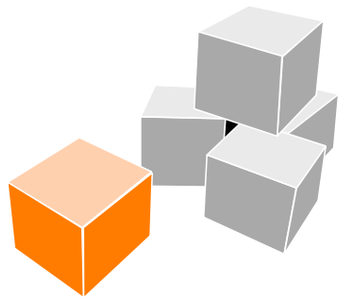
Concepts of Programming Languages

Prof. Dr. Guido Salvaneschi



Software Technology Group
TU Darmstadt | FB Informatik

COPL

SIMPLE LANGUAGES

– A language for arithmetic
– A language with names

Modeling Syntax

- Different notations for the idealized action of adding the idealized numbers (represented) by “3” and “4”:
  - `3 + 4` (infix) Java
  - `3 4 +` (postfix) FORTH
  - `(+ 3 4)` (parenthesized prefix) Scheme

- Ignoring details of concrete syntax, the essence is a tree (AST) …

- So the first question to answer in modeling languages is how to represent ASTs.

Syntax

- Concrete syntax
  - What the programmer writes
  - Comments, multiple spaces, newlines, …
- Abstract syntax
  - Internal representation of the syntax
  - Smaller to make automatic processing (e.g., type checking) and reasoning easier.
  - Example: Arithmetic Expressions

```
<AE> ::= <num>
         | {+ <AE> <AE>}
         | {- <AE> <AE>}
```

Case Classes for ASTs

AST for arithmetic expressions

```
sealed abstract class Expr

case class Num(n: Int) extends Expr
case class Add(lhs: Expr, rhs: Expr) extends Expr
case class Sub(lhs: Expr, rhs: Expr) extends Expr
```

Values of this data type:

```
Add(Num(3),Num(4))
Add(Sub(Num(3),Num(4)),Num(7))
```

Template for our Interpreters

```
def interp(expr: Expr): Int = expr match {
  case Num(n) => ???
  case Add(lhs, rhs) => ???
  case Sub(lhs, rhs) => ???
}
```

What goes into “???”

Template for our Interpreters

```
def interp(expr: Expr): Int = expr match {
  case Num(n) => ???
  case Add(lhs, rhs) => ???
  case Sub(lhs, rhs) => ???
}
```

What goes into “???”

```
def interp(expr: Expr): Int = expr match {
  case Num(n) => n
  case Add(lhs, rhs) => interp(lhs) + interp(rhs)
  case Sub(lhs, rhs) => interp(lhs) - interp(rhs)
}
```

Demo

- The AE interpreter

Next: WAE – a Language with Names

Motivation: reduce repetitions by introducing identifiers (not yet variables!)

Example programs:

```
let y = (5 + 10) in y + y
= (5 + 10) + (5 + 10)

let y = (5 + 10) in
  let x = 20 in (x + y)
= 20 + (5 + 10)
```

WAE: Abstract syntax

```
<AE> ::= <num>
       | {+ <AE> <AE>}
       | {- <AE> <AE>}
```

Extend with “let”

```
<WAE> ::= <num>
        | {+ <WAE> <WAE>}
        | {- <WAE> <WAE>}
        | {let {<id> <WAE>} <WAE>}
        | <id>
```

WAE: Concrete syntax

Quiz: What implementation steps are needed?

```
sealed abstract class Expr
case class Num(n: Int) extends Expr
case class Add(lhs: Expr, rhs: Expr) extends Expr
case class Sub(lhs: Expr, rhs: Expr) extends Expr

   ???
```

Extend with “let”

WAE: Concrete syntax

Quiz: What implementation steps are needed?

```
sealed abstract class Expr
case class Num(n: Int) extends Expr
case class Add(lhs: Expr, rhs: Expr) extends Expr
case class Sub(lhs: Expr, rhs: Expr) extends Expr

   ???
```

Extend with “let”

```
sealed abstract class Expr
case class Num(n: Int) extends Expr
case class Add(lhs: Expr, rhs: Expr) extends Expr
case class Sub(lhs: Expr, rhs: Expr) extends Expr

case class Let(name: Symbol, namedExpr: Expr, body: Expr) extends Expr
case class Id(name: Symbol) extends Expr
```

Substitution or „Name and Conquer“

Quiz: What implementation steps are needed?

```
<WAE> ::= <num>
        | {+ <WAE> <WAE>}
        | {- <WAE> <WAE>}
        | {let {<id> <WAE>} <WAE>}
        | <id>
```

```
def parse(prog: String): Expr = ...
```

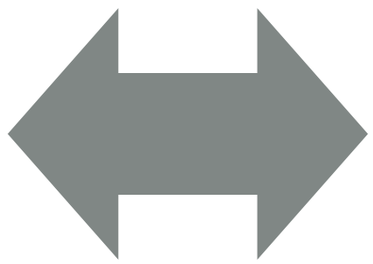
```
sealed abstract class Expr
case class Num(n: Int) extends Expr
case class Add(lhs: Expr, rhs: Expr) extends Expr
case class Sub(lhs: Expr, rhs: Expr) extends Expr

case class Let(name: Symbol, namedExpr: Expr, body: Expr) extends Expr
case class Id(name: Symbol) extends Expr
```

The interpreter …

Syntax

- We need to give a semantics to let expressions
- We do so using the concept of substitution

semantics of let  substitution

Defining Substitution

- Wanted: A definition of the process of *substitution*

- Here is one:

> *Definition (Substitution):*
>
> *To substitute identifier i in e with expression v, replace all identifier sub-expressions of e named i with v.*

- Try it out with the following WAE expressions:

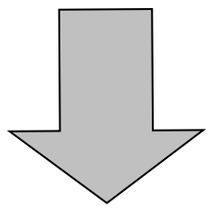
```
1. let x = 5 in x + x

2. let x = 5 in x + (let x = 3 in x)
```

Defining Substitution

```
1.  let x = 5 in x + x

2.  let x = 5 in x + (let x = 3 in x)
```



```
1.  5 + 5

2.  5 + (let 5 = 3 in 5)
```

This is not even syntactically legal!
-> it does not respect the BNF and
it would be rejected by a parser

Defining Substitution

Definition (Binding Instance):

A binding instance of an identifier is the instance of the identifier that gives it its value. In WAE, the <id> position of a ‘let’ is the only binding instance.

Definition (Scope)

The scope of a binding instance is the region of program in which instances of the identifier refer to the value bound by the binding instance.

Definition (Bound Instance)

An identifier is bound if it is contained within the scope of a binding instance of its name.

Definition (Free Instance)

An identifier not contained in the scope of any binding instance of its name is said to be free.

Defining Substitution

```
let x = 5 in
  x + (let x = 3 in
         x + x)
```

```
5 + (let x = 3 in
       5 + 5)
```

```
5 + (5 + 5)
```

```
   15
```

What can go wrong here?

-> We do not respect scoping

Defining Substitution

Definition (Binding Instance):

A binding instance of an identifier is the instance of the identifier that gives it its value. In WAE, the <id> position of a 'let' is the only binding instance.

Definition (Scope)

The scope of a binding instance is the region of program in which instances of the identifier refer to the value bound by the binding instance.

Definition (Bound Instance)

An identifier is bound if it is contained within the scope of a binding instance of its name.

Definition (Free Instance)

An identifier not contained in the scope of any binding instance of its name is said to be free.

```
let x = 5 in
  x + (let x = 3 in
         x + y)
```

Defining Substitution

Definition (Substitution):

To substitute identifier i in e with expression v, replace all free instances of i in e with v.

- This definition is implicitly using a notion of scope
- Substitute only in the scope of the identifier
- Respected binding instances, but not their scope.
- An inner binding for the same name introduces a new scope. The scope of the outer binding is shadowed or masked by the inner binding.
- Substituting the inner x is wrong.

Calculating WAE Expressions

```
def interp(expr: Expr): Int = expr match {
  case Num(n) => n
  case Add(lhs, rhs) => calc(lhs) + calc(rhs)
  case Sub(lhs, rhs) => calc(lhs) - calc(rhs)
  case Let(boundId, namedExpr, boundExpr) => ???
  case Id(name) => ???
 }
```

Use a “subst" function

```
def subst(expr: WAE, substId: Symbol, value: WAE)
```

```
def interp(expr: Expr): Int = expr match {
  case Num(n) => n
  case Add(lhs, rhs) => interp(lhs) + interp(rhs)
  case Sub(lhs, rhs) => interp(lhs) - interp(rhs)
  case Let(boundId, namedExpr, boundExpr) => {
    interp(subst(boundExpr, boundId, Num(interp(namedExpr))))
  }
  case Id(name) => sys.error("found unbound id " + name)
 }
```

Calculating WAE Expressions

- Any identifier that is in the scope of a let-expr is replaced with a value when the calculator encounters that identifier's binding instance.

- Consequently, the purpose statement of substitution said there would be no free instances of the identifier given as an argument left in the result.

- In other words, subst replaces identifiers with values before the calculator ever "sees" them.

- The calculator can't assign a value to a free identifier, so it halts with an error

Calculating WAE Expressions

```
def subst(expr: Expr, substId: Symbol, value: Expr): Expr = expr match {
  case Num(n) => ???
  case Add(lhs, rhs) => ???
  case Sub(lhs, rhs) => ???

  case Let(boundId, namedExpr, boundExpr) => ???

  case Id(name) => ...
}
```

Calculating WAE Expressions

```
def subst(expr: Expr, substId: Symbol, value: Expr): Expr = expr match {
  case Num(n) => expr
  case Add(lhs, rhs) => Add(subst(lhs, substId, value), subst(rhs, substId, value))
  case Sub(lhs, rhs) => Sub(subst(lhs, substId, value), subst(rhs, substId, value))

  case Let(boundId, namedExpr, boundExpr) => ???

  case Id(name) => ...
}
```

Calculating WAE Expressions

```
def subst(expr: Expr, substId: Symbol, value: Expr): Expr = expr match {
  case Num(n) => expr
  case Add(lhs, rhs) => Add(subst(lhs, substId, value), subst(rhs, substId, value))
  case Sub(lhs, rhs) => Sub(subst(lhs, substId, value), subst(rhs, substId, value))

  case Let(boundId, namedExpr, boundExpr) => {
      val substNamedExpr = subst(namedExpr, substId, value)
      Let(boundId, substNamedExpr, subst(boundExpr, substId, value))
    }

 case Id(name) => ...
}
```

What is wrong with this one?

Calculating WAE Expressions

```
def subst(expr: Expr, substId: Symbol, value: Expr): Expr = expr match {
  case Num(n) => expr
  case Add(lhs, rhs) => Add(subst(lhs, substId, value), subst(rhs, substId, value))
  case Sub(lhs, rhs) => Sub(subst(lhs, substId, value), subst(rhs, substId, value))

  case Let(boundId, namedExpr, boundExpr) => {
      val substNamedExpr = subst(namedExpr, substId, value)
      if (boundId == substId)
        Let(boundId, substNamedExpr, boundExpr)
      else
        Let(boundId, substNamedExpr, subst(boundExpr, substId, value))
    }

 case Id(name) => {
      if (substId == name) value
      else expr
    }
}
```

Two Substitution Regimes

Eager substitution (static and dynamic reduction): avoids re-computing the same value several times.

1

```
  {let {x {+ 5 5}} {let {y {- x 3}} {+ y y}}}
= {let {x 10} {let {y {- x 3}} {+ y y}}}
= {let {y {- 10 3}} {+ y y}}
= {let {y 7} {+ y y}}
= {+ 7 7}
= 14
```

Lazy substitution (Static reduction): the expression may be evaluated multiple times.

2

```
  {let {x {+ 5 5}} {let {y {- x 3}} {+ y y}}}
= {let {y {- {+ 5 5} 3}} {+ y y}}
= {+ {- {+ 5 5} 3} {- {+ 5 5} 3}}
= {+ {- 10 3} {- {+ 5 5} 3}}
= {+ {- 10 3} {- 10 3}}
= {+ 7 {- 10 3}}
= {+ 7 7}
= 14
```

Two Substitution Regimes

- Questions:

1. Which one have we implemented?

```
def interp(expr: Expr): Int = expr match {
 …
  case Let(boundId, namedExpr, boundExpr) => {
    interp(subst(boundExpr, boundId, Num(interp(namedExpr))))
  }
 …
```

2. Our example suggests that the eager regime generates an answer in fewer steps. Is this always true?

```
{let {x {+ 5 5}}
  {let {y 4} {+ y y}}}
```

3. Do the two regimes always produce the same result for WAE?

```
{let {x {+ z 4}}
  {let {y 4} {+ y y}}}
```

Demo

- The WAE interpreter

FOOD FOR THOUGHTS

PL success: Some Ideas

- Large popular software systems use language X. (E.g., Unix for C)

- Companies push for it.

- Market pressure. A popular language is requested and becomes even more popular

- A language introduces new concepts
  - Not necessarily applicable from day 1

More sources

- What makes a programming language
  - Popular
  - Important (?)
  - Inspiring (??)

The TIOBE index

- The TIOBE Programming Community index is an indicator of the popularity of programming languages.
  - Popular search engines are used to calculate the ratings.
  - It is not about the best programming language or the language in which most lines of code have been written.
  - http://www.tiobe.com/tiobe-index/

- Any guess?

IEEE Spectrum

- IEEE Spectrum: The Top Programming Languages 2016
  - Popular search engines, twitter, github, stack overflow, …
    - http://spectrum.ieee.org/ns/IEEE_TPL_2016/methods.html
  - http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2016

More sources

- http://githut.info
- https://github.com/blog/2047-language-trends-on-github

Disclaimer: Please take all these measures with a grain of salt (Better, with some skepticism!)