Lenguajes de Programación

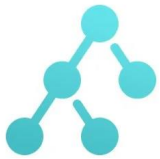
- Semantics via Interpretation
- Expressions with identifiers

Federico Olmedo
Ismael Figueroa

- Semantics via Interpretation

Interpreting a core language for arithmetic

We can now study interpretation semantics by ***executing*** the program represented in the abstract syntax tree.



Abstract Syntax Tree in abstract grammar

interpreter

Program result

What does an arithmetic language do?
It computes numbers by evaluating arithmetic expressions.

To execute our programs we must construct a calculator program.

Interpreting a core language for arithmetic: abstract syntax

```
#|
<expr> ::= (num <num>)
         | (add <expr> <expr>)
         | (sub <expr> <expr>)
         | (if0 <expr> <expr> <expr>)
|#
(deftype Expr
  (num n)
  (add l r)
  (sub l r)
  (if0 c t f))
```

The calculator program takes as input an abstract syntax tree, defined as a value of type **`Expr`**.

Interpreting a core language for arithmetic: calculator

The calculator program takes as input an abstract syntax tree, defined as a value of type **`Expr`**.

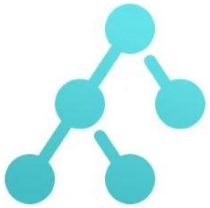
Interpretation proceeds case by case, following the variants of the abstract grammar.

```
;; calc :: Expr -> number
;; Evaluates an arithmetic expression.
```

Remember to add tests!

Complete pipeline for our interpreters

parsing



interpretation

Program result

Source code in concrete grammar

Abstract Syntax Tree in abstract grammar

```
(calc (parse '{+ 10 {+ 4 5}}))
```

In general, the interpretation function ***is the interpreter*** **and will be called `interp`**

How do we extend our language **semantics**?

Extending our language syntax involves several steps:

Extend Concrete Syntax:	how will developers write programs using the new features?
Extend Abstract Syntax:	what information will we extract from the concrete syntax, for later interpretation of programs?
Extend Parser:	update the parsing function to correctly parse the new syntax and capture the abstract information
Extend Interpreter:	implement the new semantics in terms of how to process the new abstract syntax in the interpreter

→ Expressions with identifiers: syntax

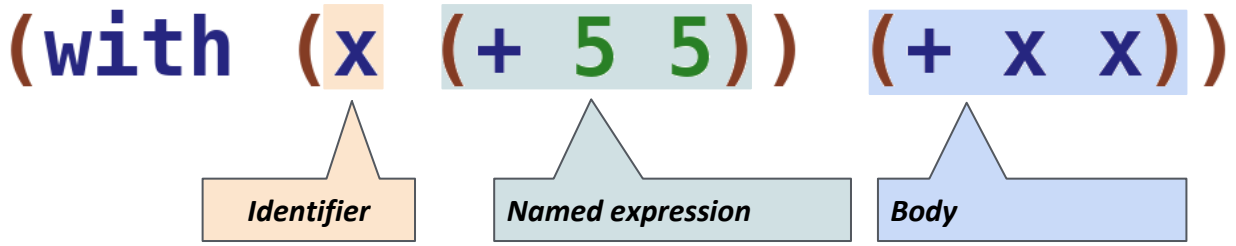
Adding the `with` expression for local binding

We want to extend our language with a **`with`** expression to declare local identifiers, similar to Racket's **`let`**.

However, the semantics of adding local identifiers is not as straightforward as the previous cases!

Let us start by adding the new syntaxes for writing programs using **`with`**

Adding `with` to the concrete syntax



IDENTIFIER: The actual identifier that is being introduced by the `with` expression.

NAMED EXPRESSION: The expression to which the introduced identifier is ***bound to***.

BODY: Scope in which the binding between the identifier and the named expression holds.

Adding `with` to the concrete syntax


```
(with (x (+ 5 5)) (+ x x))
```

— simple expression

```
(with (x (+ 5 5))
      (with (y (- 3 4))
            (+ x y)))
```

— nested expression

```
(with (x (+ 5 5))
      (with (y (- 3 x))
            (+ x y)))
```

— identifier for non-constant expression

Extending the concrete syntax

```
#|
<s-expr> ::= <num>
           | (add <s-expr> <s-expr>)
           | (sub <s-expr> <s-expr>)
           | (if0 <s-expr> <s-expr> <s-expr>)
           | (list 'with (list <sym> <s-expr>) <s-expr>)
           | <sym>
|#
```

We add the **`with`** expression itself, and also the use of symbols as identifiers inside the programs.

Notice we use inner parentheses to delimit the identifier and named expression, which are separated from the body.

Extending the abstract syntax

```
#|
<expr> ::= (num <num>)
         | (add <expr> <expr>)
         | (sub <expr> <expr>)
         | (if0 <expr> <expr> <expr>)
         | (with <sym> <expr> <expr>)
         | (id <sym>)
|#
(deftype Expr
  (num n)
  (add l r)
  (sub l r)
  (if0 c t f)
  (with x named-expr body)
  (id x))
```

We also add the `with` expression itself, and also the use of identifiers.

Notice the with node does not retain the superfluous details of the concrete syntax, just the required information.

Updating the parser

```
;; parse: s-expr -> Expr
;; Parse an s-expr into an Expr.
(define (parse s-expr)
  (match s-expr
    [(? number? n) (num n)]
    [(? symbol? x) (id x)]
    [(list '+ l r) (add (parse l) (parse r))]
    [(list '- l r) (sub (parse l) (parse r))]
    [(list 'if0 c t f) (if0 (parse c) (parse t) (parse f))]
    [(list 'with (list (? symbol? x) named-expr) body)
     (with x (parse named-expr) (parse body))]))
```

The parser now must handle symbols, which are parsed as identifiers. It must also handle **`with`** expressions.

Notice the identifier introduced by the **`with`** expression is not parsed — it's stored directly inside the **`with`** node, as defined in the inductive type definition.

Closed vs Open expressions

```
#|
<s-expr> ::= <num>
           | (add <s-expr> <s-expr>)
           | (sub <s-expr> <s-expr>)
           | (if0 <s-expr> <s-expr> <s-expr>)
           | (list 'with (list <sym> <s-expr>) <s-expr>)
           | <sym>
|#
```

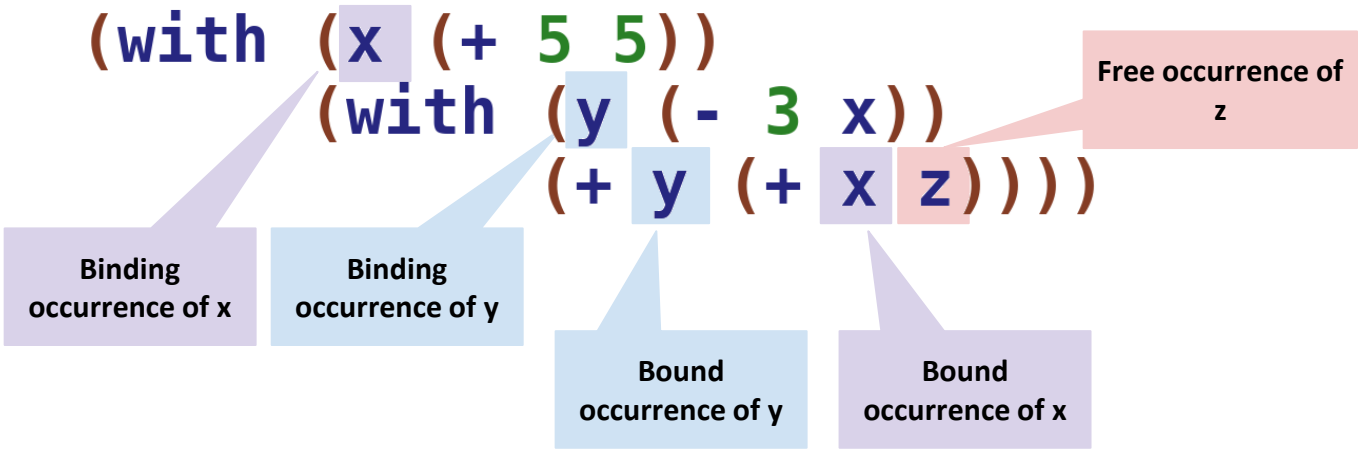
The concrete syntax, and consequently the parser, can express both meaningful and meaningless programs.

```
(with (x 5) (+ x x)) ⇝      10

(with (x 5) (+ x z)) ⇝      ??
```

The second expression cannot be interpreted because it contains so-called *free* ***identifiers***

Closed vs Open expressions



Binding occurrence: Occurrence that introduces the identifier into a **scope**.

Bound occurrence: Occurrence of a name within the scope of its binding instance.

Free occurrence: Occurrence of a name not contained in the scope of any binding instance.

Open expression: Expression with (at least) a free occurrence of an identifier

Closed expression: Expression with no free occurrences of identifiers

- Expressions with identifiers: semantics

Semantics of local identifiers — Intuition —

```
(with (x (+ 5 5)) (with (y (- 3 x)) (+ x y)))
```

[+ operation]

```
(with (x 10) (with (y (- 3 x)) (+ x y)))
```

[substitution and descent]

```
(with (y (- 3 10)) (+ 10 y))
```

[- operation]

```
(with (y -7) (+ 10 y))
```

[substitution and descent]

```
(+ 10 -7)
```

[+ operation]

```
3
```

Reducing an expression with identifiers involves a mixture of **evaluation** and **substitution**.

Semantics of local identifiers — Shadowing —

To which value should the following expression reduce?

```
(with (x 0) (with (x 1) x))
```
$\rightsquigarrow$ `1`

When there is a scope collision for a bounded occurrence of an identifier, the inner-most scope takes precedence.

Therefore, when evaluating

```
(with (id named-expr) body)
```

we have to substitute only the free occurrences of id in body.

Understanding substitution

Let us see the step-by-step interpretation/reduction of the following programs

```
'{with {x 5} {+ x x}}
```

Understanding substitution

Let us see the step-by-step interpretation/reduction of the following programs

```
'{with {x {+ 5 5}}
       {+ x x}}
```

Understanding substitution

Let us see the step-by-step interpretation/reduction of the following programs

```
'{with {x 10}
       {with {x 1} x}}
```

Understanding substitution

Let us see the step-by-step interpretation/reduction of the following programs

```
'{with {x 10}
       {with {x x} {+ x x}}}
```

Understanding substitution

Let us see the step-by-step interpretation/reduction of the following programs

```
'{with {x 5}
       {with {y x}
             y}}
```

Understanding substitution

Let us see the step-by-step interpretation/reduction of the following programs

```
'{with {x 5} {+ x {with {x 3} {+ x x}}}}
```

Understanding substitution

Let us see the step-by-step interpretation/reduction of the following programs

```
'{with {x 5} {+ x {with {y 3} {+ y x}}}}
```

Understanding substitution

Let us see the step-by-step interpretation/reduction of the following programs

```
'{with {x 5} {+ {with {x 10} {+ x x}}
                {with {y {+ x x}} {+ y x}}}}
```

Understanding substitution

Let us see the step-by-step interpretation/reduction of the following programs

```
'{with {x {+ 5 5}}
       {with {y {- x 3}}
             {with {x {+ y x}}
                   {with {z {+ x y}}
                         {with {x z}
                               {+ x y}}}}}}
```

Evaluating expressions with identifiers

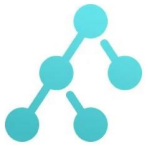
To evaluate an expression of the form

```
(with (id named-expr) body)
```

we proceed as follows:

- First, we compute the value of `named-expr`.
- Then we substitute in `body` the free occurrences of `id` with the value (of `named-expr`) above computed (and “get rid” of the identifier).
- Finally we (recursively) reduce the obtained expression.

Substitution function



Let's define the function that handles such substitution:

```
;; subst :: Expr Symbol Expr -> Expr
;; (subst in what for) substitutes all free occurrences
;; of identifier 'what' in expression 'in' for expression 'for'.
(define (subst in what for)
  (match in
    [(num n) (num n)]
    [(add l r) (add (subst l what for) (subst r what for))]
    [(sub l r) (sub (subst l what for) (subst r what for))]
    [(if0 c t f) (if0 (subst c what for)
                      (subst t what for)
                      (subst f what for))]
    [(with x e b)
     (with x
           (subst e what for)

    [(id x) (if (symbol=? x what) ;; is x the identifier to be substituted?
                for
                (id x))]))
```

Remember to add tests!

Evaluating expressions with identifiers

```
;; interp :: Expr -> number
;; Evaluates arithmetical expressions
;; with conditionals and identifiers.
(define (interp expr)
  (match expr
    [(num n) n]
    [(add l r) (+ (interp l) (interp r))]
    [(sub l r) (- (interp l) (interp r))]
    [(if0 c t f) (if (zero? (interp c))
                     (interp t) (interp f))]
    [(with x e b)
                                          ]
    [(id x) (error 'interp "Open expression (free occurrence of ~a)" x)]))
```

Remember to add tests!

Takeaways

On substitution

Reducing expressions with local definitions and function calls requires the fundamental notion of substitution.

Lecture material

Bibliography

- Programming Languages: Application and Interpretation (1st Edition)
  Shriram Krishnamurthi [Download]
  Chapters 3, 4

Source code

- Interpreter of arithmetical expressions with identifiers [Download]