CS320

First-Order Functions

Sukyoung Ryu

March 13, 2019

Q&A

- Shadowing?

```
{with {x {with {x 3} {- 5 x}}}
      {+ 1 x}}
```

```
{with {x 3}
      {with {x 5} {+ 1 x}}}
```

Functions

- $identity(x) = x$
- $twice(x) = x + x$

Functions

- $identity(x) = x$
- $twice(x) = x + x$
- `AE`

```
{- 20 {+ 10 10}}
{- 20 {+ 17 17}}
{- 20 {+ 3 3}}
```

Functions

- $identity(x) = x$
- $twice(x) = x + x$
- AE

```
{- 20 {+ 10 10}}
{- 20 {+ 17 17}}
{- 20 {+ 3 3}}
```

WAE

```
{with {x 10} {- 20 {+ x x}}}
{with {x 17} {- 20 {+ x x}}}
{with {x  3} {- 20 {+ x x}}}
```

- F1WAE

```
{deffun {identity x}
  x}
{identity 8}
```

```
{deffun {twice x}
  {+ x x}}
{twice 10}
{twice 17}
```

Functions

- $identity(x) = x$
- $twice(x) = x + x$
- AE WAE

```
{- 20 {+ 10 10}}    {with {x 10} {- 20 {+ x x}}}
{- 20 {+ 17 17}}    {with {x 17} {- 20 {+ x x}}}
{- 20 {+ 3 3}}      {with {x  3} {- 20 {+ x x}}}
```

- F1WAE

```
{deffun {identity x}    {deffun {twice x}
  x}                      {+ x x}}
{identity 8}            {twice 10}
                        {twice 17}
```

Functions

- $identity(x) = x$
- $twice(x) = x + x$
- AE WAE

```
{- 20 {+ 10 10}}    {with {x 10} {- 20 {+ x x}}}
{- 20 {+ 17 17}}    {with {x 17} {- 20 {+ x x}}}
{- 20 {+ 3 3}}      {with {x  3} {- 20 {+ x x}}}
```

- F1WAE

```
{deffun {identity x}      {deffun {twice x}
  x}                        {+ x x}}
{identity 8}              {twice 10}
                          {twice 17}
```

Functions

- $identity(x) = x$
- $twice(x) = x + x$
- AE WAE

```
{- 20 {+ 10 10}}    {with {x 10} {- 20 {+ x x}}}
{- 20 {+ 17 17}}    {with {x 17} {- 20 {+ x x}}}
{- 20 {+ 3 3}}      {with {x  3} {- 20 {+ x x}}}
```

- F1WAE

```
{deffun {identity x}
  x}
{identity 8}
```

```
{deffun {twice x}
  {+ x x}}
{twice 10}
{twice 17}
```

F1WAE: Concrete Syntax

```
<FunDef> ::= {deffun {<id> <id>} <F1WAE>}

<F1WAE> ::= <num>
          | {+ <F1WAE> <F1WAE>}
          | {- <F1WAE> <F1WAE>}
          | {with {<id> <F1WAE>} <F1WAE>}
          | <id>
          | {<id> <F1WAE>}
```

F1WAE: Concrete Syntax

```
<FunDef> ::= {deffun {<id> <id>} <F1WAE>}

<F1WAE> ::= <num>
          | {+ <F1WAE> <F1WAE>}
          | {- <F1WAE> <F1WAE>}
          | {with {<id> <F1WAE>} <F1WAE>}
          | <id>
          | {<id> <F1WAE>}
```

```
{deffun {twice x} {+ x x}}
{- 20 {twice 10}}
{- 20 {twice 17}}
{- 20 {twice 3}}
```

FWAE: Concrete Syntax

```
<FWAE> ::= <num>
         | {+ <FWAE> <FWAE>}
         | {- <FWAE> <FWAE>}
         | {with {<id> <FWAE>} <FWAE>}
         | <id>
         | {<FWAE> <FWAE>}
         | {fun {<id>} <FWAE>}
```

F1WAE: Concrete Syntax

```
<FunDef> ::= {deffun {<id> <id>} <F1WAE>}

<F1WAE> ::= <num>
          | {+ <F1WAE> <F1WAE>}
          | {- <F1WAE> <F1WAE>}
          | {with {<id> <F1WAE>} <F1WAE>}
          | <id>
          | {<id> <F1WAE>}
```

F1WAE: Concrete Syntax

```
<FunDef> ::= {deffun {<id> <id>} <F1WAE>}

<F1WAE> ::= <num>
          | {+ <F1WAE> <F1WAE>}
          | {- <F1WAE> <F1WAE>}
          | {with {<id> <F1WAE>} <F1WAE>}
          | <id>
          | {<id> <F1WAE>}
```

```
{deffun {twice x} {+ x x}}
{- 20 {twice 10}}
{- 20 {twice 17}}
{- 20 {twice 3}}
```

F1WAE: Abstract Syntax

```
case class FunDef(f: String, x: String, b: F1WAE)

trait F1WAE
case class Num(n: Int) extends F1WAE
case class Add(l: F1WAE, r: F1WAE) extends F1WAE
case class Sub(l: F1WAE, r: F1WAE) extends F1WAE
case class With(x: String, i: F1WAE, b: F1WAE)
           extends F1WAE
case class Id(x: String) extends F1WAE
case class App(f: String, a: F1WAE) extends F1WAE
```

F1WAE: Parser

```
// parser for FunDef
```

```
// parser for F1WAE
object F1WAE {
  lazy val expr: Parser[F1WAE] =
    int
    ...
    wrap(str ~ expr) ^^ { case f ~ a => App(f, a) }
  def apply(str: String): F1WAE =
    parse(expr, str).getOrElse(error(s"bad syntax: $str"))
}
```

F1WAE: Parser

```
// parser for FunDef
object FunDef {
    lazy val fundef: Parser[FunDef] =
      wrap("deffun" ~> wrap(str ~ str) ~ F1WAE.expr)
                    ^^ { case f ~ x ~ e => FunDef(f, x, e) }
    def apply(str: String): FunDef =
      parse(fundef, str).getOrElse(error(s"bad syntax: $str"))
  }
// parser for F1WAE
object F1WAE {
  lazy val expr: Parser[F1WAE] =
    int
    ...
    wrap(str ~ expr) ^^ { case f ~ a => App(f, a) }
  def apply(str: String): F1WAE =
    parse(expr, str).getOrElse(error(s"bad syntax: $str"))
}
```

F1WAE: Parser

```
// parser for FunDef
object FunDef {
    lazy val fundef: Parser[FunDef] =
      wrap("deffun" ~> wrap(str ~ str) ~ F1WAE.expr)
                    ^^ { case f ~ x ~ e => FunDef(f, x, e) }
    def apply(str: String): FunDef =
      parse(fundef, str).getOrElse(error(s"bad syntax: $str"))
  }
// parser for F1WAE
object F1WAE {
  lazy val expr: Parser[F1WAE] =
    int
    ...
    wrap(str ~ expr) ^^ { case f ~ a => App(f, a) }
  def apply(str: String): F1WAE =
    parse(expr, str).getOrElse(error(s"bad syntax: $str"))
}
```

F1WAE: Interpreter

```
type FDs = List[FunDef]
// interp : (F1WAE, Env, FDs) => Int
```

F1WAE: Interpreter

```
type FDs = List[FunDef]
// interp : (F1WAE, Env, FDs) => Int
def interp(e: F1WAE, env: Env, fs: FDs): Int = e match {
  case Num(n) => n
  case Add(l, r) => interp(l, env, fs) + interp(r, env, fs)
  case Sub(l, r) => interp(l, env, fs) - interp(r, env, fs)
  case With(x, i, b) =>
                interp(b, env + (x -> interp(i, env, fs)), fs)
  case Id(x) => lookup(x, env)
  case App(f, a) => ...
}
```

F1WAE: Interpreter

```
type FDs = List[FunDef]
// interp : (F1WAE, Env, FDs) => Int
def interp(e: F1WAE, env: Env, fs: FDs): Int = e match {
  case Num(n) => n
  case Add(l, r) => interp(l, env, fs) + interp(r, env, fs)
  case Sub(l, r) => interp(l, env, fs) - interp(r, env, fs)
  case With(x, i, b) =>
                interp(b, env + (x -> interp(i, env, fs)), fs)
  case Id(x) => lookup(x, env)
  case App(f, a) => ...
}
```

Let's make examples/tests first...

F1WAE: Interpreter

```
test(interp(F1WAE("{+ 1 1}"), Map(),
            List()),
     ?)
```

```
type FDs = List[FunDef]
// interp : (F1WAE, Env, FDs) => Int
def interp(e: F1WAE, env: Env, fs: FDs): Int = e match {
  case Num(n) => n
  case Add(l, r) => interp(l, env, fs) + interp(r, env, fs)
  case Sub(l, r) => interp(l, env, fs) - interp(r, env, fs)
  case With(x, i, b) =>
                 interp(b, env + (x -> interp(i, env, fs)), fs)
  case Id(x) => lookup(x, env)
  case App(f, a) => ...
}
```

F1WAE: Interpreter

```
test(interp(F1WAE("{+ 1 1}"), Map(),
            List()),
     2)
```

```
type FDs = List[FunDef]
// interp : (F1WAE, Env, FDs) => Int
def interp(e: F1WAE, env: Env, fs: FDs): Int = e match {
  case Num(n) => n
  case Add(l, r) => interp(l, env, fs) + interp(r, env, fs)
  case Sub(l, r) => interp(l, env, fs) - interp(r, env, fs)
  case With(x, i, b) =>
                 interp(b, env + (x -> interp(i, env, fs)), fs)
  case Id(x) => lookup(x, env)
  case App(f, a) => ...
}
```

F1WAE: Interpreter

```
test(interp(F1WAE("{+ 1 1}"), Map(),
            List(FunDef("f", "x", F1WAE("{+ x 3}")))),
     ?)
```

```
type FDs = List[FunDef]
// interp : (F1WAE, Env, FDs) => Int
def interp(e: F1WAE, env: Env, fs: FDs): Int = e match {
  case Num(n) => n
  case Add(l, r) => interp(l, env, fs) + interp(r, env, fs)
  case Sub(l, r) => interp(l, env, fs) - interp(r, env, fs)
  case With(x, i, b) =>
                 interp(b, env + (x -> interp(i, env, fs)), fs)
  case Id(x) => lookup(x, env)
  case App(f, a) => ...
}
```

F1WAE: Interpreter

```
test(interp(F1WAE("{+ 1 1}"), Map(),
            List(FunDef("f", "x", F1WAE("{+ x 3}")))),
     2)
```

```
type FDs = List[FunDef]
// interp : (F1WAE, Env, FDs) => Int
def interp(e: F1WAE, env: Env, fs: FDs): Int = e match {
  case Num(n) => n
  case Add(l, r) => interp(l, env, fs) + interp(r, env, fs)
  case Sub(l, r) => interp(l, env, fs) - interp(r, env, fs)
  case With(x, i, b) =>
                interp(b, env + (x -> interp(i, env, fs)), fs)
  case Id(x) => lookup(x, env)
  case App(f, a) => ...
}
```

F1WAE: Interpreter

```
test(interp(F1WAE("{f 1}"), Map(),
            List(FunDef("f", "x", F1WAE("{+ x 3}")))),
     ?)
```

```
type FDs = List[FunDef]
// interp : (F1WAE, Env, FDs) => Int
def interp(e: F1WAE, env: Env, fs: FDs): Int = e match {
  case Num(n) => n
  case Add(l, r) => interp(l, env, fs) + interp(r, env, fs)
  case Sub(l, r) => interp(l, env, fs) - interp(r, env, fs)
  case With(x, i, b) =>
                 interp(b, env + (x -> interp(i, env, fs)), fs)
  case Id(x) => lookup(x, env)
  case App(f, a) => ...
}
```

F1WAE: Interpreter

```
test(interp(F1WAE("{f 1}"), Map(),
            List(FunDef("f", "x", F1WAE("{+ x 3}")))),
     4)
```

```
type FDs = List[FunDef]
// interp : (F1WAE, Env, FDs) => Int
def interp(e: F1WAE, env: Env, fs: FDs): Int = e match {
  case Num(n) => n
  case Add(l, r) => interp(l, env, fs) + interp(r, env, fs)
  case Sub(l, r) => interp(l, env, fs) - interp(r, env, fs)
  case With(x, i, b) =>
                 interp(b, env + (x -> interp(i, env, fs)), fs)
  case Id(x) => lookup(x, env)
  case App(f, a) => ...
}
```

F1WAE: Interpreter

```
test(interp(F1WAE("{f 10}"), Map(),
            List(FunDef("f", "x", F1WAE("{- 20 {twice x}}")),
                 FunDef("twice", "y", F1WAE("{+ y y}")))),
     ?)
```

```
type FDs = List[FunDef]
// interp : (F1WAE, Env, FDs) => Int
def interp(e: F1WAE, env: Env, fs: FDs): Int = e match {
  case Num(n) => n
  case Add(l, r) => interp(l, env, fs) + interp(r, env, fs)
  case Sub(l, r) => interp(l, env, fs) - interp(r, env, fs)
  case With(x, i, b) =>
                 interp(b, env + (x -> interp(i, env, fs)), fs)
  case Id(x) => lookup(x, env)
  case App(f, a) => ...
}
```

F1WAE: Interpreter

```
test(interp(F1WAE("{f 10}"), Map(),
            List(FunDef("f", "x", F1WAE("{- 20 {twice x}}")),
                 FunDef("twice", "y", F1WAE("{+ y y}")))),
     0)
```

```
type FDs = List[FunDef]
// interp : (F1WAE, Env, FDs) => Int
def interp(e: F1WAE, env: Env, fs: FDs): Int = e match {
  case Num(n) => n
  case Add(l, r) => interp(l, env, fs) + interp(r, env, fs)
  case Sub(l, r) => interp(l, env, fs) - interp(r, env, fs)
  case With(x, i, b) =>
                 interp(b, env + (x -> interp(i, env, fs)), fs)
  case Id(x) => lookup(x, env)
  case App(f, a) => ...
}
```

F1WAE: Interpreter

```
type FDs = List[FunDef]
// interp : (F1WAE, Env, FDs) => Int
def interp(e: F1WAE, env: Env, fs: FDs): Int = e match {
  case Num(n) => n
  case Add(l, r) => interp(l, env, fs) + interp(r, env, fs)
  case Sub(l, r) => interp(l, env, fs) - interp(r, env, fs)
  case With(x, i, b) =>
                interp(b, env + (x -> interp(i, env, fs)), fs)
  case Id(x) => lookup(x, env)
  case App(f, a) => ... interp(a, env, fs) ...
}
```

F1WAE: Interpreter

```
type FDs = List[FunDef]
// interp : (F1WAE, Env, FDs) => Int
def interp(e: F1WAE, env: Env, fs: FDs): Int = e match {
  case Num(n) => n
  case Add(l, r) => interp(l, env, fs) + interp(r, env, fs)
  case Sub(l, r) => interp(l, env, fs) - interp(r, env, fs)
  case With(x, i, b) =>
                interp(b, env + (x -> interp(i, env, fs)), fs)
  case Id(x) => lookup(x, env)
  case App(f, a) => ... lookupFD(f, fs) ...
                    ... interp(a, env, fs) ...
}
// lookupFD : (string, FDs) => FunDef
```

Lookup

```
// lookupFD: (string, FDs) => FunDef
def lookupFD(name: String, fs: FDs): FunDef =
  ...
```

Lookup

```
// lookupFD: (string, FDs) => FunDef
def lookupFD(name: String, fs: FDs): FunDef = fs match {
  case Nil =>
    ...
  case h :: t =>
    ... h ...
    ... lookupFD(name, t) ...
}
```

Lookup

```
// lookupFD: (string, FDs) => FunDef
def lookupFD(name: String, fs: FDs): FunDef = fs match {
  case Nil =>
    error(s"lookupFD: unknown function: $name")
  case h :: t =>
    if (h.f == name) h
    else lookupFD(name, t)
}
```

F1WAE: Interpreter

```
type FDs = List[FunDef]
// interp : (F1WAE, Env, FDs) => Int
def interp(e: F1WAE, env: Env, fs: FDs): Int = e match {
  case Num(n) => n
  case Add(l, r) => interp(l, env, fs) + interp(r, env, fs)
  ...
  case App(f, a) => ... lookupFD(f, fs) ...
                    ... interp(a, env, fs) ...
}
// lookupFD : (string, FDs) => FunDef
```

F1WAE: Interpreter

```
type FDs = List[FunDef]
// interp : (F1WAE, Env, FDs) => Int
def interp(e: F1WAE, env: Env, fs: FDs): Int = e match {
  case Num(n) => n
  case Add(l, r) => interp(l, env, fs) + interp(r, env, fs)
  ...
  case App(f, a) => lookupFD(f, fs) match {
    case FunDef(fname, pname, body) =>
      val aval = interp(a, env, fs)
      ...
  }
}
```

F1WAE: Interpreter

```
type FDs = List[FunDef]
// interp : (F1WAE, Env, FDs) => Int
def interp(e: F1WAE, env: Env, fs: FDs): Int = e match {
  case Num(n) => n
  case Add(l, r) => interp(l, env, fs) + interp(r, env, fs)
  ...
  case App(f, a) => lookupFD(f, fs) match {
    case FunDef(fname, pname, body) =>
      val aval = interp(a, env, fs)
      interp(body, ..., fs)
  }
}
```

F1WAE: Interpreter

```
type FDs = List[FunDef]
// interp : (F1WAE, Env, FDs) => Int
def interp(e: F1WAE, env: Env, fs: FDs): Int = e match {
  case Num(n) => n
  case Add(l, r) => interp(l, env, fs) + interp(r, env, fs)
  ...
  case App(f, a) => lookupFD(f, fs) match {
    case FunDef(fname, pname, body) =>
      val aval = interp(a, env, fs)
      interp(body, ..., fs)
  }
}
```

Let's make more examples...

Function Calls

```
{deffun {f x} {+ 1 x}}
interp(F1WAE("{with {y 2} {f 10}}"  []))
```

⇒

```
interp(F1WAE("{f 10}"  [y=2]))
```

⇒

```
interp(F1WAE("{+ 1 x}"  [...]))
```

Interpreting function body with only one piece of information

Function Calls

```
{deffun {f x} {+ 1 x}}
interp(F1WAE("{with {y 2} {f 10}}"  []))
```

$\Rightarrow$

```
interp(F1WAE("{f 10}"  [y=2]))
```

$\Rightarrow$

```
interp(F1WAE("{+ 1 x}"  [...]))
```

Interpreting function body with only one piece of information

Function Calls

```
{deffun {f x} {+ 1 x}}
interp(F1WAE("{with {y 2} {f 10}}"  []))
⇒
interp(F1WAE("{f 10}"  [y=2]))
⇒
interp(F1WAE("{+ 1 x}"  [...]))
```

Interpreting function body with only one piece of information

Function Calls

```
{deffun {f x} {+ 1 x}}
interp(F1WAE("{with {y 2} {f 10}}"  []))
⇒
interp(F1WAE("{f 10}"  [y=2]))
⇒
interp(F1WAE("{+ 1 x}"  [...]))
```

Interpreting function body with only one piece of information

Function Calls

What goes wrong if you extend the old environment?

```
{deffun {f x} {+ y x}}
interp(F1WAE("{with {y 2} {f 10}}"  []))
⇒
interp(F1WAE("{f 10}"  [y=2]))
⇒
interp(F1WAE("{+ y x}"  [x=10 y=2]))
⇒
12      wrong!
```

Function Calls

What goes wrong if you extend the old environment?

```
{deffun {f x} {+ y x}}
interp(F1WAE("{with {y 2} {f 10}}"  []))
⇒
interp(F1WAE("{f 10}"  [y=2]))
⇒
interp(F1WAE("{+ y x}"  [x=10 y=2]))
⇒
12      wrong!
```

Function Calls

What goes wrong if you extend the old environment?

```
{deffun {f x} {+ y x}}
interp(F1WAE("{with {y 2} {f 10}}"  []))
⇒
interp(F1WAE("{f 10}"  [y=2]))
⇒
interp(F1WAE("{+ y x}"  [x=10 y=2]))
⇒
12        wrong!
```

Function Calls

What goes wrong if you extend the old environment?

```
{deffun {f x} {+ y x}}
interp(F1WAE("{with {y 2} {f 10}}"  []))
⇒
interp(F1WAE("{f 10}"  [y=2]))
⇒
interp(F1WAE("{+ y x}"  [x=10]))
⇒
"free var: y"
```

Interpreting function body with only one piece of information

Function Calls

What goes wrong if you extend the old environment?

```
{deffun {f x} {+ y x}}
interp(F1WAE("{with {y 2} {f 10}}"  []))
⇒
interp(F1WAE("{f 10}"  [y=2]))
⇒
interp(F1WAE("{+ y x}"  [x=10]))
⇒
"free var: y"
```

Interpreting function body with only one piece of information

Function Calls

What goes wrong if you extend the old environment?

```
{deffun {f x} {+ y x}}
interp(F1WAE("{with {y 2} {f 10}}"  []))
⇒
interp(F1WAE("{f 10}"  [y=2]))
⇒
interp(F1WAE("{+ y x}"  [x=10]))
⇒
"free var: y"
```

Interpreting function body with only one piece of information

Scope

- Static scope
  In a language with static scope, the scope of an identifier's binding is a syntactically delimited region.

- Dynamic scope
  In a language with dynamic scope, the scope of an identifier's binding is the entire remainder of the execution during which that binding is in effect.

Scope

```
{deffun {f p} n}
{with {n 5} {f 10}}
```

- Static scope
  In a language with static scope, the scope of an identifier's binding is a syntactically delimited region.

- Dynamic scope
  In a language with dynamic scope, the scope of an identifier's binding is the entire remainder of the execution during which that binding is in effect.

Scope

```
{deffun {f p} n}
{with {n 5} {f 10}}
```

- Static scope
  In a language with static scope, the scope of an identifier's binding is a syntactically delimited region.
  The code signals an error.
- Dynamic scope
  In a language with dynamic scope, the scope of an identifier's binding is the entire remainder of the execution during which that binding is in effect.
  The code evaluates to 5.

F1WAE: Interpreter

```
type FDs = List[FunDef]
// interp : (F1WAE, Env, FDs) => Int
def interp(e: F1WAE, env: Env, fs: FDs): Int = e match {
  ...
  case App(f, a) => lookupFD(f, fs) match {
    case FunDef(fname, pname, body) =>
      val aval = interp(a, env, fs)
      interp(body, ..., fs)
  }
}
```

F1WAE: Interpreter

```
type FDs = List[FunDef]
// interp : (F1WAE, Env, FDs) => Int
def interp(e: F1WAE, env: Env, fs: FDs): Int = e match {
  ...
  case App(f, a) => lookupFD(f, fs) match {
    case FunDef(fname, pname, body) =>
      val aval = interp(a, env, fs)
      interp(body, _____ + (pname -> aval), fs)
  }
}
```

F1WAE: Interpreter

```
type FDs = List[FunDef]
// interp : (F1WAE, Env, FDs) => Int
def interp(e: F1WAE, env: Env, fs: FDs): Int = e match {
  ...
  case App(f, a) => lookupFD(f, fs) match {
    case FunDef(fname, pname, body) =>
      val aval = interp(a, env, fs)
      interp(body, Map() + (pname -> aval), fs)
  }
}
```

Homework #2

- Available from the course webpage
- Due Wednesday, March 20 (before midnight)

Sukyoung Ryu
sryu.cs@kaist.ac.kr
http://plrg.kaist.ac.kr