

Lecture 4 – Identifiers (1)

COSE212: Programming Languages

Jihyeok Park



2025 Fall

- **ADT** for **Abstract Syntax** of AE

```
enum Expr:  
  case Num(number: BigInt)  
  case Add(left: Expr, right: Expr)  
  case Mul(left: Expr, right: Expr)
```

- **Parser** for **Concrete Syntax** of AE

```
lazy val expr: P[Expr] = ...
```

- **Interpreter** for **Semantics** of AE

```
def interp(expr: Expr): Value = ...
```

- **ADT** for **Abstract Syntax** of AE

```
enum Expr:  
  case Num(number: BigInt)  
  case Add(left: Expr, right: Expr)  
  case Mul(left: Expr, right: Expr)
```

- **Parser** for **Concrete Syntax** of AE

```
lazy val expr: P[Expr] = ...
```

- **Interpreter** for **Semantics** of AE

```
def interp(expr: Expr): Value = ...
```

- In this lecture, we will learn **identifiers**.

1. Identifiers

- Bound Identifiers

- Free Identifiers

- Shadowing

2. VAE – AE with Variables

- Concrete Syntax

- Abstract Syntax

- Examples

1. Identifiers

- Bound Identifiers

- Free Identifiers

- Shadowing

2. VAE – AE with Variables

- Concrete Syntax

- Abstract Syntax

- Examples

An **identifier** is a **name** for a certain element in a program.

In Scala, there are diverse kinds of identifiers:

```
/* Scala */

// variable names
val x: Int = 42

// function and parameter names
def f(a: Int, b: Int): Int = a + b

// class and field names
case class Person(name: String, age: Int)

...
```

```
/* Scala */  
val x: Int = 3  
  
val y: Int = x + z  
  
def add(a: Int, b: Int): Int =  
  val x: Int = a + b  
  x + add(y, z)  
  
add(x, b)
```

A **bound identifier** is an identifier that is **defined** in a program.

- A **binding occurrence** of an identifier denotes its **definition** site.
- A **scope** of an identifier denotes where the identifier is **usable**.
- A **bound occurrence** of an identifier denotes its **lookup** site.

```
/* Scala */  
val x: Int = 3  
  
val y: Int = x + z  
  
def add(a: Int, b: Int): Int =  
  val x: Int = a + b  
  x + add(y, z)  
  
add(x, b)
```

Binding
Occurrences

A **bound identifier** is an identifier that is **defined** in a program.

- A **binding occurrence** of an identifier denotes its **definition** site.
- A **scope** of an identifier denotes where the identifier is **usable**.
- A **bound occurrence** of an identifier denotes its **lookup** site.


```
/* Scala */  
val x: Int = 3  
  
val y: Int = x + z  
  
def add(a: Int, b: Int): Int =  
  val x: Int = a + b  
  x + add(y, z)  
  
add(x, b)
```

The diagram illustrates the concept of a **scope** in Scala. A red dashed arrow points from the identifier `x` in the first line (`val x: Int = 3`) to the `x` in the second line (`val y: Int = x + z`). A red box encloses the code block starting from `val y: Int = x + z` and ending at `add(x, b)`, with the word **scope** written in red to the right of the box.

A **bound identifier** is an identifier that is **defined** in a program.

- A **binding occurrence** of an identifier denotes its **definition** site.
- A **scope** of an identifier denotes where the identifier is **usable**.
- A **bound occurrence** of an identifier denotes its **lookup** site.

```
/* Scala */  
val x: Int = 3  
  
val y: Int = x + z  
  
scope  
  
def add(a: Int, b: Int): Int =  
  val x: Int = a + b  
  x + add(y, z)  
  
add(x, b)
```

A **bound identifier** is an identifier that is **defined** in a program.

- A **binding occurrence** of an identifier denotes its **definition** site.
- A **scope** of an identifier denotes where the identifier is **usable**.
- A **bound occurrence** of an identifier denotes its **lookup** site.

```
/* Scala */  
val x: Int = 3  
  
val y: Int = x + z  
  
def add(a: Int, b: Int): Int =  
  val x: Int = a + b  
  x + add(y, z)  
  + scope  
add(x, b)
```

A **bound identifier** is an identifier that is **defined** in a program.

- A **binding occurrence** of an identifier denotes its **definition** site.
- A **scope** of an identifier denotes where the identifier is **usable**.
- A **bound occurrence** of an identifier denotes its **lookup** site.

```
/* Scala */  
val x: Int = 3  
  
val y: Int = x + z  
  
def add(a: Int, b: Int): Int =  
  val x: Int = a + b  
  x + add(y, z)  
  scope  
add(x, b)
```

A **bound identifier** is an identifier that is **defined** in a program.

- A **binding occurrence** of an identifier denotes its **definition** site.
- A **scope** of an identifier denotes where the identifier is **usable**.
- A **bound occurrence** of an identifier denotes its **lookup** site.

```
/* Scala */  
val x: Int = 3  
  
val y: Int = x + z  
  
def add(a: Int, b: Int): Int =  
  val x: Int = a + b  
  x + add(y, z)  
  scope  
add(x, b)
```

A **bound identifier** is an identifier that is **defined** in a program.

- A **binding occurrence** of an identifier denotes its **definition** site.
- A **scope** of an identifier denotes where the identifier is **usable**.
- A **bound occurrence** of an identifier denotes its **lookup** site.

```
/* Scala */  
val x: Int = 3  
  
val y: Int = x + z  
  
def add(a: Int, b: Int): Int =  
  val x: Int = a + b  
  x + add(y, z)  
  scope  
add(x, b)
```

A **bound identifier** is an identifier that is **defined** in a program.

- A **binding occurrence** of an identifier denotes its **definition** site.
- A **scope** of an identifier denotes where the identifier is **usable**.
- A **bound occurrence** of an identifier denotes its **lookup** site.

```
/* Scala */  
val x: Int = 3  
  
val y: Int = x + z  
  
def add(a: Int, b: Int): Int =  
  val x: Int = a + b  
  x + add(y, z)  
  
add(x, b)
```

Bound
Occurrences

A **bound identifier** is an identifier that is **defined** in a program.

- A **binding occurrence** of an identifier denotes its **definition** site.
- A **scope** of an identifier denotes where the identifier is **usable**.
- A **bound occurrence** of an identifier denotes its **lookup** site.

```
/* Scala */  
val x: Int = 3  
  
val y: Int = x + z  
  
def add(a: Int, b: Int): Int =  
  val x: Int = a + b  
  x + add(y, z)  
  
add(x, b)
```

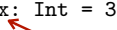
Bound
Occurrences

A **bound identifier** is an identifier that is **defined** in a program.

- A **binding occurrence** of an identifier denotes its **definition** site.
- A **scope** of an identifier denotes where the identifier is **usable**.
- A **bound occurrence** of an identifier denotes its **lookup** site.

Let's draw arrows from each bound occurrence to its binding occurrence.


```
/* Scala */  
val x: Int = 3  
val y: Int = x + z  
  
def add(a: Int, b: Int): Int =  
  val x: Int = a + b  
  x + add(y, z)  
  
add(x, b)
```

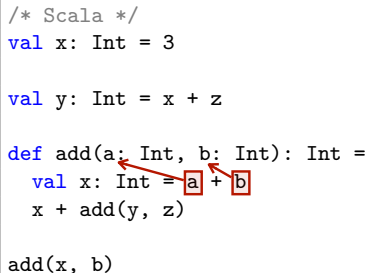


A **bound identifier** is an identifier that is **defined** in a program.

- A **binding occurrence** of an identifier denotes its **definition** site.
- A **scope** of an identifier denotes where the identifier is **usable**.
- A **bound occurrence** of an identifier denotes its **lookup** site.

Let's draw arrows from each bound occurrence to its binding occurrence.


```
/* Scala */  
val x: Int = 3  
  
val y: Int = x + z  
  
def add(a: Int, b: Int): Int =  
  val x: Int = a + b  
  x + add(y, z)  
  
add(x, b)
```



A **bound identifier** is an identifier that is **defined** in a program.

- A **binding occurrence** of an identifier denotes its **definition** site.
- A **scope** of an identifier denotes where the identifier is **usable**.
- A **bound occurrence** of an identifier denotes its **lookup** site.

Let's draw arrows from each bound occurrence to its binding occurrence.

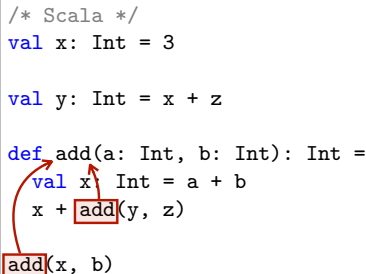
```
/* Scala */  
val x: Int = 3  
  
val y: Int = x + z  
  
def add(a: Int, b: Int): Int =  
  val x: Int = a + b  
   + add(y, z)  
  
add(x, b)
```

A **bound identifier** is an identifier that is **defined** in a program.

- A **binding occurrence** of an identifier denotes its **definition** site.
- A **scope** of an identifier denotes where the identifier is **usable**.
- A **bound occurrence** of an identifier denotes its **lookup** site.

Let's draw arrows from each bound occurrence to its binding occurrence.

```
/* Scala */  
val x: Int = 3  
  
val y: Int = x + z  
  
def add(a: Int, b: Int): Int =  
  val x: Int = a + b  
  x + add(y, z)  
  
add(x, b)
```

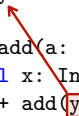


A **bound identifier** is an identifier that is **defined** in a program.

- A **binding occurrence** of an identifier denotes its **definition** site.
- A **scope** of an identifier denotes where the identifier is **usable**.
- A **bound occurrence** of an identifier denotes its **lookup** site.

Let's draw arrows from each bound occurrence to its binding occurrence.

```
/* Scala */  
val x: Int = 3  
  
val y: Int = x + z  
  
def add(a: Int, b: Int): Int =  
  val x: Int = a + b  
  x + add(y, z)  
  
add(x, b)
```

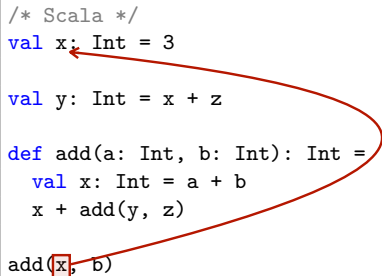


A **bound identifier** is an identifier that is **defined** in a program.

- A **binding occurrence** of an identifier denotes its **definition** site.
- A **scope** of an identifier denotes where the identifier is **usable**.
- A **bound occurrence** of an identifier denotes its **lookup** site.

Let's draw arrows from each bound occurrence to its binding occurrence.

```
/* Scala */  
val x: Int = 3  
  
val y: Int = x + z  
  
def add(a: Int, b: Int): Int =  
  val x: Int = a + b  
  x + add(y, z)  
  
add(x, b)
```

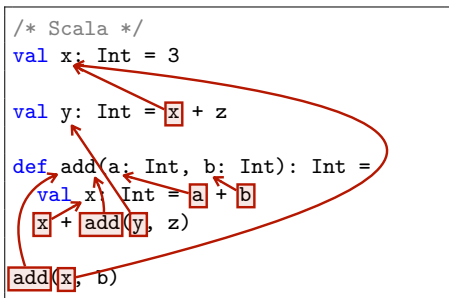


A **bound identifier** is an identifier that is **defined** in a program.

- A **binding occurrence** of an identifier denotes its **definition** site.
- A **scope** of an identifier denotes where the identifier is **usable**.
- A **bound occurrence** of an identifier denotes its **lookup** site.

Let's draw arrows from each bound occurrence to its binding occurrence.

```
/* Scala */  
val x: Int = 3  
  
val y: Int = x + z  
  
def add(a: Int, b: Int): Int =  
  val x: Int = a + b  
  x + add(y, z)  
  
add(x, b)
```



A **bound identifier** is an identifier that is **defined** in a program.

- A **binding occurrence** of an identifier denotes its **definition** site.
- A **scope** of an identifier denotes where the identifier is **usable**.
- A **bound occurrence** of an identifier denotes its **lookup** site.

Let's draw arrows from each bound occurrence to its binding occurrence.

```
/* Scala */  
val x: Int = 3  
  
val y: Int = x + z  
  
def add(a: Int, b: Int): Int =  
  val x: Int = a + b  
  x + add(y, z)  
  
add(x, z)
```

Free
Identifiers

A **free identifier** is an identifier that is **not defined** in the current scope of the program.


```
/* Scala */  
val x: Int = 3  
  
val y: Int = x + z  
  
def add(a: Int, b: Int): Int =  
  val x: Int = a + b  
  x + add(y, z)  
  
add(x, b)
```

Diagram illustrating variable shadowing in Scala:

- The first `val x: Int = 3` is labeled **Shadowed Identifier** (red text).
- The second `val x: Int = a + b` is labeled **Shadowing** (red text).
- The `add(x, b)` call is labeled **Shadowing Identifier** (red text).
- Red dashed arrows indicate the shadowing relationships: from the inner `x` to the outer `x`, and from the `add(x, b)` `x` to the inner `x`.

Shadowing means that the innermost binding occurrence shadows the outer binding occurrences of the same name.

- A **shadowing identifier** is an identifier that shadows another
- A **shadowed identifier** is an identifier that is shadowed by another.

```
/* Scala */  
val x: Int = 3  
  
val y: Int = x + z  
  
def add(a: Int, b: Int): Int =  
  val x: Int = a + b  
  x + add(y, z)  
  
add(x, b)
```

Diagram illustrating shadowing in Scala code:

- The first `val x: Int = 3` is labeled **Shadowed Identifier** (red text).
- The `val x` inside the `add` function is labeled **Shadowing** (red text).
- The `add(x, b)` call at the bottom is labeled **Shadowing Identifier** (red text).
- Red dashed arrows indicate the shadowing relationships: from the inner `x` to the outer `x`, and from the `add(x, b)` call to the inner `x`.

Shadowing means that the innermost binding occurrence shadows the outer binding occurrences of the same name.

- A **shadowing identifier** is an identifier that shadows another
- A **shadowed identifier** is an identifier that is shadowed by another.

Note that shadowing is **NOT** a mutation.

1. Identifiers

Bound Identifiers

Free Identifiers

Shadowing

2. VAE – AE with Variables

Concrete Syntax

Abstract Syntax

Examples

Now, we want to extend AE into VAE with **variables**:

```
/* VAE */  
val x = 1 + 2;  // x = 1 + 2 = 3  
val y = x + 3;  // y = x + 3 = 3 + 3 = 6  
y + 4           // 6 + 4 = 10
```

Now, we want to extend AE into VAE with **variables**:

```
/* VAE */  
val x = 1 + 2;    // x = 1 + 2 = 3  
val y = x + 3;    // y = x + 3 = 3 + 3 = 6  
y + 4             // 6 + 4 = 10
```

First, we define the **concrete syntax** of **identifiers** used in VAE:

```
<digit>      ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"  
<number>     ::= "-"? <digit>+  
<alphabet>   ::= "A" | "B" | "C" | ... | "Z" | "a" | "b" | "c" | ... | "z"  
<idstart>    ::= <alphabet> | "_"  
<idcont>     ::= <alphabet> | "_" | <digit>  
<keyword>    ::= "val"  
<id>         ::= <idstart> <idcont>* butnot <keyword>
```

Now, we want to extend AE into VAE with **variables**:

```
/* VAE */  
val x = 1 + 2;    // x = 1 + 2 = 3  
val y = x + 3;    // y = x + 3 = 3 + 3 = 6  
y + 4             // 6 + 4 = 10
```

First, we define the **concrete syntax** of **identifiers** used in VAE:

```
<digit>      ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"  
<number>     ::= "-"? <digit>+  
<alphabet>   ::= "A" | "B" | "C" | ... | "Z" | "a" | "b" | "c" | ... | "z"  
<idstart>    ::= <alphabet> | "_"  
<idcont>     ::= <alphabet> | "_" | <digit>  
<keyword>    ::= "val"  
<id>         ::= <idstart> <idcont>* butnot <keyword>
```

For example, the following are valid identifiers:

x y get_name getName add42

Then, let's define the **concrete syntax** of VAE in BNF:

```
<expr> ::= <number>
        | <expr> "+" <expr>
        | <expr> "*" <expr>
        | "(" <expr> ")"
        | "{" <expr> "}"
        | "val" <id> "=" <expr> ";" <expr>
        | <id>
```

Then, let's define the **concrete syntax** of VAE in BNF:

```
<expr> ::= <number>
        | <expr> "+" <expr>
        | <expr> "*" <expr>
        | "(" <expr> ")"
        | "{" <expr> "}"
        | "val" <id> "=" <expr> ";" <expr>
        | <id>
```

Note that each variable definition creates a **new scope**.

Then, let's define the **concrete syntax** of VAE in BNF:

```
<expr> ::= <number>
        | <expr> "+" <expr>
        | <expr> "*" <expr>
        | "(" <expr> ")"
        | "{" <expr> "}"
        | "val" <id> "=" <expr> ";" <expr>
        | <id>
```

Note that each variable definition creates a **new scope**. For example:

```
/* VAE */
val x = 1 + 2;
val y = x + 3;
y + 4
```

Then, let's define the **concrete syntax** of VAE in BNF:

```
<expr> ::= <number>
        | <expr> "+" <expr>
        | <expr> "*" <expr>
        | "(" <expr> ")"
        | "{" <expr> "}"
        | "val" <id> "=" <expr> ";" <expr>
        | <id>
```

Note that each variable definition creates a **new scope**. For example:

```
/* VAE */
val x = 1 + 2;
val y = x + 3;
y + 4
```

means

```
/* VAE */
val x = 1 + 2;
{ // scope of x
  val y = x + 3;
  { // scope of y
    y + 4
  }
}
```

Let's define the **abstract syntax** of VAE in BNF:

Numbers	$n \in \mathbb{Z}$	(BigInt)	Expressions	$e ::= n$	(Num)
Identifiers	$x \in \mathbb{X}$	(String)		$e + e$	(Add)
				$e * e$	(Mul)
				val $x = e; e$	(Val)
				x	(Id)

Let's define the **abstract syntax** of VAE in BNF:

Numbers	$n \in \mathbb{Z}$ (BigInt)	Expressions	$e ::= n$	(Num)
Identifiers	$x \in \mathbb{X}$ (String)		$ e + e$	(Add)
			$ e * e$	(Mul)
			$ \text{val } x = e; e$	(Val)
			$ x$	(Id)

We can define an **ADT** for the abstract syntax of VAE in Scala:

```
enum Expr:
  case Num(number: BigInt)
  case Add(left: Expr, right: Expr)
  case Mul(left: Expr, right: Expr)
  // variable definition
  case Val(name: String, init: Expr, body: Expr)
  // variable lookup
  case Id(name: String)
```

```
enum Expr:
  case Num(number: BigInt)
  case Add(left: Expr, right: Expr)
  case Mul(left: Expr, right: Expr)
  case Val(name: String, init: Expr, body: Expr)
  case Id(name: String)
```

Parser implementation is given and you don't need to implement it.

You can freely use Expr to parse VAE programs as follows:

```
Expr("val x = 1; x + 2")
// Val("x", Num(1), Add(Id("x"), Num(2)))

Expr("val a = 1; val b = 2; a + b")
// Val("a", Num(1), Val("b", Num(2), Add(Id("a"), Id("b"))))
```

For each VAE program, please draw:

- an **arrow** from each **bound occurrence** to its **binding occurrence**.
- a **dotted arrow** from each **shadowing variable** to its **shadowed one**.
- an **X** mark on each **free variable**.

```
/* VAE */  
val x = 1; x
```

```
/* VAE */  
val x = x + 1;  
val y = x * 2;  
val x = y + x;  
x * z
```

```
/* VAE */  
val x = 1;  
val y = {  
    val x = 2 * x;  
    { val y = x; y } + { val y = 3; y }  
};  
x + y
```

1. Identifiers

- Bound Identifiers

- Free Identifiers

- Shadowing

2. VAE – AE with Variables

- Concrete Syntax

- Abstract Syntax

- Examples

- Identifiers (2)

Jihyeok Park

jihyeok_park@korea.ac.kr

<https://plrg.korea.ac.kr>