CSCI-344
Programming Language Concepts (Section 3)

Lecture 2

Impcore: an Imperative Core Language

Instructor: Hossein Hojjat

August 24, 2016

Programming Paradigm

- A **programming paradigm** is a model of programming based on concepts that shapes the way programmers think, design, organize and write programs
- Main programming paradigms:
  - Imperative programming
  - Functional programming
  - Logic programming
- Orthogonal to it:
  - Object-oriented programming

Programming Paradigm

- A **programming paradigm** is a model of programming based on concepts that shapes the way programmers think, design, organize and write programs
- Main programming paradigms:
  - Imperative programming ← Today's lecture
  - Functional programming
  - Logic programming
- Orthogonal to it:
  - Object-oriented programming

Imperative Programming

- **Imperative programming**
  - Modifying mutable variables using assignments
  - Sequence of statements, with side effects
  - Imperative: "do this, then do that, then do this other thing, ..."
  - Control structures if-branches, loops, break, continue, etc

Imperative Programming

- **Imperative programming**
  - Modifying mutable variables using assignments
  - Sequence of statements, with side effects
  - Imperative: "do this, then do that, then do this other thing, ..."
  - Control structures if-branches, loops, break, continue, etc

- **ImpCore (IMPerative CORE)**
  - Simple imperative programming language
  - Embodies a tiny core fragment of conventional mainstream languages such as C++, Java, ...
  - Eliminate superficial differences, focus on main features

Example to find number is prime

```
(val prime 1)
(val i 2)
(define is-prime? (n)
  (begin
    (while (<= i (/ n 2))
      (begin
        (if (= (mod n i) 0)
          (set prime 0)
          (set prime prime)   ;; don't change
        )
        (set i (+ i 1))
      )
    )
    prime
  )
)
(print (is-prime? 31))
```

Example to find number is prime

```
(val prime 1)
(val i 2)
(define is-prime? (n)
  (begin
    (while (<= i (/ n 2))
      (begin
        (if (= (mod n i) 0)
          (set prime 0)
          (set prime prime)   ;; don't change
        )
        (set i (+ i 1))
      )
    )
    prime
  )
)
(print (is-prime? 31))
```

Global Variable Definition

Example to find number is prime

```
(val prime 1)
(val i 2)
```

Function Definition

```
(define is-prime? (n)
  (begin
    (while (<= i (/ n 2))
      (begin
        (if (= (mod n i) 0)
          (set prime 0)
          (set prime prime)  ;; don't change
        )
        (set i (+ i 1))
      )
    )
    prime
  )
)
```

```
(print (is-prime? 31))
```

Example to find number is prime

```
(val prime 1)
(val i 2)
(define is-prime? (n)                          Begin Expression
  (begin
    (while (<= i (/ n 2))
      (begin
        (if (= (mod n i) 0)
          (set prime 0)
          (set prime prime)   ;; don't change
        )
        (set i (+ i 1))
      )
    )
    prime   <-------- Return Value
  )
)
(print (is-prime? 31))
```

Example to find number is prime

```
(val prime 1)
(val i 2)
(define is-prime? (n)
  (begin
    (while (<= i (/ n 2))
      (begin
        (if (= (mod n i) 0)
          (set prime 0)
          (set prime prime)  ;; don't change
        )
        (set i (+ i 1))
      )
    )
    prime
  )
)
(print (is-prime? 31))
```

While Expression (returns 0)

Example to find number is prime

```
(val prime 1)
(val i 2)
(define is-prime? (n)
  (begin
    (while (<= i (/ n 2))
      (begin
        (if (= (mod n i) 0)
          (set prime 0)
          (set prime prime)  ;; don't change
        )
        (set i (+ i 1))
      )
    )
    prime
  )
)
(print (is-prime? 31))
```

If Expression

LISP-like Syntax

Prefix notation: $(+\ (*\ x\ y)\ 1)$

Infix notation: $(x * y) + 1$

- ImpCore has fully parenthesized prefix notation (as in Lisp)
- Variables and names are basic atoms
- Other constructs bracketed with parentheses
  (Possible keyword after opening bracket)
- **Pros:** Trivial to parse
- **Cons:** Unreadable in complex expressions

ImpCore Syntax

- Two most important syntactic categories: **definitions** and **expressions**
- ImpCore has no statements
- An ImpCore program is a sequence of definitions

ImpCore Definitions

- `(val x exp)`
- `(define f (x ... y) exp)`
- `exp`

ImpCore Definitions

Global variable definition (no local variables)

- Example: `(val prime 1)`
- Equivalents construct in C: `int prime = 1;`

ImpCore Definitions

Global variable definition (no local variables)

- Example: `(val prime 1)`
- Equivalents construct in C: `int prime = 1;`

Function definition

- Example: `(define max (x y) (if (> x y) x y))`
- Equivalents construct in C:

```
int max(int x, int y) {
    return (x>y) ?  x :  y;
}
```

ImpCore Definitions

Global variable definition (no local variables)

- Example: `(val prime 1)`
- Equivalents construct in C: `int prime = 1;`

Function definition

- Example: `(define max (x y) (if (> x y) x y))`
- Equivalents construct in C:

```
int max(int x, int y) {
    return (x>y) ?  x :  y;
}
```

Top-level expression definition

- Top-level expression definitions are treated as implicit definition of a standard variable `it`

- Example: `(+ 5 6)`
- Equivalents definition: `(val it (+ 5 6))`

ImpCore Expressions

- Expression produces a value and may have side effects

  - `100` (integer literal)
  - `x` (variable)
  - `(if exp1 exp2 exp3)`
  - `(while exp1 exp2)`
  - `(set x exp)`
  - `(begin exp1 ... expn)`
  - `(f exp1 ... expn)`

- Primitive Functions: `+` , `-` , `*` , `/` , `=` , `<` , `>` , `print`
- Only type of data is "machine integer" (deliberate oversimplification)

Concrete vs. abstract syntax

Abstract Syntax

- The "essential" constructs of a programming language

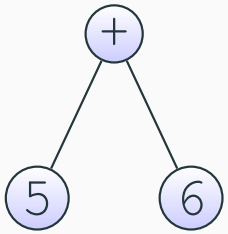
Concrete Syntax

- The actual syntax of a programming language

Concrete vs. abstract syntax

Abstract Syntax

- The "essential" constructs of a programming language
- An additive expression has two operand expressions as its "essential" parts



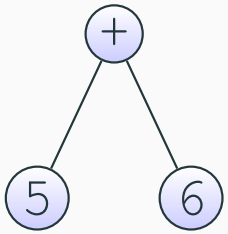
Concrete Syntax

- The actual syntax of a programming language

Concrete vs. abstract syntax

Abstract Syntax

- The "essential" constructs of a programming language
- An additive expression has two operand expressions as its "essential" parts



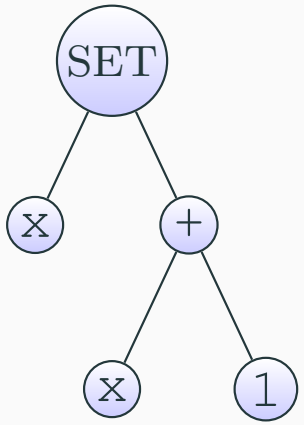
Concrete Syntax

- The actual syntax of a programming language
  - `5 + 6` Java (infix)
  - `(+ 5 6)` ImpCore (prefix)
  - `bipush 5`
`bipush 4`
`iadd` JVM (postfix)

Concrete vs. abstract syntax

Abstract Syntax

- The "essential" constructs of a programming language



Concrete Syntax

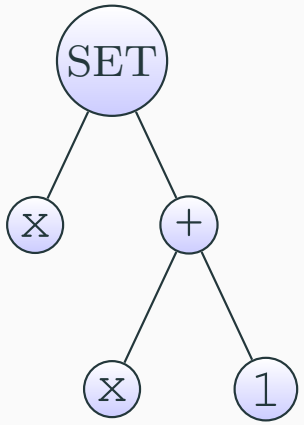
- The actual syntax of a programming language

  - `x = x + 1;` Java
  - `(set x (+ x 1))` ImpCore
  - `x := x + 1;` Pascal

Concrete vs. abstract syntax

Abstract Syntax

- The "essential" constructs of a programming language



Concrete Syntax

- The actual syntax of a programming language

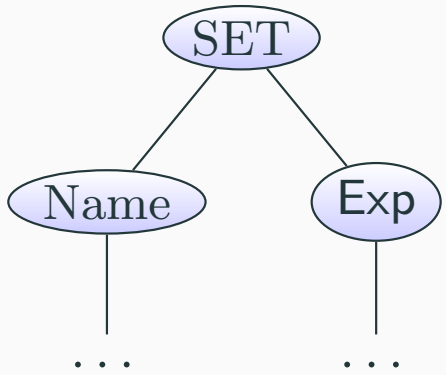
  - `x = x + 1;` Java
  - `(set x (+ x 1))` ImpCore
  - `x := x + 1;` Pascal

Abstract Syntax Tree (AST): Ordered-tree representation of the abstract syntactic structure of source code

Abstract Syntax Tree (AST) Representation

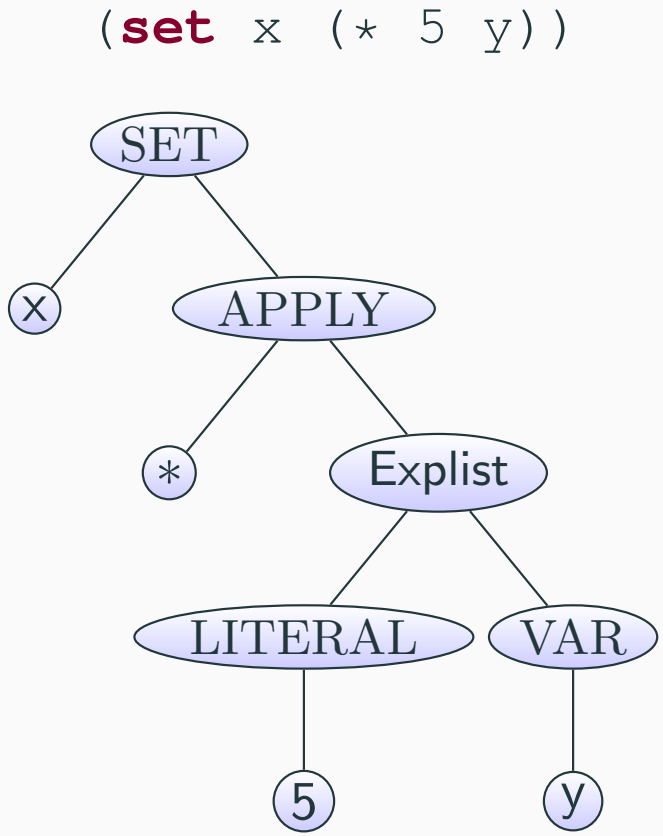
- We represent AST with context-free grammars
- Label each node with all capitalized constructor
- Specify children in parentheses

$$
\begin{array}{lcll}
\text{Exp} & = & \text{LITERAL} & \text{(Value)} \\
& | & \text{VAR} & \text{(Name)} \\
& | & \text{SET} & \text{(Name, Exp)} \\
& | & \ldots &
\end{array}
$$

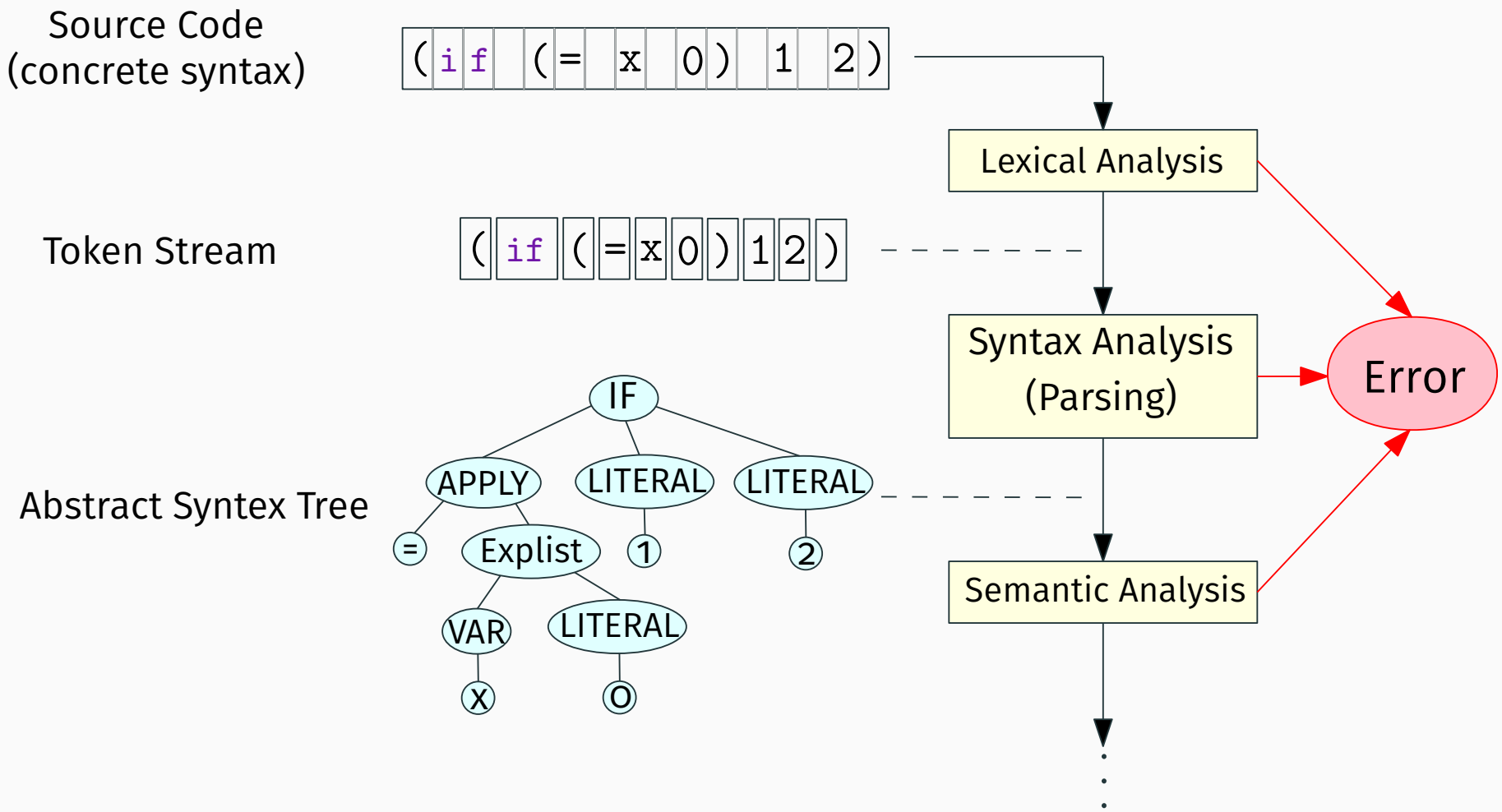


ImpCore Abstract Syntax

Def	=	VAL	(Name, Exp)
	|	EXP	(Exp)
	|	DEFINE	(Name, Namelist, Exp)
Exp	=	LITERAL	(Value)
	|	VAR	(Name)
	|	SET	(Name, Exp)
	|	IF	(Exp, Exp, Exp)
	|	WHILE	(Exp, Exp)
	|	BEGIN	(Explist)
	|	APPLY	(Name, Explist)



Typical Interpreter



Semantic Errors

- Syntax analysis does not correlate variable definitions with the uses of variables
- Example: Syntactically correct program

```
(val x 0)
(set x (+ y 1))
```

Semantic Errors

- Syntax analysis does not correlate variable definitions with the uses of variables
- Example: Syntactically correct program

```
(val x 0)
(set x (+ y 1))
```

- `(+ y 1)` cannot be evaluated by itself: we need to know what `y` is
- Programming-language theory uses **environments** to define the meanings of names
- Environment is a mapping from names to meanings
- Environments are often implemented as hash tables or search trees, so they are sometimes called **symbol tables**

Environments

- An environment is a set of bindings denoted $\mapsto$
- We write $\rho\{x \mapsto v\}$ to mean the environment ρ extended by a binding of the name x to v

$$\rho\{x \mapsto v\}(y) = \begin{cases} v, & \text{when } y = x \\ \rho(y), & \text{when } y \neq x \end{cases}$$

- ξ: global variables, ρ: formal parameters
- There are no local variables: if you need temporaries use formal parameters

Example to find number is prime

$$\xi_0 = \{\}$$

```
(val prime 1)
(val i 2)
(define is-prime? (n)
  (begin
    (while (<= i (/ n 2))
      (begin
        (if (= (mod n i) 0)
          (set prime 0)
          (set prime prime)
        )
        (set i (+ i 1))
      )
    )
    prime
  )
)
(print (is-prime? 31))
```

Example to find number is prime

$$\xi_0 = \{\}$$

$$\xi_1 = \xi_0\{\mathit{prime} \mapsto 1\}$$

```
→ (val prime 1)
  (val i 2)
  (define is-prime? (n)
    (begin
      (while (<= i (/ n 2))
        (begin
          (if (= (mod n i) 0)
            (set prime 0)
            (set prime prime)
          )
          (set i (+ i 1))
        )
      )
      prime
    )
  )
  (print (is-prime? 31))
```

Example to find number is prime

$$\xi_0 = \{\}$$

$$\xi_1 = \xi_0\{prime \mapsto 1\}$$

$$\xi_2 = \xi_1\{i \mapsto 2\}$$

```
   (val prime 1)
→  (val i 2)
   (define is-prime? (n)
     (begin
       (while (<= i (/ n 2))
         (begin
           (if (= (mod n i) 0)
             (set prime 0)
             (set prime prime)
           )
           (set i (+ i 1))
         )
       )
       prime
     )
   )
   (print (is-prime? 31))
```

Example to find number is prime

$$\xi_0 = \{\}$$

$$\xi_1 = \xi_0\{prime \mapsto 1\}$$

$$\xi_2 = \xi_1\{i \mapsto 2\}$$

$$\xi_2 \ , \ \rho_0 = \{n \mapsto 31\}$$

```
   (val prime 1)
   (val i 2)
→  (define is-prime? (n)
     (begin
       (while (<= i (/ n 2))
         (begin
           (if (= (mod n i) 0)
             (set prime 0)
             (set prime prime)
           )
           (set i (+ i 1))
         )
       )
       prime
     )
   )
   (print (is-prime? 31))
```

Environmental Abuse

- The environment ϕ contains the definition of functions
- Abuse of separate environments

```
-> (val x 1)
1
-> (define x (y) (- x y))
-> (define y (x) (x x))
-> (y 1)
0
```

- Which xs correspond to globals?
- Which xs correspond to formals?
- Which xs correspond to functions?

Next Week

- We study **Semantics**: the meaning of a program, or how it behaves
- What is the meaning of `(+ x y)`?
- Semantics helps to understand a particular language
  - what you can depend on as a programmer
  - what you must provide as a compiler writer