Lecture 4 – Identifiers (1)

COSE212: Programming Languages

Jihyeok Park

2023 Fall

Recall

- **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 = ...
```

Recall

- **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**.

Contents

Contents

Identifiers

PLRG

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

In Scala, there are diverse kinds of identifiers:

```
// 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)

...
```

Bound Identifiers

PLRG

```
val x: Int = 3
val y: Int = x + 1
def f(a: Int, b: Int): Int = {
  val x: Int = a + b
  x + y + z
}
f(x, b)
```

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

- A **binding occurrence** of an identifier is the occurrence in its *definition* position.
- A **scope** of an identifier is a *code region* where the identifier is usable.
- A **bound occurrence** of an identifier is an occurrence in a *lookup* position in its scope.

Bound Identifiers

PLRG

```
val x: Int = 3
val y: Int = x + 1
def f(a: Int, b: Int): Int = {
  val x: Int = a + b
  x + y + z
}
f(x, b)
```

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

- A **binding occurrence** of an identifier is the occurrence in its *definition* position.
- A **scope** of an identifier is a *code region* where the identifier is usable.
- A **bound occurrence** of an identifier is an occurrence in a *lookup* position in its scope.

Bound Identifiers

PLRG

```
val x: Int = 3
val y: Int = x + 1
def f(a: Int, b: Int): Int = {
  val x: Int = a + b
  x + y + z
}
f(x, b)
```

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

- A **binding occurrence** of an identifier is the occurrence in its *definition* position.
- A **scope** of an identifier is a *code region* where the identifier is usable.
- A **bound occurrence** of an identifier is an occurrence in a *lookup* position in its scope.

Bound Identifiers

```
val x: Int = 3
val y: Int = x + 1
def f(a: Int, b: Int): Int = {
  val x: Int = a + b
  x + y + z
}
f(x, b)
```

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

- A **binding occurrence** of an identifier is the occurrence in its *definition* position.
- A **scope** of an identifier is a *code region* where the identifier is usable.
- A **bound occurrence** of an identifier is an occurrence in a *lookup* position in its scope.

Free Identifiers

```
val x: Int = 3
val y: Int = x + 1
def f(a: Int, b: Int): Int = {
  val x: Int = a + b
  x + y + z
}
f(x, b)
```

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

Shadowing

```
val x: Int = 3
val y: Int = x + 1
def f(a: Int, b: Int): Int = {
  val x: Int = a + b
  x + y + z
}
f(x, b)
```

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

Shadowing

```
val x: Int = 3
val y: Int = x + 1
def f(a: Int, b: Int): Int = {
  val x: Int = a + b
  x + y + z
}
f(x, b)
```

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

Note that this is **NOT** a mutation because the value stored in the shadowed identifier is unchanged.

Contents

VAE – AE with Variables

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
```

VAE – AE with Variables

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:

```
<alphabet> ::= "A" | "B" | "C" | ... | "Z" | "a" | "b" | "c" | ... | "z"
<idstart>  ::= <alphabet> | "_"
<idcont>   ::= <alphabet> | "_" | <digit>
<id>       ::= <idstart> <idcont>*
```

VAE – AE with Variables

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:

```
<alphabet> ::= "A" | "B" | "C" | ... | "Z" | "a" | "b" | "c" | ... | "z"
<idstart>  ::= <alphabet> | "_"
<idcont>   ::= <alphabet> | "_" | <digit>
<id>       ::= <idstart> <idcont>*
```

For example, the following are valid identifiers:

```
x    y    get_name    getName    add42
```

Concrete Syntax

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

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

Concrete Syntax

Now, 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**.

Concrete Syntax

Now, 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;
  {
    val y = x + 3;
    y + 4
  }
}
```

Abstract Syntax

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

$$
\begin{array}{rcll}
e & ::= & n & (\texttt{Num}) \\
& | & e + e & (\texttt{Add}) \\
& | & e \times e & (\texttt{Mul}) \\
& | & \texttt{val}\ x = e;\ e & (\texttt{Val}) \\
& | & x & (\texttt{Id})
\end{array}
$$

Abstract Syntax

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

$$
\begin{array}{rcll}
e & ::= & n & \texttt{(Num)} \\
 & | & e + e & \texttt{(Add)} \\
 & | & e \times e & \texttt{(Mul)} \\
 & | & \texttt{val } x = e\texttt{; } e & \texttt{(Val)} \\
 & | & x & \texttt{(Id)}
\end{array}
$$

```
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)
```

Abstract Syntax

PLRG

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

$$
\begin{array}{rcll}
e & ::= & n & \texttt{(Num)} \\
 & | & e + e & \texttt{(Add)} \\
 & | & e \times e & \texttt{(Mul)} \\
 & | & \texttt{val}\ x = e;\ e & \texttt{(Val)} \\
 & | & x & \texttt{(Id)}
\end{array}
$$

```
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)
```

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

Exercise

PLRG

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
```

Summary

PLRG

1. Identifiers
    - Bound Identifiers
    - Free Identifiers
    - Shadowing

2. VAE – AE with Variables
    - Concrete Syntax
    - Abstract Syntax
    - Exercise

Next Lecture

- Identifiers (2)

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr