Principles of Programming Languages
Lecture 4: Abstract syntax and semantics of expressions

Andrei Arusoaie[1]

[1]Department of Computer Science

October 19, 2020

Outline

Alphabet. Lexical analysis. Parsing.

Parse Trees

Abstract syntax trees

Sentences in a programming language

Which phrases are correct?

- `int x; x = x + 2`
- `int x; x = x + 2;`
- `if (a > 0) then x = 1; else x = -1;`
- `(a > 0) ? x = 1 : x = -1;`

Sentences in a programming language

Which phrases are correct?

- `int x; x = x + 2`
- `int x; x = x + 2;`
- `if (a > 0) then x = 1; else x = -1;`
- `(a > 0) ? x = 1 : x = -1;`

Sentences in a programming language

Which phrases are correct?

- `int x; x = x + 2`
- `int x; x = x + 2;`
- `if (a > 0) then x = 1; else x = -1;`
- `(a > 0) ? x = 1 : x = -1;`

Overview: alphabet, lexical analysis, syntax.

- *Alphabet* : set of (allowed) symbols
- *Lexical analysis*: identify the sequence of symbols constituting the *words* (or *tokens*)
  - *Lexical rules*
- *Syntax*: describes which sequences of words constitute "legal" phrases
  - *Grammars*

Overview: alphabet, lexical analysis, syntax.

- *Alphabet* : set of (allowed) symbols
- *Lexical analysis*: identify the sequence of symbols constituting the *words* (or *tokens*)
  - *Lexical rules*
- *Syntax*: describes which sequences of words constitute "legal" phrases
  - *Grammars*

Overview: alphabet, lexical analysis, syntax.

- *Alphabet* : set of (allowed) symbols
- *Lexical analysis*: identify the sequence of symbols constituting the *words* (or *tokens*)
  - *Lexical rules*
- *Syntax*: describes which sequences of words constitute “legal” phrases
  - *Grammars*

Alphabet

The Alphabet of C from the Standard has 96 symbols:

- `a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,z`
- `A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W,X,Y,Z`
- `0,1,2,3,4,5,6,7,8,9`
- `! " # % & ' ( ) * + , - . /`
- `: ; < = > ? [ \ ] ^ _ { | } ~`
- *Separators*: space, horizontal and vertical tab, form feed, newline

Lexical analysis

Problem: Given a sequence of characters, find the pieces with assigned meaning from that sequence: words or *tokens*

Example:

- Input: `if (a > 0) then x = 1; else x = -1;`
- Output: `if`, `(`, `a`, `>`, `0`, `)`, `then`, `x`, `=`, `1`, `;`, `else`, `x`, `=`, `-1`, `;`
- *Tokens = pieces with assigned/identified meaning*

Lexical analyzer (lexer) = a program that implements an algorithm that solves the problem above

Lexical analysis

Problem: Given a sequence of characters, find the pieces with assigned meaning from that sequence: words or *tokens*

Example:

- Input: `if (a > 0) then x = 1; else x = -1;`
- Output: `if`, `(`, `a`, `>`, `0`, `)`, `then`, `x`, `=`, `1`, `;`, `else`, `x`, `=`, `-1`, `;`
- *Tokens = pieces with assigned/identified meaning*

Lexical analyzer (lexer) = a program that implements an algorithm that solves the problem above

Lexical analysis

Problem: Given a sequence of characters, find the pieces with assigned meaning from that sequence: words or *tokens*

Example:

- Input: `if (a > 0) then x = 1; else x = -1;`
- Output: `if`, `(`, `a`, `>`, `0`, `)`, `then`, `x`, `=`, `1`, `;`, `else`, `x`, `=`, `-1`, `;`
- *Tokens = pieces with assigned/identified meaning*

Lexical analyzer (lexer) = a program that implements an algorithm that solves the problem above

Lexical analysis

Problem: Given a sequence of characters, find the pieces with assigned meaning from that sequence: words or *tokens*

Example:

- Input: `if (a > 0) then x = 1; else x = -1;`
- Output: `if`, `(`, `a`, `>`, `0`, `)`, `then`, `x`, `=`, `1`, `;`, `else`, `x`, `=`, `-1`, `;`
- *Tokens = pieces with assigned/identified meaning*

Lexical analyzer (lexer) = a program that implements an algorithm that solves the problem above

Example I - Lexical rules

- Integers: $6, 0, -2, +3$
- The *alphabet* $A = \{+, -\} \cup \mathbb{N}$
- *Lexical rules*: used to describe atomic language constructions: numbers, identifiers, ...
- *Lexical rules* are expressed using *regular grammars* (see LFAC course)
- Regular expressions, a.k.a regex
  - Regex for integers: `[\+-]?\d+`

Example I - Lexical rules

- Integers: $6, 0, -2, +3$
- The *alphabet* $A = \{+, -\} \cup \mathbb{N}$
- *Lexical rules*: used to describe atomic language constructions: numbers, identifiers, ...
- *Lexical rules* are expressed using *regular grammars* (see LFAC course)
- Regular expressions, a.k.a regex
  - Regex for integers: `[\+-]?\d+`

Example I - Lexical rules

- Integers: $6, 0, -2, +3$
- The *alphabet* $A = \{+, -\} \cup \mathbb{N}$
- *Lexical rules*: used to describe atomic language constructions: numbers, identifiers, ...
- *Lexical rules* are expressed using *regular grammars* (see LFAC course)
- Regular expressions, a.k.a regex
  - Regex for integers: `[\+-]?\d+`

Example I - Lexical rules

- Integers: $6, 0, -2, +3$
- The *alphabet* $A = \{+, -\} \cup \mathbb{N}$
- *Lexical rules*: used to describe atomic language constructions: numbers, identifiers, ...
- *Lexical rules* are expressed using *regular grammars* (see LFAC course)
- Regular expressions, a.k.a regex
  - Regex for integers: `[\+-]?\d+`

Example I - Lexical rules

- Integers: $6, 0, -2, +3$
- The *alphabet* $A = \{+, -\} \cup \mathbb{N}$
- *Lexical rules*: used to describe atomic language constructions: numbers, identifiers, ...
- *Lexical rules* are expressed using *regular grammars* (see LFAC course)
- Regular expressions, a.k.a regex
  - Regex for integers: `[\+-]?\d+`

Parsing

Problem: how to combine the tokens in (valid) sentences?

- Answer: we define the *grammar* of the language
- Noam Chomsky: *generative grammar*
- Grammars allow us to transform a program given as an sequence of characters into a *syntax tree*
- Parser = program which attempts to do this transformation
- Only valid programs can be parsed!

Parsing

Problem: how to combine the tokens in (valid) sentences?

- Answer: we define the *grammar* of the language
- Noam Chomsky: *generative grammar*
- Grammars allow us to transform a program given as an sequence of characters into a *syntax tree*
- Parser = program which attempts to do this transformation
- Only valid programs can be parsed!

Parsing

Problem: how to combine the tokens in (valid) sentences?

- Answer: we define the *grammar* of the language
- Noam Chomsky: *generative grammar*
- Grammars allow us to transform a program given as an sequence of characters into a *syntax tree*
- Parser = program which attempts to do this transformation
- Only valid programs can be parsed!

Parsing

Problem: how to combine the tokens in (valid) sentences?

- Answer: we define the *grammar* of the language
- Noam Chomsky: *generative grammar*
- Grammars allow us to transform a program given as an sequence of characters into a *syntax tree*
- Parser = program which attempts to do this transformation
- Only valid programs can be parsed!

Parsing

Problem: how to combine the tokens in (valid) sentences?

- Answer: we define the *grammar* of the language
- Noam Chomsky: *generative grammar*
- Grammars allow us to transform a program given as an sequence of characters into a *syntax tree*
- Parser = program which attempts to do this transformation
- Only valid programs can be parsed!

Example II - Grammar

- Language of palindromic strings using symbols a and b
- The *alphabet* $A = \{a, b\}$
- Can we describe palindromes using regex?

Example II - Grammar

- Language of palindromic strings using symbols a and b
- The *alphabet* $A = \{a, b\}$
- Can we describe palindromes using regex?

Example II - Grammar

- Language of palindromic strings using symbols *a* and *b*
- The *alphabet* $A = \{a, b\}$
- Can we describe palindromes using regex?

Example II - Grammar

- How do we “formally” describe palindromic strings?
  - Note that there is a simple recursion of a palindromic string
  - Base: a and b are palindromic strings
  - Recursion: if s is a palindromic string then so are asa and bsb
- Examples: “aba”, “aabaa”, “bab”, etc
- *Problem?* yes: “aa”, “abba“.
- Fix: add the empty string to base, hereafter denoted by ϵ

Example II - Grammar

- How do we “formally” describe palindromic strings?
  - Note that there is a simple recursion of a palindromic string
  - Base: a and b are palindromic strings
  - Recursion: if s is a palindromic string then so are asa and bsb
- Examples: “aba”, “aabaa”, “bab”, etc
- *Problem?* yes: “aa”, “abba“.
- Fix: add the empty string to base, hereafter denoted by ϵ

Example II - Grammar

- How do we "formally" describe palindromic strings?
  - Note that there is a simple recursion of a palindromic string
  - Base: a and b are palindromic strings
  - Recursion: if s is a palindromic string then so are asa and bsb
- Examples: "aba", "aabaa", "bab", etc
- *Problem?* yes: "aa", "abba".
- Fix: add the empty string to base, hereafter denoted by ϵ

Example II - Grammar

- How do we “formally” describe palindromic strings?
  - Note that there is a simple recursion of a palindromic string
  - Base: a and b are palindromic strings
  - Recursion: if s is a palindromic string then so are asa and bsb
- Examples: “aba”, “aabaa”, “bab”, etc
- *Problem?* yes: “aa”, “abba“.
- Fix: add the empty string to base, hereafter denoted by ϵ

Example II - Grammar

- How do we "formally" describe palindromic strings?
  - Note that there is a simple recursion of a palindromic string
  - Base: *a* and *b* are palindromic strings
  - Recursion: if *s* is a palindromic string then so are *asa* and *bsb*
- Examples: "aba", "aabaa", "bab", etc
- *Problem?* yes: "aa", "abba".
- Fix: add the empty string to base, hereafter denoted by ϵ

Example II - Grammar

- How do we “formally” describe palindromic strings?
  - Note that there is a simple recursion of a palindromic string
  - Base: a and b are palindromic strings
  - Recursion: if s is a palindromic string then so are asa and bsb
- Examples: “aba”, “aabaa”, “bab”, etc
- *Problem?* yes: “aa”, “abba“.
- Fix: add the empty string to base, hereafter denoted by ϵ

Example II - Grammar

- How do we "formally" describe palindromic strings?
  - Note that there is a simple recursion of a palindromic string
  - Base: *a* and *b* are palindromic strings
  - Recursion: if *s* is a palindromic string then so are *asa* and *bsb*
- Examples: "aba", "aabaa", "bab", etc
- *Problem?* yes: "aa", "abba".
- Fix: add the empty string to base, hereafter denoted by ϵ

Example II - Grammar

- Base case:
  - $P \rightarrow \epsilon$
  - $P \rightarrow a$
  - $P \rightarrow b$
- Recursion:
  - $P \rightarrow aPa$
  - $P \rightarrow bPb$
- Context-free grammar (you study this in your compiler course!)

Example II - Grammar

- Base case:
  - $P \rightarrow \epsilon$
  - $P \rightarrow a$
  - $P \rightarrow b$
- Recursion:
  - $P \rightarrow aPa$
  - $P \rightarrow bPb$
- Context-free grammar (you study this in your compiler course!)

Example II - Grammar

- Base case:
  - $P \rightarrow \epsilon$
  - $P \rightarrow a$
  - $P \rightarrow b$
- Recursion:
  - $P \rightarrow aPa$
  - $P \rightarrow bPb$
- Context-free grammar (you study this in your compiler course!)

Example II - Grammar

- Base case:
  - $P \rightarrow \epsilon$
  - $P \rightarrow a$
  - $P \rightarrow b$
- Recursion:
  - $P \rightarrow aPa$
  - $P \rightarrow bPb$
- Context-free grammar (you study this in your compiler course!)

Example II - Grammar

- Base case:
  - $P \rightarrow \epsilon$
  - $P \rightarrow a$
  - $P \rightarrow b$
- Recursion:
  - $P \rightarrow aPa$
  - $P \rightarrow bPb$
- Context-free grammar (you study this in your compiler course!)

Example II - Grammar

- Base case:
  - $P \rightarrow \epsilon$
  - $P \rightarrow a$
  - $P \rightarrow b$
- Recursion:
  - $P \rightarrow aPa$
  - $P \rightarrow bPb$
- Context-free grammar (you study this in your compiler course!)

Example II - Grammar

- Base case:
  - $P \to \epsilon$
  - $P \to a$
  - $P \to b$
- Recursion:
  - $P \to aPa$
  - $P \to bPb$
- Context-free grammar (you study this in your compiler course!)

Example II - Grammar

- Base case:
  - $P \rightarrow \epsilon$
  - $P \rightarrow a$
  - $P \rightarrow b$
- Recursion:
  - $P \rightarrow aPa$
  - $P \rightarrow bPb$
- Context-free grammar (you study this in your compiler course!)

Backus-Naur Form (BNF)

- *Meta-language* introduced by Backus and Naur to define ALGOL60
- Vocabulary:
  - Terminals : simple language strings; typically: *tokens* or symbols
  - Non-terminals: complex language constructions

Backus-Naur Form (BNF)

- *Meta-language* introduced by Backus and Naur to define ALGOL60
- Vocabulary:
  - Terminals : simple language strings; typically: *tokens* or symbols
  - Non-terminals: complex language constructions

Backus-Naur Form (BNF)

- *Meta-language* introduced by Backus and Naur to define ALGOL60
- Vocabulary:
  - Terminals : simple language strings; typically: *tokens* or symbols
  - Non-terminals: complex language constructions

BNF - example I

- Palindromic strings:

$$
\begin{array}{llll}
\mathtt{P} & \mathtt{::=} & \epsilon & (1) \\
 & \mid & \mathtt{a} & (2) \\
 & \mid & \mathtt{b} & (3) \\
 & \mid & \mathtt{a\ P\ a} & (4) \\
 & \mid & \mathtt{b\ P\ b} & (5)
\end{array}
$$

Derivations

- How to obtain a *derivation*: read the production as rewrite rules and find a finite sequence of rewrite steps
- Example: derivation for `abba`
  $\mathsf{P} \rightarrow^4 \texttt{a}\mathsf{P}\texttt{a} \rightarrow^5 \texttt{ab}\mathsf{P}\texttt{ba} \rightarrow^1 \texttt{abba}$

Parse trees

- Derivation: P $\rightarrow^4$ aPa $\rightarrow^5$ abPba $\rightarrow^1$ abba
- Parse tree:
  - contains nodes labeled with terminals, nonterminals, and ϵ

```
      P
     /|\
    / | \
   a  P  a
     /|\
    / | \
   b  P  b
      |
      |
      e
```

Parse trees

- Derivation: P $\rightarrow^{4}$ $\texttt{a}$**P**$\texttt{a}$ $\rightarrow^{5}$ $\texttt{ab}$**P**$\texttt{ba}$ $\rightarrow^{1}$ $\texttt{abba}$
- Parse tree:
  - contains nodes labeled with terminals, nonterminals, and ϵ

```
      P
     /|\
    / | \
   a  P  a
     /|\
    / | \
   b  P  b
      |
      |
      e
```

BNF - example II

- Simple expressions language:

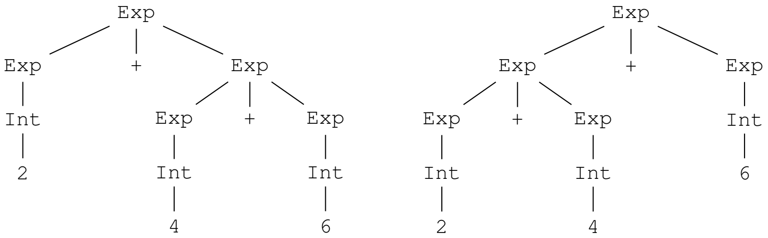
```
Int  ::=  [\+-]?[0-9]+
Exp  ::=  Int
       |  Exp "+" Exp
       |  Exp "*" Exp
       |  Exp "/" Exp
       |  "(" Exp ")"
```

Multiple parses available

Possible parse trees for `2 + 4 + 6`:

Multiple parses available

Possible parse trees for `2 + 4 + 6`:



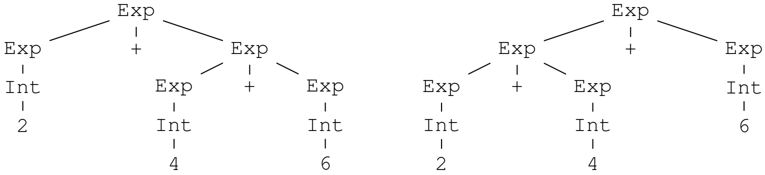
Ambiguities

Possible parse trees for `2 + 4 + 6`:

- Solutions?
  - Use the parentheses defined in the syntax: '`(`' and '`)`'
  - Encode some kind of associativity: left or right

Ambiguities

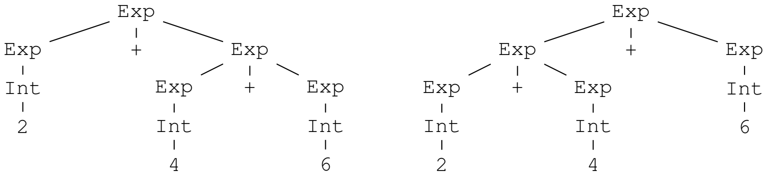
Possible parse trees for `2 + 4 + 6`:



- Solutions?
  - Use the parentheses defined in the syntax: '`(`' and '`)`'
  - Encode some kind of associativity: left or right

Ambiguities

Possible parse trees for `2 + 4 + 6`:



- Solutions?
  - Use the parentheses defined in the syntax: '`(`' and '`)`'
  - Encode some kind of associativity: left or right

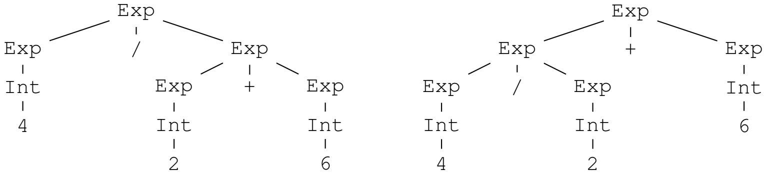
Priorities

Possible parse trees for `4 / 2 + 6`:

- What parse tree is correct in this case?

Priorities

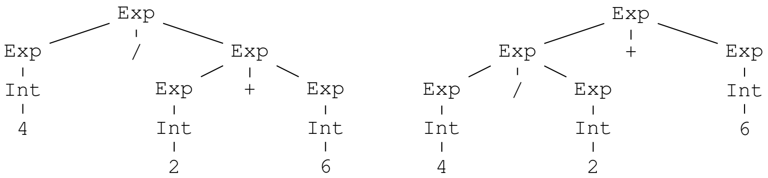
Possible parse trees for `4 / 2 + 6`:



- What parse tree is correct in this case?

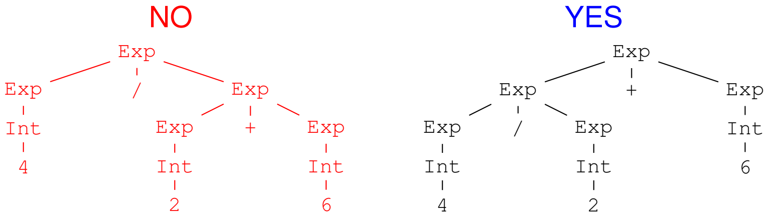
Priorities

Possible parse trees for `4 / 2 + 6`:



- What parse tree is correct in this case?

Priorities



Solutions:

- establish priorities between various constructs
- filtering vs. modify the grammar

Abstract Syntax Trees

- Grammars define *concrete syntax*
- Arithmetic expressions:
  - $1 + 2$ – infix notation
  - $(+\ 1\ 2)$ – prefix notation
  - $(1\ 2\ +)$ – postfix notation
- Each variant has a particular grammar production:
  - $E \rightarrow E + E$ // infix
  - $E \rightarrow + E E$ // prefix
  - $E \rightarrow E E +$ // postfix

Abstract Syntax Trees

- Grammars define *concrete syntax*
- Arithmetic expressions:
  - 1 + 2 – infix notation
  - (+ 1 2) – prefix notation
  - (1 2 +) – postfix notation
- Each variant has a particular grammar production:
  - $E \rightarrow E + E$ // infix
  - $E \rightarrow + E E$ // prefix
  - $E \rightarrow E E +$ // postfix

Abstract Syntax Trees

- Grammars define *concrete syntax*
- Arithmetic expressions:
  - 1 + 2 – infix notation
  - (+ 1 2) – prefix notation
  - (1 2 +) – postfix notation
- Each variant has a particular grammar production:
  - $E \rightarrow E + E$ // infix
  - $E \rightarrow + E\ E$ // prefix
  - $E \rightarrow E\ E\ +$ // postfix

Abstract syntax trees

- Variants:
  - 1 + 2 – infix notation
  - (+ 1 2) – prefix notation
  - (1 2 +) – postfix notation
- Parse trees:

```
     infix              prefix                postfix

       E                   E                     E
      /|\                 /|\                   /|\
     / | \               / | \                 / | \
    1  +  2             +  1  2               1  2  +
```

Abstract syntax trees

- Variants:
  - 1 + 2 – infix notation
  - (+ 1 2) – prefix notation
  - (1 2 +) – postfix notation
- Parse trees:

```
       infix              prefix                 postfix

         E                  E                       E
        /|\                /|\                     /|\
       / | \              / | \                   / | \
      1  +  2            +  1  2                 1  2  +
```

Abstract syntax trees

- Parse trees:

```
     infix            prefix            postfix

       E                 E                  E
      /|\               /|\                /|\
     / | \             / | \              / | \
    1  +  2           +  1  2            1  2  +
```

- *abstract* representation of all the above trees:

```
        add
        / \
       /   \
      1     2
```

Abstract syntax trees

- Parse trees:

```
     infix            prefix            postfix

       E                E                  E
      /|\              /|\                /|\
     / | \            / | \              / | \
    1  +  2          +  1  2            1  2  +
```

- *abstract* representation of all the above trees:

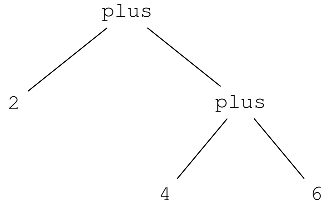
```
      add
      / \
     /   \
    1     2
```

Abstract Syntax Trees

- Grammars define *concrete syntax*
- An AST is a *tree* representation of the structure of a program where the syntactical details are ignored
- For instance, addition has the same abstract tree even if in some languages the syntax different (e.g., C vs. Haskell)
- Compilers use ASTs as the main data structure

Example: arithmetic expressions

AST for `2 + (4 + 6)`:




```
Inductive Exp :=
| number : nat -> Exp
| plus : Exp -> Exp -> Exp.

Coercion number : nat >-> Exp.

Check (plus 2 (plus 4 6)).
plus 2 (plus 4 6)
     : Exp
```

Abstract Syntax in Coq

The BNF grammar of arithmetic expressions:

- E ::= *nat* | E + E | E * E

The corresponding Coq encoding is:

```
Inductive Exp : Type :=
  | num : nat -> Exp
  | plus : Exp -> Exp -> Exp
  | mul : Exp -> Exp -> Exp.
```

Abstract Syntax in Coq

The BNF grammar of arithmetic expressions:

- E ::= *nat* | E + E | E * E

The corresponding Coq encoding is:

```
Inductive Exp : Type :=
  | num : nat -> Exp
  | plus : Exp -> Exp -> Exp
  | mul : Exp -> Exp -> Exp.
```

Coercion

Complicated:

```
Check (plus(num1) (num2)).
```

Less complicated:

```
Check (plus 1 2).
```

Coercion

Complicated:

```
Check (plus(num1) (num2)).
```

Less complicated:

```
Check (plus 1 2).
```

Notations

```
Inductive AExp :=
| avar : string -> AExp
| anum : nat -> AExp
| aplus : AExp -> AExp -> AExp
| amul : AExp -> AExp -> AExp.

Coercion anum : nat >-> AExp.
Coercion avar : string >-> AExp.
Notation "A +' B" := (aplus A B)
    (at level 50, left associativity).
Notation "A *' B" := (amul A B)
    (at level 40, left associativity).
```

IMP

Demo

- Arithmetic expressions
- Boolean expressions
- Statements

Bibliography

- Sections 2.1-2.4 from *Programming Languages: Principles and Paradigms, Maurizio Gabbrielli, Simone Martini; 2010.* Link: `http://websrv.dthu.edu.vn/attachments/newsevents/content2415/Programming_Languages_-_Principles_and_Paradigms_thereds1106.pdf`