5. From high-level language to machine language

Assembly languages

- Machine language is raw number language in binary form (or, in shorter notations, hexadecimal form), so it's not very readable
- In the previous lecture we used symbolic machine language
- More advanced symbolic machine languages are actually called assembly languages
  - Key improvements include i.e. labels for program and memory locations & expressions (no need to think about memory addresses, calculations can be done with one command…)
  - Code written using these can be transformed to binary machine code using an assembler
- Assembly languages have words that specify the operations better than numerical op-codes, but these words still correspond to operations that are performed by the computer's physical components
- Still quite low level of abstraction

High-level languages

- On order to make programming easier, higher-level programming languages have been developed – first ones already in the late 1950s (FORTRAN, LISP)
- More advanced ones soon followed (C/C++, Python)
- High-level languages contain advanced control- and data structures
  - No need for primitive JUMP-commands
  - No need to think about memory addresses or contents of the accumulator
  - Emphasis on logic (”what happens”) instead of execution details (”how it happens”)
- Programming is easier, programs are shorter and easier to debug & modify
- Programs written on high-level languages are portable to nearly any device
- …so, have high-level languages made lower ones obsolete?

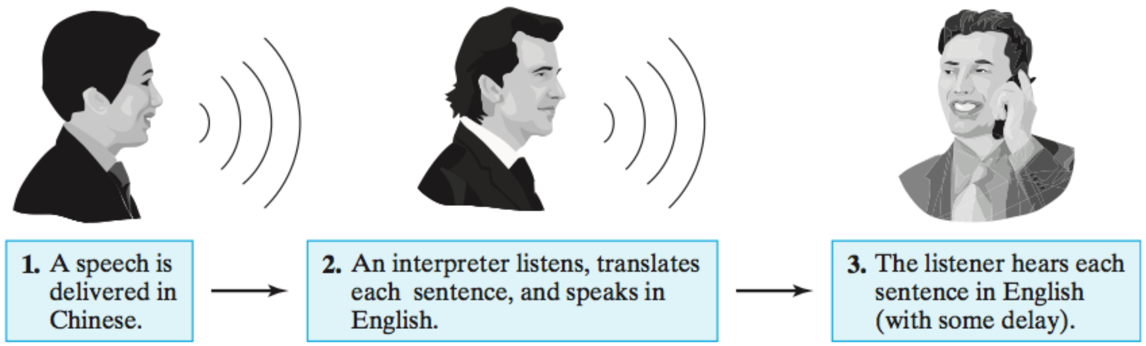
High-level vs. low-level languages

- Short answer is no
- Lower-level programs require less memory and execute quicker
- Also, lower-level programs are capable of making better use of the resources available (memory, registers, processor use)
- Therefore, if the programs needed are simple but require
  - As fast execution time as possible
  - Low energy consumption
- …it’s good to consider a lower-level program
  - For example: Automotive applications (ECU, ESP system), wearables (sports watches etc.)
- Otherwise, high-level languages are strongly favorable

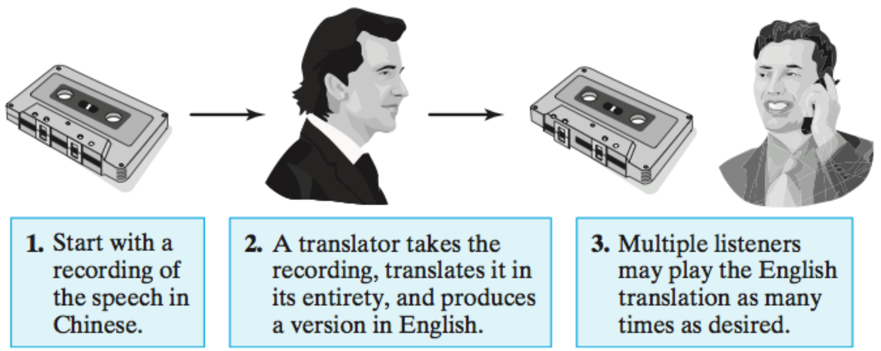
Compiler vs. interpreter

- A program written on high-level language must be translated to machine code in order to execute it
- This translation can be done in two ways: using a compiler or by using an interpreter
  - The difference can be understood by using an analogy: translating a speech from Chinese to English

Interpreter



Compiler



Compiler vs. interpreter

- Both have their advantages and disadvantages
- Interpreter:

  + Translation is (almost) immediate

  - Translation is not stored anywhere for further use
- Compiler:

  + Translation must be done only once; after it's done, the translation can be listened to multiple times by multiple people

  - Translation takes some time to complete
- Interpreter is a good choice in development phase, when it's important to test and debug the program quickly
- When the program is ready, compiling it is a good idea due to increased speed
  - Also, end user doesn't need an interpreter to run the program; it runs as a stand-alone
- Interpreters are commonly used in web pages (JavaScript etc.)

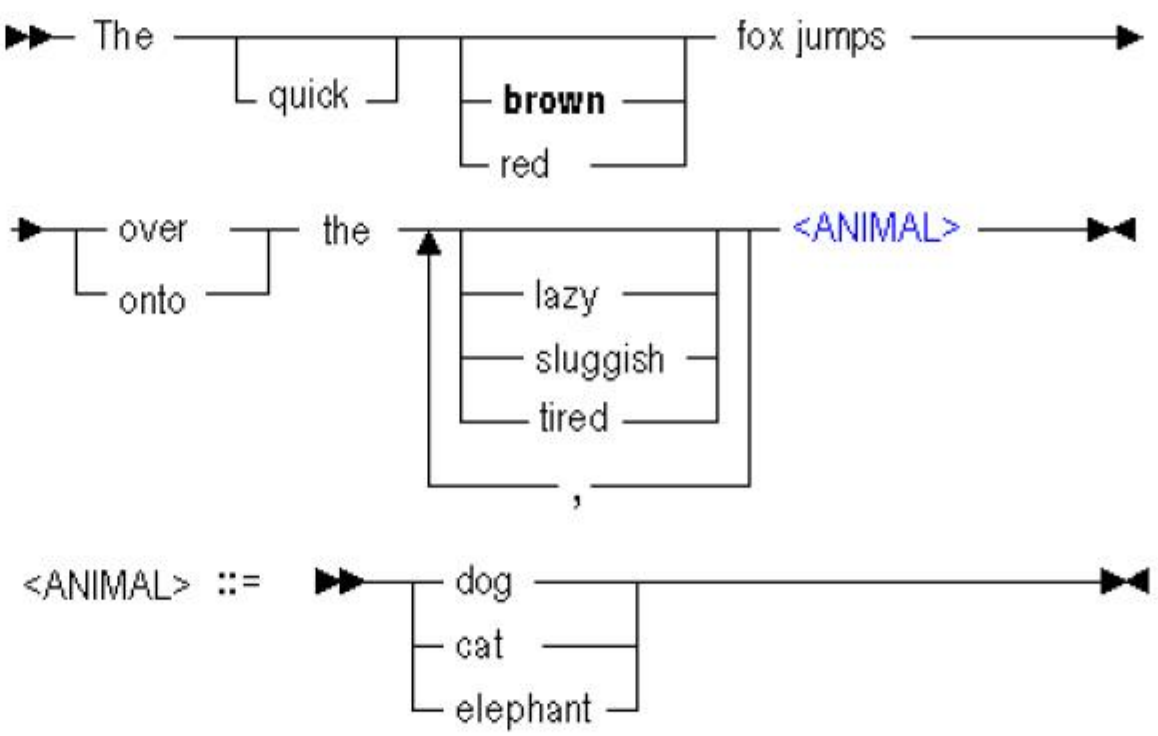
Syntax and grammar

- In order to function automatically, the translation process requires a well defined set of rules for converting a long string of characters to a program
- This set of rules is called the syntax of the language
- The syntax is defined by grammar (like with natural languages)
- Grammar rules are called productions, which specify which kind of characters & character strings are included in the language
  - If some character string is not included in the language, the translation process is halted
- We’ll take a closer look at these a bit later…

Syntax and grammar

- Grammar of programming languages is not that different from natural languages
  - Definitions are more accurate, though
  - Also, there are no exceptions to rules

PRODUCTION	SEMANTIC RULE
$L \to E\mathrm{n}$	$print(E.val)$
$E \to E_1 + T$	$E.val := E_1.val + T.val$
$E \to T$	$E.val := T.val$
$T \to T_1 * F$	$T.val := T_1.val * F.val$
$T \to F$	$T.val := F.val$
$F \to (E)$	$F.val := E.val$
$F \to \mathrm{digit}$	$F.val := \mathrm{digit}.lexval$

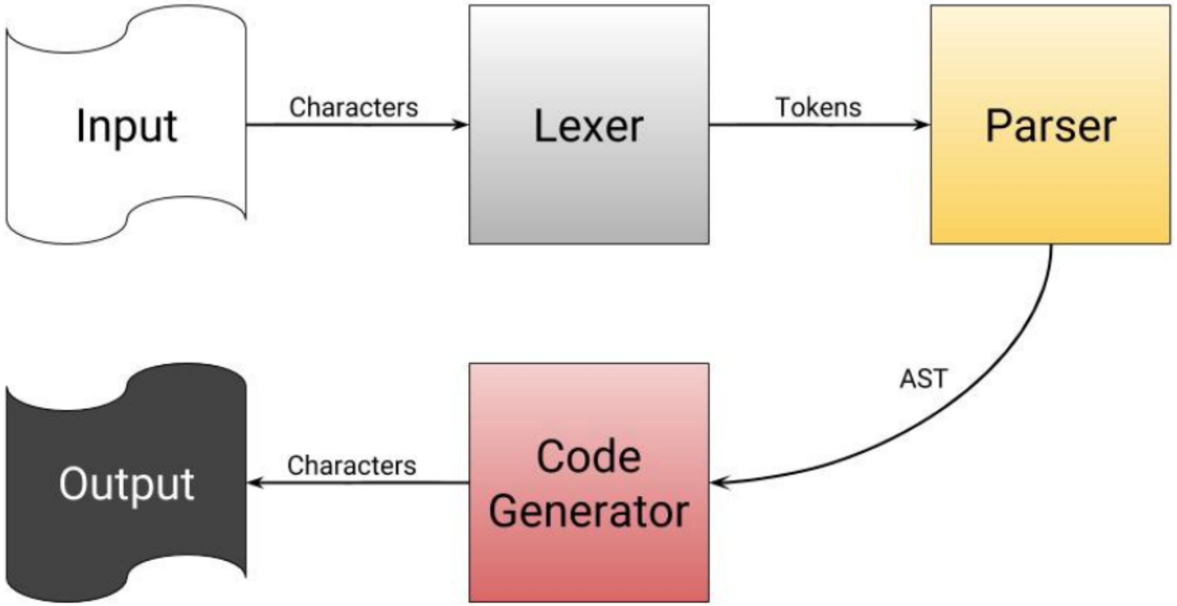


Context-free grammar

- The syntax of programming languages is often described using context-free grammar (also known as Backus-Naur-Form grammar; BNF)
- This means that the structuring options of components is not dependent on their surroundings
  - Natural languages are not context-free; for example, English word ”lead” can be interpreted in at least three possible ways (noun, verb, adjective (”lead singer”))
- Context-free grammar is comprised of four elements:
  - Terminals
  - Nonterminals
  - Start symbol (actually a nonterminal, but a special case)
  - Production rules
- In BNF, nonterminals only appear singly on the left sides of productions

Compiling process

- A compiler translates the high-level language program (input) to machine level language (output) in three stages:
  - Lexing (lexical analysis)
  - Parsing (syntactic analysis)
  - Code generation
- Optimization of output is possible



Lexing

- Lexing identifies the characters that belong to a single text element
- These text elements are called tokens, which can be categorized as follows:
  - Keywords (IF, WHILE, FOR, END)
  - Operators (+, -, *, <)
  - Separators ([, ;) :)
  - Identifiers (x, y, val)
  - Literals (7, 523, pi)
  - Comments (optional)
- These tokens represent the terminals
- Identifiers are stored in a symbol table

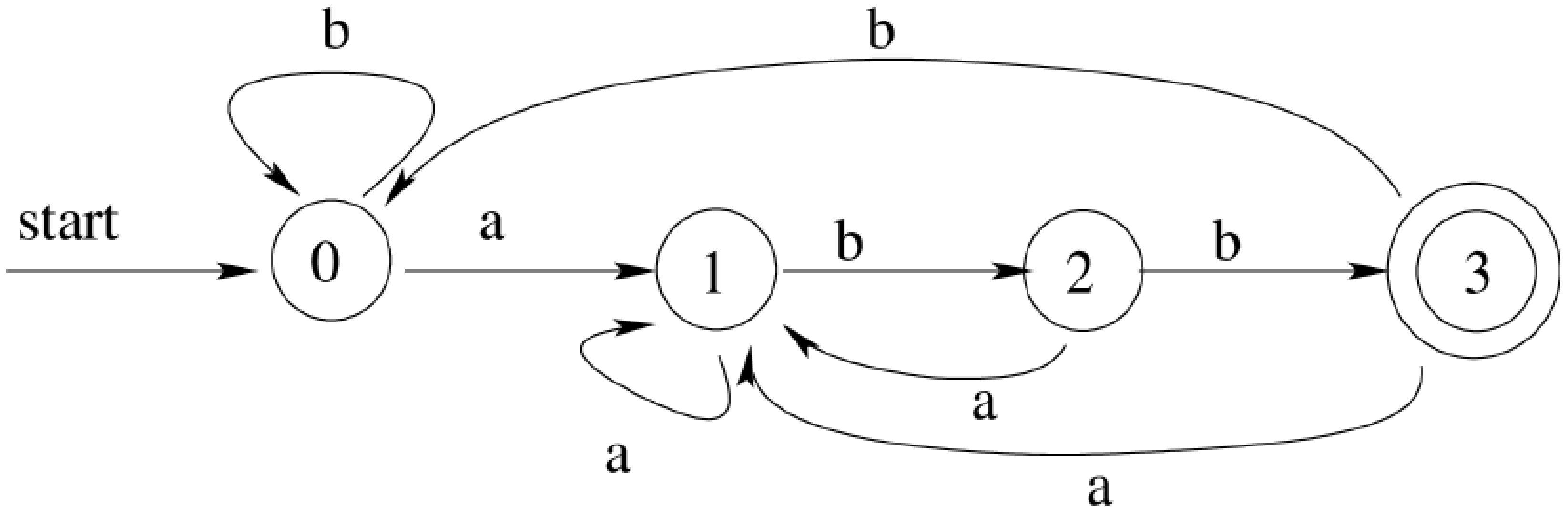
IF x < 5 THEN x:= x + 1

K I O L K I O I O L

State machines

- Lexer is basically a state machine that returns the identified tokens
- State machine can be defined using regular expressions
  - State machine below identifies, for example, inputs (a | b)*abb
  - Try inputs i) ababb ii) abbbaabb iii) abbab. Which ones of these it identifies as tokens?

This means ”a or b, repeated 0 to infinite number of times”!

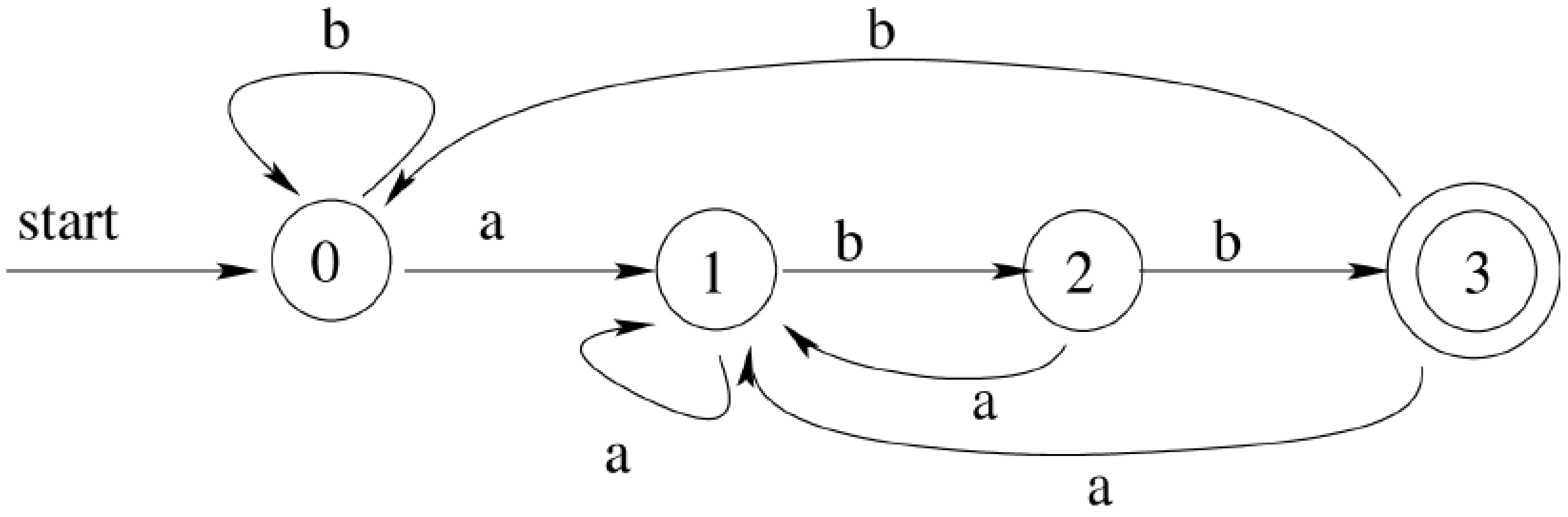


Inputs that end in double circled dots are identified!

State machines

- Lexer is basically a state machine that returns the identified tokens
- State machine can be defined using regular expressions
  - State machine below identifies, for example, inputs (a | b)*abb
  - Try inputs i) ababb ii) abbbaabb iii) abbab. Which ones of these it identifies as tokens?
    - Inputs i and ii, but not iii!

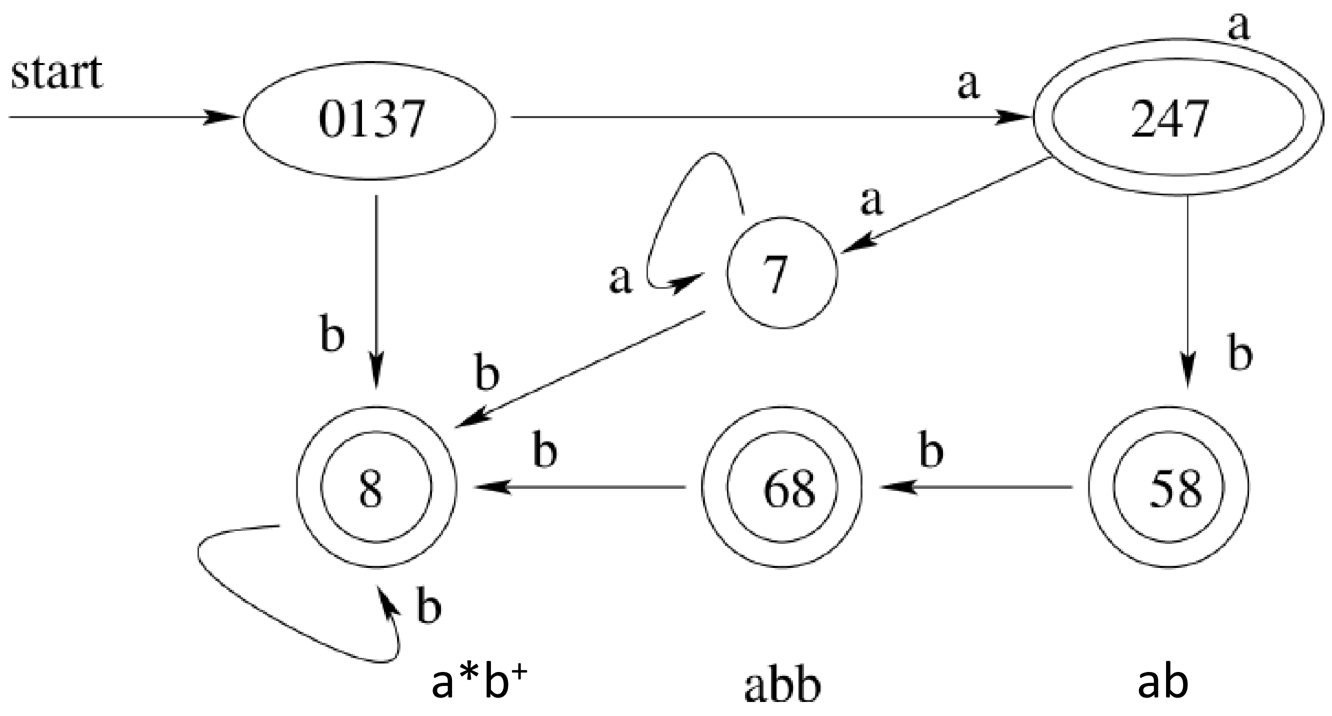
This means ”a or b, repeated 0 to infinite number of times”!



Inputs that end in double circled dots are identified!

State machines

- A state machine can identify multiple different tokens
- For example, the one below identifies tokens of forms a and a^*b^+
  - State 247 identifies a, state 58 ab, state 68 abb and state 8 all others



a^* = 0 to infinite pcs of a's
b^+ = 1 to infinite pcs of b's

Parsing

- Lexing produces us a list of identified tokens
- The idea of parsing is to find out the syntactic structure of tokens
- The result is usually shown as a parse tree
  - Start symbol is at the root of the tree
  - Leaf nodes are terminals
  - Interior nodes are non-terminals
  - Resulting sentence is read from left to right
- If the grammar of the language is properly defined, the resulting parse tree should be unambiguous

Parsing

- There are two possible strategies for parsing:
  - Top-down (from root to leaves)
  - Bottom-up (from leaves to the root)
- Top-down parsing is a logical choice
  - Grammar productions are used in order to modify the root
  - If the root can be modified to match the input, the parse tree can be formed
- Bottom-up parsing goes the other way
  - Grammar productions are used in order to modify the input
  - If the input can be modified to match the root, the parse tree can be formed
- Parsing process (especially bottom-up) is quite nondeterministic, if the precedence of productions is not defined in the grammar
  - Need to back off from “bad guesses”

Parse tree

- Example: grammar is defined as

"id" is a terminal, E acts as a start symbol as well as a non-terminal

$$E \rightarrow E+E \mid E*E \mid -E \mid (E) \mid \mathbf{id}$$

Parse tree

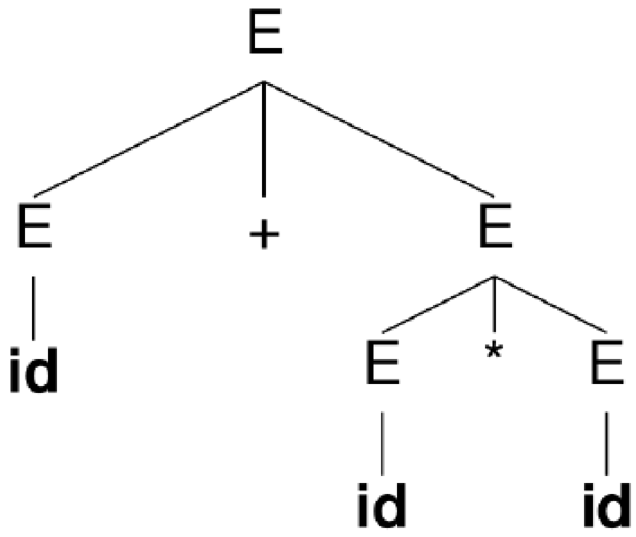
- Example: grammar is defined as

"id" is a terminal, E acts as a start symbol as well as a non-terminal

$$E \to E+E \mid E*E \mid -E \mid (E) \mid \mathbf{id}$$

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow \mathbf{id} + E \\ &\Rightarrow \mathbf{id} + E * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \end{aligned}$$

- Parse tree from input id+id*id:



Parse tree

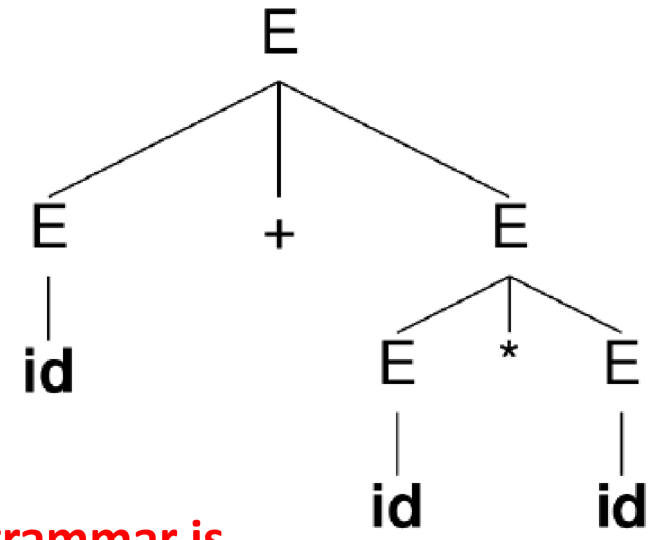
- Example: grammar is defined as

"id" is a terminal, E acts as a start symbol as well as a non-terminal

$$E \to E+E \mid E*E \mid -E \mid (E) \mid \mathbf{id}$$

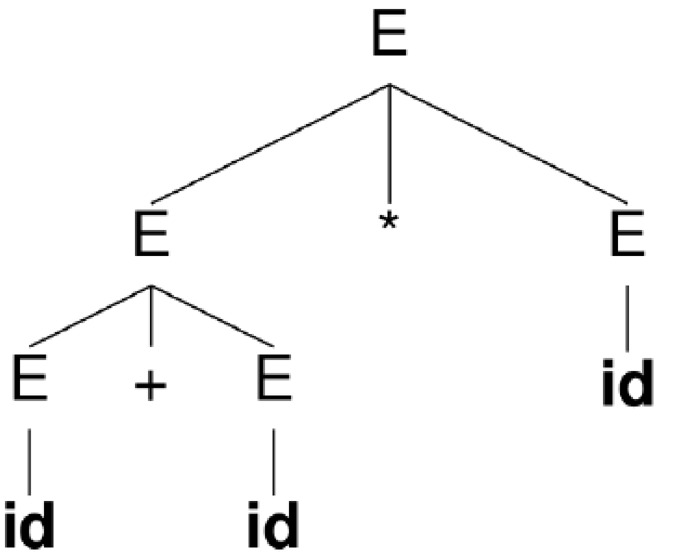
- Parse tree from input id+id*id:

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow \mathbf{id} + E \\ &\Rightarrow \mathbf{id} + E * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \end{aligned}$$



- …or, if we use productions in another order:

$$\begin{aligned} E &\Rightarrow E * E \\ &\Rightarrow E + E * E \\ &\Rightarrow \mathbf{id} + E * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \end{aligned}$$



Two different parse trees! So, the grammar is poorly defined and parsing will result in an error.

Parse tree

- The previous example was top-down; let's use bottom-up now for a change
- In the previous example, the problem was that all the productions were at same level; let's try a grammar which has more non-terminals
- Can we now make a parse tree out of, say, input id*id?

$$E \rightarrow E+T \mid T$$
$$T \rightarrow T*F \mid F$$
$$F \rightarrow (E) \mid \mathbf{id}$$

Parse tree

- The previous example was top-down; let's use bottom-up now for a change
- In the previous example, the problem was that all the productions were at same level; let's try a grammar which has more non-terminals
- Can we now make a parse tree out of, say, input id*id?
  - YES!

$$E \rightarrow E+T \mid T$$
$$T \rightarrow T*F \mid F$$
$$F \rightarrow (E) \mid \textbf{id}$$

id*id	→	F *id	→	T* id	→	T* F	→	T	→	E
		id		F		F id		T*F		T
				id		id		F id		T * F
								id		F id
										id

Old (replaced) productions shown below.

Parse tree

if E_1 then if E_2 then S_1 else S_2

- Especially a case of multiple ifs can result in ambiguous parse trees
- Is the last “else” associated with E_1 or E_2?
- Grammar rules are needed to make the distinction!
  - Brackets
  - Tabulations

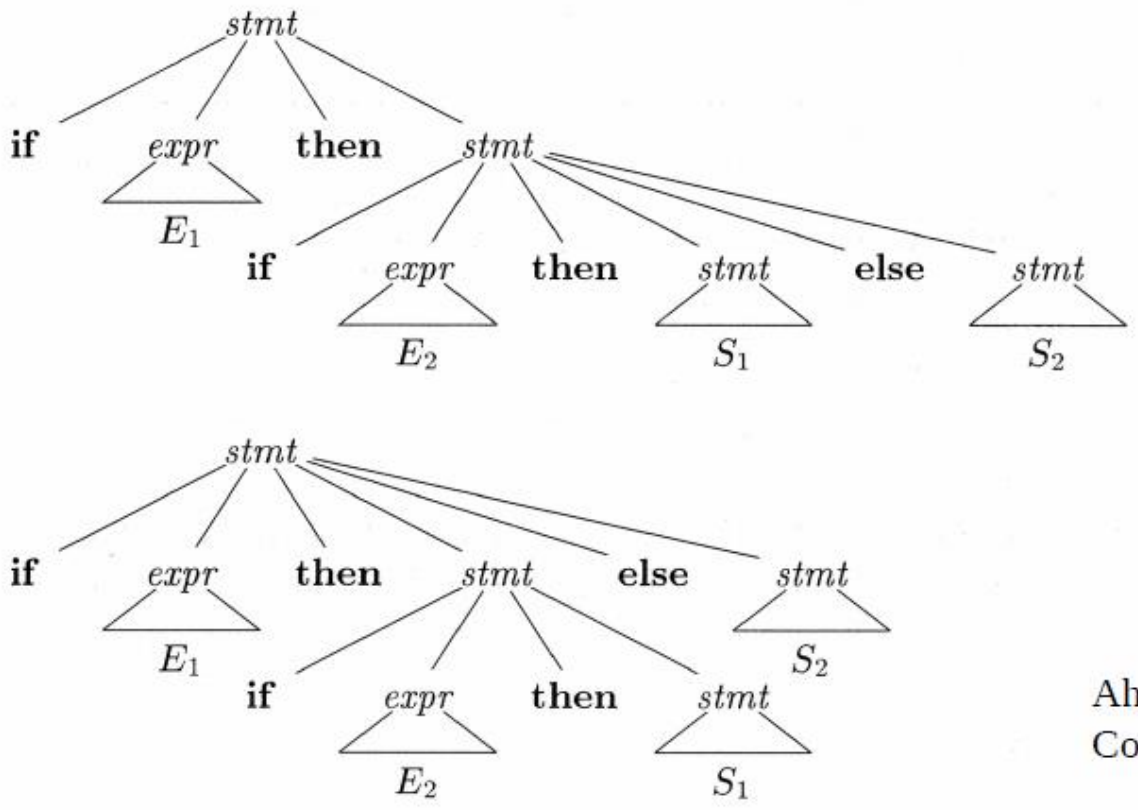


Figure 4.9: Two parse trees for an ambiguous sentence

Aho, Lam, Sethi, Ullman: Compilers. Pearson, 2007.

From parsing table to parse tree

- Grammar rules can be expressed in the form of a parsing table
- During this course, we will not consider how to formulate a parsing table from grammar; we will only learn how to use the table in order to build a parse tree
- Example grammar:

$$
\begin{array}{lll}
E & \rightarrow & TE' \\
E' & \rightarrow & +TE' \mid \epsilon \\
T & \rightarrow & FT'
\end{array}
\qquad
\begin{array}{lll}
T' & \rightarrow & *FT' \mid \epsilon \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

- Parsing table (top-down):

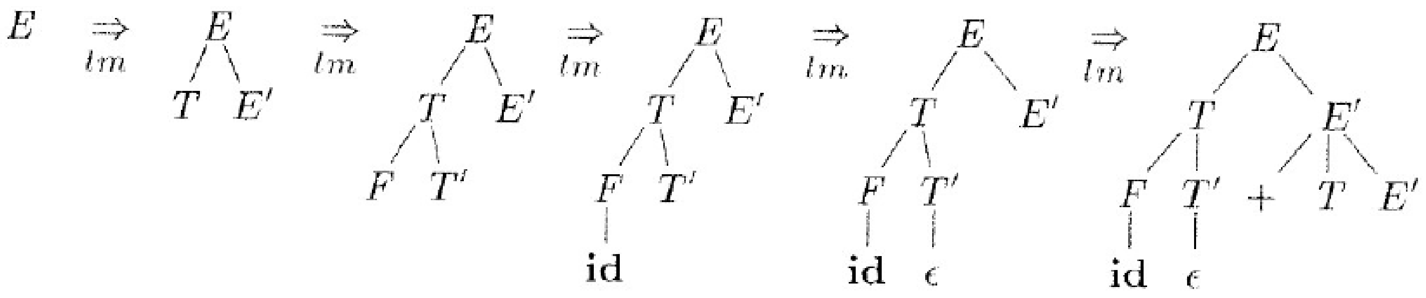
Non-terminal	Input Symbol					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \mathbf{id}$			$F \rightarrow (E)$		

ϵ is a "zero symbol" which is used to terminate branches we don't want

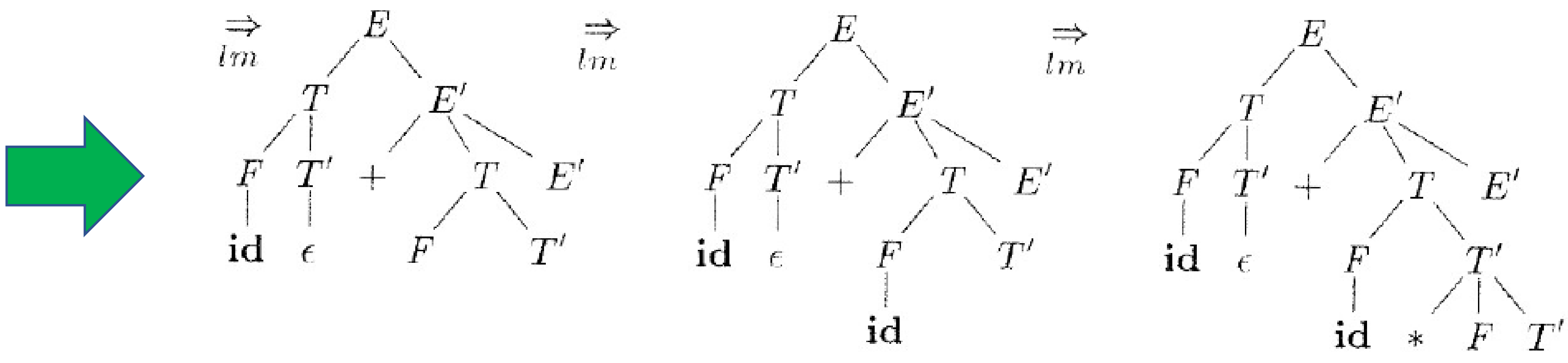
Parsing table row = what should be replaced
Parsing table column = what is our target
→ specifies which production should be used, so the process is now deterministic!

From parsing table to parse tree

- Parse tree iteration for input id+id*id in the previous grammar (top-down):



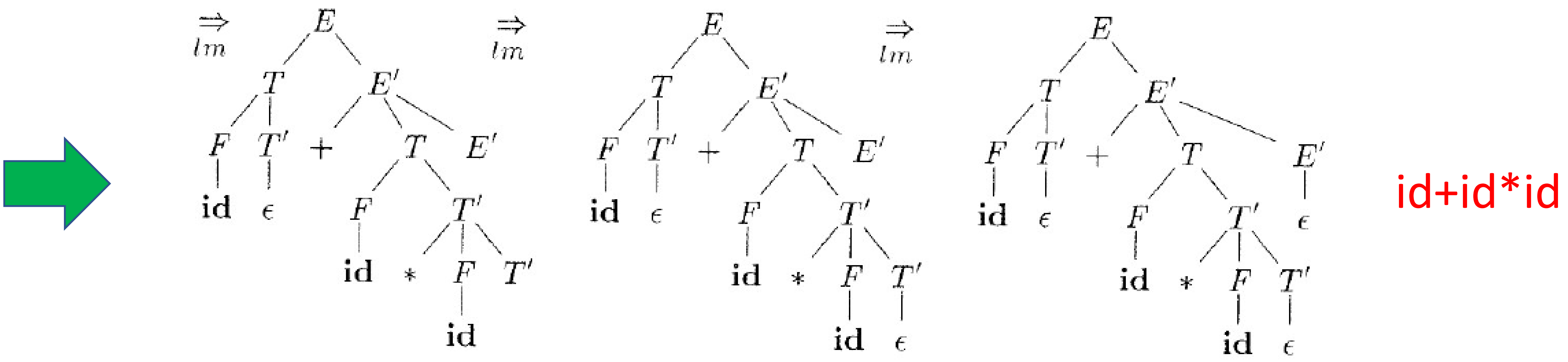
id+TE’



id+id*FT’

From parsing table to parse tree

- Parse tree iteration for input id+id*id in the previous grammar (top-down):



- Success!

Condensed parse trees

- As you can see, parse trees for even simple inputs look a bit complex
- However, the syntactic structure of an expression can be determined without writing all terminals and non-terminals
- Parse trees can be condensed
  - Simpler look
  - Easier to read and draw
  - Sufficient for illustration purposes

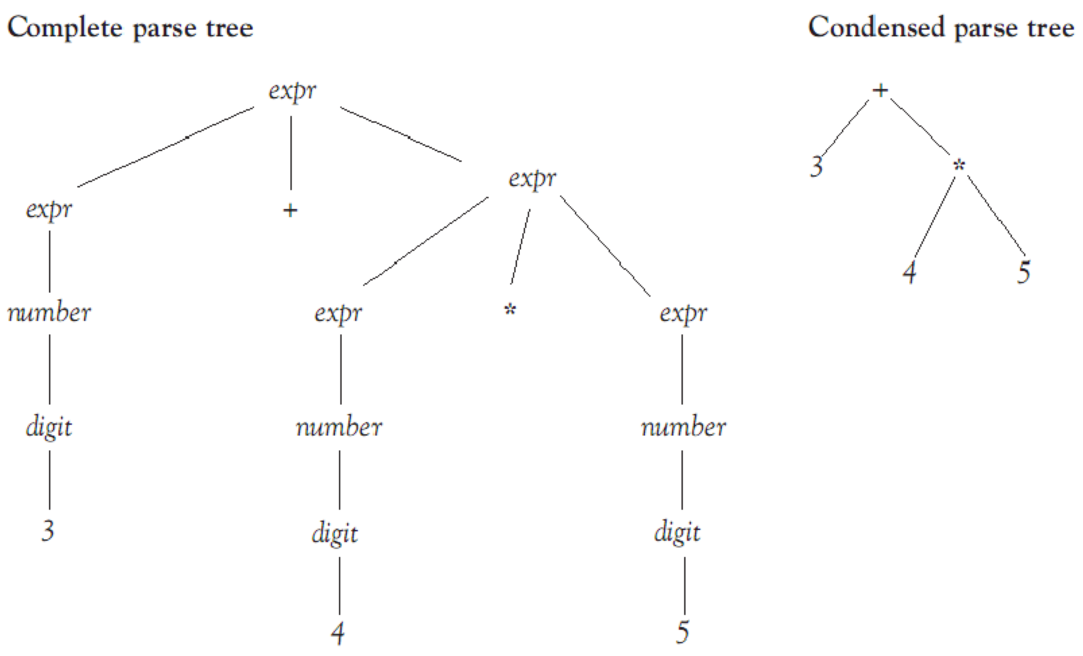
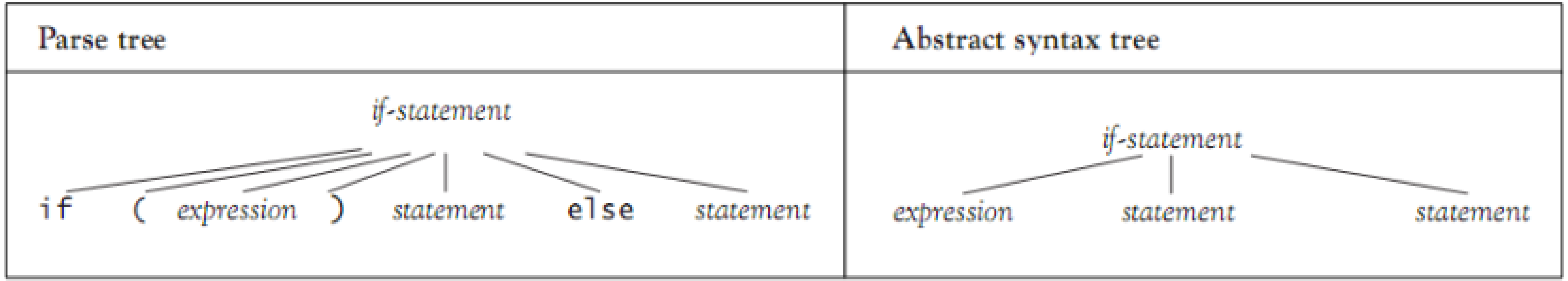


Figure 6.10: Condensing parse tree for 3 + 4 * 5

Abstract syntax tree (AST)

- Abstract syntax trees (ASTs) are a special case of condensed parse trees
- Require understanding of the structure (so, not just the syntax, but also semantics)
- Redundant terminals can be removed

if-statement → `if` `(` *expression* `)` *statement* `else` *statement*



Code generation

- After parse tree (or AST) of the program is formed by the parser, comes the 3rd phase – generating the actual machine language program:
  - Memory allocation
  - Formulation of machine language commands
- It is common that programs are not translated straight from high-level language to machine language, but the program is first translated to some mid-level language
  - Machine language programs are computer-specific
  - Mid-level language program is easier to translate further to several different computers
  - Beneficial, if the same program will be launched to different platforms (for example, 32-bit & 64-bit versions – or a desktop version & tablet/cellphone version)

Memory allocation & symbol table

- Memory will be allocated to identifiers according to their type & value
  - Floating-point numbers, integers etc. require different amounts of memory
- Memory addresses of identifier values will be decided
  - Addresses can be physical or virtual
- These will be written in the symbol table that was generated during lexing phase

Identifier	Type	Size	Address
X	Integer	4	245
Y	Integer	4	249
Sig	Sign	1	253
Rem	Float	4	254

Formulation of machine language commands

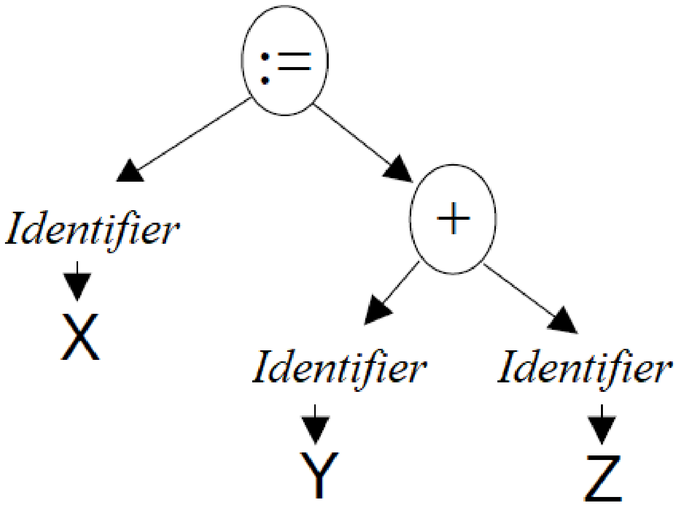
- From parse tree to symbolic machine language

Input & output:

```
x := y+z

LOAD y
ADD z
STORE x
```

Parse module:



Code generation module:

```
MODULE set(x, y, o, z)

   Print "LOAD y"

   CASE o of
       '+'  : print "ADD z"
       ´-´  : print "SUBTRACT z"
       '*'  : print "MULTIPLY z"
       '/'  : print "DIVIDE z"
   ENDCASE

   Print "STORE x"

ENDMODULE
```

Optimization

- Mechanically generated code can usually be improved
- Different compilers vary in their efficiency to optimize the code they generate
  - Identifying and removing unnecessary parts of the programs
  - Use the operations which are most efficient to perform on the destination computer
- Example:

```
X=Y+Z;
B=X*A;
```

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

No need to store and then again load X, because X is already in ACC after 2nd line!

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

Possible pitfall: now no value is stored in X, so if we later would need it for something else, this step shouldn't be done!

Thank you for listening!