Principle of Programming Languages

Hien D. Nguyen, ph.D

University of Information Technology

Lecture slides prepared by:

Quan Thanh Tho (qttho@hcmut.edu.vn)

Hien D. Nguyen

- **Personal Information**
  - Email: loveyou238us@gmail.com
  - Affiliation: University of Information Technology, VNU-HCM
- **Working**
  - 2008 - now: Lecturer at Computer Science Faculty, UIT
  - March. 2017 – Sept. 2017: Researcher at Inference and Learning lab., National Institute of Informatics (NII), Japan
  - Jan. 2018 – Feb. 2018: Visiting researcher at Artificial Intelligence lab., Wakayama University, Japan
- **Research areas**
  - Knowledge representation, automated reasoning, intelligent problem solver, expert system

- Tel: 0918735299
- Email: loveyou238us@gmail.com
- Facebook:
https://www.facebook.com/loveyou.nguyen
- Website: https://mathsolve.edu.vn/home.html
- YouTube:
https://www.youtube.com/channel/UC033mzOlYLlZifCigHTkszg?view_as=subscriber

Schedule

- Session 1: Introduction – Warm up
- Session 2: Lexicon Analysis
- Session 3: Grammar – Parse Tree
- Session 4: Grammar Analysis (Precedence – Association)
- Session 5: OOP – Polymorphism
- Session 6: Design Pattern – Adapter Pattern
- Session 7: Exercises for Revision
- Session 8: Visitor Pattern

→ **Midterm**

Schedule (cont.)

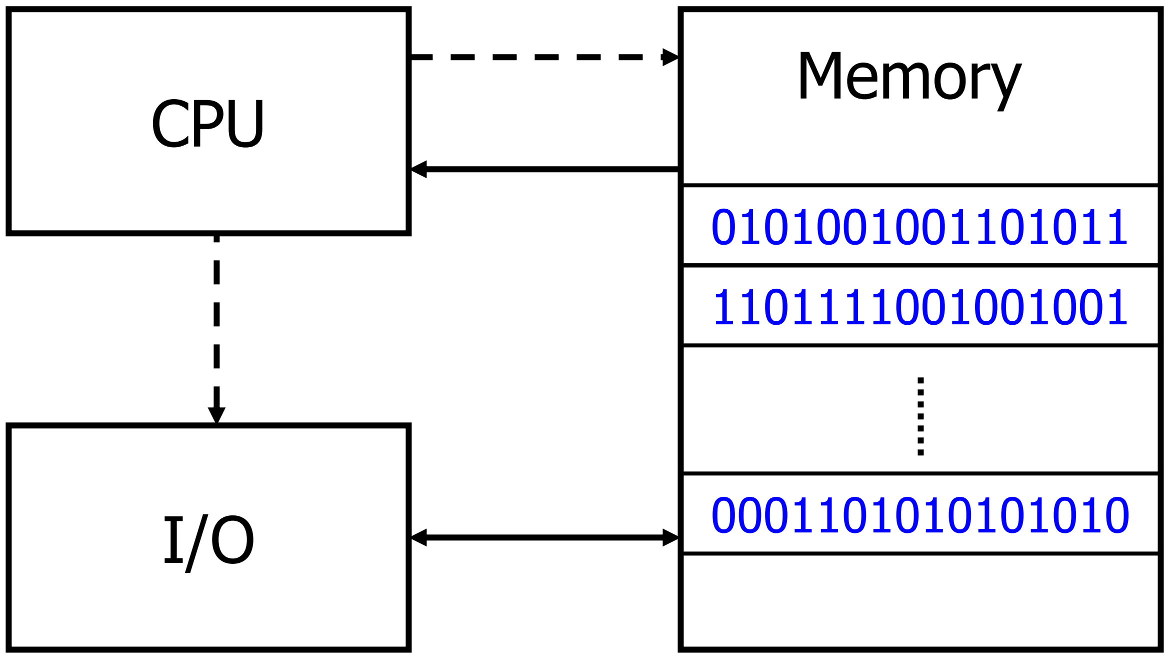
- Session 9: AST tree
- Session 10: Expression evaluation
- Session 11: Functional Programming - Introduction
- Session 12: Functional Programming – Higher Order Function
- Session 13: Functional Programming – Exercises & Revision
- Session 14: Parametter Passing
- Session 15: Revision

→ **Final test**

Contents

- Evolution and classification
- Formal syntax and semantics
- Compilation and interpretation

Machine Language



Machine Language

Instruction:

Operation Code	Operands

10110011010010010011010110110001

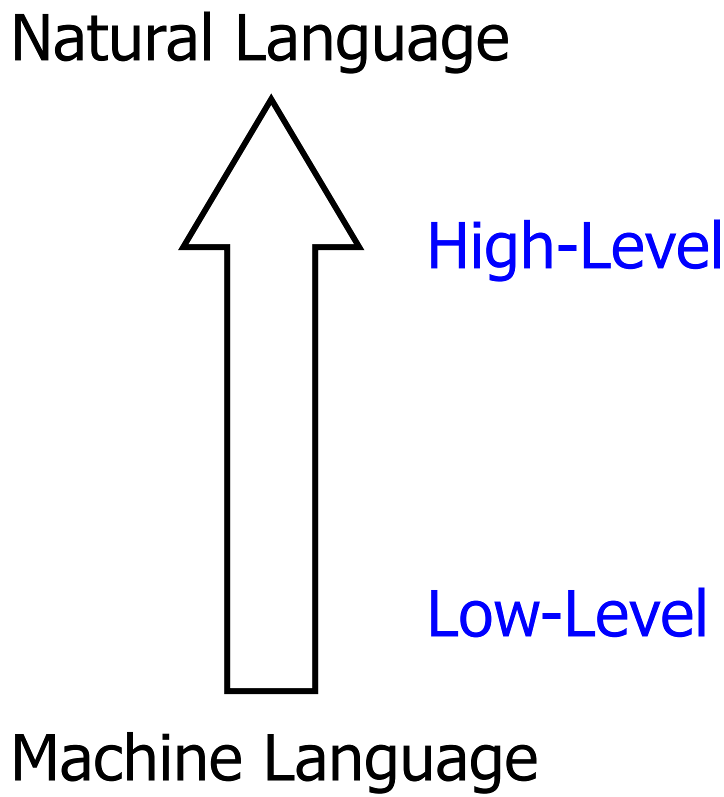
Assembly Language

A := B + C

if A = 0 then *body*

```
    MOV r0, B   ; move B into register r0
    ADD r0, C   ; add
    MOV A, r0   ; store
    BNE L1      ; branch if result not equal 0
    body
L1:
```

Language Levels



What Makes a Good Language?

- Clarity, simplicity, unity of language concepts
- Clarity of program syntax
- Naturalness for the application
- Support for abstraction
- Ease of program verification

What Makes a Good Language?

- Programming environment
- Portability of programs
- Cost of use
  - program execution
  - program translation
  - program creation, testing, use
  - program maintenance

Language Classification

- Imperative
  - von Neumann — Fortran, Pascal, Basic, C
  - object-oriented — Smalltalk, Eiffel, C++, Java
- Declarative
  - functional — Lisp, ML, Haskell
  - dataflow — Id, Val
  - logic — Prolog, VisiCalc

Von Neumann Languages

- Most familiar and successful
- Imperative statements
- Modification of variables

Fortran, Pascal, Basic, C, Python …

Object-Oriented Languages

- Imperative statements
- Message passing among objects

Smalltalk, Eiffel, C++, Java, Python

Functional Languages

- Recursive definition of functions (lambda calculus)
- Expressions of function composition

Lisp, ML, Haskell, Python

Logic Languages

- Logical facts and rules
  (predicate logic)
- Computation as theorem proving

Prolog, VisiCalc

Contents

- Evolution and classification
- Formal syntax and semantics
- Compilation and interpretation

Formal Syntax and Semantics

- Computer languages must be precise
- Both their form (syntax) and meaning (semantics) must be specified without ambiguity
- Both programmers and computers can tell what a program is supposed to do

Formal Syntax

- Abstract syntax
- Context-free grammars
- Backus-Naur formalism (BNF)
- Syntax diagrams
- Derivations and parse trees

Context-Free Grammars

- Start symbol
- Non-terminals
- Terminals
- Productions $A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$

(Noam Chomsky, 1959)

Example: Unsigned Integers

6 2 5 7 3

<digit> <unsigned_integer>

Example: Unsigned Integers

- Start symbol <unsigned_integer>
- Non-terminals <unsigned_integer>, <digit>
- Terminals 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- Productions

  <unsigned_integer> → <digit> |
  <digit> <unsigned_integer>

Backus-Naur Formalism

<unsigned_integer> ::= <digit> |
<digit> <unsigned_integer>

(John Backus, 1960)

Example: Expressions

<factor>

12 * 3 + 4

<term>

Example: Expressions

- Start symbol <expression>
- Non-terminals <expression>, <term>, <factor>, <unsigned_integer>, <term_op>, <factor_op>
- Terminals 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, –, *, /

Example: Expressions

- Productions:

<expression> → <term> |
<expression> <term_op> <term>

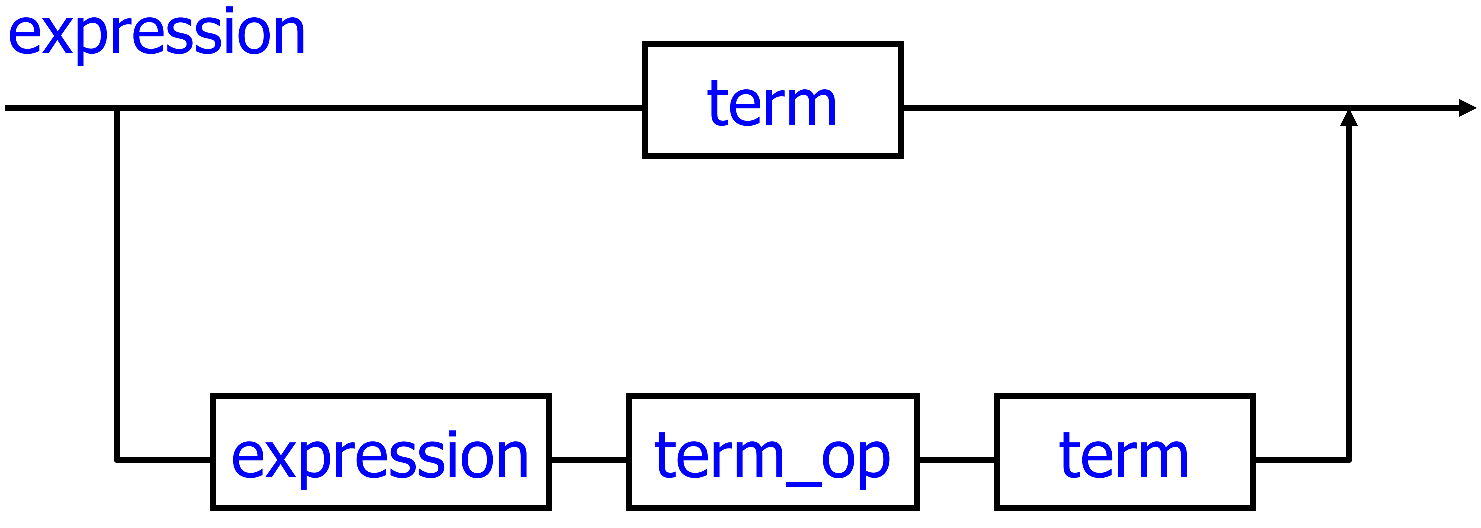
<term> → <factor> | <term> <factor_op> <factor>

<factor> → <unsigned_integer> | (<expression>)

<term_op> → + | −

<factor_op> → ∗ | /

Syntax Diagrams



Derivations

<expression>

⇒ <expression> <term_op> <term>

⇒ <term> + <factor>

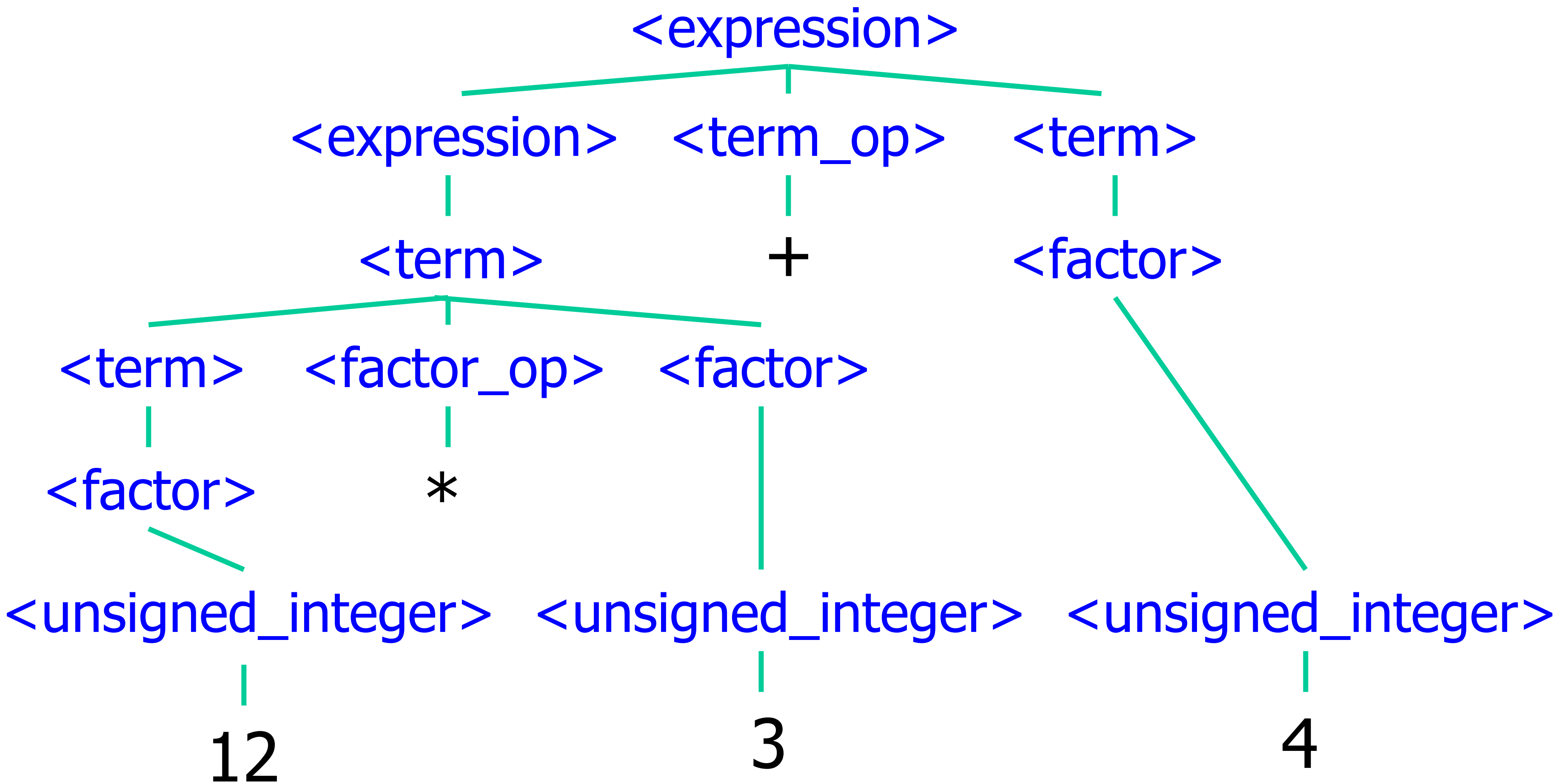
⇒ <term> <factor_op> <factor> + <unsigned_integer>

⇒ <factor> ∗ <unsigned_integer> + 4

⇒ <unsigned_integer> ∗ 3 + 4

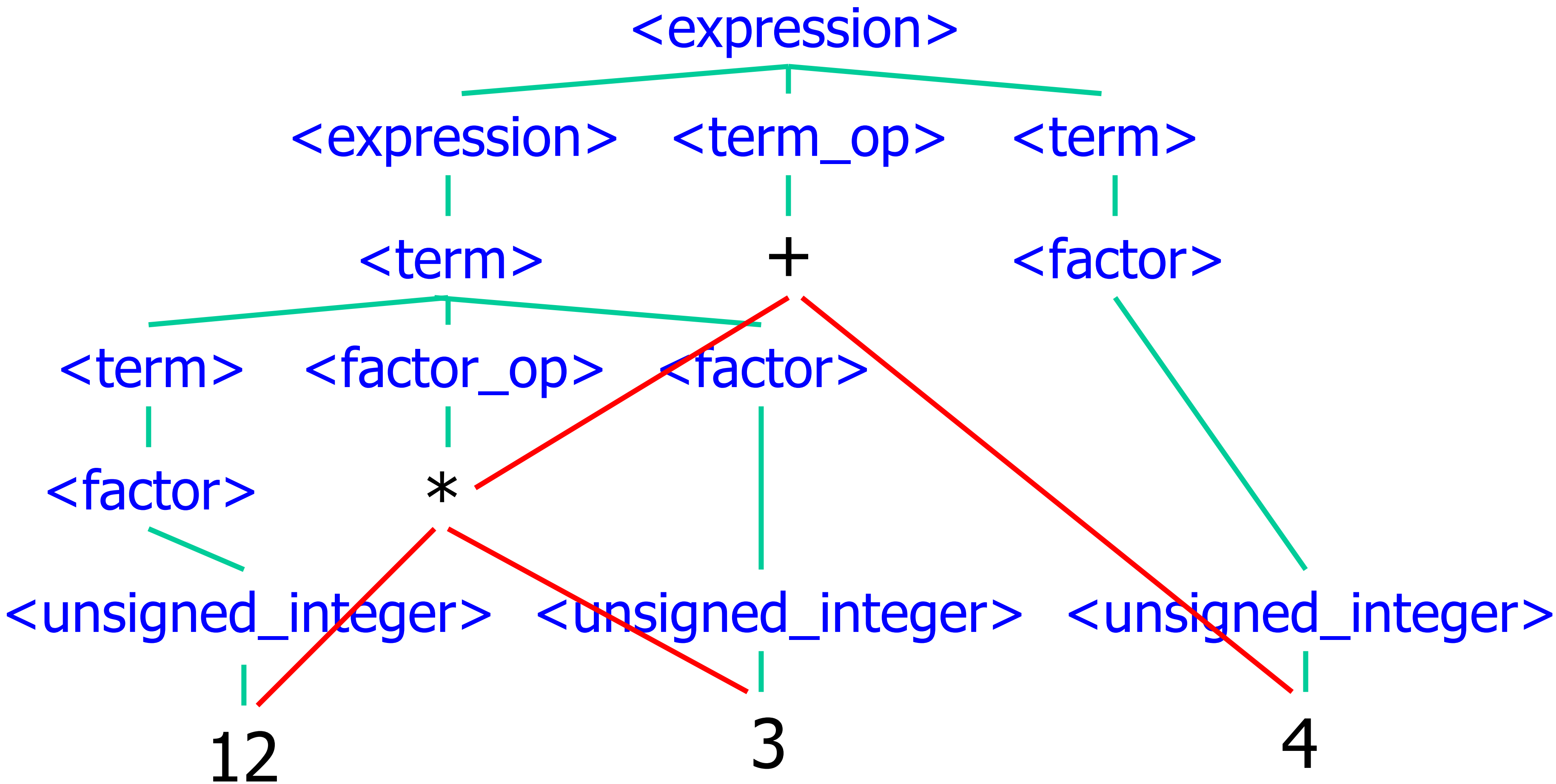
⇒ 12 ∗ 3 + 4

Parse Trees



<expression>
<expression>
<term_op>
<term>
<term>
+
<factor>
<term>
<factor_op>
<factor>
<factor>
*
<unsigned_integer>
<unsigned_integer>
<unsigned_integer>
12
3
4

Parse Trees



<expression>
<expression>
<term_op>
<term>
<term>
+
<factor>
<term>
<factor_op>
<factor>
<factor>
*
<unsigned_integer>
<unsigned_integer>
<unsigned_integer>
12
3
4

Expressions

- Control mechanism
- Syntax
- Execution-time representation
- Evaluation

Control Mechanism

- Functional composition:

$$(A + B) * (C - A)$$

$$* \; (+ \; (A, B), - \; (C, A))$$

Syntax

- Infix:

$$A * B + C$$

  - binary operations only
  - computation order ambiguity

Syntax

- Prefix:
  - ordinary

    * (+ (A, B), - (C, A))

  - Cambridge Polish

    (* (+ A B) (- C A))

  - Polish

    * + A B - C A

Syntax

- Prefix:
    - different numbers of operands
    - ordinary/ Cambridge Polish: cumbersome with parentheses
    - Polish: number of operands known in advance

Syntax

- Postfix:
  - ordinary

    ((A, B) +, (C, A) -) *

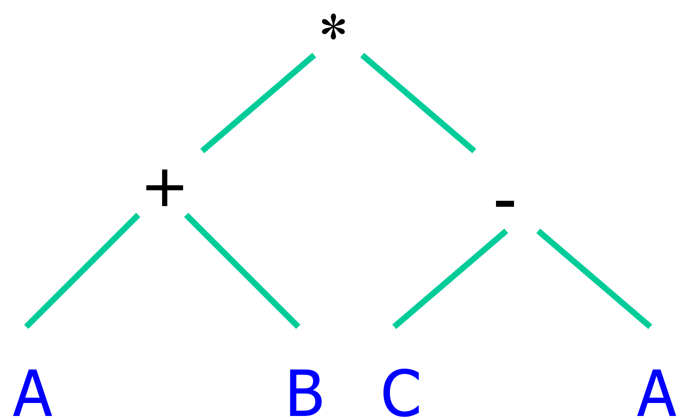
  - Polish

    A B + C A - *

  - suitable execution-time representation

Evaluation

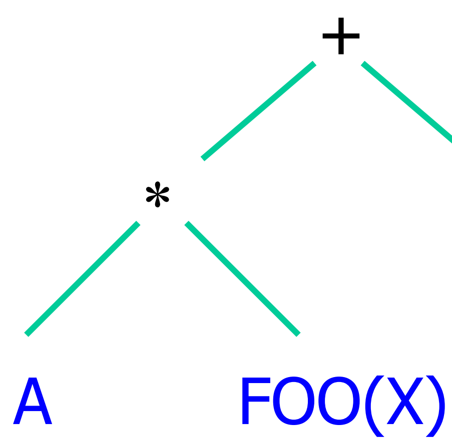
- No simple uniform evaluation rule is satisfactory:



Evaluation

- Side effects:

$$A * FOO(X) + A$$



Evaluation

- Side effects:

$$A * B * C = 10^{20} * 10^{-20} * 10^{-20}$$

$$(A * B) * C = 1 * 10^{-20} = 10^{-20}$$

$$A * (B * C) = 10^{20} * 0 = 0$$

Evaluation

- Short-circuit Boolean expressions:

if (A = 0) or (B/A > C) then ...

Evaluation

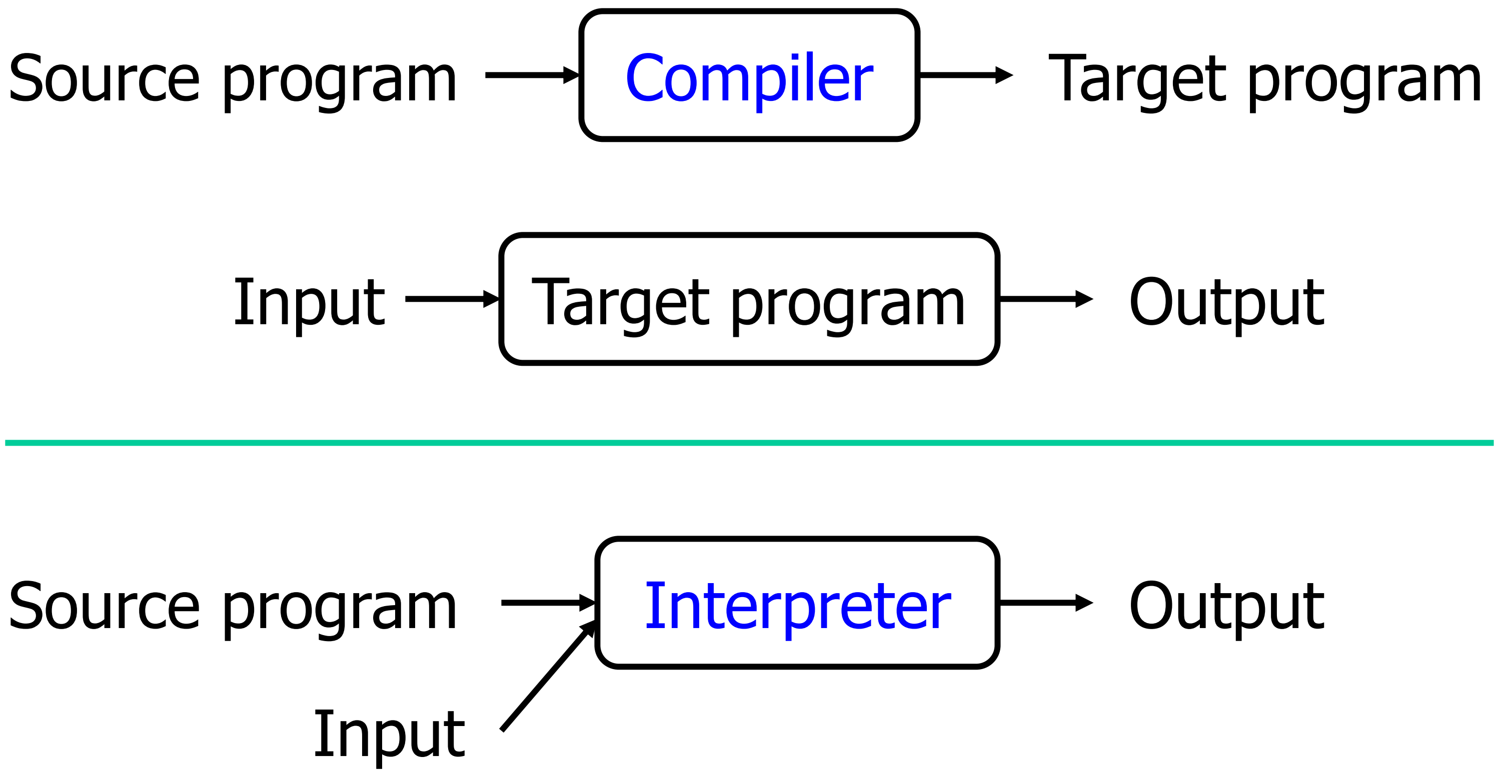
- Short-circuit Boolean expressions:

if (A = 0) or else (B/A > C) then ...

Contents

- Evolution and classification
- Formal syntax and semantics
- Compilation and interpretation

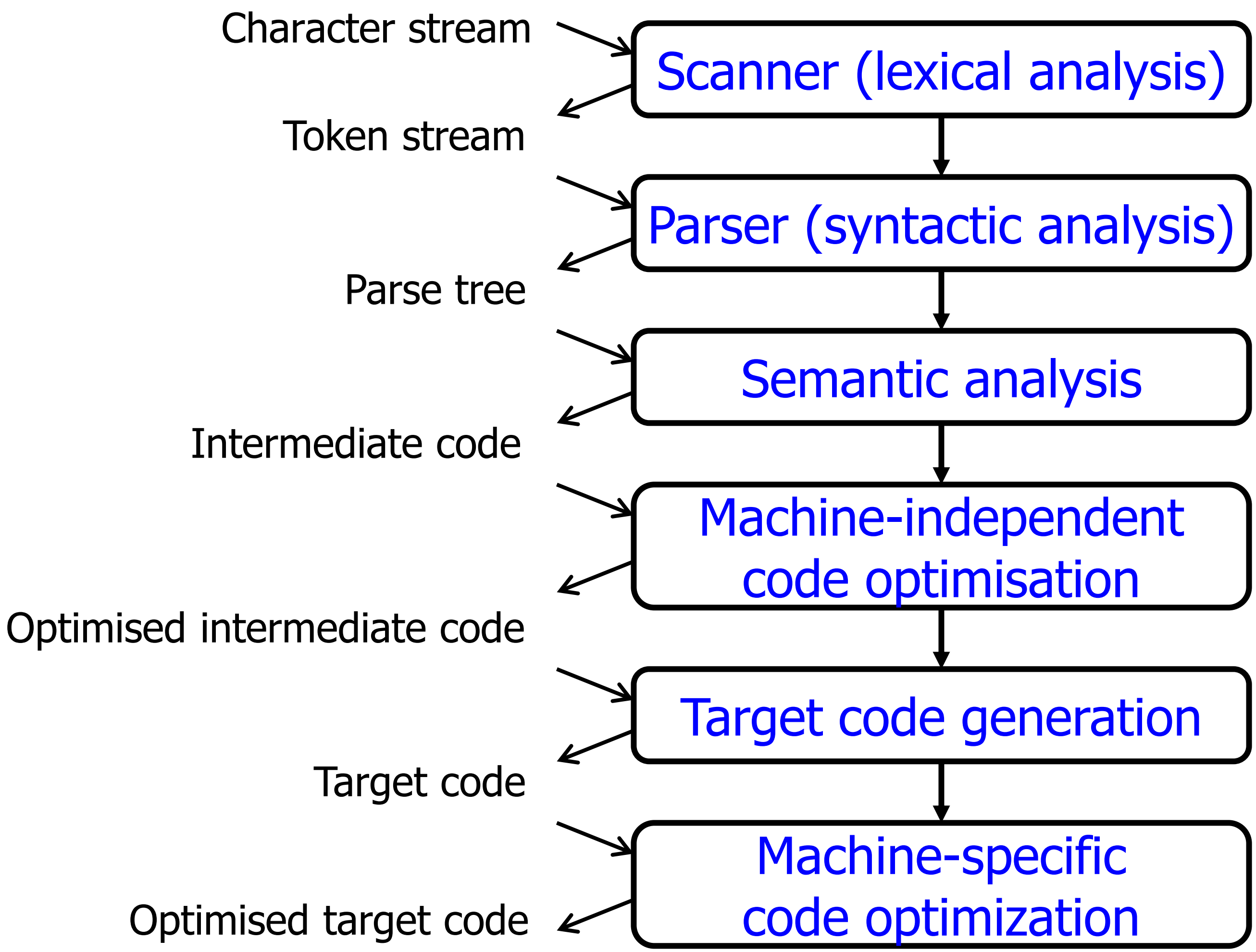
Compilation and Interpretation



Compilation and Interpretation

- Interpreter: better flexibility and diagnostics
- Compiler: better performance

Phases of Compilation



Phases of Compilation

$c := a + b * 7$

$\downarrow$

Scanner (lexical analysis)

$\downarrow$

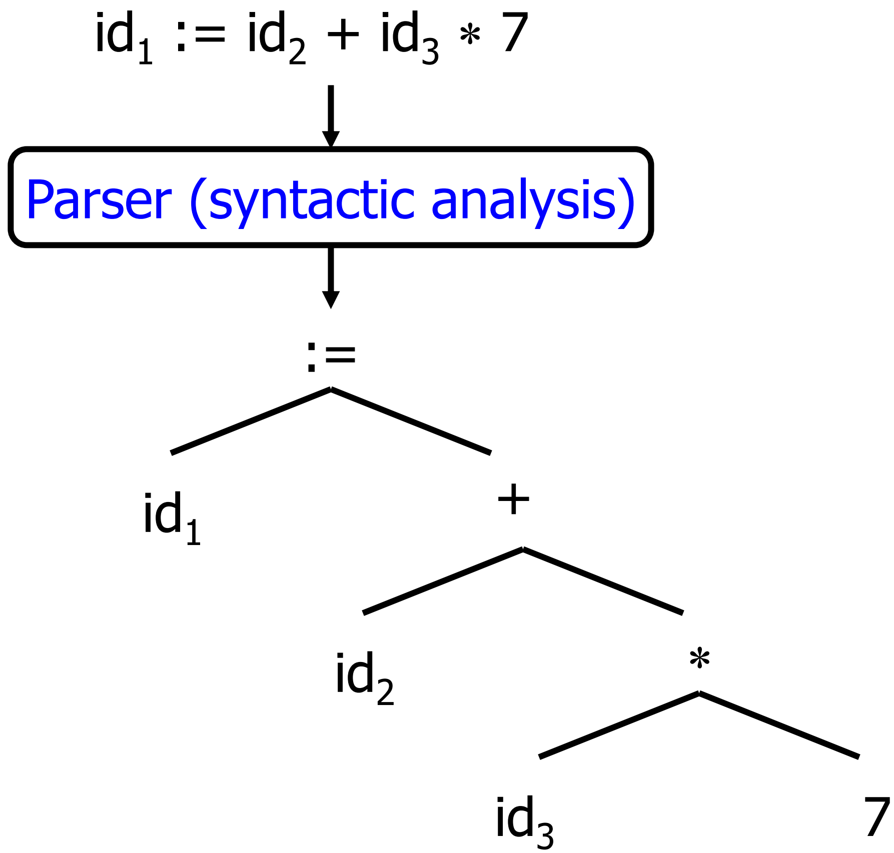
$id_1 := id_2 + id_3 * 7$

$\downarrow$

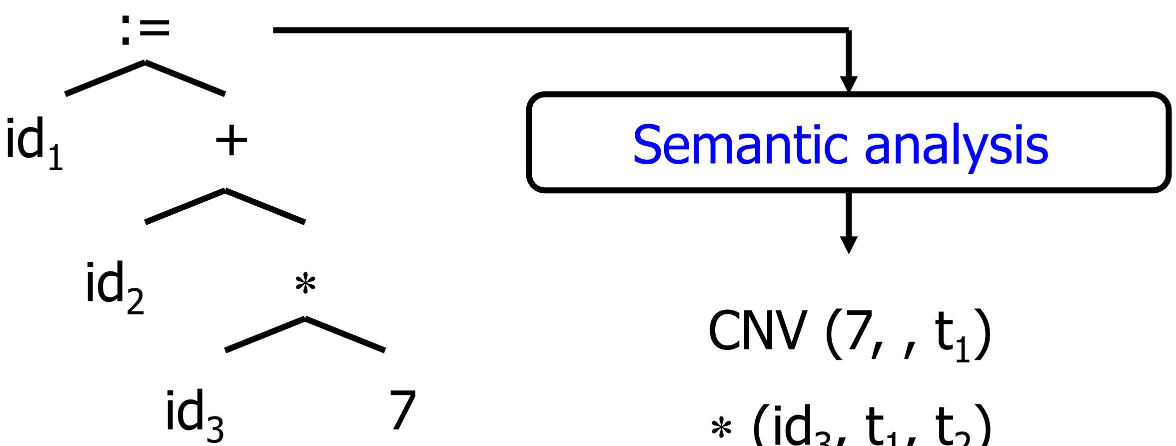
Parser (syntactic analysis)

$\downarrow$

Phases of Compilation



Phases of Compilation



CNV (7, , t_1)

∗ (id_3, t_1, t_2)

+ (id_2, t_2, t_3)

ASS (t_3, , id_1)

Phases of Compilation

CNV (7, , t_1)

$*$ (id_3, t_1, t_2)

\+ (id_2, t_2, t_3)

ASS (t_3, , id_1)

↓

Machine-independent code optimisation

↓

$*$ (id_3, 7.0, t_1)

\+ (id_2, t_1, id_1)

Phases of Compilation

$* \ (id_3, 7.0, t_1)$

$+ \ (id_2, t_1, id_1)$

↓

Target code generation

↓

MOV reg, id_3

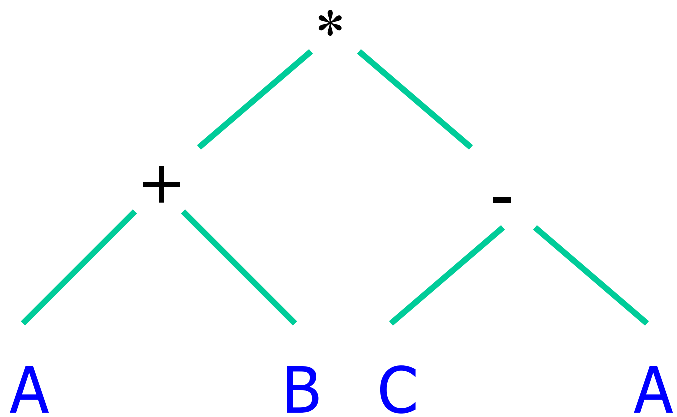
MUL reg, 7.0

ADD reg, id_2

MOV id_1, reg

Execution-Time Representation

- Interpretation:
    - tree structures

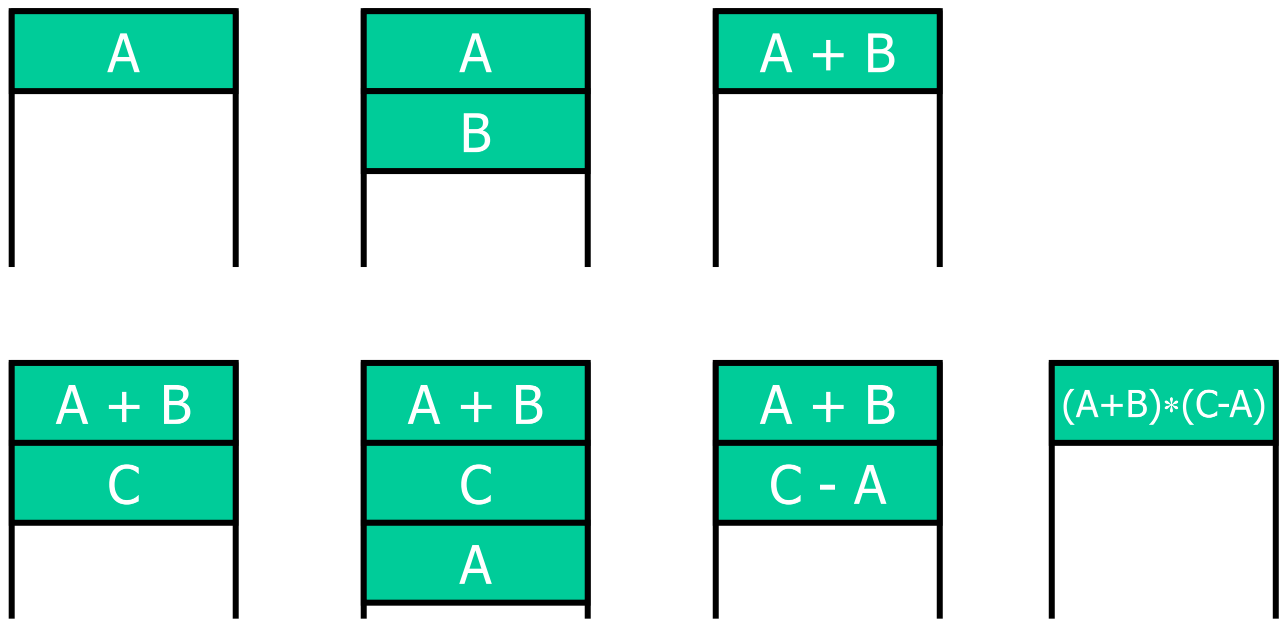


    - prefix or postfix

Execution-Time Representation

- Compilation: machine code sequences

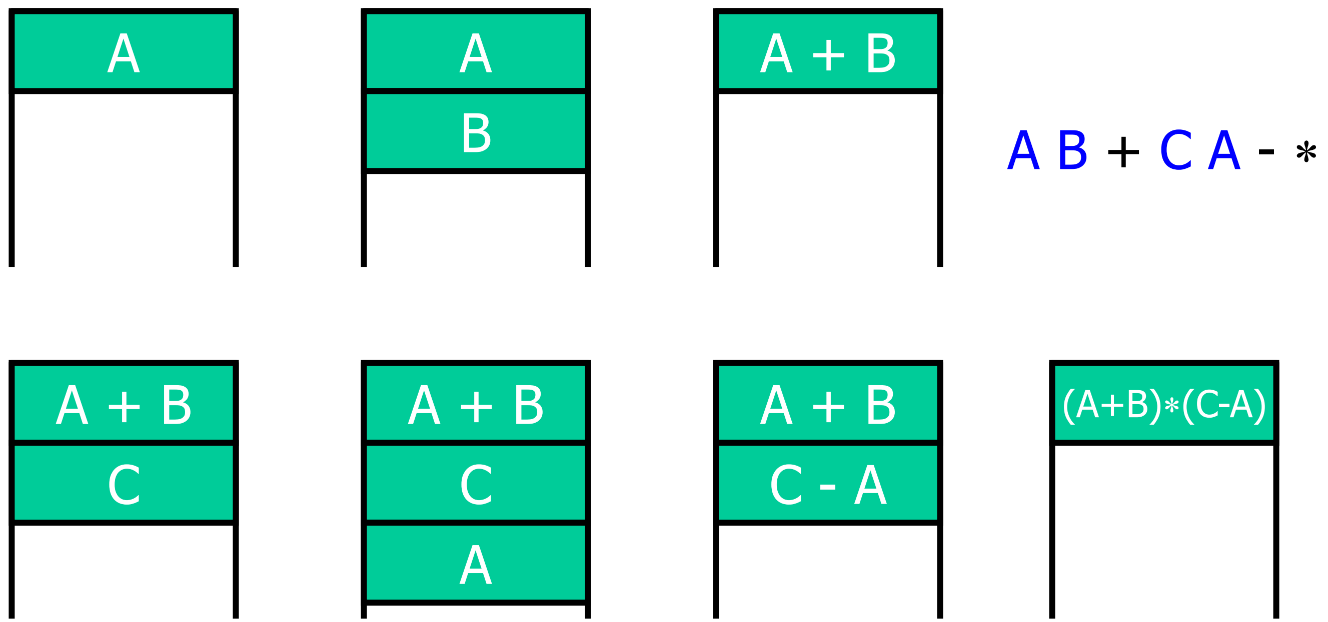
PUSH A
PUSH B
ADD
PUSH C
PUSH A
SUB
MUL



Execution-Time Representation

- Compilation: machine code sequences

PUSH A
PUSH B
ADD
PUSH C
PUSH A
SUB
MUL



A B + C A - *