SECURITY ENGINEERING OF VIRTUAL MACHINES AND COMPILERS

Software engineering

Content:

1. Introduction:

- **Software engineering;**
- **Software re-engineering;**
- **Reverse engineering;**

2. Compiler Design:

- **Introduction;**
- **Components and stages;**
- **Lexical analysis;**
- **Syntactic analysis;**

Software engineering

Content:

2. Compiler Design:

- **Semantic analysis;**
- **Generating the intermediate code;**
- **Optimization of intermediate code;**
- **Treating the errors;**
- **Generating the object code;**

3. Binary encoding of assembly instructions to machine/native code;

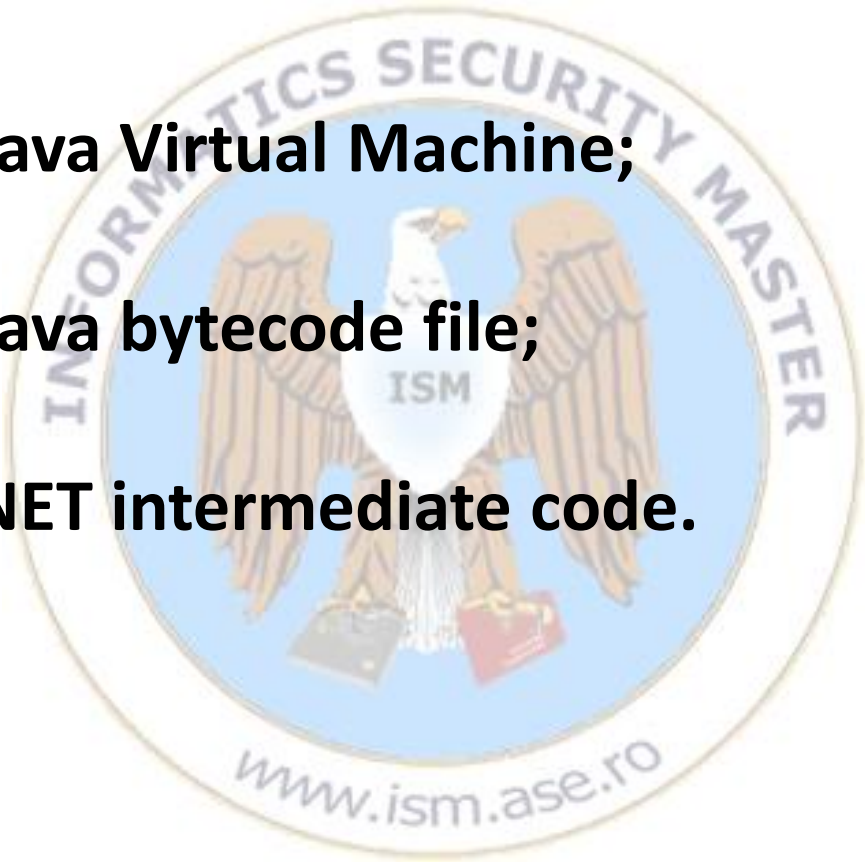
Software engineering

Content:

4. Structure of Java Virtual Machine;

5. Structure of Java bytecode file;

6. Structure of NET intermediate code.



1. Introduction

Software engineering

Definition:

- **Systemic, disciplined approach, quantifiable for software development, usage and maintenance;**

- **Study of approaches for software development, usage and maintenance.**

Software engineering

Goal:

- **Creating software with the following characteristics:**
  - **High quality level;**
  - **Price accessible to users;**
  - **Easy to maintain;**
  - **Easy and fast to develop.**

Software engineering

Sub-disciplines:

- **Software requirements: identification, specification, validation of software requirements;**

- **Software design: usually made with CASE tools and uses standard fors such as UML;**

Software engineering

Sub-disciplines:

- **Software development: build a software application by using programming languages;**
- **Software testing: use standard procedures for checking software functionalities;**

Software engineering

Sub-disciplines:

- **Software maintenance: treating the aspects related to functioning problems and the need of software improvement;**

- **Software configuration management: using structured and standard methods in the software complexity management (application versions, sources etc.);**

Software engineering

Sub-disciplines:

- **Softare engineering management: inspired from the project management – resource planning, organization and management for achieving successfully the objectives;**

- **The software development process: choosing and using development methodologies;**

Software engineering

Sub-disciplines:

- **Software engineering tools: using CASE – Computer-Aided Software Engineering for obtaining high quality level software, defect-free and easy to maintain;**

- **Software quality: measuring the quality level: projecting vs. requirements, implementation conformity vs. project.**

System engineering

Definition:

- **Inter-disciplinary feature;**
- **The way of projecting and managing complex projects;**
- **Complexity aims at: logistics, coordination, control.**

Software re-engineering

Definition:

- **Examining and modifying the system for its rebuilding in a new form;**
- **Modification of a software system for adding new functionalities or correcting errors.**

Software re-engineering

Reverse engineering:

- **Initial examination of the system;**
- **Identification of technological principles related software structure, functions and operations;**

Software re-engineering:

- **Later modification of the software system.**

2. Compiler Design

Compilers - introduction

The need to study compiler development:

- **Fundamental branch of informaics;**
  - **Problem structuring: separation between analysis and synthesis – analysis of input -> semantic representation of input data -> synthesis of output; analysis and synthesis are divided in the compiling phases;**
  - **Description of algorithm mechanisms;**
  - **Using utilities;**

Compilers - introduction

Need to study compiler development:

- **Very wide applicability: techniques and algorithms are successfully used also in other programming fields;**
- **Making the programming experience close to perfect: choosing the adequate data structures, efficient algorithm implementation.**

Compilers - introduction

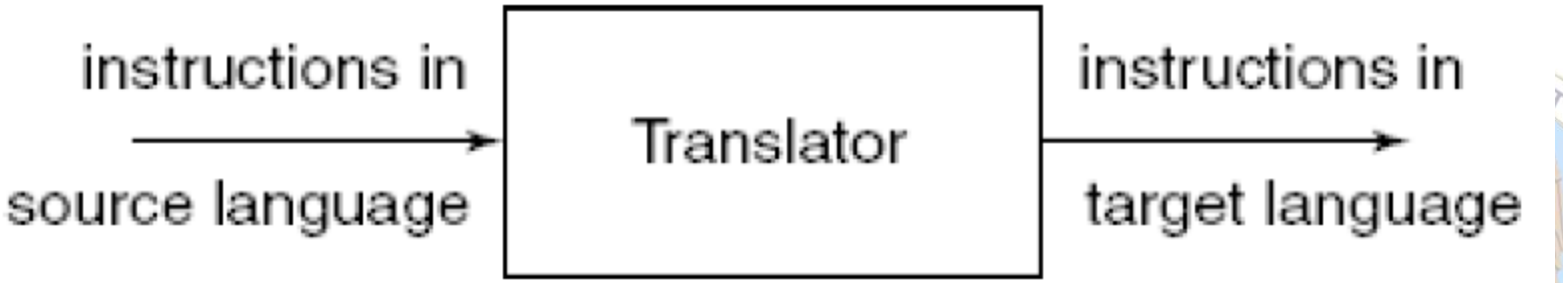
The programmer:

- **User of a compiler, which allows to:**
  - **Easy solving the compiling and running errors;**
  - **Maximum usage of the programming language facilities.**

Compilers - introduction

The translator:

- **Input: source program written in a programming language;**
- **Output: program written in another language, keeping the significance of the source program.**



Compilers - introduction

The assembler:

- **Input: source program written in an assembly language;**
- **Output: executable or object (almost executable) code;**
- **Method: replacement of symbolic information by numeric (binary or hex) information; it needs 2 tables: op-code table and symbol table.**

Compilers - introduction

The interpreter:

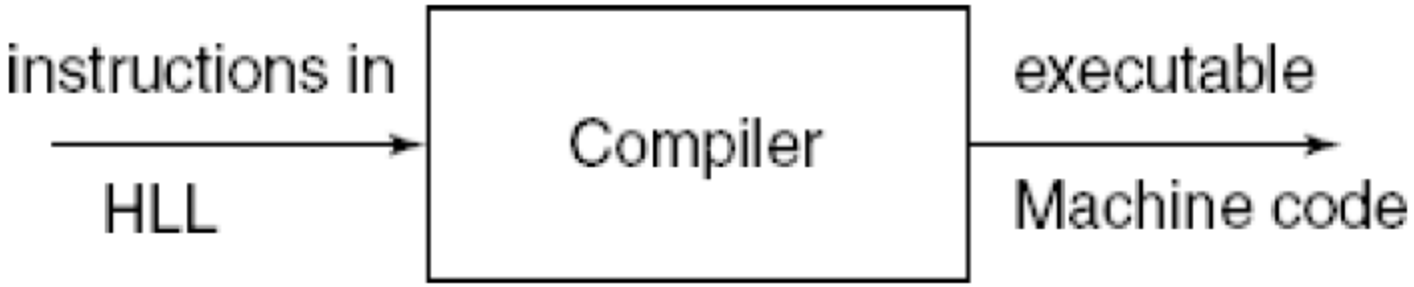
- **Input: source program written in a programming language;**
- **Output: direct execution of the input; analysis of each statement in the source code.**

HLL – High Level Language

Compilers - introduction

The compiler:

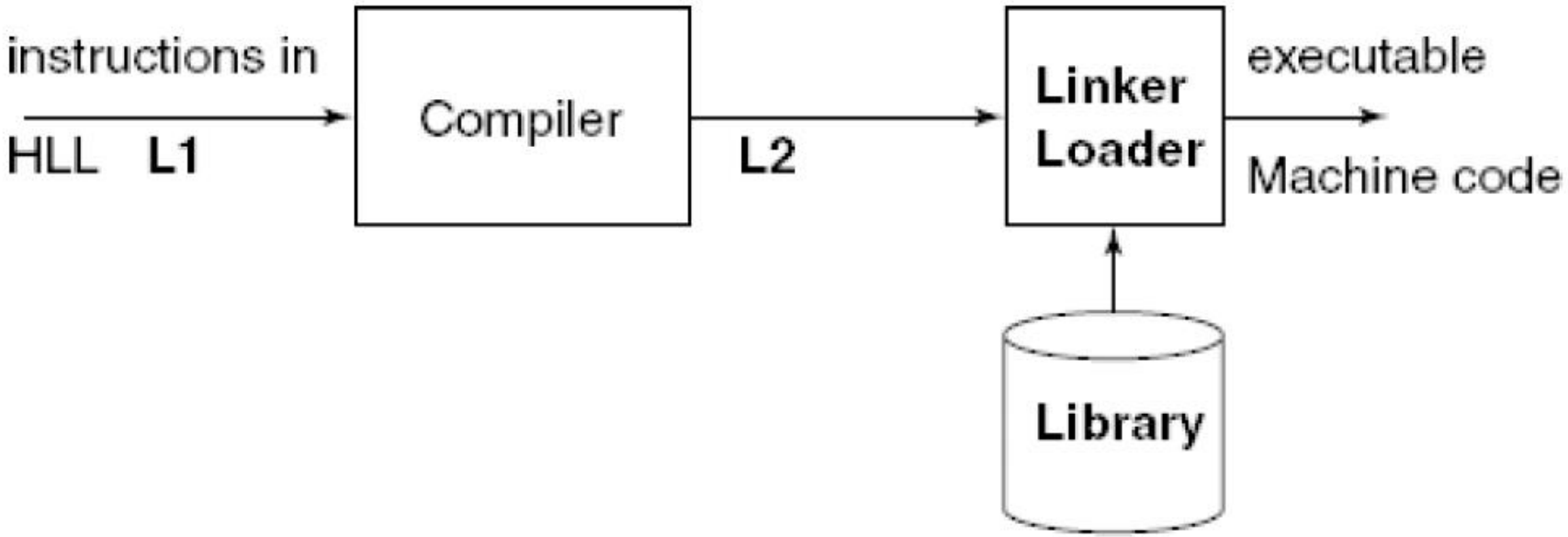
- **Translator for which the object program is in machine code or almost machine code.**



Compilers - introduction

The compiler actions:

- **Compiler - a large set of programs / modules;**
- **L1 – usually a HLL source;**
- **L2 – usually a form of the binary machine language; it is not a pure machine language.**



Compilers - introduction

The linker:

- **Program linking together several object modules and libraries to form a single, coherent program;**
- **Applied for data and program entry points in one module are possible from other modules;**
- **Object module – machine code output from an assembler/compiler and contains executable machine code and data together with information necessary to the linker.**

Compilers - introduction

The object module structure

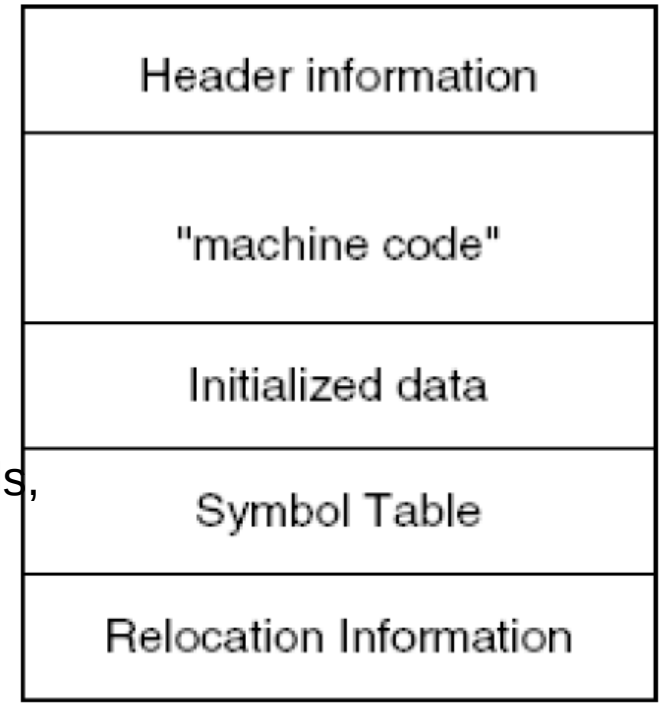
Header – lengths of each of the remaining sections and length of the uninitialized data area (bss);

Machine code – generated binary executable code; (txt)

Initialized data – data values known to the compiler, constants and initialized variables; (data)

Symbol table – local symbols, undefined external symbols, public symbols;

Relocation information – including linking information:
1. A list of all the symbols and their positions within this module which can be referenced by other modules. (esd)
2. A list of all the symbols not defined in this module but referenced. (esd)
3. A list of address constants, their positions and how to adjust them. (rld)



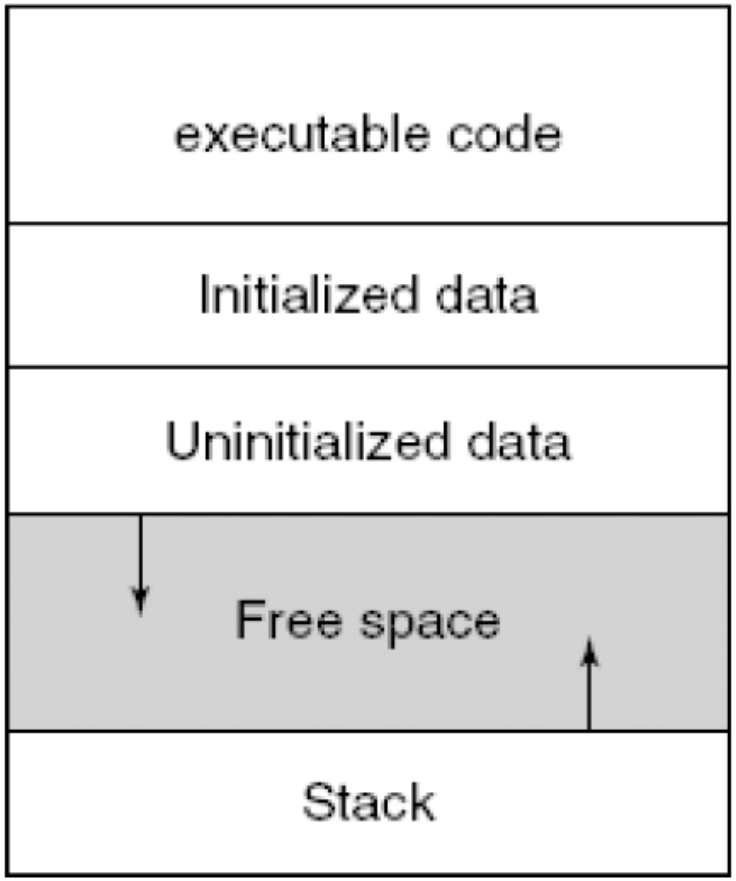
Compilers - introduction

The loader:

- **Utility program which:**
- **Reads an executable program file in a particular for from a secondary storage unit;**
- **Puts proper bitpatterns in the memory at specified place (instructions and data);**
- **Initiates execution of that program;**

Compilers - introduction

The process memory structure



Compilers - introduction

Transformation of the source program in machine code or other language:

- **Analysis:**
  - **Determining the lexical and synthetical structures in the program;**
  - **Determining the significance of the structures in the program;**
  - **Creating an abstract, language-independent representation;**

Compilers - introduction

Transforion of the source program in machine code or other language:

- **Synthesis:**
  - **Taking over an abstract representation and generating an object code;**
  - **Are taken into account the characteristics of the computer on which the code is executed.**

Compilers - introduction

Analysis and synthesis: complex processes run in several steps:

1. **Lexical analysis:**

- **Identification of lexical tokens;**
- **tokens: substrings of characters delimited in the source program: key words, operators, constants, identifiers, separators;**

Compilers - introduction

Analysis and synthesis: complex processes run in several steps:

2. Syntactic analysis:

- **Determining the syntactic entities: statements, expressions, instructions;**
- **Syntactic correctness of the program;**
- **Build a syntactic analysis tree: equivalent representation of source program: syntactic entities and relations between them;**

Compilers - introduction

Analysis and synthesis: complex processes run in several steps:

3. Semantic analysis:

- **Verification of consistency when using variables, types, expressions;**
- **Build an analysis tree enriched with attributes – abstract representation;**

Compilers - introduction

Analysis and synthesis: complex processes run in several steps:

4. Generating the Intermediate code:

- **Obtaining the simple form of the program;**
- **Contains instructions with established for which works with operators and variables;**
- **Easy generation of machine code for computers as diverse as possible.**

Compilers - introduction

Analysis and synthesis: complex processes run in several steps:

5. Code optimization:

- **Applied on the Intermediate code and/or object code;**
- **Compromise between object code efficiency and compilation duration;**
- **Optimization operations: registers allocation, load and save;**

Compilers - introduction

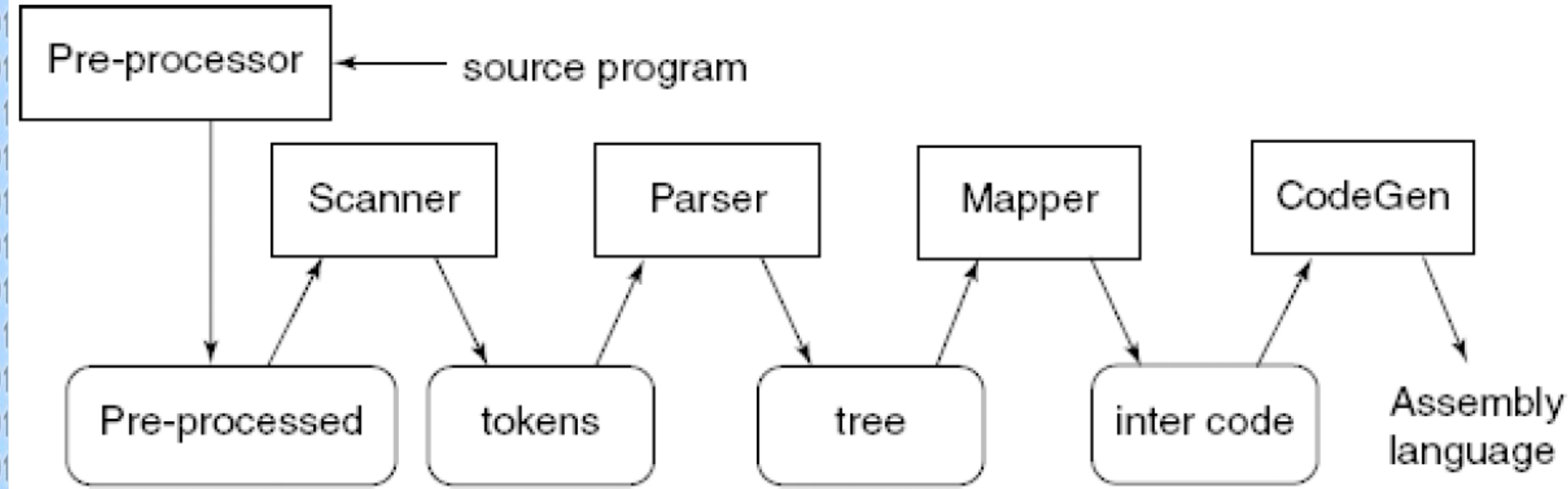
Analysis and synthesis: complex processes run in several steps:

6. Generating object code:

- **Register allocation and management;**
- **Loading registers from variables;**
- **Translating the Intermediate code instructions into object code.**

Compilers - introduction

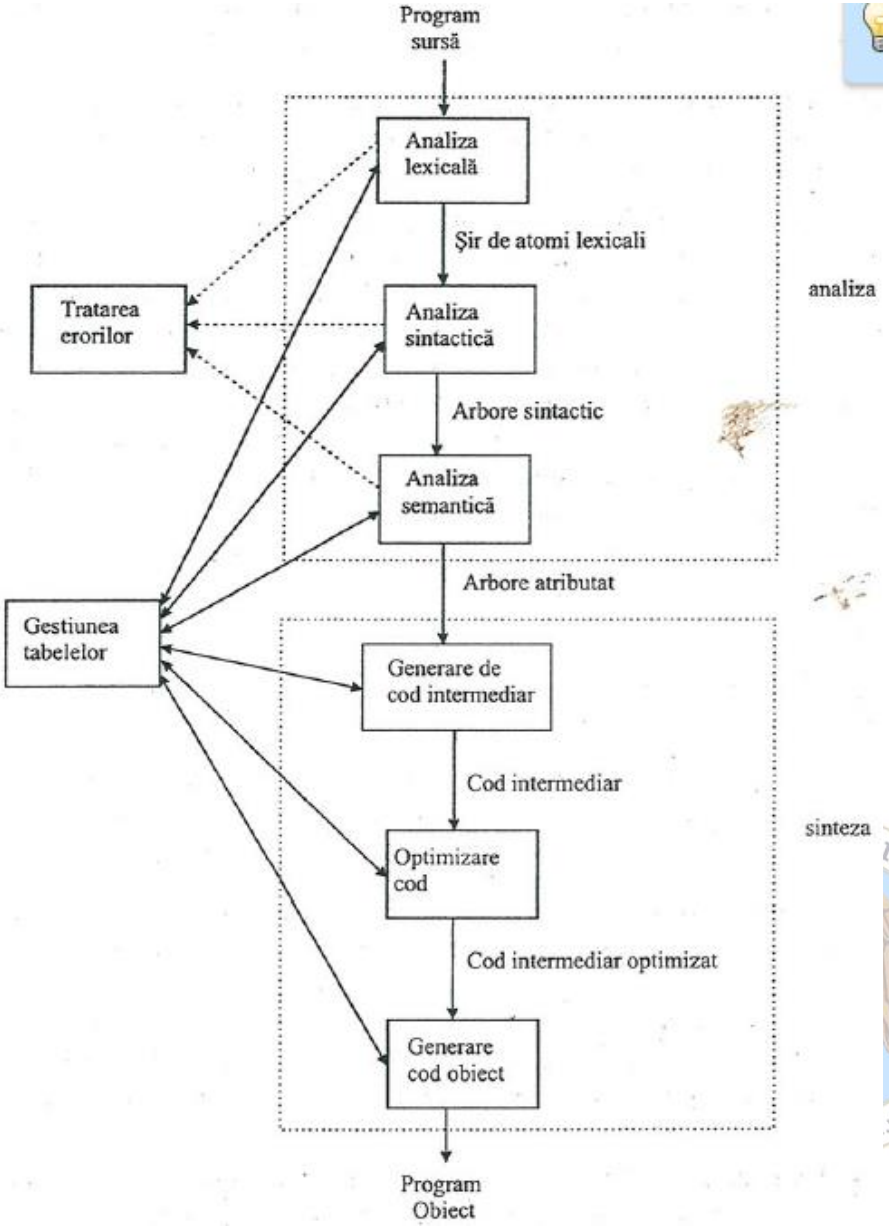
Compile process steps



Scanner – Lexical Analysis
Parser – Syntax Analysis
Mapper – Semantic Analysis

Compilers - introduction

Compile process steps



Compilers - introduction

Components of the compiling process:

1. Symbol tables management:
   - Creating, accessing, updating information for the symbolical names in the program;
   - Created in the lexical analysis phase and used in the other phases;

2. Treating errors: routines to treat the errors that appear in the phases of the compiling process.

Compilers - introduction

The programming language:

- Notation to describe some computations;
- Problem describing and implementing properties.

Compilers - introduction

Meta-language:

- Describing language;
- Rules according to the syntax of a language;
- BNF Backus-Naur-Form notation: basic or terminal symbols (key words, operators, separators), meta-linguistic variables (non-terminal symbols), meta-linguistic connectors (equality, alternativeness).

Lexical analysis

The first phase of compiling: transforming the source program:

- **Token detection: identifying substrings;**
- **Token classification: dividing tokens into categories;**
- **Token codification: writing in a simpler form.**

Lexical analysis

A. Detection: difficulties in projecting a lexical analyzer:

- Importance of blank spaces;
- Avoiding the incorrect detection of tokens by the “look ahead” technique (== vs. =, <= vs. < etc.); cases met when dealing with operators and operations of attribution with equality test.

Lexical analysis

B. Classification:

- Every token must belong to a single class; otherwise, the lexical analysis fails (lexical error);

- Token classes: describe all the important elements in a language:
  - Identifiers: character string that mandatory starts with a letter and does not contain space;
  - Constants: character string that contains only numbers, it may start with +, - or .;

Lexical analysis

B. Classification:

- **Token classes: describe all the important elements in a language (continued) :**
  - **Reserved words (key words): defined in the programming language: if, else, for, while etc.;**
  - **Operators: arithmetical, relational, logical;**
  - **Separators (delimiters): ( ) ; { } etc..**

Lexical analysis

C. Codification:

- Reduce the space occupied by lexical tokens;
- To every token is associated a code (small integer number);
- Build a symbol table with tokens (identifiers, constants) that appear a single time;

Lexical analysis

C. Codification (continued):

- **Building the internal form of the program as a result of lexical analysis: pairs sequence (token code, position in the symbol table / pointer to the position); for reserved words, operators, separators, the position is filled with a non-existent value**

Lexical analysis

C . Codification (continued):

- **Algorithm for lexical analysis:**
  - **Token identification;**
  - **Complete the PIF (program internal form) with the token having the code and the place position in the symbol table (ST)**
  - **The position of the atom token in the symbol table; if the token is not found in ST, then it is added to the table.**

Lexical analysis

C . Partial codification – example:

Atom	Cod
Identificator	1
Constantă	2
+	3
-	4
...	...
==	10
(	11
)	12
{	13
}	14
=	15
;	16
if	17
else	18
for	19
while	20
do	21
...	...

Lexical analysis

C . Program internal form – example:

```
if ( x == z) {
      x = y ;
}
```

Cod Atom	Poziţie în TS
17	0
11	0
1	1
10	0
1	3
12	0
13	0
1	1
15	0
1	2
16	0
14	0

Position 0 means no entry in symbol table for that language token.

Lexical analysis

Finite automaton:

- **5-tuple $M = (Q, \sum, \delta, q_0, F)$:**
  - **Q finite set of statuses;**
  - **$\sum$ finite alphabet;**
  - **δ transition function $\delta : Q \times \sum \rightarrow P(Q)$;**
  - **q_0 initial status of the finite automaton;**
  - **F included in Q the set of final status;**

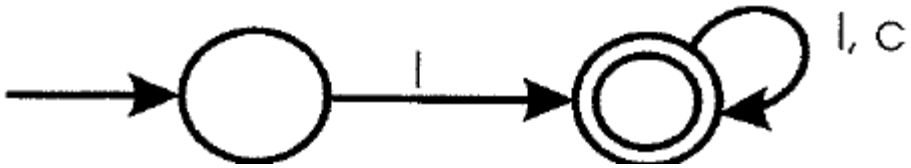
Lexical analysis

Finite automaton:

- **The transition function and an input symbol shape the detection mechanism of a lexical token;**
- **Lexical analyzer: finite automaton of the union of finite automaton for all the lexical tokens;**

Lexical analysis

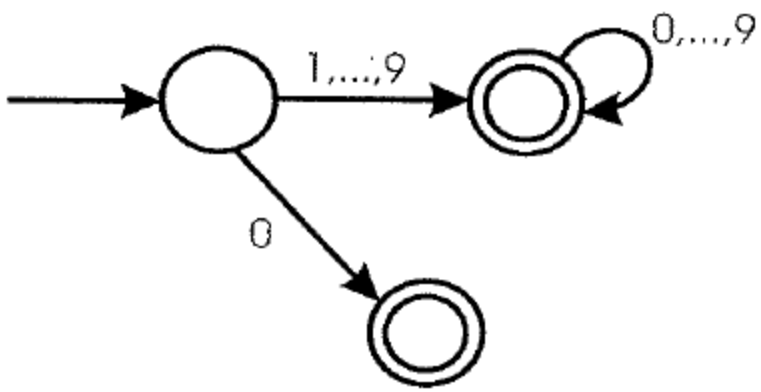
Finite automaton – example:



Finite automaton for identifiers declared in source code

Lexical analysis

Finite automaton – example:



Finite automaton for unsigned integer constants

Lexical analysis

Symbol table:

- All the information related to symbolic names;

- Alternatives:
  - Unique tables: all the symbolical names, no matter their type;
  - Distinct tables: for each type of symbolical names (identifier, constant);

Lexical analysis

Symbol table:

- Information differs from a compiler to another and allows to get the type, memory address, value;
- For each symbolical name, there is an unique entry in the table;

Lexical analysis

Symbol table – support data structures:

- **Operations on the ST: search, insertion; the result is various organization forms which optimize the operations:**
  - **Un-ordered table: insertion in the identification order in the program; sequential search;**
  - **Ordered table : insertion so that the table remains ordered; binary search;**
  - **Balanced search binary tree;**
  - **Hash tables: the most effective for symbol tables;**

Lexical analysis

Hash tables:

- Using the hash function in order to determine the location where is memorized the symbolical name;
- The hash function must uniquely map the identifiers to integers;

Lexical analysis

Hash tables:

- **Unique symbol table: constants treated as strings;**
- **Collision: Several symbolic names have the same value associated by the hash function;**

Lexical analysis

Hash tables:

- **Solving collisions:**

  - Open addressing: memorizing symbolic names in the hash table by searching the first free position if the initial position is occupied; at the end of the table, is searched the free position at the beginning of the table; the completion of the table solved by:

  a. Creating an additional table;

  b. Modifying the hash function for re-sizing the hash table;

  c. Increasing the table and re-hash of the elements;

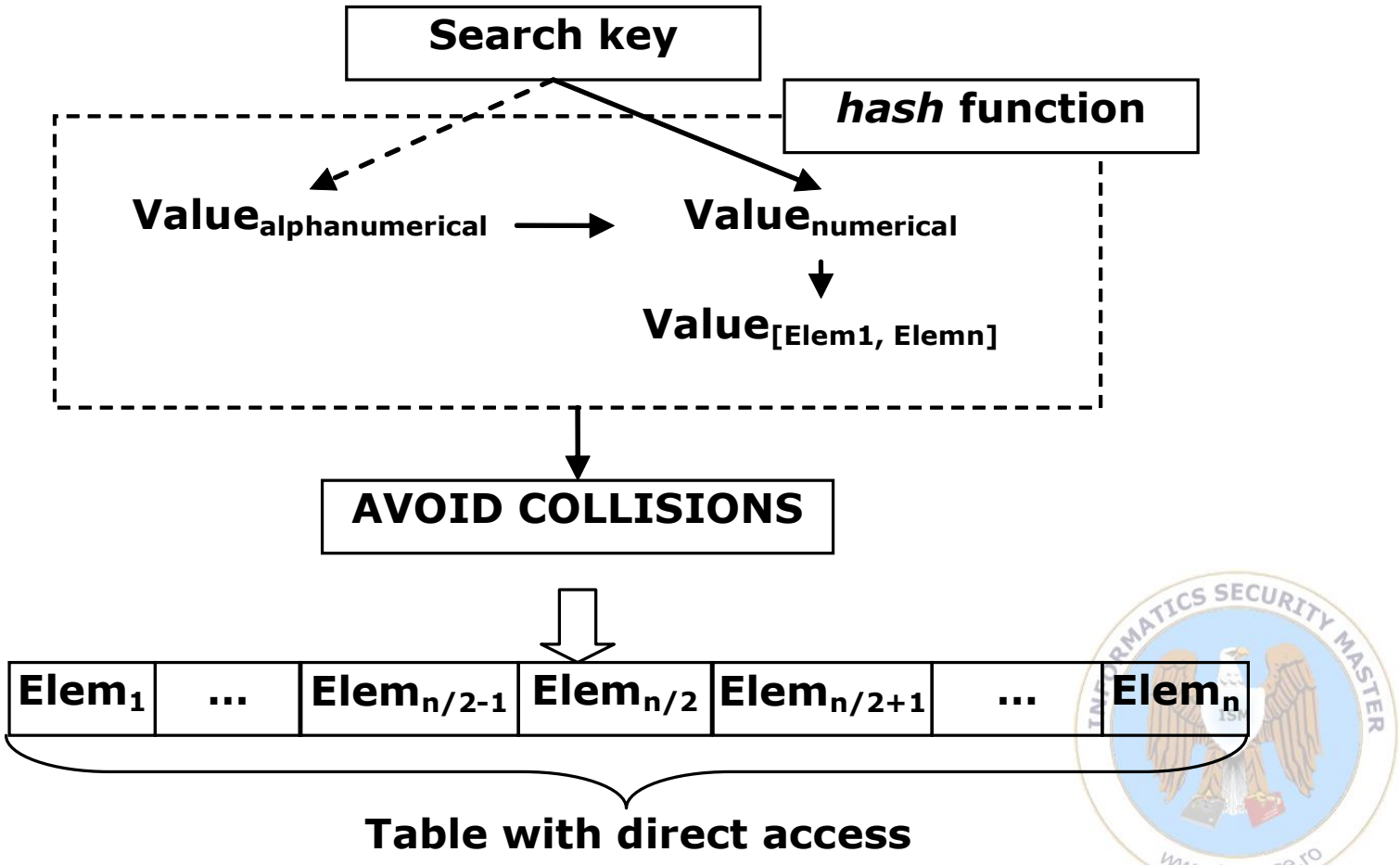
Lexical analysis

Hash tables:

- **Solving collisions (continued):**
  - **Chaining (auxiliary area): memorizing the symbolical names in a list; ST will contain for every record a pointer to the list;**

Lexical analysis

Hash tables:



Lexical analysis

Math models of the hash function:

- **Division in module:**
  - **Low complexity;**
  - **Easiness in implementation;**
  - **The key search is transformed into a numerical value and then transposed in the set [0; n-1];**

table_position = key_val % basis_val

table_position: the hash value obtained
key _val: the numerical key value
basis_val: the size of the hash table

Lexical analysis

Math models of the hash function:

- **Multiplication by a random real number:**
  - **Random real number in [0;1);**
  - **Process the decimal part included in [0 ;1);**
  - **Multiply the result by the size of the hash table;**

$$hash\_val=((key\_val*random_{[0;1)})-[(key\_val*random_{[0;1)})])*n$$

hash_val: hash value
key_val: the value of the search key
$random_{[0;1)}$: random number from [0;1)
n: the size of the table

Lexical analysis

Math models of the hash function:

- **Processing the ASCII codes:**
  - **Alphanumerical characters;**

$$hash\_vals_1 = string\_key[0] \% 255$$

$hash\_vals_1$: hash value
string_key: the value of the search key

- **Inefficient model: generates many collisions for different strings which begin with the same character**

Lexical analysis

Math models of the hash function:

- **Refining the model: taking over some characters in the string for which is determined the hash value (the first and the last character):**

$$hash\_vals_2=(string\_key[0]+string\_key[length_{string\_key}]) \% n$$

$hash\_vals_2$: hash value
string_key: search key
$length_{string\_key}$: the size of the character string
n: the size of the hash table

Lexical analysis

Avoiding collisions:

- Methods to find elements described by keys with different values, but which lead to identical hash values;
- Chaining, re-hashing, linear probing, quadratic probing, overflow area

Lexical analysis

Avoiding collisions:

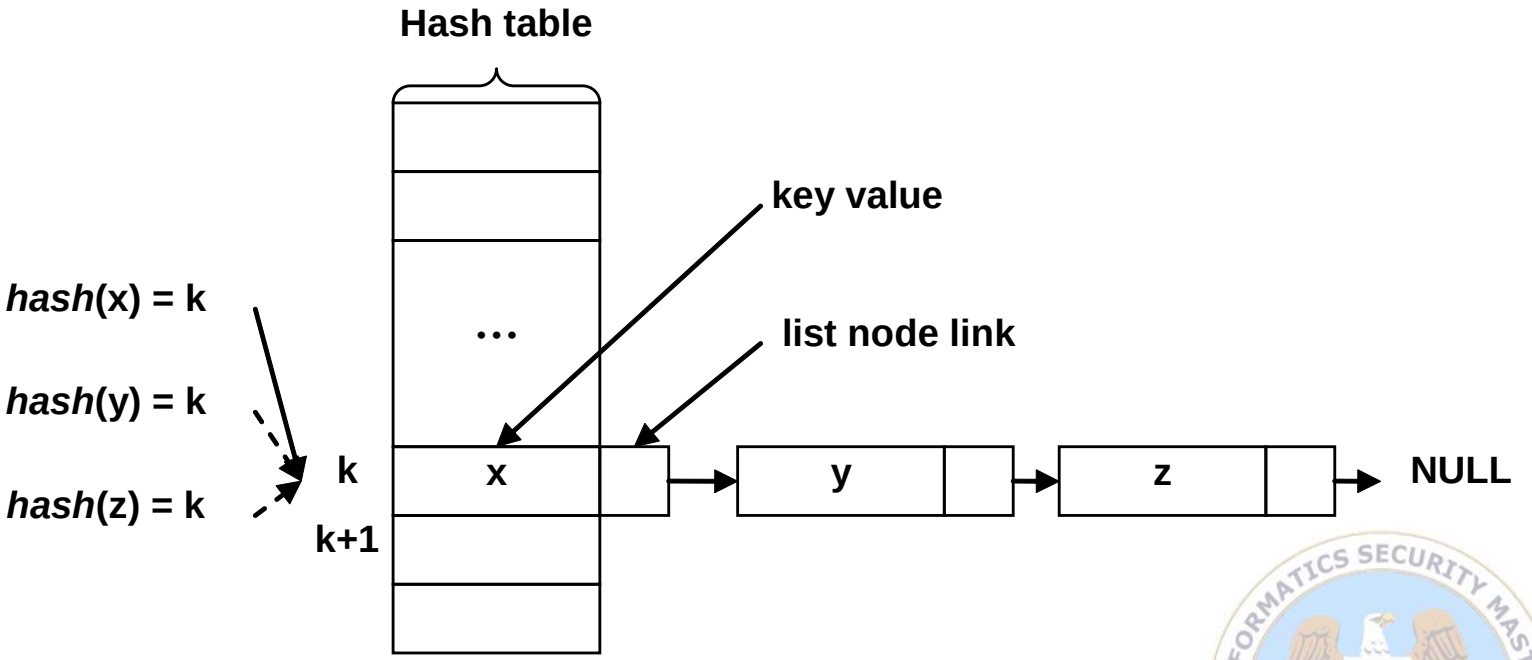
Chaining:

- **Implements working with lists;**
- **Every position in the hash table contains the address of a list of elements with equal hash values;**
- **Finding an element: determine the position in the table by computing the hash value and sequentially go through the list attached to that position.**

Lexical analysis

Avoiding collisions:

Chaining:



Lexical analysis

Avoiding collisions:

Re-hashing:

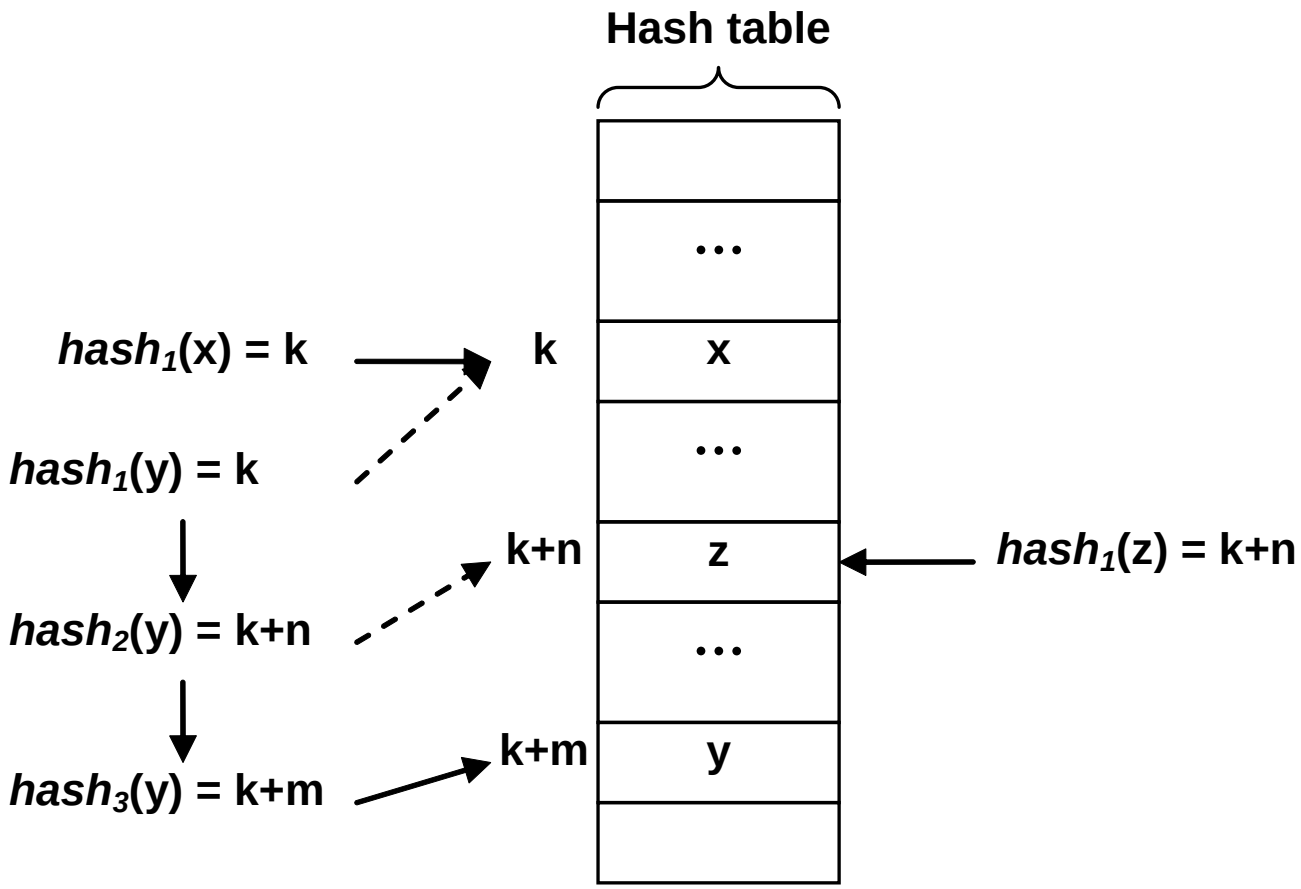
- **Applying in cascade the same hash function or another model in a set of functions until the obtained value represents a free position in the hash table;**

- **At each step of the search process: the value of the search key is introduced in a list of hash functions until the element is identified with the searched value or there are no more possibilities to re-compute the hash value.**

Lexical analysis

Avoiding collisions:

Re-hashing:



Lexical analysis

Avoiding collisions:

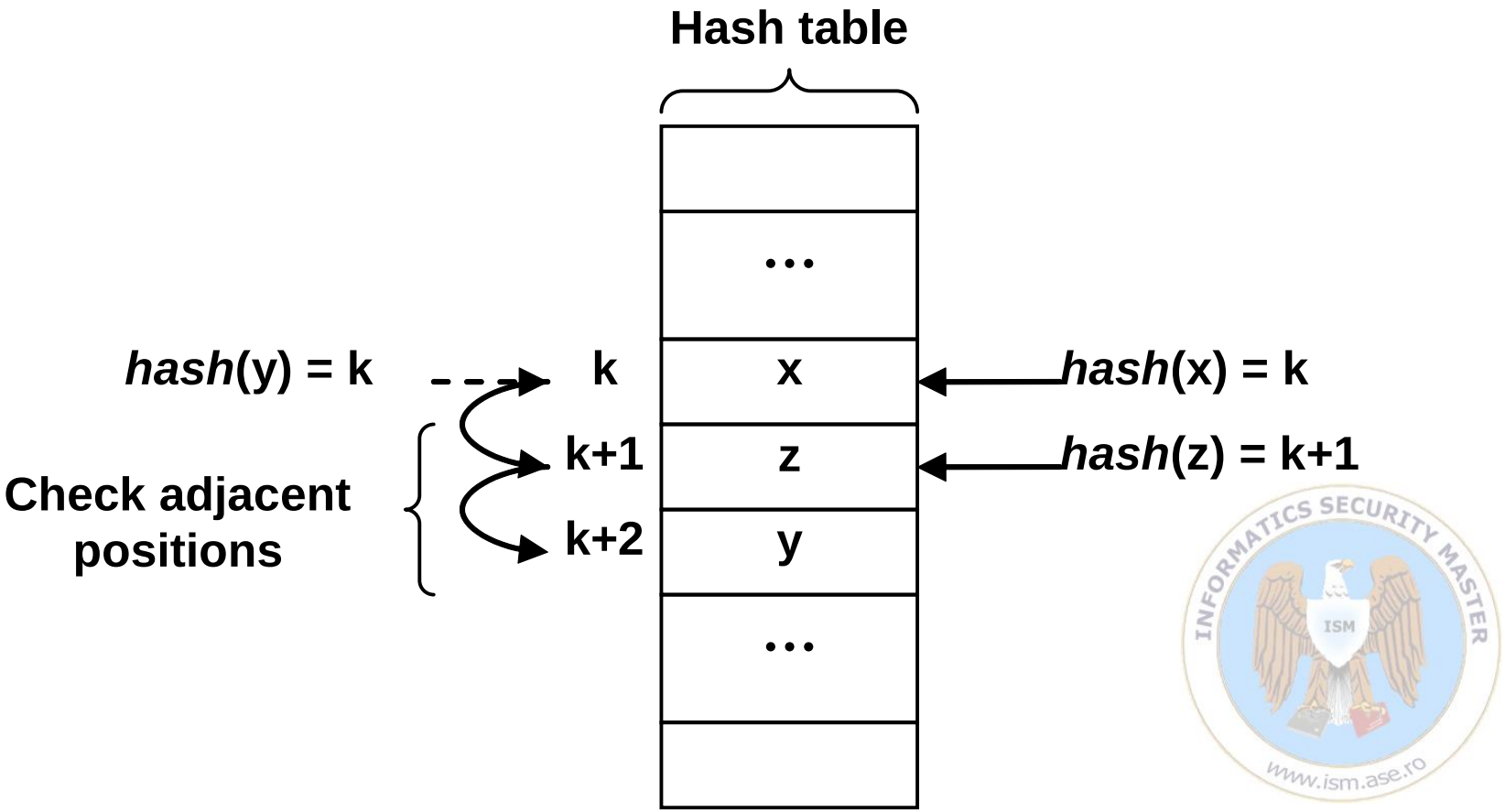
Linear probing:

- **Sequentially searching for the first free position where to insert the new element (at the right or left side of the collision);**
- **When searching: check the elements adjacent to the position indicated by the hash value;**
- **Grouping the collisions of the same type in the same cluster; the result is the increase of the probability of collision occurrence for adjacent hash values.**

Lexical analysis

Avoiding collisions:

Linear probing :



Lexical analysis

Avoiding collisions:

Quadratic probing:

- **Avoids creating collision groups by using a supplementary phase of finding the next free position different to 1; leaps in the hash table from two to two positions or from four to four;**

Lexical analysis

Avoiding collisions:

Quadratic probing (continued):

- **Determining the next position to be inserted:**

$$position = hash(X)+c*i^2$$

position: the new position on the table for insertion or search of a new element

X: the key associated to the element

hash(X): position indicated by the hash value of the element

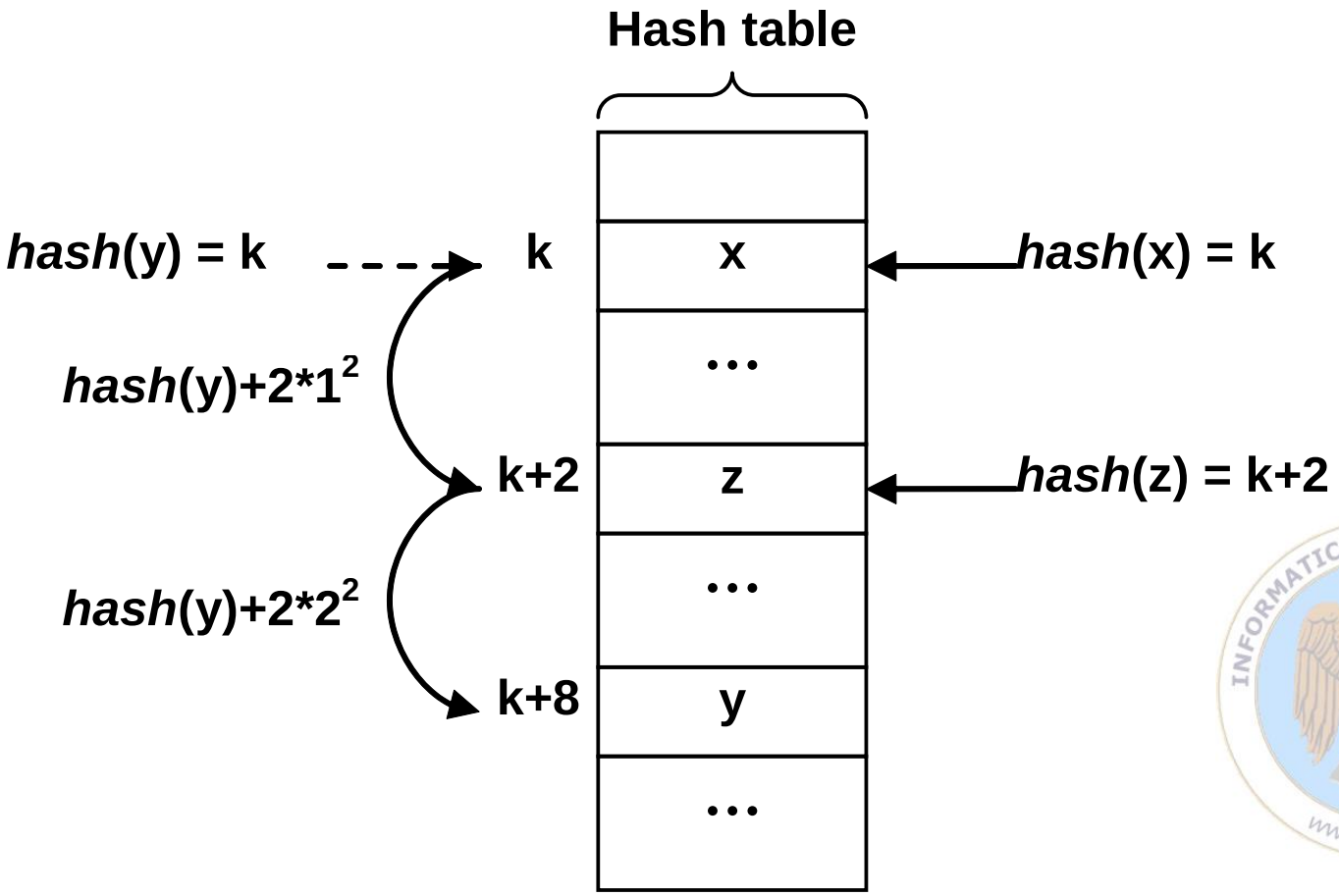
c: constant value {1, 2, 4}

i: the number of the re-hash operations or the number of verified positions

Lexical analysis

Avoiding collisions:

Quadratic probing:



Lexical analysis

Avoiding collisions:

Overflow area:

- **Splits the hash table into:**

  - Primary area: stores the initial elements;

  - Secondary area: allocated to the elements that generate collisions;

- **An element of the secondary area is used to store the new value or to continue the search;**

Lexical analysis

Avoiding collisions:

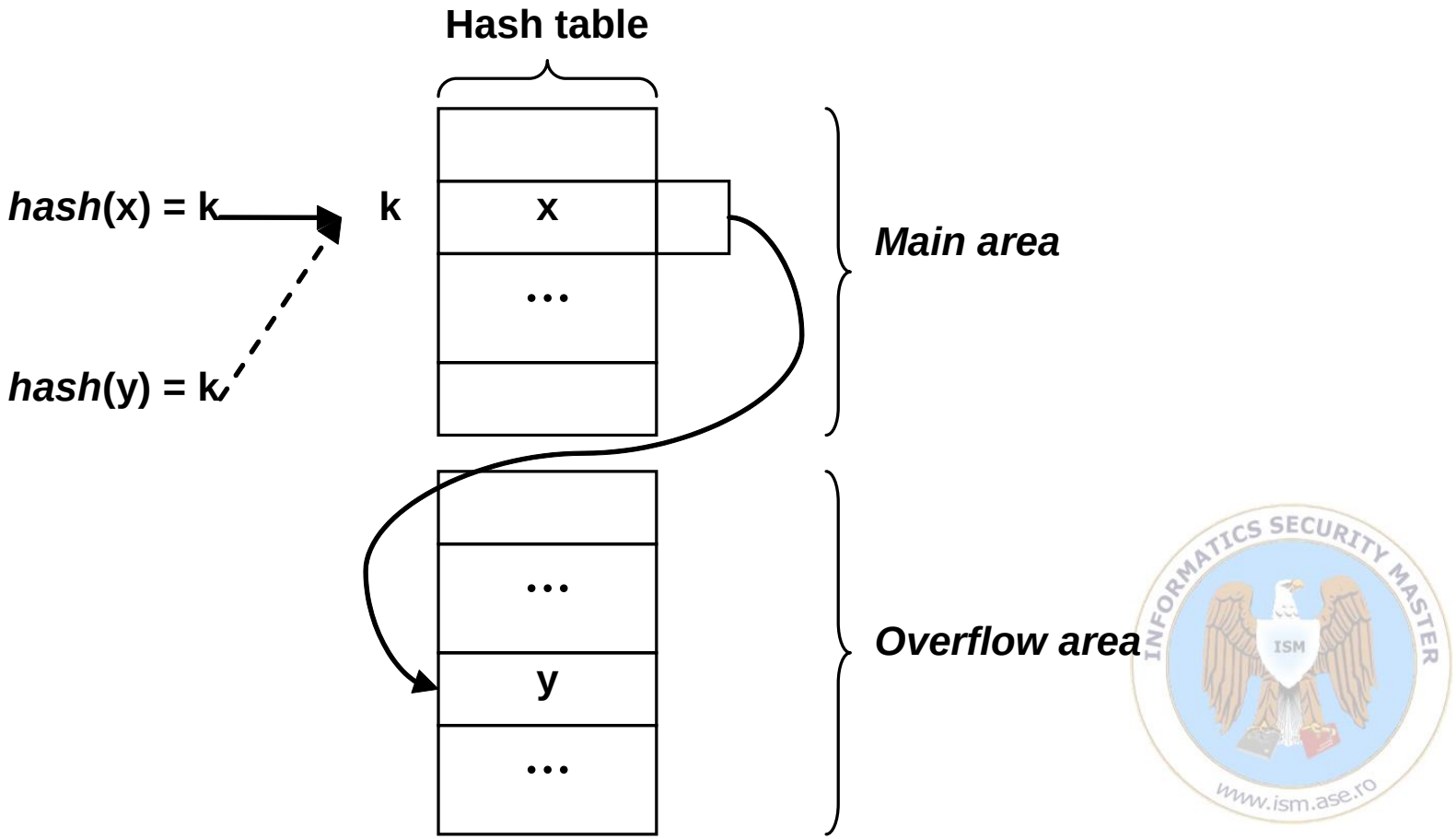
Overflow area (continued):

- **Access to the secondary area: by pointer from the primary area;**
- **Finding information is faster than in the chaining method.**

Lexical analysis

Avoiding collisions:

Overflow area:



Lexical analysis

Advantages of using the hash table:

- More efficient use of the memory resource: elements which are not used are not stored;
- Implementing alphanumerical keys: the type of the search key can be alphanumerical.

Lexical analysis

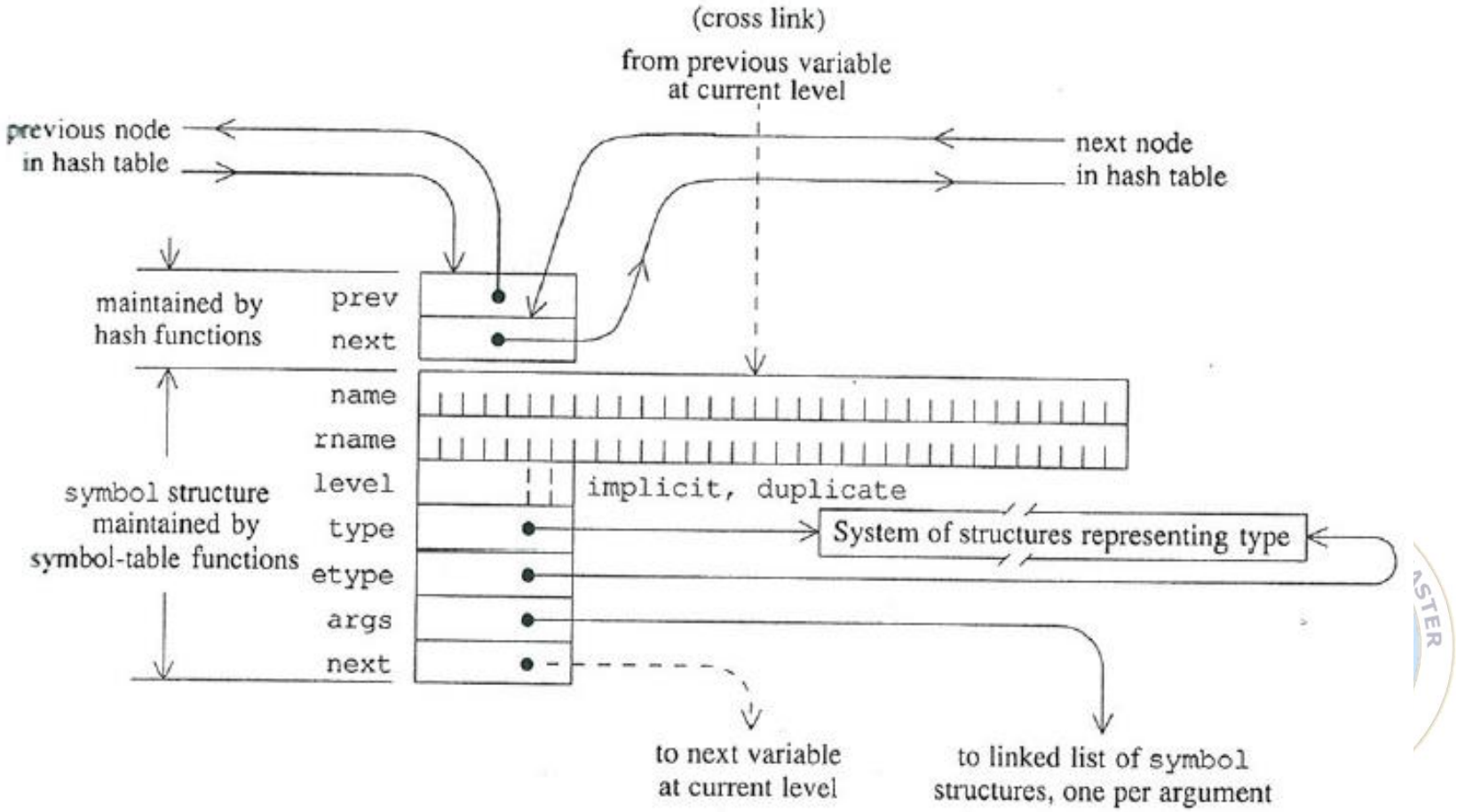
Disadvantages of using the hash table:

- **Supplementary processing given by the hash function which can sometimes have high complexity level;**
- **Collision occurrence in the table.**

Lexical analysis

Symbol table:

Possible structure of an symbol-table element (fields in the symbol structure):



Lexical analysis

Symbol table:

Possible structure of an element in a symbol table (fields in the symbol structure):

- **prev: the address of the previous node in the collision chain;**
- **next: the address of the next node in the collision chain;**
- **name: the name of the identifier as it appears in the input string;**

Lexical analysis

Symbol table:

- rname: the name of the identifier as it appears in the output string; example: global variables to which is attached the underscore symbol in front of the input identifier, arbitrary identifiers attributed by the compiler in the case of local static variables (name conflict) etc;
- level: the statement level of the symbol: 0 global symbols, 1 for the highest level block etc.; used for error checking: identify duplicate statements;

Lexical analysis

Symbol table:

- duplicate: bit that marks the symbols created by duplicate statements; the duplicate symbol is not stored in the table, but the field allows to identify the duplicate statements for later deletions;
- implicit: bit used for the implicit statement for identifiers not stated; use not stated variables in function calls (the bit is released); it becomes an unstated variable if the bit was not released and it leads to error occurrence;

Lexical analysis

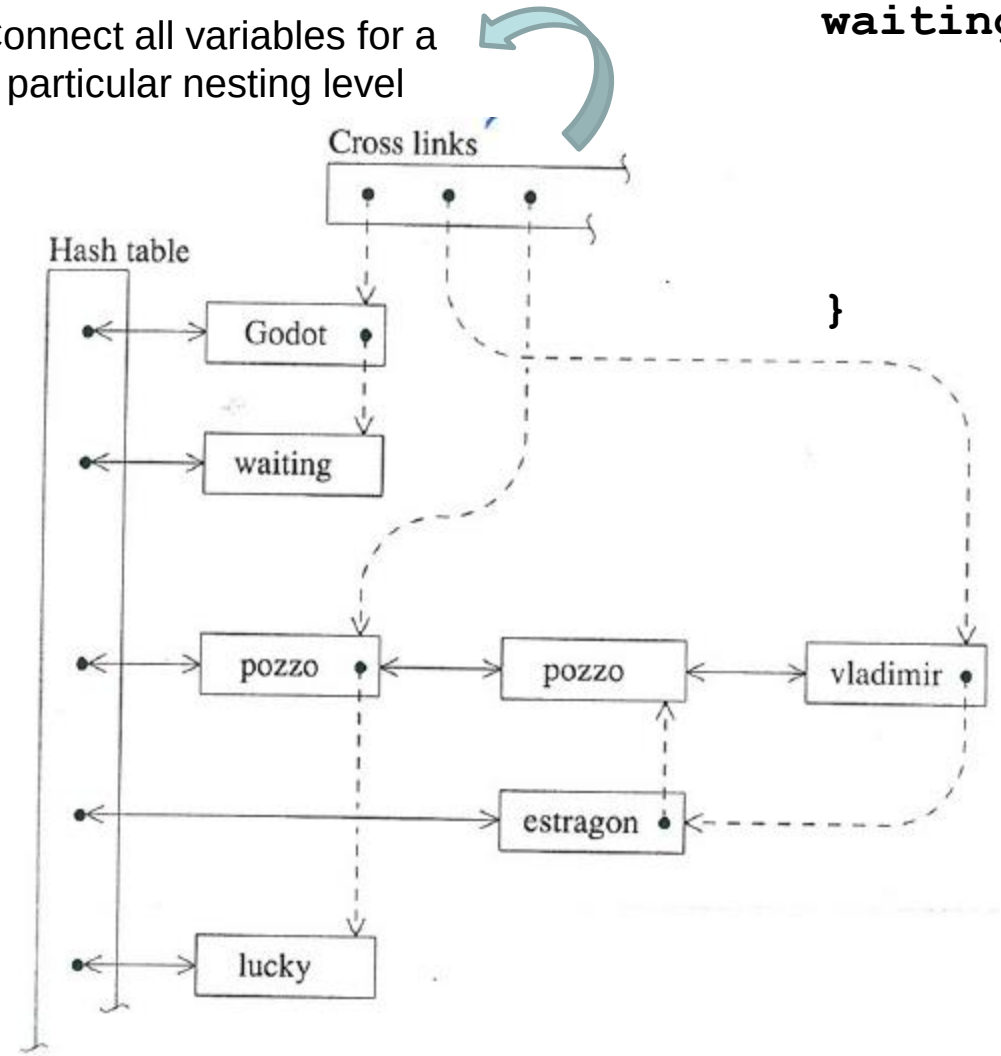
Symbol table:

- type, etype: pointers to a data structure that describes the object type;
- args: stores the list of function arguments until these are inserted in the symbol table; the beginning address of a chained list of symbol structures;
- next: pointer to the next variable (symbol component) on the same level.

Lexical analysis

Cross-Linked Symbol table – example:

Connect all variables for a particular nesting level




```
int Godot;
waiting (vladimir, estragon) {
        int pozzo;
        while(condition) {
                int pozzo, lucky;
        }

}
```

Lexical analysis

Symbol table:

Characteristics that the symbol table must have:

- **Speed: search time as short as possible; the table must be loaded in the internal memory and not accessed from files stored on disk;**
- **Maintenance: easy to maintain; the symbol table is the most complex data structure in the compiler; the support functions must be organized such that any compiler developer can use and update;**

Lexical analysis

Symbol table:

- **Flexibility: representing variables of arbitrary type, spread the table as symbols are identified;**
- **Accepting duplicate entries: stating variables with the same name, but on different inclusion levels; are different variables, active depending on the block where the program is running;**
- **Rapid deletion of elements: delete local variable group efficiently, without individually searching for them.**

Lexical analysis

Symbol table:

Organization of the symbol table – 2 levels:

- **Database level: table maintenance: entries insertion, search, deletion etc.;**

- **Maintenance level: table management, creating data structure systems for representing and inserting symbols, deleting and going through symbol classes (variables stated at the same level).**

Lexical analysis

Example of input system:

- **Projecting criteria:**

1. **Fast routines with reduced volume or without copying strings;**
2. **Characters for pushback or lookahead;**
3. **Lexemes (lexical tokens) of reasonable length;**
4. **Availability of the current, but also for the previous lexeme;**
5. **Efficient access to the data stored on the disk;**

Lexical analysis

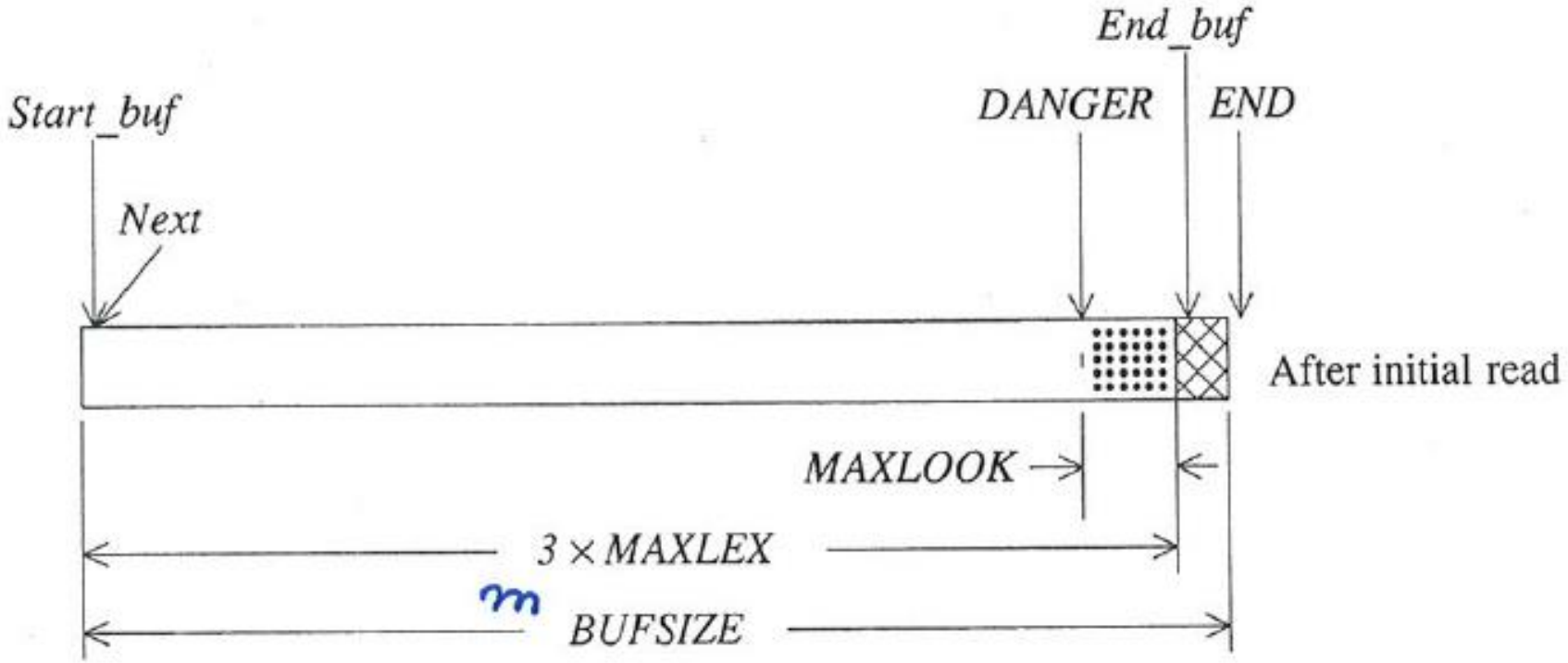
Example of input system:

- **Criterion 5: disk organized in sectors (512 bytes); reading from the disk: block of sectors; allocating unit: minimum number of bytes read from the disk;**

- **Criteria 1-4: ensured by pointers and a single input buffer;**

Lexical analysis

Example of input system: Initial reading



Lexical analysis

Example of input system:

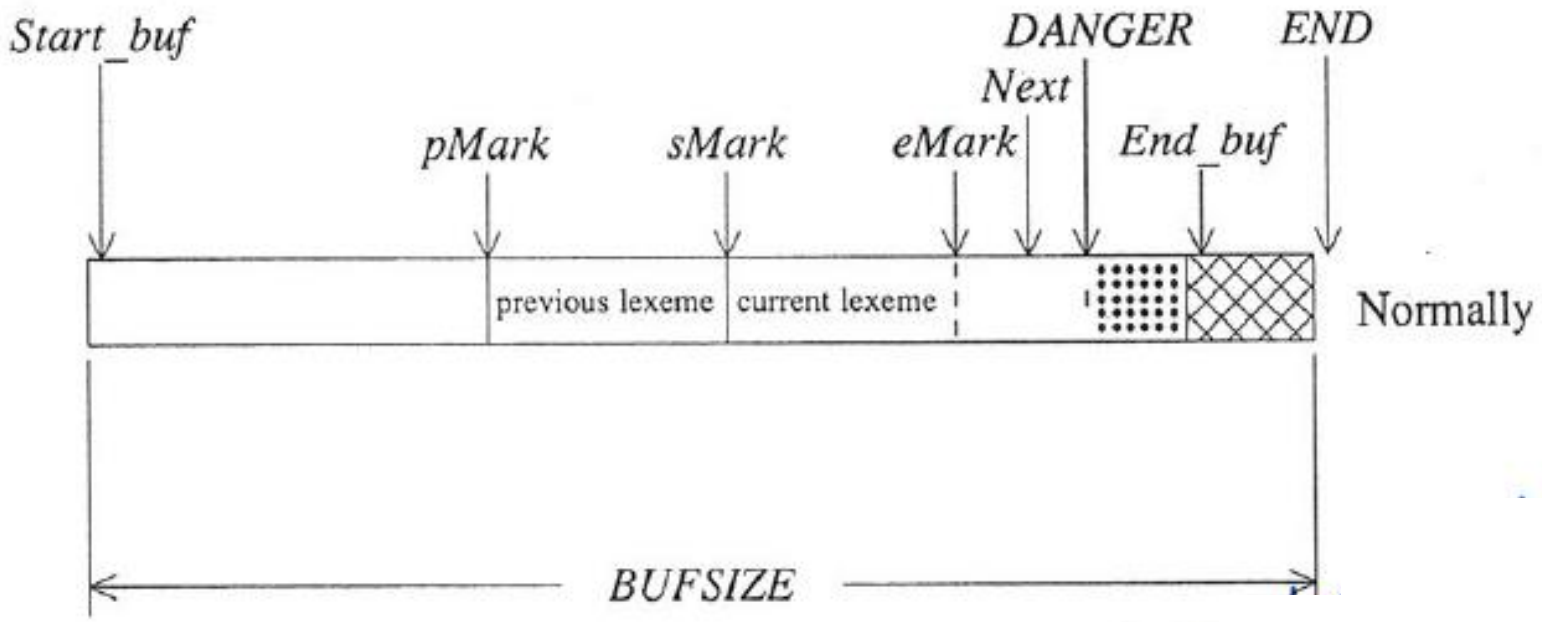
- **Possible states of the buffer:**

1. Initial reading: buffer state at the first loading:

- **BUFFSIZE: buffer size;**
- **MAXLEX: maximum size of the lexeme; disk readings: MAXLEX multiples;**
- **Start_buf: physical start of the buffer;**
- **END: physical end of the buffer;**
- **End_buf: logical end of the buffer; points to the last valid symbol in the buffer;**
- **Next: the address of the next input character;**

Lexical analysis

Example of input system: After token processing



Lexical analysis

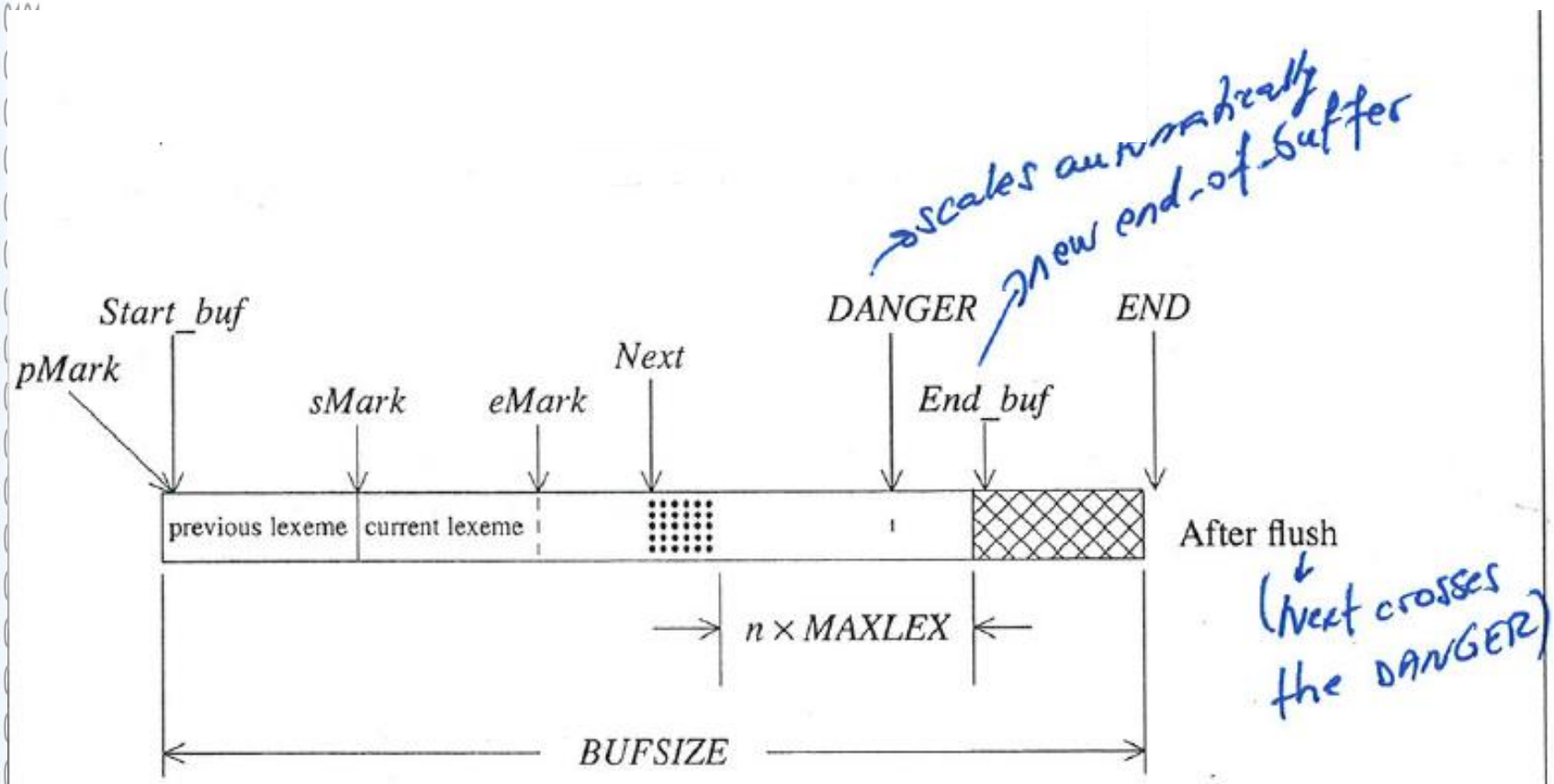
Example of input system:

2. After token processing:

- pMark: address of previous lexeme start;
- sMark: address of current lexeme start;
- eMark: address of current lexeme end;
- Next: > eMark; if a lexeme higher than the current one is identified, eMark is moved on the input symbol position; if Next goes far in the right, then Next is moved in eMark;
- MAXLOOK: constant for the number of lookahead characters admitted by the lexical analyzer;
- DANGER: marker for the beginning of the area close to the end of the buffer; signal when Next gets on DANGER; in the right side of Next there must be at least MAXLOOK symbols;

Lexical analysis

Example of input system: After token processing



Lexical analysis

Example of input system:

3. Buffer flush: Next gets over DANGER:

- Move the symbols between pMark and End_buf at the beginning of the buffer;
- Fill the buffer of size multiple-MAXLEX by reading from the disk;
- Adjust End_buf;
- Scale DANGER depending on End_buf;

Lexical analysis

Example of input system:

- Few copying operations (return sMark);
- sMark: address of the next lexeme;
- pushback: Next = eMark.

Syntactic analysis

Characteristics:

- **Structuring the sequence depending on certain rules;**
- **Input: sequence of lexical tokens;**
- **Output: correctitude of token sequence; build syntactic analysis tree;**

Syntactic analysis

Characteristics:

- **Based on grammars independent on the context: G=(N, ∑, P, S), where:**

- N: syntactic constructions of language;

- ∑: grammar alphabet (tokens);

- P: rules of forming syntactic constructions;

- S: start symbol;

Syntactic analysis

Characteristics:

- **Classification of syntactic analyzers:**

- The direction of building the tree: descendent, ascendent;

- Number of possibilities at one step of the construction: recursive (more), liniar (a single choice is possible);

Syntactic analysis

Characteristics:

- **Recursive descendent syntactic analyzer: simple from the algorithm point of view, restriction: grammar is not recursive on the left side; low efficiency and does not apply to large grammars;**

- **Recursive ascendant syntactic analyzer: has the same principles, different building direction for the tree;**

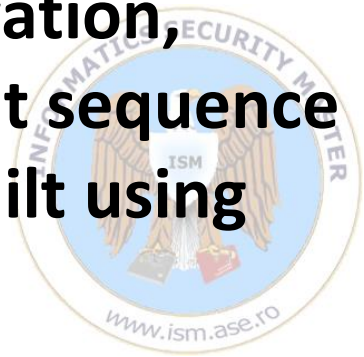
Syntactic analysis

Characteristics:

- **Linear algorithms: the highest performance and most often used:**

- LL(k) [Left-to-right scan, Leftmost derivation, Number of look-ahead symbols]: input sequence is read from left to right, the tree is built using derivations to the left;

- LR(k) [Left-to-right scan, Rightmost derivation, Number of look-ahead symbols] : input sequence is read from left to right, the tree is built using derivations to the right.



Syntactic analysis

Recursive-Descent syntactic analyzer:

- Grammar;
- Sequence;
- Syntactic analysis of the sequence;

Syntactic analysis

Recursive-Descent syntactic analyzer :

- The model of the descendant syntactic analyzer with returns:

- push-down auto – configuration: (s, i, α, β), where:
  - s: the state of the auto: q normal, r return, t termination, e error;
  - i: the position of the current symbol in the input sequence;
  - α: the work stack; the way in which the tree is obtained;
  - β: the input stack; the undeveloped part of the tree;

Syntactic analysis

Recursive-Descent syntactic analyzer:

- Initial configuration: (q, 1, ϕ, S);
- Final configuration: (t, n+1, α, ϕ);
- Transitions defined on the configuration set:
  - Expansion: the top of the input stack is not terminal; the expansion of the non terminal takes place;
  - Advance: the top of the input stack is the same with the i current symbol in the input sequence;

Syntactic analysis

Recursive-Descent syntactic analyzer :

- Temporary failure: the terminal symbol at the peak of the input stack and the current symbol in the input stack are different; the passing to a returning state is marked;
- Return: remaking the right side of a production which was not well selected;
- Another trial: exiting from the return state: trying another production, staying in the return state for rebuilding the right side of the production, error in the input sequence;
- Succes: acceptance of the sequence by the grammar.

Syntactic analysis

Recursive-Descent syntactic analyzer - example

Grammar:

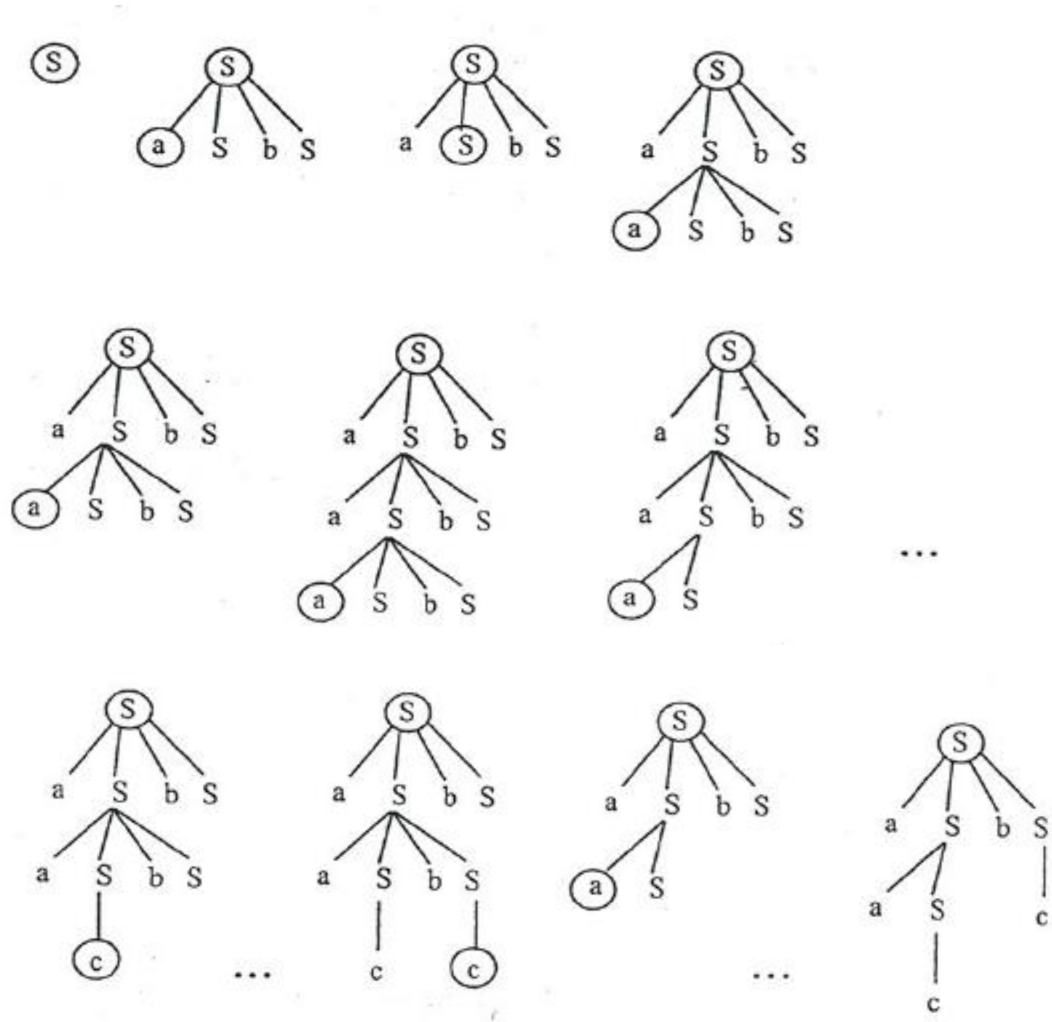
G = ({S}, {a, b, c}, {S -> aSbS|aS|c}, S)

Input string

ω = aacbc

Syntactic analysis

Recursive-Descent syntactic analyzer – example



Syntactic analysis

Syntactic analyzer LL(1):

- Length prediction 1;
- Predictions memorized in the analysis table LL(1);
- Build the analysis table:

- Values of set FIRST;
- Values of set FOLLOW;
- Includes by rows terminal symbols, non-terminal symbols and a special symbol $;

Syntactic analysis

Syntactic analyzer LL(1):

- Includes by columns terminal columns and the special symbol $;
- Completion rules;
- Example of analysis table;

Syntactic analysis

Syntactic analyzer LL(1):

- **Syntactic analyzer functioning:**

- Configuration: the input stack α, the work stack β, the exit lane π;

- Initial configuration: (ω$, S$, ε), where the entry ω$ contains the sequence to analyze finished with $, the work stack contains the stack end marker $ and S the start symbol, the exit lane is null ε;

Syntactic analysis

Syntactic analyzer LL(1):

- Final configuration: ($, $, π), where input stack and work stack are null, the exit lane contains string of predictions; acceptance criterion: null stack;
- Transitions:
  - push: adding to the stack;
  - pop: taking out from the stack (are taken out the both the stack peaks if they are identical);
  - acc: acceptance of the final configuration;
  - err: error cases;

Syntactic analysis

Syntactic analyzer LL(1):

- **Syntactic analysis LL(1) is applied for LL(1) grammar;**

- **Grammar LL(1): if A -> B | C, then FIRST(B) ∩ FIRST(C) = ϕ**

Syntactic analysis

Syntactic analyzer LL(1) – example:

- **Grammar**

G = ({S, A, B, C, D}, {+, *, (,), a}, P, S)

- **Production rules:**

1. S -> BA

2. A -> +BA

3. A -> ε

4. B -> DC

5. C -> *DC

6. C -> ε

7. D -> (S)

8. D -> a

Syntactic analysis

Syntactic analyzer LL(1) – example:

- **FIRST and FOLLOW sets**

Non-Terminal	FIRST	FOLLOW
S	{(, a}	{), ε}
A	{+, ε}	{), ε}
B	{(, a}	{+,), ε}
C	{*, ε}	{+,), ε}
D	{(, a}	{+, *,), ε}

Syntactic analysis

Syntactic analyzer LL(1) – example:

- **Building rules of analysis table:**

1. **M(A, a) = (α, i) , ∀ a ∈ FIRST(α), a ≠ ε, A -> α rule no. i**

M(A, b) = (α, i) , if ε ∈ FIRST(α), ∀ b ∈ FOLLOW(A), A -> α rule no. i

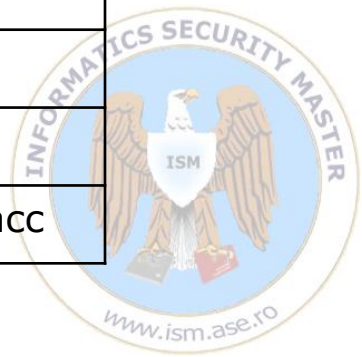
2. **M(a, a) = pop**
3. **M($, $) = acc**
4. **M(x, a) = err, otherwise**

Syntactic analysis

Syntactic analyzer LL(1) – example:

- **Analysis table**

	+	*	(	)	a	$
S			BA, 1		BA, 1	
A	+BA, 2			ε, 3		ε, 3
B			DC, 4		DC, 4	
C	ε, 6	*DC, 5		ε, 6		ε, 6
D			(S), 7		a, 8	
+	pop					
*****		pop				
(			pop			
)				pop		
a					pop	
$						acc



Syntactic analysis

Syntactic analyzer LL(1) – example:

- Grammar

G = ({S, E, A}, {a, b, (,), +, *}, P, S)

- Production rules:

1. S -> E
2. E -> a
3. E -> (E)
4. E -> bA
5. A -> +E
6. A -> *E

- Validate the input: (b + a) * a

Syntactic analysis

Syntactic analyzer LR(k):

- Ascendant syntactic analysis (leaves -> root);
- Strategy: reducing the input sequence to the start symbol;
- Efficient algorithm: lack of several ched between the intermediate form and the grammar productions;
- Based on the *moving-reducing* method;

Syntactic analysis

Syntactic analyzer LR(k):

- **Working principles:**

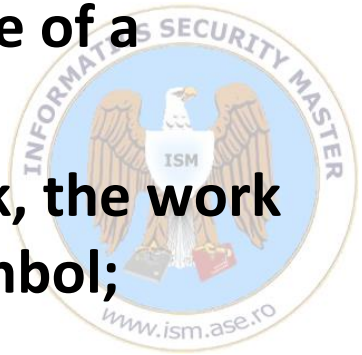
  - Push-down autoon;

  - The symbols in the input stack are introduced in the work stack - moving;

  - At the top of the work stack is obtained a form identical to the right side of a production;

  - Replacing in the work stack of the from identical to the right side of a production with the left side of a production – reducing;

  - Acceptance of the sequence: null input stack, the work stack contains only the grammar start symbol;



Syntactic analysis

Syntactic analyzer LR(k):

- **Characteristics of the method:**

- Auto configuration: triplet (α, β, π) – the work stack, input stack, exit lane with the string of used productions;

- Possible transitions: moving, reducing, accepting, error;

Syntactic analysis

Syntactic analyzer LR(0):

- Finishing function;
- Set of states;
- Determining the set of states;
- Building the LR(0) table:

- A line for each state;
- A column for each action; a state is associated to an action;
- The goto area with a column for each grammar symbol;

Syntactic analysis

Syntactic analyzer LR(0):

- Rules of filling in the LR(0) table;
- Syntactic analyzer way of working:

- Initial configuration: ($s0, ω$, ε);
- Final configuration: ($sacc, $, π);

Syntactic analysis

Syntactic analyzer LR(0):

- **Transitions:**
  - **Moving: if s_m is associated to the moving action and a_i is the stack peak, then a_i is moved to the work stack, and the new state in the stack peak is $s_j = goto(s_m, a_i)$;**
  - **Reducing: if s_m is associated to the reducing action, then the right side of the production is searched for in the work stack, it is replaced with the left side and is computed the state that was reached $s_j = goto(s_{m-p}, A)$; the exit lane is updated with the production used;**
  - **Acceptance: ($ s_i, $, π) = acc if action (i)=acc;**
  - **Error in the other cases.**

Syntactic analysis

Syntactic analyzer LR(0) – example:

- **Grammar**

G = ({S', S, A}, {a, b, c}, P, S')

- **Production rules:**

1. S' -> S

2. S -> aA

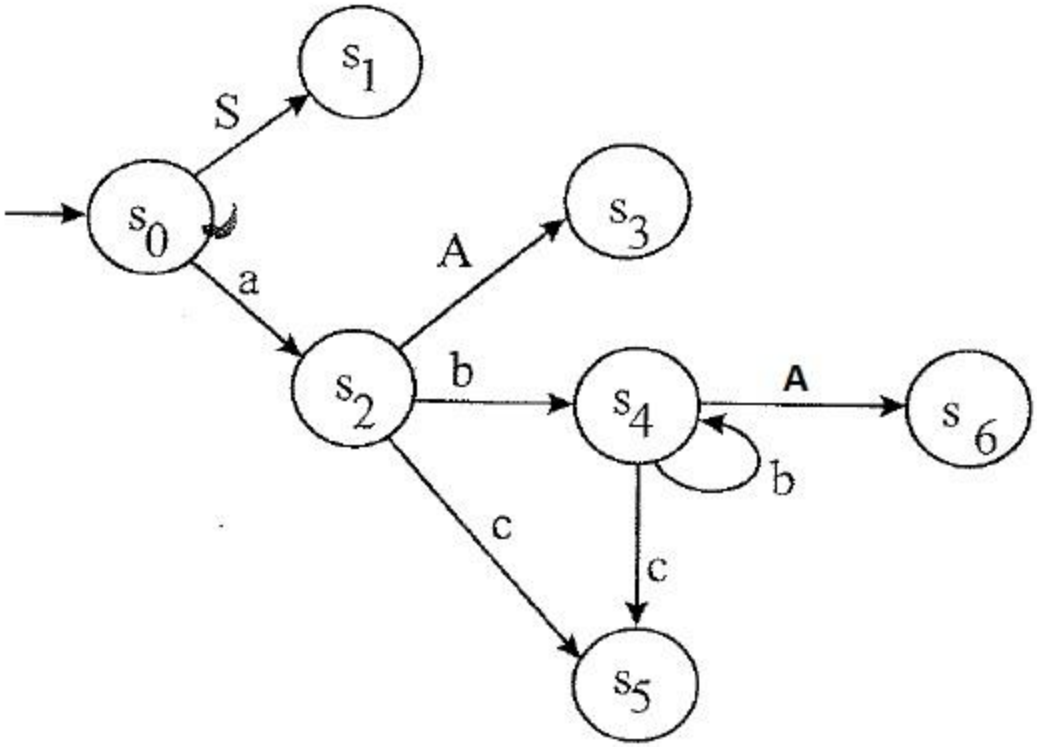
3. A -> bA

4. A -> c

Syntactic analysis

Syntactic analyzer LR(0) – example:

- **Finite automaton after transitions established by goto function:**



Syntactic analysis

Syntactic analyzer LR(0) – example:

- **Building rules of analysis table:**

1. If $[A \rightarrow \alpha.a\beta] \in s_i$, then action(i) = M;
2. If $[A \rightarrow \beta.] \in s_i$ and $A \neq S'$, then action(i) = Rn, where n is the number of production rule $A \rightarrow \beta$
3. If $[S' \rightarrow S.] \in s_i$, then action(i) = ACC
4. If $goto(s_i, X) = s_j$ then goto(i, X) = j
5. Initial state corresponds to set containing [S' -> .S]
6. Otherwise, error

Syntactic analysis

Syntactic analyzer LR(0) – example:

- **Analysis table:**

State	action	goto				
		S	A	a	b	c
0	M	1		2		
1	ACC					
2	M		3		4	5
3	R2					
4	M		6		4	5
5	R4					
6	R3					

Semantic analysis

Characteristics:

- **Operations having as result an Intermediate form represented as an attributed tree or resulted program (translator);**
- **Operations:**

- Verifications of the proprieties of the program entities;

- Verification of the type;

- Identification of the operations;

- Other transforions;

Semantic analysis

Characteristics:

- **Tools used – operations applied to the nodes in the abstract syntax tree:**

- Grammar of attributes;

- Manual methods.

Semantic analysis

Abstract syntax tree:

- Syntactic analysis tree: structure of representation of the way of obtaining program entities on the basis of syntactic rules;

- Abstract syntax tree: more efficient structure resulted by eliminating the details and uninteresting branches in the syntactic analysis tree;

Semantic analysis

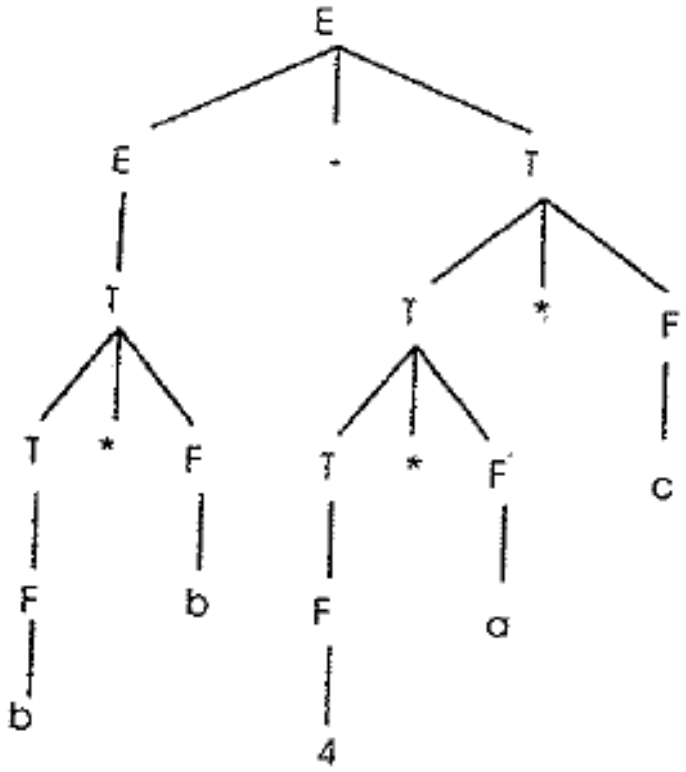
Abstract syntax tree:

- Difference in the trees: practical degree;
- Motivation of the simplification:

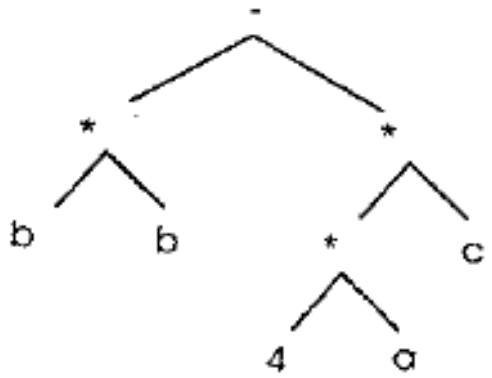
- Syntactic constructions used only as information (variable statements, constants);
- Syntactic constructions with role of transmitting value (identifier, expression, factor, term).

Semantic analysis

Abstract syntax tree – example:



Syntactic analysis tree



Abstract syntax tree

Semantic analysis

Identification:

- **Verification of type: conditions dependent on the context (correct usage of types, expressions are used correctly in terms of types):**
- **Using an identifier has its corresponding statement;**
- **Identifier occurrence context is allowed;**

Semantic analysis

Identification:

- **Determining the variable type: stating the identifier in the program;**
- **Occurrence of the identifier:**

- Definition: stating the identifier;

- Usage: other occurrences beside stating the identifier;

- **Identification in the syntactic analysis phase: partially or completely depending on the statement manner of the identifier (mandatory before usage or not);**

Semantic analysis

Identification:

- **Operations of the identification process:**

- Definition of identifiers: completion in the symbol table (type from the statement, initial value, implicit or imposed by the statement);

- Usage of identifiers: checks if the symbol table contains the needed information (type, value) corresponding to a possible update of the information in the symbol table;

Semantic analysis

Identification:

- **Categories of types:**

- Primitive: character, Boolean, integer, real;

- Structured: massive, article, reference, function etc.

Semantic analysis

Type verification:

- The compiler attributes types and verifies them;
- Type attribution: determine the construction type (variable, expression, function);
- Type verification: correct usage of type attribution;
- Type verification components:

- Static verification: during compilation; it ensures that there are no errors in the object program
- Dynamic verification: during execution; increases execution duration, decrease the reliability of the implemented code;

Semantic analysis

Type verification:

- Type equivalence:

- Structural equivalence: 2 types are equivalent if and only if they are of the same primitive type or they were built on the basis of the same constructor for structural equivalent types;

- Name equivalence: 2 types are equivalent if and only if they have identical names;

Semantic analysis

Type verification:

- Type conversions: different implementations of operators applied to some operands of different types; implicit if a compiler inserts the conversion autoically;
- Conversion principle: without information loss when converting from a type to another (usually a type with high capacity to one with low capacity);
- Explicit conversion: implemented by the programmer through a call of a function defined between types.

Semantic analysis

Attribute grammars:

- Grammars independent on the context expanded with two characteristics:

- Attribute association for each terminal or non-terminal symbol;

- Evaluation rule association for each production rule;

Semantic analysis

Attribute grammars:

- **Semantic analysis: assimilated to an attribute evaluator:**

- Applies a traversal strategy of the abstract syntax tree;

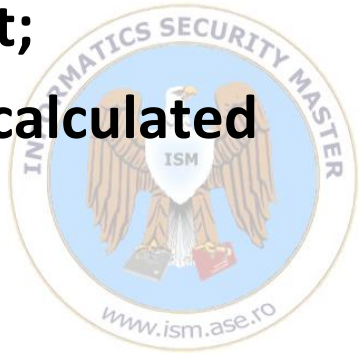
- Executes calculus calls;

- Propagates the value calculated throughout the tree.

Semantic analysis

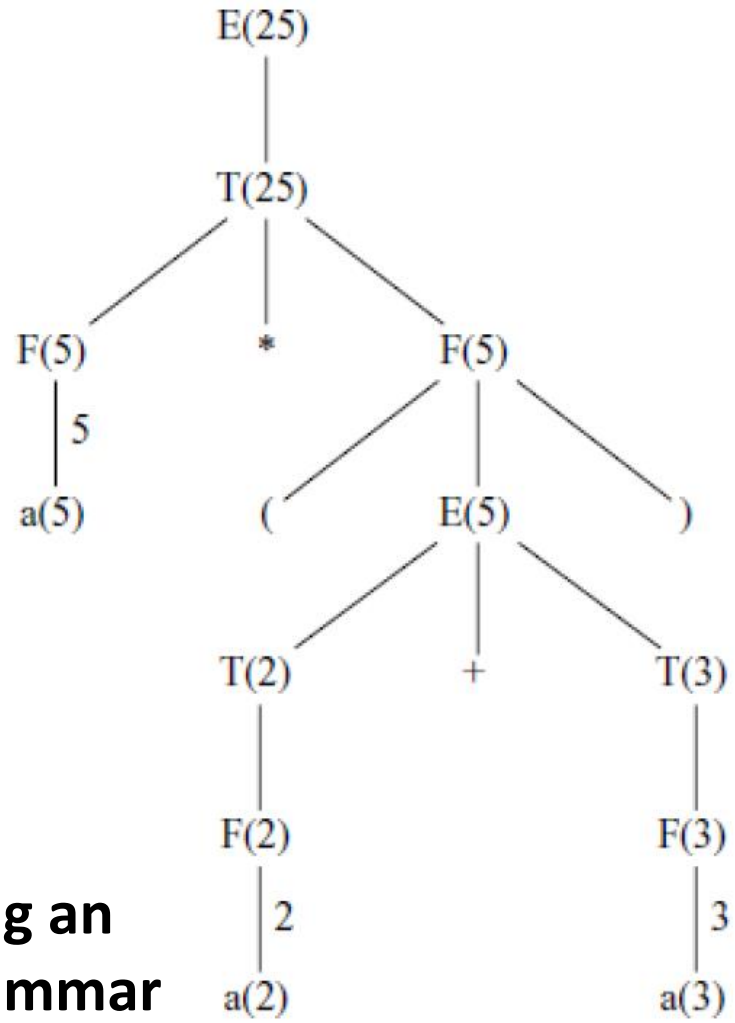
Attribute grammars:

- **Principles of building an attribute evaluator:**
  - **Allocate spate for the attributes associated to each node;**
  - **Application for each tree built based on attribute grammar;**
  - **Parse the tree accompanied by determining attributes;**
  - **Respect the order of calculus execution: an attribute receives value a single time and the dependencies between attributes are taken into account;**
  - **Detect the case when attributes can not be calculated (avoid the infinite calculus cycles).**



Semantic analysis

Attribute grammars - example:



Attribute evaluator using an attribute expression grammar

Semantic analysis

Manual methods:

- **Disadvantages of attribute grammar:**

- Space allocated to the attributes;

- Slowing down the compiling process;

- Implementation of attribute evaluators is dependent on the way of defining the attributes;

Semantic analysis

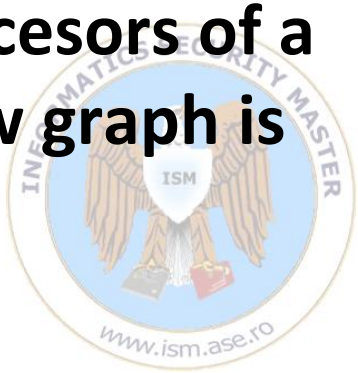
Manual methods:

- Defining the attributes and evaluation rules performed manually led to searching for manual methods:

- Symbolical interpretation;

- Data flow equations;

- Needs information on the control transfer throughout the tree: the succesor/succesors of a node in a control flow; the control flow graph is necessary;



Semantic analysis

Manual methods:

- Control flow graph:

- Nodes: basic blocks of the program;
- Edges: way of transferring the control from a block to another;
- Extracting information, when compiling, on what is going to happen during execution;
- Built by fragmenting the abstract syntax tree through a fragmentation routine for each node;

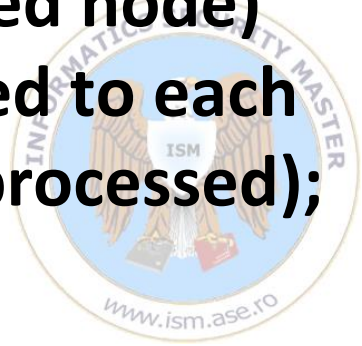
Semantic analysis

Manual methods:

- Control flow graph (continued):

- Fragmentation routine: receives a pointer to the processed node, determines production, calls the fragmentation routines of the descendants;

- All the routines update the global variable ultim_nod (pointer to the last processed node) through the successor variable attached to each node (pointer to the next node to be processed);



Semantic analysis

Manual methods:

- **Symbolical interpretation:**

- Program execution: control flow crosses a possible path in the graph;

- In each node code is executed on the basis of the input variable values and determines the values of the output variables;

Semantic analysis

Manual methods:

- **Symbolical interpretation (continued):**

- Also called stack simulation: simulating the execution during compilation, by attaching a stack to every edge in the control flow graph; an input for each identifier (variables, constants) visible at that point, with all the information available at the moment of compiling;

- Applied also when optimizing the code for: detecting the usage of uninitialized variables, propagating constants, using variables, verifying the types etc.

Semantic analysis

Manual methods:

- **Data flow equations:**

- In the program there is also a data flow, besides control flow;

- Describes the way of propagating the values of the variables and constants;

- Method used for detecting the state of the variables: declared/undeclared, initialized/uninitialized, modified/unmodified value;

Semantic analysis

Manual methods:

- **Data flow equations (continued):**

- Methods of data flow analysis:

- **Forward: propagate values in the direction of the control flow; the general form of the flow equations: *out[I] = gen[I] U (in[I] – kill[I])* -> information at the finish of an instruction I is either generated in I, or transmitted at the beginning without being destroyed while the flow crosses the instruction;**

Semantic analysis

Manual methods:

- **Data flow equations (continued):**

- Methods of data flow analysis:

- **Backwards: propagate values in the opposite sense of the control flow; the general form of the flow equations: *in[I] = gen[I] U (out[I] – kill[I])* -> information at the beginning of an instruction I is either generated inside the instruction, or transmitted at finish without being destroyed while the flow crosses the instruction.**

Generating intermediate code

Characteristics:

- Multiple crossing over the source program: less space for code, shorter execution time;
- Intermediate code: interface between the analysis part and the synthesis part;
- Intermediate code requirements: highlighting the functionality, adequate form for optimization and object code generation;

Generating intermediate code

Characteristics:

- **Possibility of re-using the Intermediate code for compiling families;**

- **Main qualities that the Intermediate code must fulfill:**

- Easiness to copy the abstract syntax tree;

- Easiness to translate into machine code;

Generating intermediate code

Characteristics:

- **Forms of representations for the Intermediate language:**

- Attributed tree;

- Post-fixed form;

- Code with three addresses.

Generating intermediate code

Attributed tree:

- The most used internal representation;
- The phase of semantic analysis overlapped with the generation of Intermediate code;
- Built in the phase of syntactic analysis;

Generating intermediate code

Attributed tree:

- **Structure easy to cross;**
- **Every construction in the source program is represented by a node that contains additional attributes and information used in the synthesis part.**

Generating intermediate code

Post-fixed form (reverse Polish notation):

- Three types of notations for an expression:

- Infix: usual form of writing in which the operator is framed by operands: a+b*c;

- Prefixed: the operator precedes the operands: +a*bc;

- Postfixed: the operands precede the operator: abc*+;

Generating intermediate code

Post-fixed form (reverse Polish notation):

- Using the stack model:

- The operands are put in the stack;
- Operations performed in the stack peak: the operands are taken out of the stack and the result is put in the stack;

Generating intermediate code

Post-fixed form (reverse Polish notation):

- **Important properties:**

- Does not contain parenthesis;

- The operators appear in the order of execution;

- **Using equivalent binary operators for the unary operators: 0-b (-b) or special notation @b.**

Generating intermediate code

Code with three addresses:

- Sequence of instructions with general form: <rez> = <arg1> <op> <arg2>, where:

- <rez>, <arg1>, <arg2> are variables, constants in the program or temporary created by the compiler;

- <op> binary operator;

- Each instruction contains addresses: two for operands, one for result;

Generating intermediate code

Code with three addresses:

- During implementation are used pointers to the addresses in the symbol table for variables and constants in the program; for temporary symbols is created space in symbol tables and are used pointers for reference;

- Advantage: simplicity of instructions, contains one operator that allows the generation of the object code almost directly;

Generating intermediate code

Code with three addresses:

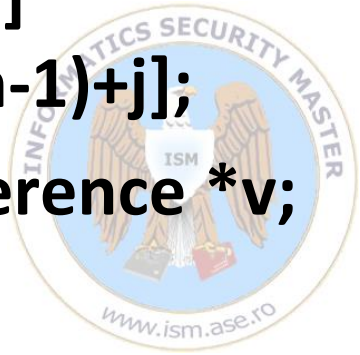
- The code is a linear representation of the abstract syntax tree;
- Abstract syntax tree and code with three addresses;
- Transforion of the code with three addresses in particular forms of instructions:

- Using unary operators: expressions treated as if the first operator is missing: <rez> = <op> <arg2>;
- Attribution instruction: lack of operator and operand two;
- Unconditioned leap: goto L, where L represents the label of an instruction with three addresses;

Generating intermediate code

Code with three addresses:

- Conditioned leap: if c goto L – unconditioned leap to the instruction with label L (if c=true) or the very next instruction is executed (if c=false);
- Procedure call: parameters are evaluated and procedure call with n parameters (call p,n);
- Using indexed variables: arrays with several sizes are transformed into linear ones; a[i][j] transformed into the expression a[i*(n-1)+j];
- Addressing the variable: address &v, reference *v;



Generating intermediate code

Code with three addresses:

- Representation of the code with three addresses:

- Record panel, each record represents an instruction;

- Record forms: quadruples, triplets, indirect triplets;

Generating intermediate code

Code with three addresses:

- **Quadruple:**

- Recording structure with four fields:

<op> <rez> <arg1> <arg2>;

Generating intermediate code

Code with three addresses:

- **Triplets:**

- Similar to the quadruples: <op> <arg1> <arg2>;

- The result is not presented;

- Creating temps and loading the symbol table are avoided;

- Disadvantage: instructions can suffer modifications in the optimization phase; modifications must be performed also on the instruction references;

Generating intermediate code

Code with three addresses:

- **Indirect triplets:**

- Solves the disadvantage of the triplets during optimization;

- Supplementary lists that contain order numbers for the triplets regarding their execution.

Generating intermediate code

Transforming the syntactic tree into a code with three addresses:

- **Building the code: cross the attributed tree;**

- **Starting from the production rule, semantic actions are associated:**

- Computing attributes used in generating code for the entire sub-tree;

- Generating instructions in code with three addresses.

Generating intermediate code

Example – the source code:

```
int a, b, c, d;
a = 3;
b = 5;
c = 7;
d = a * (b + c);
```

Generating intermediate code

Example – reverse Polish notation:

3 a = 5 b = 7 c = a b c + * d =

Generating intermediate code

Example – The 4-tuple output:

1: LD	3	T1	– –
2: =	T1	a	– –
3: LD	5	T2	– –
4: =	T2	b	– –
5: LD	7	T3	– –
6: =	T3	c	– –
7: LD	a	T4	– –
8: LD	b	T5	– –
9: LD	c	T6	– –
10: ADD	T5	T6	T7
11: MUL	T4	T7	T8
12: =	T8	d	– –

Intermediate code optimization

Characteristics:

- Transformations applied with the view of obtaining a more efficient program: smaller occupied space, shorter execution time;
- Analysis of the allocated resources for optimization vs. improvement of the code generated;
- Components:

- Control flow analysis;
- Data flow analysis;
- Transformations.

Intermediate code optimization

Program flow graph:

- nodes: basic blocks represented as sequences of instructions in code with three addresses;
- Edges: way of transferring execution control from a block to another;
- Basic block: sequence of consecutive instructions, without leap or branching, when the flow enters at the first instructions and finishes after the last instruction;

Intermediate code optimization

Program flow graph:

- **Rules of dividing a program into basic blocks:**

  - Determine the first instruction in the block;

  - **First instruction in the program;**
  - **The instruction resulted after a conditioned or unconditioned leap;**
  - **The instruction that is next to a conditioned or unconditioned leap;**

  - Building the basic block: the first instruction and all the instruction until the next “first instruction”;

Intermediate code optimization

Program flow graph:

- Building edges: there is an edge between block i and block j if block j is the next after block i, according to the situations:

- There is a leap from the last instruction in block i to the first instruction in block j;

- Block j follows immediately after block i, and the last instruction in block i is not an unconditioned leap;

Intermediate code optimization

Program flow graph:

- **Example:**

(1) i = 0

(2) s = 0

(3) t1 = s + i

(4) s = t1

(5) t2 = i + 1

(6) i = t2

(7) if i < 10 goto(3)

Intermediate code optimization

Program flow graph:

- Transformations: local (inside the block), global (the other ones);
- Optimization techniques:

- Computations during compiling;
- Elimination of redundant computations;
- Elimination of inaccesible code;
- Optimization of cycles.

Intermediate code optimization

A. Computations during compiling:

- Instruction to attribute operands – constant values; the value of the variable is calculated during the moment of compiling: a = 3 + 2; a = 3 * pi;

- Implicit conversions: a = a + 1, a real number; intermediate computation for conversion of the integer number 1 to the real number 1.

Intermediate code optimization

B. Eliminating redundant computations:

- Based on repeated use of certain computations;
- Algorithm for eliminating redundancies:

- Scanning triplets in the order of the performed operations;

- Using the dependency relationship: dep(V) represents the number of dependencies for every variable V in the program;

Intermediate code optimization

B. Eliminating redundant computations:

- Using the dependency relation:

- dep(V)=0, initial value; does not depend on any triplet;

- dep(V)=i, if triplet i attributes value to variableV;

- For triplet i: dep(i) = 1 + (maxdep(V)), where V is operand in triplet i;

- If triplet i is identical to triplet j, j<i, then i is redundant if and only if dep(i)=dep(j).

Intermediate code optimization

C. Eliminating inaccesible code:

- Inaccesible code: sequence that is not executed on any path in the control graph;
- There is no code inaccessible on purpose; it may result from evaluating conditions in report with constant values.

Intermediate code optimization

D. Cycle optimization:

- Detection of the cycles: easy in high level languages; difficult in low level languages (flow graph analysis);

- Code optimization techniques:

- Create factors for cycle invariants;

- Reducing the power of the operations;

Intermediate code optimization

D. Cycle optimization:

1. Create factors for cycle invariants:

- **Characteristics of cycle invariants: every operand is: constant or has its definition outside the cycle or has its definition as a cycle invariant;**
- **Performing instruction at every iteration;**
- **Solution: taking out the instruction before the cycle, executing it a single time**

Intermediate code optimization

D. Cycle optimization:

1. Create factors for cycle invariants:

- **Example:**

```
for(i=0; i<n; i++){
        x = y + z;
        a[i] = i * x;
}
```

Intermediate code optimization

D. Cycle optimization:

2. Reducing the power of the operations:

- **Replacing expensive operations (multiplication) with simpler operations (addition) in the cycling variables (indexing) and the variables calculated in cycles (value determined on the basis of the value in the previous iteration)**

Generating the object code

Characteristics:

- Translating the intermediate code instruction into object code instructions;

- Characteristics of the machine code:

- Finite number of registries;
- Variable stored in memory;
- Use the execution stack and instructions to access it;
- Instructions corresponding to the object language;

Generating the object code

Characteristics:

- **Issues related to generating object code from intermediate code:**

  - Registry allocation: how variables are store and handled;

  - Selecting instructions: how and mapping order of intermediate code instructions into machine instructions;

- **Generating object code: depends on the type of the computer – techniques:**

  - Computer with accumulator;

  - Computer with general registries;

Generating the object code

Generating object code for accumulator computer:

- Stack machine model: use the value stack for intermediate results;

- Stack for storage and handling values and two types of instructions:

- Move/copy values in/from the peak of the stack into memory;

- Operations on elements at the peak of the stack: remove operands from the stack, executing operation, store result in the stack;

Generating the object code

Generating object code for accumulator computer:

- **Registries are not used;**
- **Handling stack with reference at the stack peak and reference to the beginning of the area allocated to the stack (the basis of the stack).**

Generating the object code

Generating object code for registry computer:

- The model of the registry machine: memory for storing variables, registry set for executing operations, two types of instructions:

- Loading/saving values between memory and registries;

- Performing operations in the intermediate code;

Generating the object code

Generating object code for registry computer:

- Needs storing information related to registries and variables:

- The set VAR(R): associated to registry R; contains the variables whose values are in R;

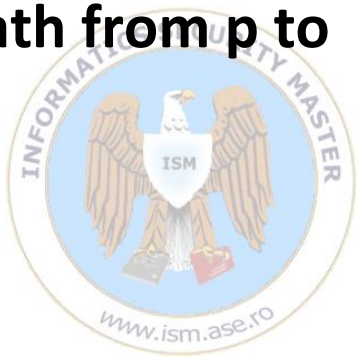
- The set MEM(v): associated to variable v; contains locations where is found the value of variable v; field in in the symbol table;

Generating the object code

Registry allocation:

- Objective: selecting variables that must be loaded in registries;
- Limited number of registries;
- Reuse the registry to store several values during the program execution;
- Active variable in point p:

  - Defined before point p, on any path;
  - Can be used in instruction s and there is a path from p to s;
  - It is not destroyed between p and s;



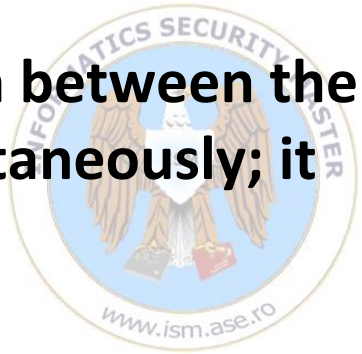
Generating the object code

Registry allocation:

- Active interval: active variable between point i and point j;
- Allocation algorithms: versions of the algorithm for coloring graphs;
- Interference graph: un-oriented graph in which:

  - Nodes: variables (identifiers or temps) that needs allocation;

  - Edges (interferences): there is an instruction between the two nodes (variables) that are active simultaneously; it is considered that the variables interfere;



Generating the object code

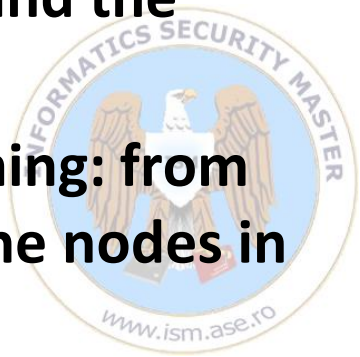
Registry allocation:

- k-colorable graph: every node in the graph is associated to the one in the k colors such that a pair of adjacent has distinct colors;
- Every color is mapped on a registry;
- Different colors: there no two variables that interfere and that are allocated to the same registry.

Generating the object code

Selecting instructions:

- Analysis of the intermediate code and modify the order of executing the instructions;
- Semantic equivalence is ensured;
- Modalities to establish the order:
  - Algorithm greedy: from the tree root the sub-tree is selected with maximum number of nodes and the descendants are visited recursively;
  - Ascendant algorithm for dynamic programming: from leaves to the root, is attributed a label to the nodes in the tree (Sethi-Ullman algorithm);



Generating the object code

Selecting instructions:

- **Sethi-Ullman algorithm: labeling nodes:**

- Leaves with 0 (less than the one in further left, with 1);

- Inner nodes: $et(N)=max(et(c_i)+i-1)$;

- Are used two stacks:

  - **Registry stack: at the beginning contains all the available registries;**
  - **Location stack: allocating temporary memory locations;**

Generating the object code

Selecting instructions:

- Working procedures for stacks:
  - push: introducing an element in the stack;
  - pop: extracting an element from the stack;
  - swap: interchange the first two elements in the stack.

Generating the object code

The object code form:

- Object code generating: machine instructions in a for decided by the compiler author;
- Does not have the usual binary file for .obj;

Generating the object code

The object code form:

- **Another conversion of the machine code to object code; complex operation avoided by the options:**

  - Assembler: transforms instructions from machine code into object code; compiling lasts longer (includes assembling), but it is compensated by avoiding to write code for the binary system;

  - Compiler for another programming language: translating the source program in program written in another language; it can be made directly from the attributed tree without generating intermediate code.

Treating errors

Characteristics:

- Minimal requirement: detection and localization (row nr., lexical token, diagnosis);
- Demanding approach: continuing the analysis on the following levels:

  - Error recovery: exit from the error area and continue the program (relatively big number of compilers);
  - Error repair: try to modify the area so that it becomes correct (small number of compilers);
  - Error correction: assumptions on the error, modify the source text (theoretical objective);

Treating errors

Characteristics:

- **Classification of compiling errors:**

- Lexical errors;

- Syntactic errors;

- Semantic errors;

- **Lexical and semantic errors: local, treated without forward-backward traversal;**

Treating errors

Characteristics:

- **Syntactic errors: global, possible to modify another lexical token than the one that caused the error;**

- **Semantic errors: detectable partially during compiling: lack of definition or multiple definitions of an identifier, using incorrectly a variable etc.**

Treating errors

Treating errors in lexical analysis:

- **Unique error type: impossibility to classify a lexical token;**

- **Error diagnosis: not respecting the lexical rules by the identifier, constants with unaccepted characters etc.**

Treating errors

Treating errors in syntactic analysis:

- Easy to implement thanks to the grammars independent of the context;
- Testing the existing of transitions based on syntactic analysis tables;
- Error recovery: implies a configuration starting from which analysis can be continued, with minimal number of modifications on the input structure, work stack, analyzer state, output structure;

Treating errors

Treating errors in syntactic analysis:

- **Error treating strategies:**

  - Panic mode: stop the algorithm until identifying a reserved word in the input structure or extracting from the stack lexical tokens until accepting the first lexical token;

  - Codification of errors in the analysis tables: analysis of the error types frequencies and call on procedure of error treating in the table location where the error occurred;

  - Error productions added to the grammar;

Treating errors

Treating errors in syntactic analysis:

- **Treating errors in syntactic analysis: LL(1): the method of the acceptable set – steps:**

1. Determine the acceptable set, including the FOLLOW set;

2. Omitting unacceptable symbols until the occurrence of an acceptable symbol in the input structure;

Treating errors

Treating errors in syntactic analysis:

3. Re-synchronizing the syntactic analyzer: continue the analysis with modifications – push şi pop transitions are attempted; if they succeed, the analyzer returns to the normal state, if not:
   - If the stack peak contains a non-terminal, is chosen the shortest way for re-copy;
   - If the stack peak contains a terminal, the prediction is inserted;
4. Step 3 is repeated until a successful transition.

Language translating & generating

- **Allow code and data for migration;**

- **The translator:**
- **Maps input constructions on output constructions;**
- **Dificulty: the number and order of factors, forward references etc;**

Language translating & generating

- **Translation – Characteristics:**

- **Understanding the input sequence from syntactic and semantic points of view;**
- **Choose the adequate output construction;**
- **Populate the output construction with elements in the input sequence;**

Language translating & generating

Translation strategies:

- **Syntax-Directed Translator :**
  - **parser with embedded actions that immediately generates the output;**
  - **An internal representation is not created;**
  - **Forward references can not be taken into consideration;**
  - **Does not generate output that significantly differs to the input in terms of operation order;**

Language translating & generating

Translation strategies (continued):

- **Rule-Based Translator:**
  - **Rule engine combined with the grammar of the input syntax;**
  - **Allows any input-output transforion wanted, on the basis of rules defined in the rule engine;**
  - **Difficulty to learn and debug the translator;**

Language translating & generating

Translation strategies (continued):

- **Model-Driven Translation:**
  - **Internal representation of the input by the parser;**
  - **Generate text during the tree crossing and not the token string parsing (Syntax-Directed);**

3. Java Virtual Machine

Bytecode interpreters

Bytecode interpreters:

Simulates a computer with the following elements (architecture):

- ***Memory for code*** **– vector of bytes with instruction codes and operands; addresses are integer;**
- ***IP Registry*** **– points to the next instruction to be executed in the code memory;**
- ***Global memory*** **– fixed number of slots for variables which points to: Integer, Float, String or struct instances; accessing variables is made through addresses and not through names (as it is done with high level interpreters)**

Bytecode interpreters

Bytecode interpreters:

Architecture (continued):

- *CPU* – is simulated and named *instruction dispatcher*: switch on bytecode for selecting the instruction;
- *Constant pool* – includes any value which can not be stored on an integer operand on 4 bytes
- *Function call stack* – parameters, local variables, the return address in the call;
- *FP – Frame Pointer – Registry* points to the stack peak for the function call.

Bytecode interpreters

Bytecode Assembler:

Characteristics:

- **bytecode – byte vector with operations and operators codes;**
- **example: memory dump for the instruction *print 1 + 2*;**
- **The sum is saved in the stack, the display takes over the stack peak;**

Bytecode interpreters

Stack-Based Bytecode Interpreter:

Simulation of hardware processor without general-scope registries:

- ***Operands stack* – contains temporary values; operands are either in the code memory, or in the operands stack;**
- ***SP – Stack Pointer – Registry* the operands stack peak.**

Bytecode interpreters

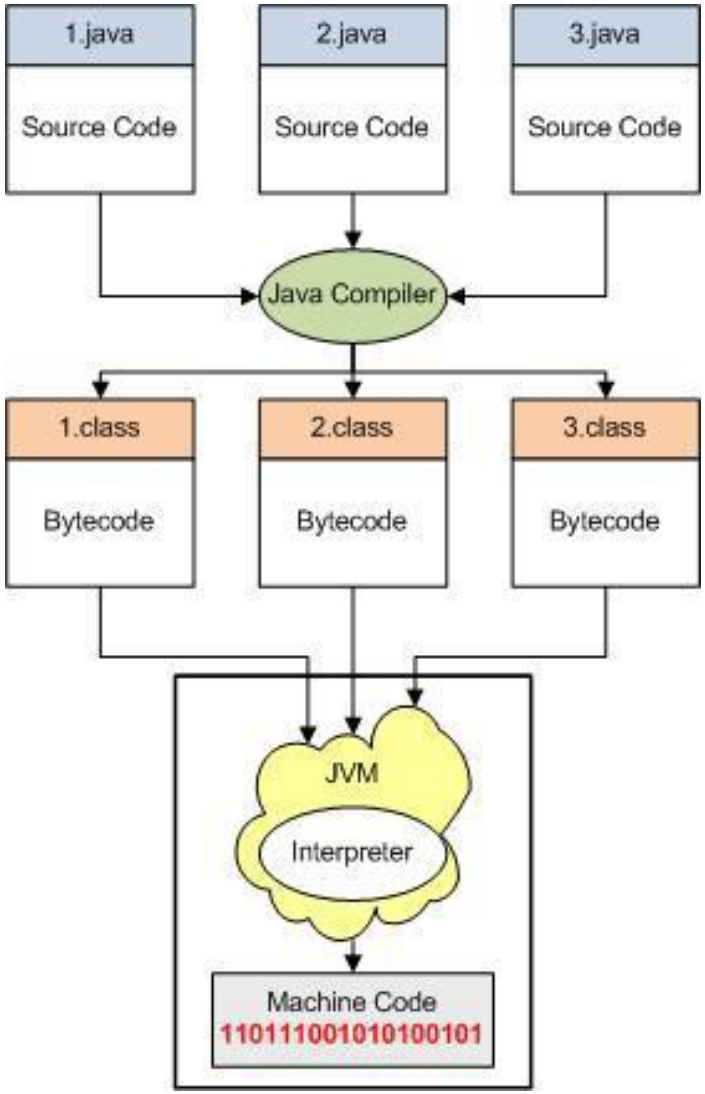
Register-Based Bytecode Interpreter:

Characteristics:

- **Does not contain operands stack, but contains a registry machine (simulated registries);**
- **Establish the registries' roles: r_0 the function returned value, r_1, …, r_p registries for retaining parameters, r_{p+1}, …, r_{p+t} local, temporary variables registries;**
- **Functions can access any registry in the registry vector, not only at one end of a structure such as the stack;**

Bytecode interpreters

Java compiling and interpreting



Java Bytecode

- Source file Angajat.java – Java 2 SDK Standard Edition v1.2.1;
- Bytecode – intermediate representation in Java language; Assembler – intermediate representation in C/C++ language;
- Assembler knowledge – adjust the execution performance and memory usage;

Java Bytecode

- **Bytecode:**

- **Generated by the javac compiler;**
- **Helps the Java programmer to improve the execution performance and memory usage;**
- **Important part of the code size and execution speed;**

Java Bytecode

- `javac Angajat.java`
- `javap -c Angajat > Angajat.bc` dis-assambler file .class;

Java Bytecode

Angajat.bc

```
Compiled from Angajat.java
class Angajat extends java.lang.Object {
    public java.lang.String Nume;
    public int id;
    public Angajat(java.lang.String,int);
    public int IDAngajat();
    public java.lang.String NumeAngajat();
    public void prelDate(java.lang.String, int);
}
```

Java Bytecode

Angajat.bc (continued)

```
Method Angajat(java.lang.String,int)
    0 aload_0
    1 invokespecial #3 <Method java.lang.Object()>
    4 aload_0
    5 aload_1
    6 putfield #4 <Field java.lang.String Nume>
    9 aload_0
   10 iload_2
   11 putfield #5 <Field int id>
   14 aload_0
   15 aload_1
   16 iload_2
   17 invokevirtual #6 <Method void
prelDate(java.lang.String, int)>
   20 return
```

Java Bytecode

Angajat.bc (continued)

JVM Instruction	Opcode	Operation	Description
`aload_0`	0x2A	Load reference from local variable.	*#0* is an index into the local variable array of the current frame. The *object reference* in the local variable is pushed onto the operand stack.
`invokespecial`	0xB7	Invoke instance method. Special handling for superclass, private, and instance initialization.	Parameters are used to construct an index (symbolic reference to a method) into the runtime constant pool.
`aload_1`	0x2B	Load reference from local variable.	*#1* is an index into the local variable array of the current frame. The *object reference* in the local variable is pushed onto the operand stack.
`putfield`	0xB5	Set field in object.	Parameters are used to construct an index (symbolic reference to a field) into the runtime constant pool.
`iload_2`	0x1A	Load `int` from local variable	*#2* is an index into the local variable array of the current frame and it must contain an `int`. The *value* of the local variable at *#2* is pushed onto the operand stack

Java Bytecode

Angajat.bc (continued)

JVM Instruction	Opcode	Operation	Description
`invokevirtual`	0xB6	Invoke instance method. Dispatch based on class	Parameters are used to construct an index (symbolic reference to a method) into the runtime constant pool. The method must not be an instance initialization method or the class or interface initialization method
`return`	0xB1	Return **`void`** from method.	The current method must have return type **`void`**. Any values on the operand stack of the current frame are discarded.

Java Bytecode

Angajat.bc (continued)

```
Method int IDAngajat()
    0 aload_0
    1 getfield #5 <Field int id>
    4 ireturn

Method java.lang.String NumeAngajat()
    0 aload_0
    1 getfield #4 <Field java.lang.String Nume>
    4 areturn

Method void prelDate(java.lang.String, int)
    0 return
```

Java Bytecode

Angajat.bc (continued)

JVM Instruction	Opcode	Operation	Description
`getfield`	0xB4	Fetch field from object.	A type reference is popped from the operand stack. Parameters are used to construct an index (symbolic reference to a field) into the runtime constant pool.
`ireturn`	0xAC	Return **`int`** from method.	The current method must have return type **`boolean`**, **`byte`**, **`short`**, **`char`**, or **`int`**. The *value* must be of type **`int`**. The value is popped from the operand stack of the current frame and pushed onto the operand stack. Any other values on the operand stack of the current method are discarded.
`areturn`	0xB0	Return *reference* from method.	The object reference is popped from the operand stack of the current frame and pushed onto the operand stack. Any other values on the operand stack of the current method are discarded.

Java Bytecode

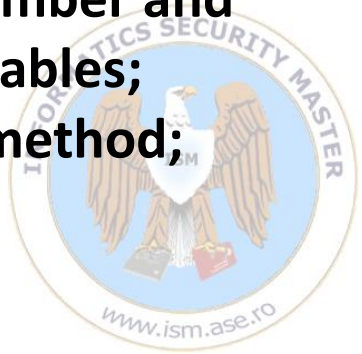
Java Virtual Machine (JVM):

- Bytecode execution;
- Stack-based machine;
- Every thread has a JVM stack that stores frames;

Java Bytecode

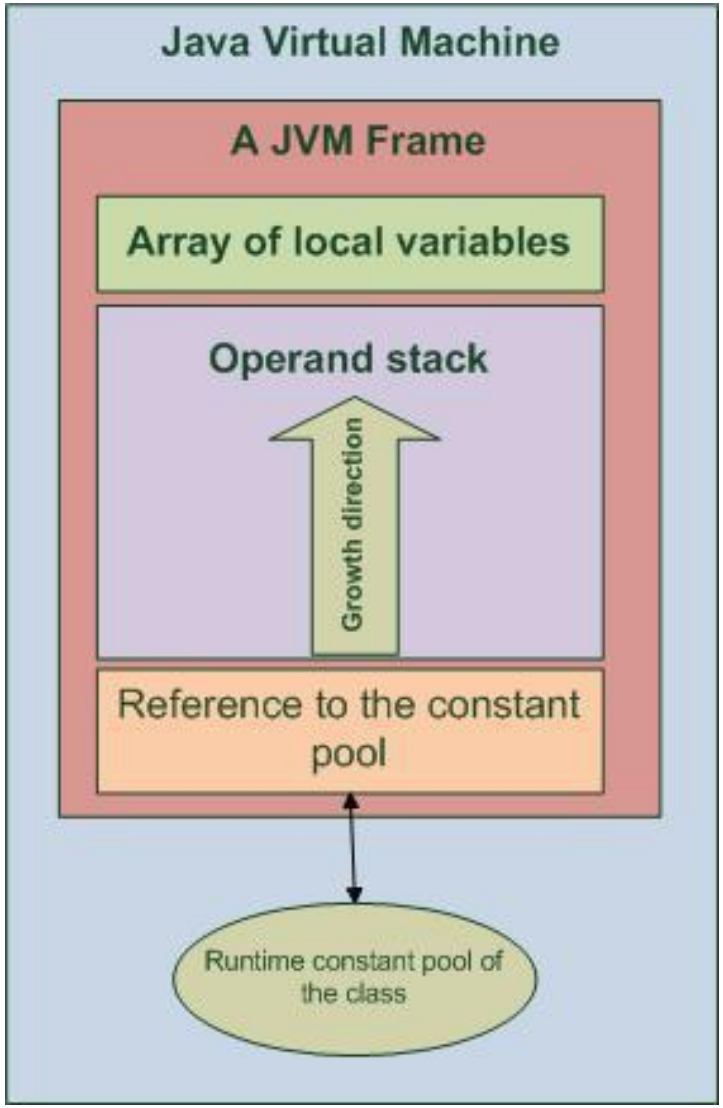
Java Virtual Machine (JVM):

- A frame is created at each method invoking; is formed by:
  - Operand stack: size computed at the compiling moment; instructions can make push, pop or pop combined with push operations; used also for values returned by methods;
  - Local variables vector (table): method parameters, variables defined locally; parameters are stored beginning with index 0; if the method is constructor or belong to an instant (not static), then the object reference is memorized at index 0, following the method parameters and the local variables; size: computed at the moment of compiling and depends on the number and types associated to the local parameters and variables;
  - Reference to class constant pool for the current method;



Java Bytecode

Java Virtual Machine (JVM) – frame structure:



Java Bytecode

Bytecode method NumeAngajat():

- ***aload_0*** **– push index value 0 from the local variable table on the operand stack (this);**
- ***getfield*** **– extracting the field from an object; pop stack (this), accessing the field address on the basis of index #4 and push reference nume angajat on the operand stack;**
- ***areturn*** **– returns a reference from the method; pop reference nume angajat from the stack and push this reference on the operand stack in the calling method;**

Java Bytecode

- Every method has attached a bytecode vector;
- Values attached to the bytecode for the method NumeAngajat(): 0, 1 and 4 – indexes of the bytecode vector where are the instructions and their operands;
- The code of an instruction occupied 1 byte; an instruction can have or not operands:
  - *aload_0* has1 byte;
  - *getfield* has 3 bytes (2 operands);
  - *areturn* has 1 byte;

Java Bytecode

Instructions for constructor method:

- ***aload_0 – push*** **reference *this* onto the operand stack;**
- ***invokespecial*** **– constructor call to superclass of Angajat class; *pop this*;**
- ***aload_0, aload_1*** **– push the first two locations in the table of the local variables: this and the first formal parameter;**
- ***putfield*** **– pop the first two values in the stack and stores the reference to employee name *Nume*;**
- ***aload_0, iload_2*** **– push this and push formal parameter 2;**
- ***putfield*** **– pop the first two values in the stack and stores id employee;**
- ***aloadd_2*** **– push this, formal parameter 1, formal parameter 2; preparing call method prelDate;**
- ***invokevirtual*** **– call method prelDate;**
- ***return*** **– close the constructor method;**

JVM structure at abstract level

- Executes compiled code under the bytecode form: binary for independent by hardware and operating system;

- Bytecode: usually stored in a file in a *class* for;

JVM structure at abstract level

- Data types:

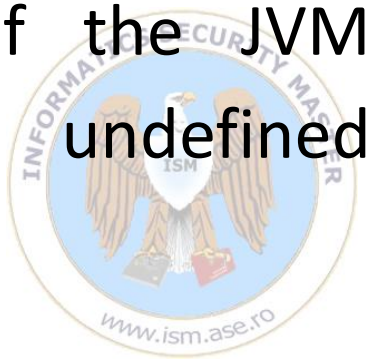
- Primitive: *byte, short, int, long, char, float, double, boolean, returnAddress*;
- Reference: classes, arrays, interfaces;
- Verification of types: at the moment of compiling and it is not done by JVM;
- JVM contains instructions specific to values of different types;

JVM structure at abstract level

- Data areas at the moment of execution:

- Some areas are created at the startup of JVM and destroyed when shutting down; others exist at thread level only during its existence;

- JVM supports several execution threads at a certain moment; an JVM thread has a ***program counter – pc*** registry; pc contains the address of the JVM instruction that is executed or has undefined content (*native* method);



JVM structure at abstract level

- Data areas at the moment of execution (continued):

- ***JVM stacks***:
  - a JVM thread is associated to an JVM stack;
  - An JVM stack is organized in frames;
  - The stack contains: local variables, partial results, method invocation and result returns;
  - Can have fixed or dynamical dimension – the stack management problem;

JVM structure at abstract level

- Data areas at the moment of execution (continued):

- ***Heap***:
  - Shared by all threads;
  - Used in space allocation for objects and arrays;
  - Created when starting up JVM;
  - Dis-allocation of heap space: automatic space management system (*garbage collector*);
  - Can have fixed or dynamically expandable dimension;

JVM structure at abstract level

- Data areas at the moment of execution (continued):

- ***Area allocated to methods***:
  - Shared by all threads;
  - Stores class structures: constant pool, fields, data şi code associated to methods;
  - Created when starting up JVM;
  - logically, it is a part of Heap;
  - Can have fixed or dynamically expandable dimension;

JVM structure at abstract level

- Data areas at the moment of execution (continued):

- ***Constant pool during execution***:
  - Representation at class or interface level of the *constant_pool* table in the *class* file;
  - Contains constants, numerical literals, method references and fields needed at the moment of execution (similar to the symbol table in conventional programming languages);
  - allocated by the area reserved for methods and built when a class or interface is created by JVM;

JVM structure at abstract level

- Data areas at the moment of execution (continued):

- ***Native method stacks***:
  - Native method – method written in another programming language than Java;
  - Allocated by thread, when it is created;
  - Can have fixed or dynamically expandable dimension;

JVM structure at abstract level

- Frame:

- Contains: data and partial results, returned values, dynamic links and exceptions;
- Created when invoking a method; destroyed at the finish of the method execution;
- Allocated by the JVM stack of the thread;
- Structure: local variables masive, operand stack, reference to class's constant pool for the current method;

JVM structure at abstract level

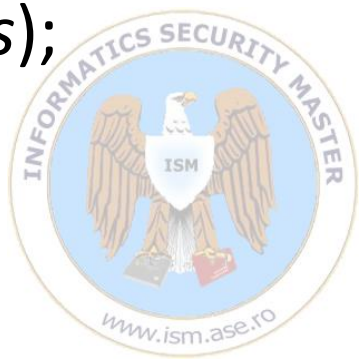
- Frame (continued):

- Active only for the method in execution:
  - Current frame of the current method;
  - Current method is defined in current class;
  - The current frame stops when the current methods calls another method, which becomes current and the frame associated to it becomes the new current frame;
- It is local to the thread that it creates and can not be referred to another thread;

JVM structure at abstract level

- Frame (continued):

- ***Local variables***:
  - The length of the massive is determined at the moment of compiling;
  - Referred to by indexing;
  - Used for transmitting the method parameters;
  - The first variable (index 0) contains the address of the object whose method is invoked (*this*);



JVM structure at abstract level

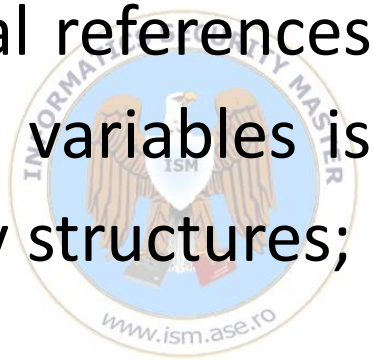
- Frame (continued):

- ***Operand stack***:
  - The depth of the stack is determined at the moment of compiling;
  - Is completed while executing the JVM instructions or is downloaded for certain JVM instructions whose result is put again in the stack;
  - Used also for transmitting the parameters and retaining the results after invoking the methods;

JVM structure at abstract level

- Frame (continued):

- ***Dynamical links***:
  - Reference to the *constant pool* for the current method type for linking to the method code;
  - The code in the *class* file contains methods that will be invoked and variables that will be accessed through symbolical references;
  - Dynamical links transform the symbolical references into concrete references, and accessing variables is transformed into offsets in their memory structures;



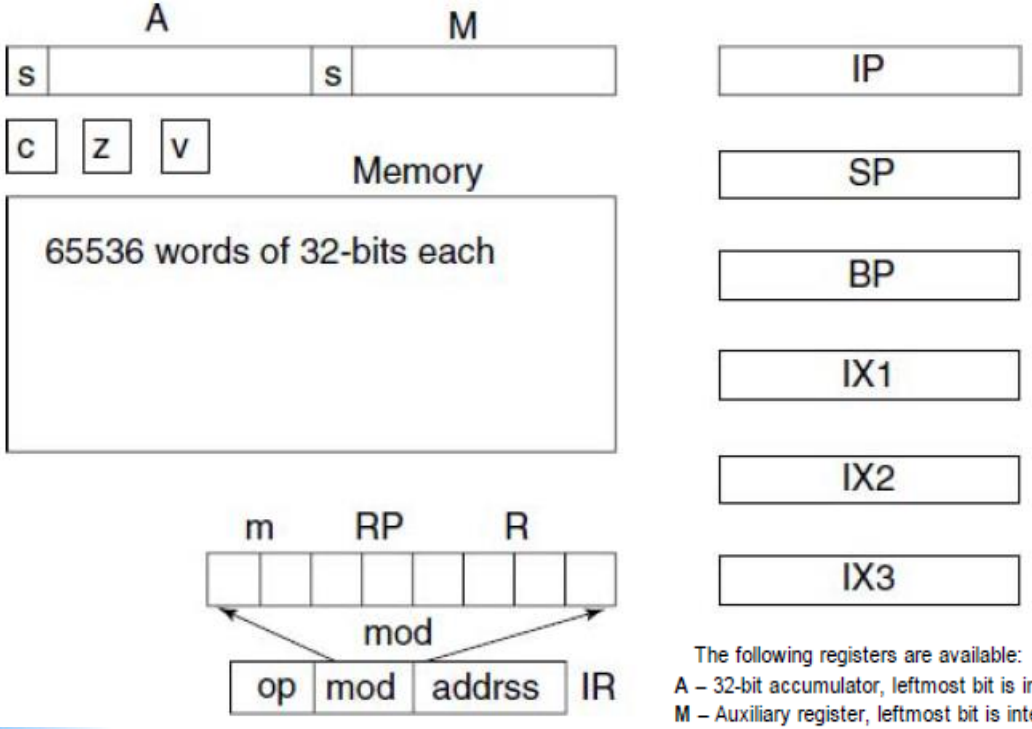
JVM structure at abstract level

- Frame (continued):

- ***Normal method termination***: the invoked method did not generate any exception either directly from JVM, or from the *throw* specification;
- ***Abrupt method termination***: generates an exception that is not managed by the method; a method terminated abruptly does not return any value to its caller;
- ***Supplementary information***: a frame can be expanded with information specific to the implementation (debugging information).

JVM structure at abstract level

- VM example programmer's view (copyright Parag Dave - Compilers Principles and Practice)



The following registers are available:

A – 32-bit accumulator, leftmost bit is interpreted as its sign bit. Register no. 0.
M – Auxiliary register, leftmost bit is interpreted as its sign bit. Register no. 4.
IP – Instruction pointer, 32-bits, but only lower 16-bits significant at present. Register no. 7.
SP – Stack Pointer, 32-bits, but only lower 16-bits significant. Register no. 6.
BP – Base Pointer, 32-bits, but only lower 16-bits significant. Register no. 5.
IX1 – Index Register 1, 32-bits; Register no. 1.
IX2 – Index Register 2, 32-bits; Register no. 2.
IX3 – Index Register 3, 32-bits; Register no. 3.
IR – Instruction Register, 32-bits; divided into OP (1-byte), MOD (1-byte) and ADDRS (2-bytes);holds an instruction being executed.
MOD – A part of IR, consisting of m (2-bits), RP (3-bits), R (3-bits); m determines the addressing mode used:

00 – Direct address
10 – Indirect w.r.t RP (the numbered registers R)
01 – Indexed w.r.t RP
11 – Indirect (indexed w.r.t RP)

JVM structure at abstract level

- Op-codes of the VM (copyright Parag Dave - Compilers Principles and Practice)

Hex	Mnemonic	Hex	Mnemonic	Hex	Mnemonic
00	NOP	10	PUSH	08	JMP
01	ADD	11	ADDI	09	JZ
02	SUB	12	SUBI	0A	JP
03	MUL	13	MULI	0B	JN
04	DIV	14	DIVI	0C	
05	LD	15	LDI	0D	
06	ST	16		0F	
07	LA	17	POP	FF	HLT

JVM structure at abstract level

- Sourcecode example compiled for VM (copyright Parag Dave - Compilers Principles and Practice)

```
        program
          int a ;
          let a = 10 ;
loop:     print (a) ;
          let a = a - 1 ;
          if a goto loop ;
          end
```

JVM structure at abstract level

- Binary code example for VM (copyright Parag Dave - Compilers Principles and Practice)

```
<load addrss> <OP> <M> <RP> <R> <ADDRS>

0100 | 15 0 0 0 000A | Load value 10 in A-reg

   1 | 06 0 0 0 0110 | store in a

   2 | 06 0 0 0 0000 | display (store in 0)

   3 | 05 0 0 0 0110 | load a

   4 | 12 0 0 0    1 | subtract 1

   5 | 0A 0 0 0 FFFB | Jump if +ve (relative)

   6 | FF 0 0 0 0    | Halt

0110 | XX X X X X    | value of a stored here
```

4. Binary encoding of assembly instructions to machine/native code

EXE Files

COM Files:

- There is not an organizing for of the binary code (only one segment);
- Execution from the 1st byte ;
- Size is not greater than 64KB ;
- Image of the application in the memory.

EXE Files

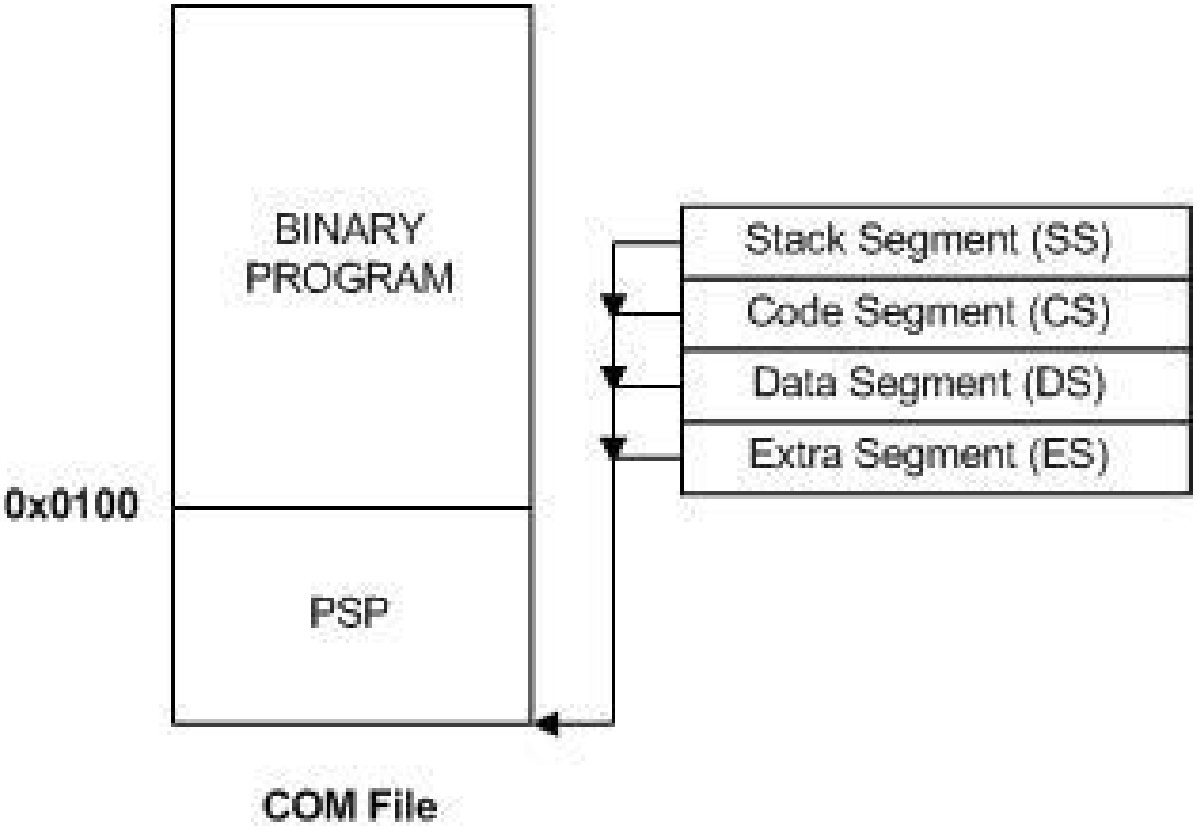
COM Files:

PSP - Program Segment Prefix

- **Control block introduced by operating system (DOS);**
- **Contains information for operating system (DOS);**
- **Length: 256 bytes;**
- **Also used to pass the arguments in DOS command prompt;**

EXE Files

Segment registers for COM Files



EXE Files

COM Files:

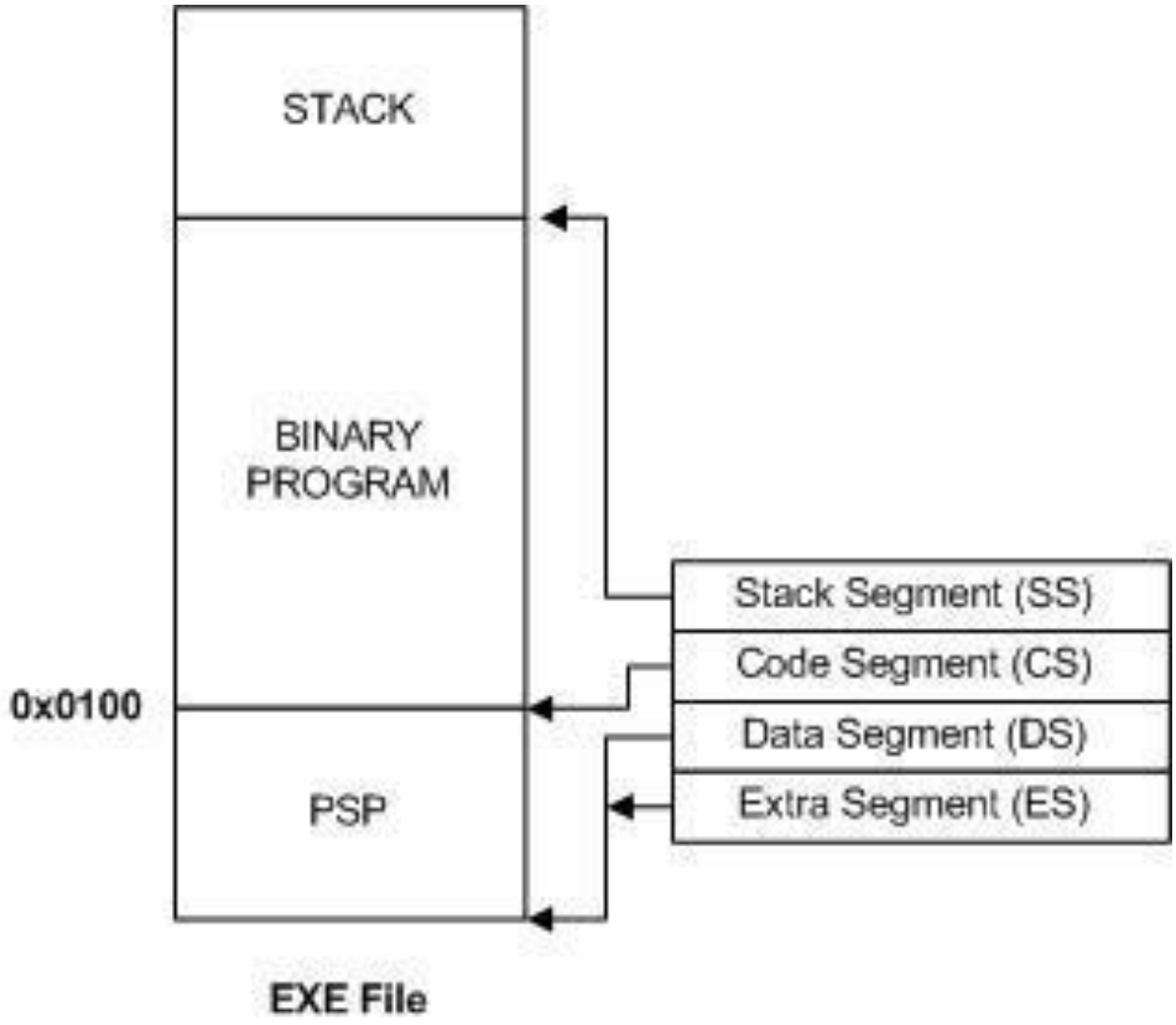
PSP - Program Segment Prefix

- **Structure:**

```
PSP struc

pspInt20                dw ?                    ; Int 20 instruction
pspNextParagraph        dw ?                    ; seg address of next paragraph
                        db ?                    ; not used
pspDispatcher           db   5 dup ( ? )        ; long call to MS-DOS
pspTerminateVect        dd ?                    ; Terminate Vector (Int 22h)
pspControlCVect         dd ?                    ; Control-C Vector (Int 23h)
pspCritErrorVect        dd ?                    ; CritError Vector (Int 24h)
pspParentId             dw ?                    ; PSP seg of parent
pspHandleTable          db  20 dup ( ? )        ; Allocated Handle Table
pspEnvironment          dw ?                    ; seg address of environment block
pspUserStack            dd ?                    ; user stack
pspFileHandleCount      dw ?                    ; handles allocated
pspFileHandlePtr        dd ?                    ; pointer to handle table
pspShareChain           dd ?                    ; Share chain pointer
                        db   4 dup ( ? )        ; unused
pspVersion              dw ?                    ; Major, Minor version (VERS)
                        db  14 dup ( ? )        ; unused
pspDosCall              db   3 dup ( ? )        ; int 21/ retf
                        db   9 dup ( ? )        ; unused
pspFCB_1                db  16 dup ( ? )        ; FCB 1
pspFCB_2                db  16 dup ( ? )        ; FCB 2
                        db   4 dup ( ? )
pspCommandTail          db 128 dup ( ? )        ; command tail (also default DTA )

        PSP ends
```

EXE Files

Segment registers for Windows EXE-NE Files (16 bits):



5. Structure of NET intermediate code

EXE Files

Windows EXE-NE Files (16 bits):

1. Old-style header info (at 0x00 offset);

2. Reserved (at 0x20 offset);

3. Offset to segmented header (see 5) (at 0x3C offset);

4. Relocation table and stub program (at 0x40 offset):
- list of pointers (fixups);
- pointers are changed when the loader relocates the program;

EXE Files

Windows EXE-NE Files (16 bits):

5. Segmented EXE header:

- general information about the EXE file;

- EXE file larger than one segment (64K);

- information on the location and size of the other sections;

- copied by Windows loader into the module table in the system data;

6. Segment table:

- one entry for each segment in EXE;

- number of segment table entries are defined in segmented EXE header (at 0X1C offset);

- one entry: 8 bytes;

EXE Files

Windows EXE-NE Files (16 bits):

7. Resource table:

- entries for each resource in EXE file;

- resource entry defines: resource ID or name ID, location and size of the resource;

8. Resident name table:

- contains module's name string and resident exported procedure name strings;

9. Module reference table:

- each entry contains an offset for the module name string within imported names table;

- each entry: 2 bytes;

EXE Files

Windows EXE-NE Files (16 bits):

10. Imported names tables:

- names of modules and procedures imported by the EXE file;

- one entry: 1 byte field for the string length, the characters;

11. Entry table:

- contains entry-point definitions grouped into bundles;

- an entry is accessed by an ordinal number adjusted to bundle index;

12. Non-Resident name table:

- module description;

- nonresident exported procedure name strings;

EXE Files

Windows EXE-NE Files (16 bits):

13. Seg 1 data:

- location and size defined in segment table entry;

14. Seg 1 info;

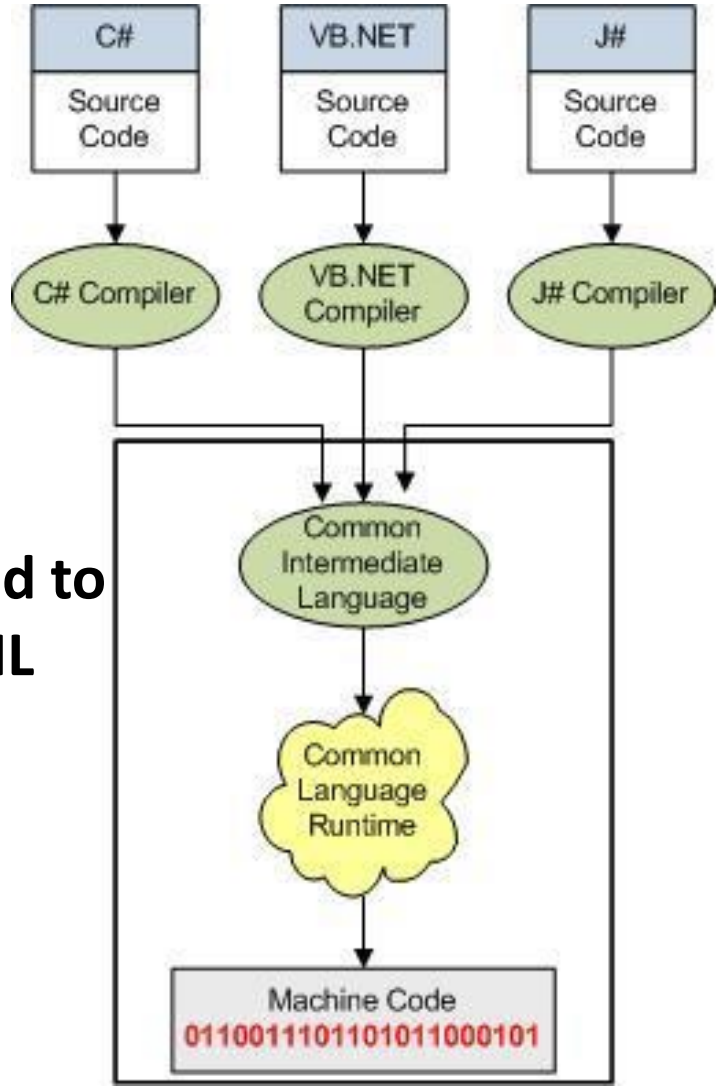
....

13+n. Seg n data;

14+n. Seg n info.

EXE Files

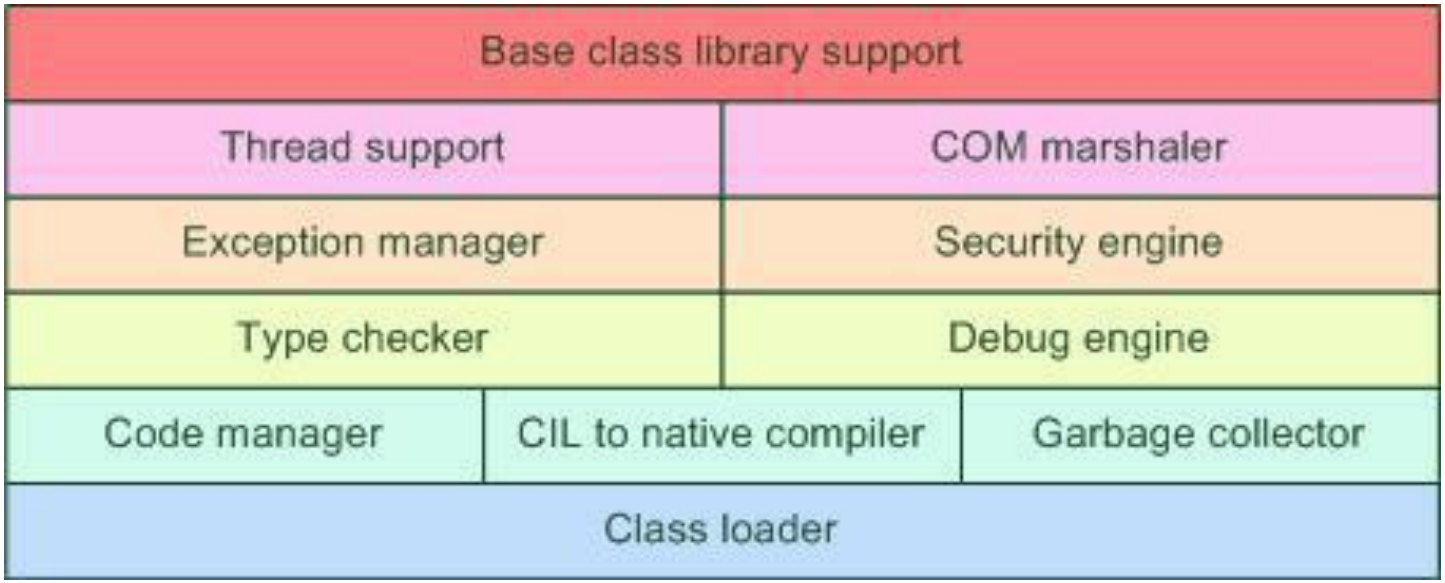
Windows EXE-PE Files (32 bits) - .NET compiling and interpreting



Source code compiled to intermediate code CIL

EXE Files

Windows EXE-PE Files (32 bits) - Common Language Runtime architecture



EXE Files

Windows EXE-PE Files (32 bits) – State of .NET method

EXE Files

Windows EXE-PE Files (32 bits):

File for structure:

1. MS-DOS information:

1.1 IMAGE_DOS_HEADER:

```
typedef struct _IMAGE_DOS_HEADER {  // DOS .EXE header
    uint16_t e_magic;              // semnătură fişier .EXE "MZ"
    uint16_t e_cblp;               // număr bytes ultima pagină din fişier (lungime imagine MOD
    512)
    uint16_t e_cp;                 // număr de pagini din fişier (pagină = 512 bytes)
    uint16_t e_crlc;               // număr realocări după DOS header
    uint16_t e_cparhdr;            // dimensiune header în paragrafe de 16 bytes
    uint16_t e_minalloc;           // număr minim de paragrafe
    uint16_t e_maxalloc;           // număr maxim de paragrafe
    uint16_t e_ss;                 // valoare (relativă) iniţială SS
    uint16_t e_sp;                 // valoare iniţială SP
    uint16_t e_csum;               // număr control
    uint16_t e_ip;                 // valoare iniţială IP
    uint16_t e_cs;                 // valoare (relativă) iniţială CS
    uint16_t e_lfarlc;             // addresă tabelă de relocare
    uint16_t e_ovno;               // overlay number (0 pentru program root)
    uint16_t e_res[4];             // bytes rezervaţi
    uint16_t e_oemid;              // OEM (original equipment manufacturer – producător promovat
    de alt producător) id
    uint16_t e_oeminfo;            // OEM info
    uint16_t e_res2[10];           // bytes rezervaţi
    uint32_t   e_lfanew;           // addresă header tip EXE-PE
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

EXE Files

Windows EXE-PE Files (32 bits):

File for structure:

1. MS-DOS information:

1.2 MS-DOS Stub Program (disassembly image);

```
seg000:0000 start proc near
seg000:0000 push cs
seg000:0001 pop ds
seg000:0002 assume ds:seg000
seg000:0002 mov dx, 0Eh
seg000:0005 mov ah, 9
seg000:0007 int 21h ; DOS - PRINT STRING
seg000:0007 ; DS:DX -> string terminated by"$"
seg000:0009 mov ax, 4C01h
seg000:000C int 21h ; DOS - 2+ - QUIT WITH EXIT CODE(EXIT)
seg000:000C start endp ; AL = exit code
seg000:000C
seg000:000C ;---------------------------------------------------------------------------
seg000:000E aThisProgramCan db 'This program cannot be run in DOS mode.',0Dh,0Dh,0Ah
seg000:000E db '$',0
```

EXE Files

Windows EXE-PE Files (32 bits):

File for structure:

2. Windows NT information `IMAGE_NT_HEADERS`:

```
typedef struct _IMAGE_NT_HEADERS
{
    DWORD Signature; //semnătură fişier EXE-PE
    IMAGE_FILE_HEADER FileHeader; //header fişier EXE-PE
    IMAGE_OPTIONAL_HEADER32 OptionalHeader; //header maşină
} IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;
```

2.1 Signature: PE signature (“PE..”);

EXE Files

Windows EXE-PE Files (32 bits):

File for structure:

2. Windows NT information `IMAGE_NT_HEADERS`:

2.2 `IMAGE_FILE_HEADER`:

```
typedef struct _IMAGE_FILE_HEADER {
    WORD Machine; //tipul maşinii pentru EXE-PE
    WORD NumberOfSections; //număr secţiuni din fişier
    DWORD TimeDateStamp; //data şi ora creării fişierului
    DWORD PointerToSymbolTable; //offset tabelă de simboluri
    DWORD NumberOfSymbols; //număr intrări în tabela de simboluri
    WORD SizeOfOptionalHeader; //dimensiune header opţional
    WORD Characteristics; //caracteristici generale
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

EXE Files

Windows EXE-PE Files (32 bits):

File for structure:

2. Windows NT information IMAGE_NT_HEADERS:

2.3 IMAGE_OPTIONAL_HEADER32:

```
typedef struct _IMAGE_OPTIONAL_HEADER {
      USHORT Magic; //diferenţiere PE 32 biţi cu PE 64 biţi
      UCHAR MajorLinkerVersion; //versiune principală linker
      UCHAR MinorLinkerVersion; //versiune secundară linker
      ULONG SizeOfCode; //dimensiune cumulată a secţiunilor de cod; de regulă este dimensiunea
      secţiunii .code
      ULONG SizeOfInitializedData; //dimensiune cumulată a secţiunilor cu date iniţializate; de
      regulă este dimensiunea secţiunii .data
      ULONG SizeOfUninitializedData; //dimensiune cumulată a secţiunilor cu date ne-
      iniţializate; de regulă este dimensiunea secţiunii .bss sau .rdata
      ULONG AddressOfEntryPoint; //adresă virtuală relativă a punctului de intrare în program
      ULONG BaseOfCode; //adresă virtuală relativă a începutului de zonă cod
      ULONG BaseOfData; //adresă virtuală relativă a începutului de zonă date
      ULONG ImageBase; //adresă încărcare fişieru executabil în memorie; adresele virtuale
      relative sunt raportate la această adresă
      ULONG SectionAlignment; //aliniere secţiuni încărcate în memorie; dimensiunea fiecărei
      secţiuni este multiplu de această valoare
      ULONG FileAlignment; //aliniere secţiuni fişier pe disk; dimensiunea fiecărei secţiuni
      este multiplu de această valoare
```

EXE Files

Windows EXE-PE Files (32 bits):

File for structure:

2. Windows NT information IMAGE_NT_HEADERS:

2.3 IMAGE_OPTIONAL_HEADER32:

```
USHORT MajorOperatingSystemVersion; //versiunea principală SO pentru execuţie fişier
USHORT MinorOperatingSystemVersion; //versiunea secundară SO pentru execuţie fişier
USHORT MajorImageVersion; //versiune principală imagine; nu mai este utilizate
USHORT MinorImageVersion; //versiune secundară imagine; nu mai este utilizate
USHORT MajorSubsystemVersion; //versiune principală subsistem de execuţie
USHORT MinorSubsystemVersion; //versiune secundară subsistem de execuţie
ULONG Reserved1; //zonă rezervată
ULONG SizeOfImage; //dimensiune fişier în memorie
ULONG SizeOfHeaders; //dimensiune cumulată a header-elor; trebuie să fie multiplu de
FileAlignment
ULONG CheckSum; //sumă de control executabil; nu este utilizată
USHORT Subsystem; //subsistem de execuţie
USHORT DllCharacteristics; //caracteristici suplimentare dacă fişierul este de tip DLL
ULONG SizeOfStackReserve; //dimensiune maximă stack
ULONG SizeOfStackCommit; //dimensiune stack care urmează a se aloca
ULONG SizeOfHeapReserve; // dimensiune maximă heap
ULONG SizeOfHeapCommit; // dimensiune heap care urmează a se aloca
ULONG LoaderFlags; //nu este utilizat
ULONG NumberOfRvaAndSizes; //număr elemente masiv DataDirectory
IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];

} IMAGE_OPTIONAL_HEADER3, *PIMAGE_OPTIONAL_HEADER;
```

EXE Files

Windows EXE-PE Files (32 bits):

File for structure:

2. Windows NT information `IMAGE_NT_HEADERS`:

2.3 IMAGE_OPTIONAL_HEADER32: `IMAGE_DATA_DIRECTORY[16]`:

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD VirtualAddress;
    DWORD Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

EXE Files

Windows EXE-PE Files (32 bits):

File for structure:

2. Windows NT information `IMAGE_NT_HEADERS`:

2.3 IMAGE_OPTIONAL_HEADER32: `IMAGE_DATA_DIRECTORY[16]`:

- Export Table;
- Import Table;
- Resource Table;
- Exception Table;
- Certificate File;
- Relocation Table;
- Debug Data;
- Architecture Data;
- Global Ptr; TLS Table;
- Load Config Table;
- Bound Import Table;
- Import Address Table;
- Delay Import Descriptor;
- COM+ Runtime Header;
- Reserved;

EXE Files

Windows EXE-PE Files (32 bits):

File for structure:

3. Sections information:

3.1 IMAGE_SECTION_HEADER[0]:

```
typedef struct _IMAGE_SECTION_HEADER {
    UCHAR Name[IMAGE_SIZEOF_SHORT_NAME]; //denumire secţiune
    union {
        ULONG PhysicalAddress;
        ULONG VirtualSize; //dimensiune secţiune la încărcarea în memorie
    } Misc;
    ULONG VirtualAddress; //adresă virtuală relativă a secţiunii; multiplu
    de SectionAlignment
    ULONG SizeOfRawData; //Dimensiune secţiune (pe disk); aliniere la
    FileAlignment
    ULONG PointerToRawData; //offset început secţiune (pe disk); aliniere
    la FileAlignment
    ULONG PointerToRelocations; //adresă secţiune în tabela de relocări
    ULONG PointerToLinenumbers; //-
    USHORT NumberOfRelocations; //număr relocări
    USHORT NumberOfLinenumbers; //-
    ULONG Characteristics; //caracteristici secţiune
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

EXE Files

Windows EXE-PE Files (32 bits):

File for structure:

3. Sections information:

…

3.(n+1) `IMAGE_SECTION_HEADER[n]`:

3.(n+2) `SECTION[0];`

…

3.(2n+3) `SECTION[n]`.

EXE Files

Windows EXE-PE Files (32 bits):

File for structure:

3. Sections names:

Name	Content
.bss	Uninitialized data (free format)
.cormeta	CLR metadata that indicates that the object file contains managed code
.data	Initialized data (free format)
.debug$F	Generated Frame Pointer Omission (FPO) debug information (object only, x86 architecture only, and now obsolete)
.debug$P	Precompiled debug types (object only)
.debug$S	Debug symbols (object only)
.debug$T	Debug types (object only)
.drective	Linker options
.edata	Export tables
.idata	Import tables
.idlsym	Includes registered Structured Exception Handler (SEH) (image only) to support Interface Definition Language (IDL) attributes.
.pdata	Exception information

EXE Files

Windows EXE-PE Files (32 bits):

File for structure:

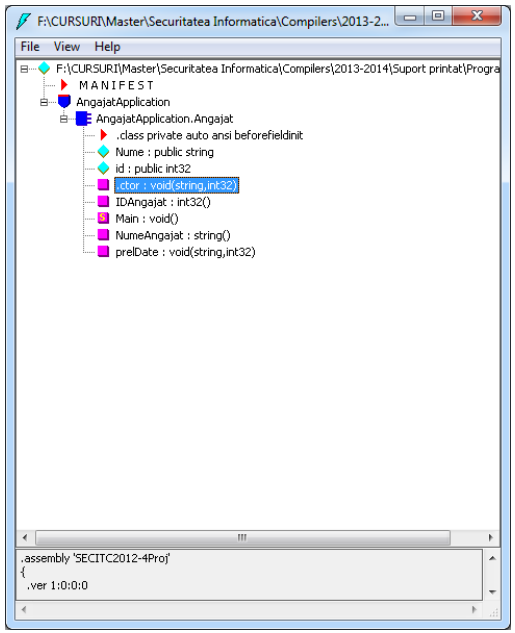
3. Sections names:

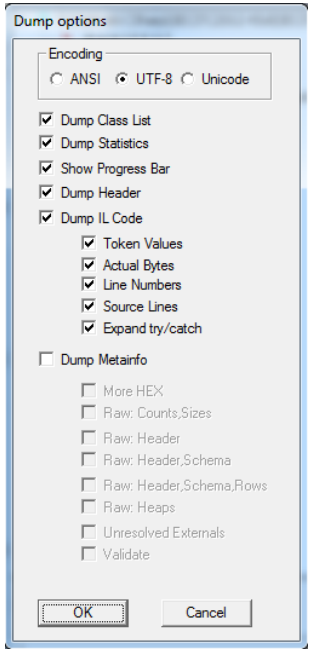
Name	Content
.rdata	Read-only initialized data
.reloc	Image relocations
.rsrc	Resource directory
.sbss	Global Pointer (GP)-relative uninitialized data (free format)
.sdata	GP-relative initialized data (free format)
.srdata	GP-relative read-only data (free format)
.sxdata	Registered exception handler data (free format and x86/object only)
.text	Executable code (free format)
.tls	Thread-local storage (object only)
.tls$	Thread-local storage (object only)
.vsdata	GP-relative initialized data (free format and for ARM, SH4, and Thumb architectures only)
.xdata	Exception information (free format)

EXE Files

Windows EXE-PE Files (32 bits) – ILDASM disassembly:

Ildasm.exe takes a portable executable (PE) file that contains intermediate language (IL) code and creates a text file suitable as input to Ilasm.exe. This tool is automatically installed with Visual Studio.





6. Structure of Java bytecode file

Java Bytecode File

Class file structure:

```
ClassFile {
        u4 magic;
        u2 minor_version;
        u2 major_version;
        u2 constant_pool_count;
        cp_info constant_pool[constant_pool_count-1];
        u2 access_flags;
        u2 this_class;
        u2 super_class;
        u2 interfaces_count;
        u2 interfaces[interfaces_count];
        u2 fields_count;
        field_info fields[fields_count];
        u2 methods_count;
        method_info methods[methods_count];
        u2 attributes_count;
        attribute_info attributes[attributes_count];
    }
```

Java Bytecode File

Class file structure:

- **`magic`** – magic number identifying the class file for; it has the value **`0xCAFEBABE`**;

- **`minor_version, major_version`** – together, determine the version of the class file for;

- **`constant_pool_count`** – the number of entries in the **`constant_pool`** table plus one;

Java Bytecode File

Class file structure:

- `constant_pool[constant_pool_count-1]`:

  - table of structures – string constants, class and interface names, field names, and other constants that are referred to within the `ClassFile` structure and its substructures;

  - the for of each entry – indicated by its first "tag" byte;

  - is indexed from 1 to `constant_pool_count-1`;

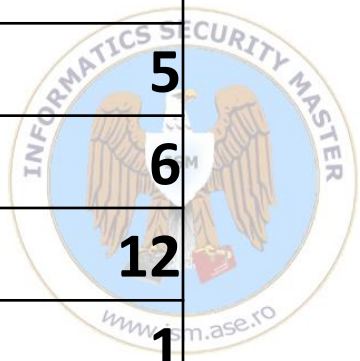
```
cp_info {
u1 tag;
u1 info[];

}
```

- `tag` – the kind of `cp_info` entry
- `info` – vary with the value of `tag`; information about the specific constant;

Java Bytecode File

Class file structure:

Constant Type	*Value*
CONSTANT_Class	7
CONSTANT_Fieldref	9
CONSTANT_Methodref	10
CONSTANT_InterfaceMethodref	11
CONSTANT_String	8
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_NameAndType	12
CONSTANT_Utf8	1

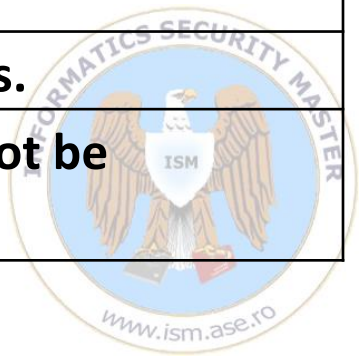


Java Bytecode File

Class file structure:

- `access_flags` - a mask of flags used to denote access permissions to and properties of this class or interface;

Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared public; may be accessed from outside its package.
ACC_FINAL	0x0010	Declared final; no subclasses allowed.
ACC_SUPER	0x0020	Treat superclass methods specially when invoked by the *invokespecial* instruction.
ACC_INTERFACE	0x0200	Is an interface, not a class.
ACC_ABSTRACT	0x0400	Declared abstract; may not be instantiated.



Java Bytecode File

Class file structure:

- `this_class` – a valid index into the constant_pool table (a `CONSTANT_Class_info structure`);

- `super_class`:
  - zero - this class file the class `Object`;
  - a valid index into the `constant_pool` table (`CONSTANT_Class_info`) – direct superclass of the class;
  - interface – always be a valid index into the `constant_pool` table;

- `interfaces_count` – the number of direct superinterfaces of this class or interface type;

Java Bytecode File

Class file structure:

- **`interfaces[interfaces_count]` – each value – a valid index into the `constant_pool table` (`CONSTANT_Class_info`);**

- **`fields_count` – the number of field_info structures in the fields table;**

- **`fields[fields_count]`:**
  - **a complete description of a field in the class or interface;**
  - **includes only fields declared by the class or interface;**
  - **does not include fields inherited from superclasses or superinterfaces.**

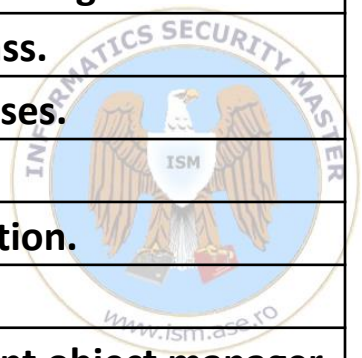
Java Bytecode File

Class file structure:

```
field_info {
u2 access_flags;
u2 name_index;
u2 descriptor_index;
u2 attributes_count;
attribute_info attributes[attributes_count];
}
```

- `access_flags` – a mask of flags used to denote access permission to and properties of this field;

Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared public; may be accessed from outside its package.
ACC_PRIVATE	0x0002	Declared private; usable only within the defining class.
ACC_PROTECTED	0x0004	Declared protected; may be accessed within subclasses.
ACC_STATIC	0x0008	Declared static.
ACC_FINAL	0x0010	Declared final; no further assignment after initialization.
ACC_VOLATILE	0x0040	Declared volatile; cannot be cached.
ACC_TRANSIENT	0x0080	Declared transient; not written or read by a persistent object manager.



Java Bytecode File

Class file structure:

- `name_index` – a valid index into the `constant_pool` table (`CONSTANT_Utf8_info`), a valid field name;
- `descriptor_index` – a valid index into the `constant_pool` table, a valid field descriptor;
- `attributes_count` – number of additional attributes of the field;
- `attributes[attributes_count]` - a field can have any number of attributes associated with it;

Java Bytecode File

Class file structure:

```
attribute_info {
    u2 attribute_name_index;
    u4 attribute_length;
    u1 info[attribute_length];
}
```

- `attribute_name_index` – a valid unsigned 16-bit index into the constant pool of the class (`CONSTANT_Utf8_info`), the name of the attribute;
- `attribute_length` - the length of the subsequent information in bytes; does not include the initial six bytes;
- `info[attribute_length];`

Java Bytecode File

Class file structure:

- `methods_count` – the number of `method_info` structures in the methods table

- `methods[methods_count]`:

  - a complete description of a method in this class or interface;

  - `method_info` structures represent all methods declared by this class or interface type; does not include methods inherited from superclasses or superinterfaces;

Java Bytecode File

Class file structure:

```
method_info {
u2 access_flags;
u2 name_index;
u2 descriptor_index;
u2 attributes_count;
attribute_info attributes[attributes_count];
}
```

- `access_flags` – denote access permission to and properties of this method

Java Bytecode File

Class file structure:

Flag Name	Value	Interpretation
ACC_PUBLIC	0x0001	Declared public; may be accessed from outside its package.
ACC_PRIVATE	0x0002	Declared private; accessible only within the defining class.
ACC_PROTECTED	0x0004	Declared protected; may be accessed within subclasses.
ACC_STATIC	0x0008	Declared static.
ACC_FINAL	0x0010	Declared final; may not be overridden.
ACC_SYNCHRONIZED	0x0020	Declared synchronized; invocation is wrapped in a monitor lock.
ACC_NATIVE	0x0100	Declared native; implemented in a language other than Java.
ACC_ABSTRACT	0x0400	Declared abstract; no implementation is provided.
ACC_STRICT	0x0800	Declared strictfp; floating-point mode is FP-strict

Java Bytecode File

Class file structure:

- **`name_index` – a valid index into the `constant_pool` table (`CONSTANT_Utf8_info`), representing either one of the special method names `<init>` or `<clinit>`, or a valid method name;**

- **`descriptor_index` – a valid index into the `constant_pool` table (`CONSTANT_Utf8_info`), representing a valid method descriptor;**

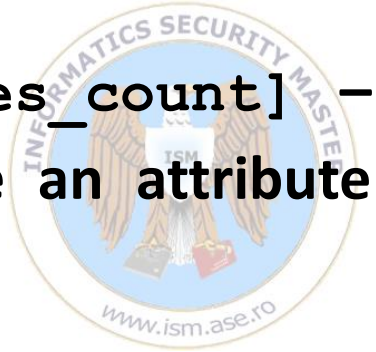
Java Bytecode File

Class file structure:

- `attributes_count` – the number of additional attributes;
- `attribute_info attributes[attributes_count]` – each value of the attributes table must be an attribute structure;

- `attributes_count` – the number of attributes in the attributes table of the class;

- `attribute_info attributes[attributes_count]` – each value of the attributes table must be an attribute structure.



Java Bytecode File

Example: ByteStr.java

```
public class ByteStr {

  public static void main(String argv[]){
     int x, y, z;
     x = 1;
     y = 2;
     z = x+y;

  }
}
```

Java Bytecode File

Example: ByteStr.class

```
CAFEBABE00000033000F0A0003000C07
000D07000E0100063C696E69743E0100
03282956010004436F646501000F4C69
6E654E756D6265725461626C65010004
6D61696E010016285B4C6A6176612F6C
616E672F537472696E673B295601000A
536F7572636546696C6501000C427974
655374722E6A6176610C00400050100
074279746553747201001 06A6176612F
6C616E672F4F626A656374 0021000200
030000000000200010004 0005000100
060000001D0001000100000052AB700
01B1000000100070000006 00010000
000400090008000900010 0060000002D
000200040000009043C053D1B1C603E
B1000000100070000012 0004000000
080002000900040 00A0008000C000100
0A00000002000B
```

Java Bytecode File

Example: ByteStr.java

attribute_info attributes[1]

2A B7 00 01 B1 code[5]

2A *aload_0* //load the reference #0 to operands stack

B7 *invokespecial* **00 01** //call super constructor

//**00 01** argument of **invokespecial** (index in constant_pool):

// #1 -> #12 -> #4 (name: (**<init>**) #5 (descriptor: **()V**)

B1 *return* // close the method

Java Bytecode File

Example: ByteStr.java

attribute_info attributes[2]

04 3C 05 3D 1B 1C 60 3E B1 code[9]

04 *iconst_1* // load the constant #1 to operands stack

3C *istore_1* // take the value #1 from operand stack and store in local variable

05 *iconst_2* // load the constant #2 to operands stack

3D *istore_2* // take the value #2 from operand stack and store in local variable

1B *iload_1* // load the local variable #1 to operand stack

1C *iload_2* // load the local variable #2 to operand stack

60 *iadd* // add 2 integers of the stack top

3E *istore_3* // load the local variable #3 to operand stack

B1 *return* // close the method