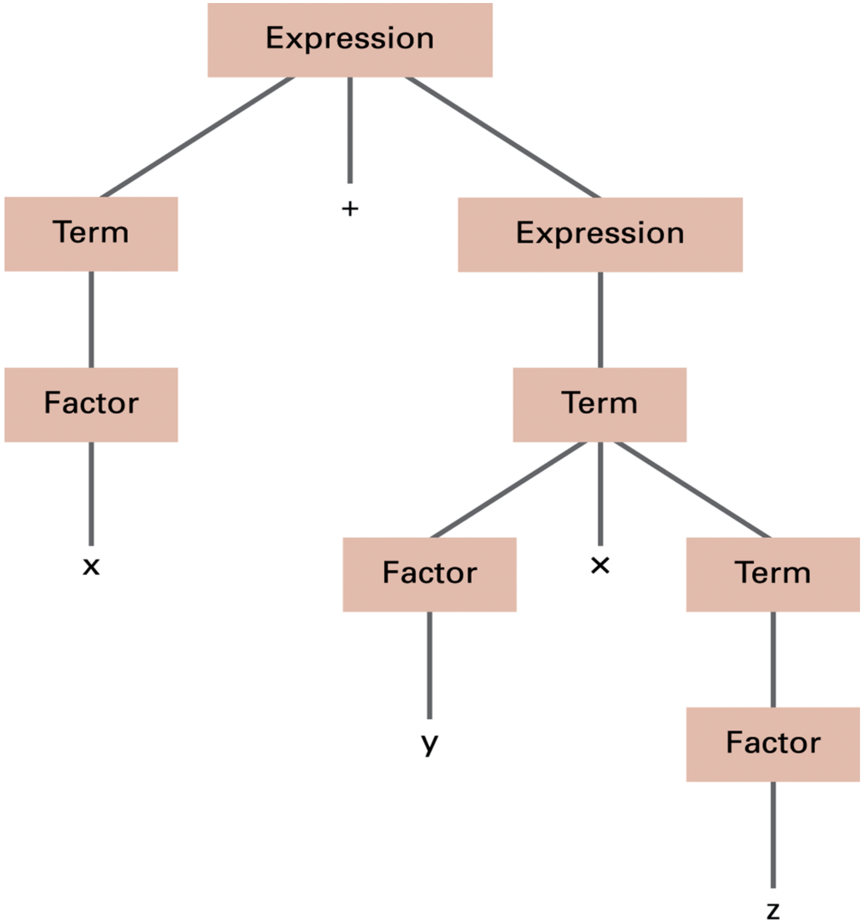
What is wanted to be defined here?

The structure of simple arithmetic expressions.

Sparse trees

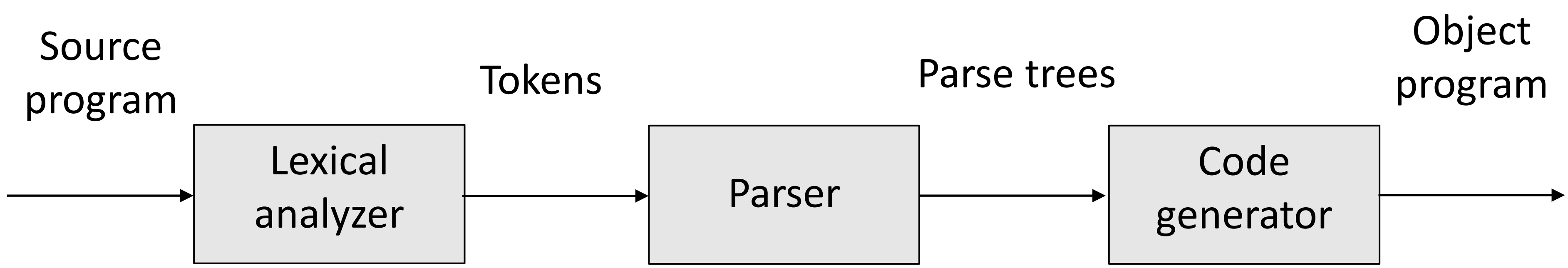
The parse tree for the string
x + y x z
based on the syntax diagrams shown on the previous page.





The translation process

- The source program (high-level language) is translated to the object program (machine-language) as follows:
  1. The lexical analyzer recognizes which strings of symbols represent a single entity (a token) and sets the types of tokens (the keywords of the language, etc.).
  2. The parser performs the syntactic analysis symbol by symbol.
  3. The code generator produces the machine-language program.



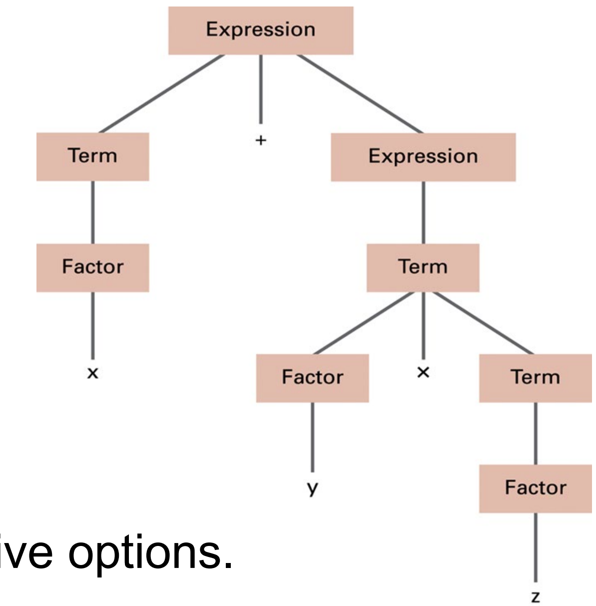
Lexical analysis

- Characters belonging to one string (token) are recognized based on separators.
- Defines the type of each token.
- Tokens are terminal symbols (terminals):
  - Keywords (reserved words): MODULE, IF, THEN, ELSE, CASE, OF, etc.
  - Operators: =:, =, <>, <, >, >=, <=, +, -, *, /
  - Punctuation marks: parentheses, spaces, line breaks, commas, semicolons, etc.
  - Factors: variables, named constants, names of modules.
  - Numbers and strings: -354.786, ”A”.
- The factors are stored to a symbol table for later use: the name and the type.
- Example: Analyze lexically the following part of a program:

  IF x < 5 THEN x := x + 1 ENDIF

Symbols:	IF	x	<	5	THEN	x	:=	x	+	1	ENDIF
Types:	keyword	factor	oper.	num.	keyword	factor	oper.	factor	oper.	num.	keyword

Parsing



- The *syntactic structure* of the source program is defined using *parser trees* based on syntax.
- Parsing approaches:
  - Top-down: from the root to the leaves.
    - Easy to derive from grammar rules.
  - Bottom-up: from the leaves to the root.
    - Backtracking might be needed due to alternative options.
- Two concepts makes parsing easier to implement:
  - Context-free grammar: each symbol is always interpreted in the same way regardless of the surrounding context.
  - Uniqueness of symbols: the unambiguous definition of each text element.
- Example:

Based on the corresponding syntax of the selection statement

IF <Boolean expression> THEN <Statements> [ELSE <Statements>] ENDIF

our example IF x < 5 THEN x := x + 1 ENDIF can be analyzed by parsing.

Code generation

- The syntactic structure of the original source program presented in the sparse tree is used.
- The machine-language-based program is generated as follows:
  - Symbolic machine-language => machine-language (the assembler compiler).
  - Finally, the program is converted to the language of *microprogramming* (microcode) which is implemented by a micro interpreter bit by bit between the different components of a computer in clock cycles.
- The following steps are needed:
  - Memory allocation.
  - Construction of machine-language instructions.
  - Optimization of the code.
- Microprogramming is to be considered in the course “Foundations of Computer Science”.

Memory allocation

- Memory is allocated for the factors, usually variables.
- The corresponding address of each memory cell is attached.
- This is done in the symbol table.

sum:=value+1

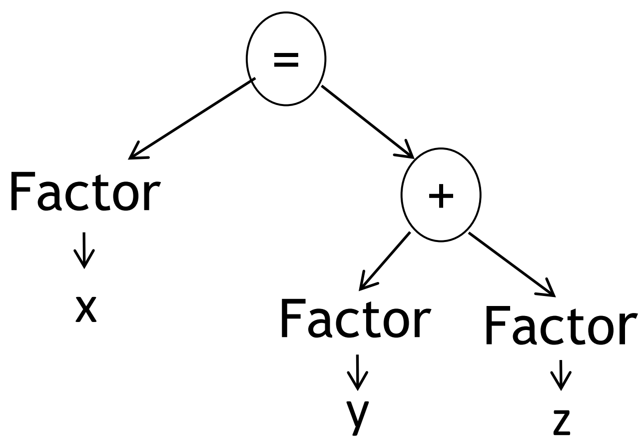
symbol:='a'

factors	type	size	address
sum	integer	4	245
value	integer	4	249
symbol	character	1	253
...			254

Construction of machine-language instructions

- Generate symbolic machine-language based on the corresponding parse tree.
- Arithmetic operations are computed using the special register (memory cell) called the accumulator.

x := y + z




```
MODULE assignment_statement(x, y, o, z)
  print LOAD y
  CASE o of
    '+' : print ADD z
    '-' : print SUBTRACT z
    '*' : print MULTIPLY z
    '/' : print DIVIDE z
  ENDCASE
  print STORE x
ENDMODULE
```

LOAD y	Set y as the accumulator value
ADD z	z + the accumulator value stored in the accumulator
STORE x	Assign x to the accumulator value

Optimization of the code

- In the direct translation each instruction (sentence) is converted instruction by instruction

  => unnecessary or extra parts might be generated.
- Generated machine-language instructions must be considered together, also related to each other.
- Change the code to be more efficient for a computer:

```
LOAD Y
ADD Z
STORE X
LOAD X
MULTIPLY A
STORE B
```

→

```
LOAD Y
ADD Z
MULTIPLY A
STORE B
```

- The intermediate result X (the result of the addition of Y and Z) is not needed to store and to access when the next instructions are known.
- X is used automatically in MULTIPLY A from the memory address (the acumulator) where the result of ADD Z is stored.
- This is one of the properties of the machine-language.

Summary

- **In interpretation** each statement is converted (translated) from the source program to the machine-language program and is executed immediately, statement by statement.
- **In compilation** the whole algorithm is converted first and then the whole converted program is executed.
- Each statement of a **high-level** programming language is translated to one or more **machine-language** statements (instructions).
- The translation process consists of **lexical analysis** and **parsing** of the source program, and **code generation** of the object program, usually the machine-language program.