Case Studies

David Basin

Department of Computer Science
ETH Zurich

Overview

- Present **interpreters** for **arithmetic** and for **mini-Haskell**

  Important: Interpreters are everywhere!

  Programming languages, database systems, text processors, hardware description languages, search engines, . . .

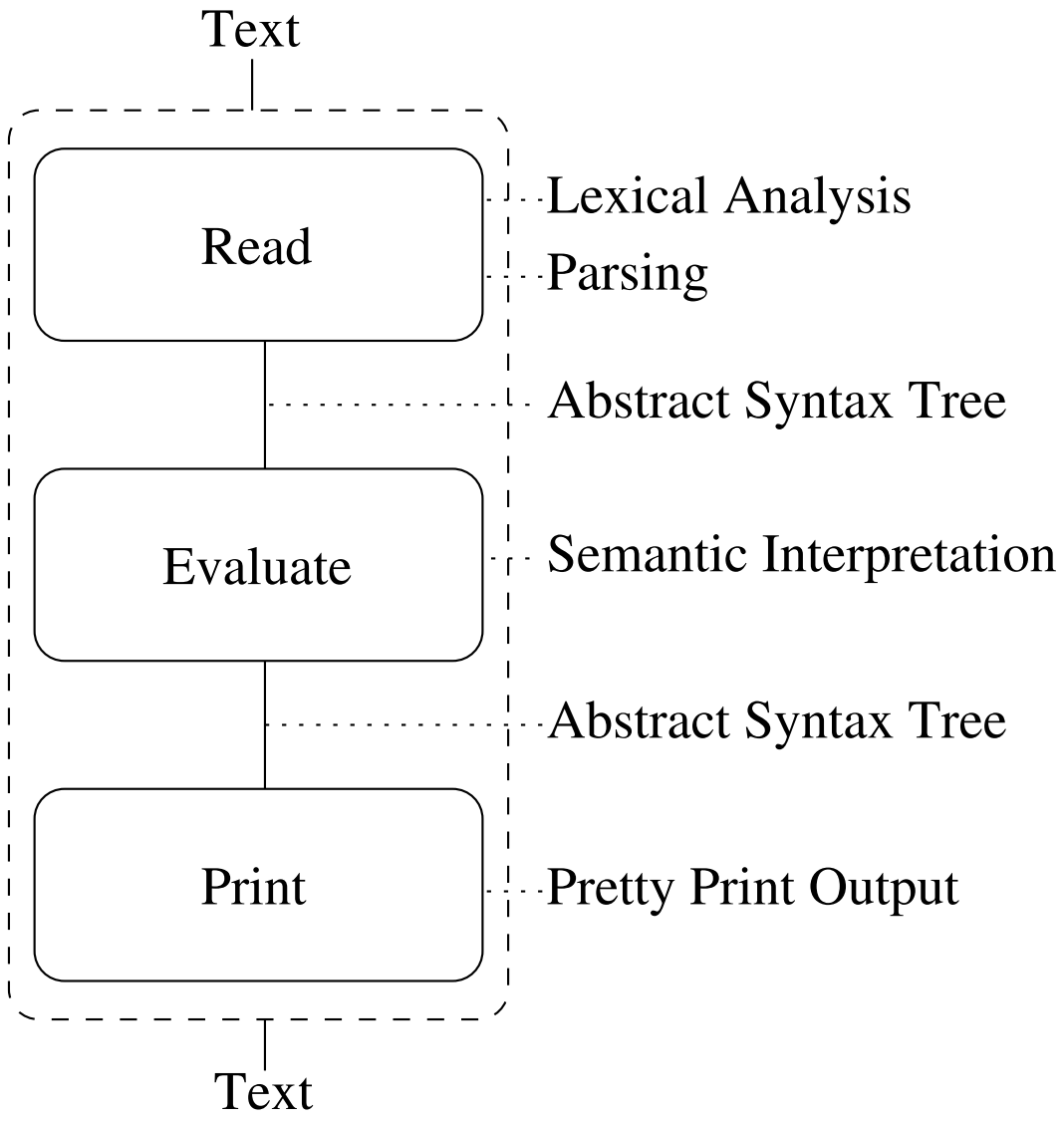
  Conceptually simple: read, evaluate, print

  Concretization less trivial: elegant application for higher-order programming and lazy evaluation

- An interpreter for mini-Haskell illustrates Haskell itself

- Source code available from course web-page. Try it!

Interpreter skeleton



Overview — lexical analysis

- Convert source code to tokens

```
position := initial + rate + 60
```

- Code translated to:

1. Identifier `position`
2. Assignment symbol `:=`
3. Identifier `initial`
4. Addition symbol `+`
5. Identifier `rate`
6. Addition symbol `+`
7. Number `60`

- White-spaces and comments are removed

Overview — parsing

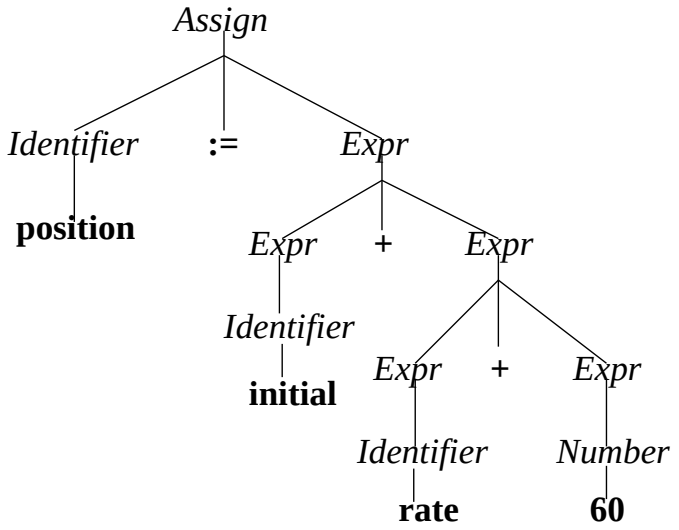
- Syntax specified by a grammar

$$\begin{array}{rcl} Expr & ::= & Identifier \mid Number \mid Expr\ \text{`+'}\ Expr \\ Assign & ::= & Identifier\ \text{`:='}\ Expr \end{array}$$

- Corresponds to a data type in Haskell

```
data Expr   = Identifier Ident | Number Num | Plus Expr Expr
data Assign = Assignment Ident Expr
type Ident  = String
type Num    = Int
```

- Parser constructs an "abstract syntax tree"
  - In Haskell: element of data type



parse tree for **position := initial + rate + 60**

Overview — other phases

- Further processing depends on application

  In general: conversion between data types, e.g.,

  $$\begin{array}{ll} \textit{Compiler:} & \textit{AST} \rightarrow \textit{CODE} \\ \textit{Calculator:} & \textit{AST} \rightarrow \textit{Int} \\ \textit{Mini-Haskell} & \textit{AST} \rightarrow \textit{AST} \end{array}$$

- For example, in ghci there are additional phases such as:

  Parsing, dependency analysis, type checking, ...

Case Study 1: Arithmetic Interpretation

Interpreter #1

- A calculator for arithmetic expressions

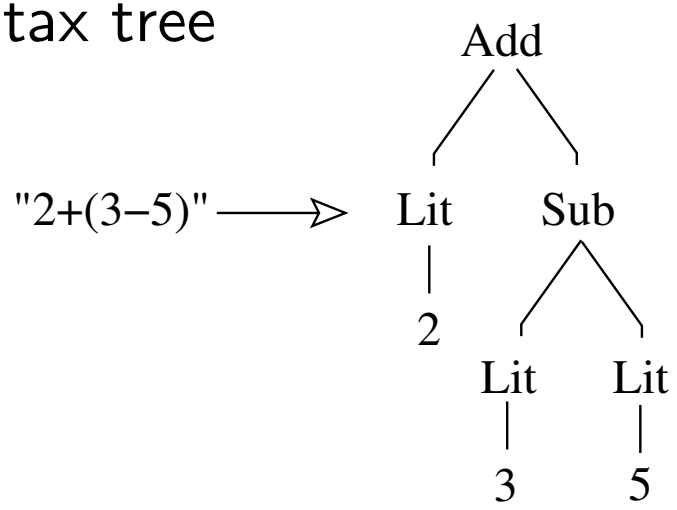
$$Expr \ ::= \ Int \ \mid \ Expr \ \text{`}+\text{'} \ Expr \ \mid \ Expr \ \text{`}-\text{'} \ Expr$$

- Lexical analysis: recognition of integers, '+', and '−'

  As well as parentheses and white space.

- Parsing: convert tokens into abstract syntax tree

```
data Expr = Lit Int | Add Expr Expr
                    | Sub Expr Expr
```



Example (cont.)

- Evaluation functions already given earlier

```
eval :: Expr -> Int

eval (Lit n)     = n
eval (Add e1 e2) = (eval e1) + (eval e2)
eval (Sub e1 e2) = (eval e1) - (eval e2)
```

- Pretty printer as instance of type class `Show`

```
instance Show Expr where
    show (Lit n)     = show n
    show (Add e1 e2) = "(" ++ show e1 ++ "+" ++ show e2 ++ ")"
    show (Sub e1 e2) = "(" ++ show e1 ++ "-" ++ show e2 ++ ")"
```

Alternative: just use the standard `show` by adding `deriving Show` to the data-type definition

Lexical analysis and parsing

- Problem important: necessary for all systems with nontrivial input
- Interesting too from computer science perspective

  Formal language theory
  Relationship between different language classes and recognizers

  Software engineering
  Interpreters built from subsystems with well-defined interfaces

  Computer linguistics
  Analysis and classification of human languages

- Motivation for many software tools, e.g. LEX, YACC, . . .
  For Haskell: Alex and Happy

Combinatory parsing — overview

- Idea: modular construction and composition of parser functions

  first parse this and afterwards that

- Result is powerful, e.g., can handle ambiguous grammars

- Approach elegantly uses lazy & higher-order programming

  Also uses monads (new!) to package "higher-order plumbing"

Parser type

- A parser is a **function** taking a string as input
- **Result** is an element of type a (typically a data type like `Expr`)
- A parser may process only part of input, leaving a **remainder**

  This supports composition: p parses first bit and q continues
- Allow **multiple results** from parsing via **list of successes**

$$[(\text{result}_1,\text{remainder}_1), \ldots, (\text{result}_n,\text{remainder}_n)]$$

- Hence: `data Parser a = Prs (String -> [(a,String)])`

Parser application

```
data Parser a = Prs (String -> [(a,String)])
```

- Apply a parser to an input string:

```
parse :: Parser a -> String -> [(a,String)]
parse (Prs p) inp = p inp
```

- So `parse p inp` = [(result$_1$,remainder$_1$), ..., (result$_n$,remainder$_n$)]

  A pair with the remainder `""` is a **complete parse** of the input.

- We are interested in the result of the first complete parse:

```
completeParse :: Parser a -> String -> a
completeParse p inp
    | results == [] = error "Parse unsuccessful"
    | otherwise     = head results
  where results     = [res | (res,"") <- parse p inp]
```

Primitive parsers

```
data Parser a = Prs (String -> [(a,String)])
```

- Primitive parsers serve as basic building blocks.
- **Fails** trivially ([] signifies 'unsuccessful parse'):

```
failure :: Parser a
failure = Prs (\inp -> [])
```

- **Succeeds** trivially **without progress**:

```
return :: a -> Parser a
return x = Prs (\inp -> [(x,inp)])
```

- **Succeeds** trivially **with progress**:

```
item :: Parser Char
item = Prs (\inp -> case inp of
                      "" -> []
                      (x:xs) -> [(x,xs)])
```

Primitive parser (cont.)

- Parse a **single character** with property p

```
sat :: (Char -> Bool) -> Parser Char
sat p = Prs (\inp -> case inp of
                       "" -> []
                       (x:xs) -> if p x then [(x,xs)] else [])
```

- Examples (also showing types)

```
? parse (return "foo") "3+5"
[("foo","3+5")] :: [([Char],String)]

? parse failure "3+5"
[] :: [(a,String)]

? parse item "3+5"
[('3',"+5")] :: [(Char,String)]

? parse (sat (\x -> '0'<=x && x<='9')) "3+5"  -- (sat isDigit) "3+5"
[('3',"+5")] :: [(Char,String)]

? parse (sat (\x -> x=='+' || x=='-')) "3+5"
[] :: [(Char,String)]
```

Gluing parsers together

- **Mutual selection:** Apply both first and second parser

```
(|||) :: Parser a -> Parser a -> Parser a
p ||| q = Prs (\s -> parse p s ++ parse q s)
```

- **Alternative selection:** If first parser fails, apply second parser

```
(+++) :: Parser a -> Parser a -> Parser a
p +++ q = Prs (\s -> case parse p s of
                       [] -> parse q s
                       res -> res)
```

- Examples

```
? parse (return '!' ||| sat isDigit) "3+5"
[('!',"3+5"), ('3',"+5")]

? parse (return '!' +++ sat isDigit) "3+5"
[('!',"3+5")]
```

Gluing parsers together (cont.)

- **Sequencing:** first parser `p` then parser `q` to results

```
(>>) :: Parser a -> Parser b -> Parser b
p >> q = Prs (\s -> [ (u,s'') | (t,s') <- parse p s,
                                (u,s'') <- parse q s' ])
```

- A schematic example:

$$\texttt{parse}\ p\ s \hookrightarrow [\ldots(t, s')\ldots] \quad \text{and} \quad \texttt{parse}\ q\ s' \hookrightarrow [\ldots(u, s'')\ldots]$$
$$\Longrightarrow \quad \texttt{parse}\ (p\ \texttt{>>}\ q)\ s \hookrightarrow [\ldots(u, s'')\ldots]$$

- Problem: the **result** of first parser (t above) is lost

```
? parse (sat isDigit >> sat (=='+')) "3+5"
[('+',"5")] :: [(Char,String)]

? parse (sat isDigit >> sat isDigit) "31+5"
[('1',"+5")] :: [(Char,String)]
```

Sequencing

- **Solution:** use as second argument a **"parser generator"** that takes as input the result of the first parser

```
(>>=) :: Parser a -> (a -> Parser b) -> Parser b
p >>= g = Prs (\s -> [ (u,s'') | (t,s' ) <- parse p s,
                                 (u,s'') <- parse (g t) s' ])
```

- Examples

```
? parse (sat isDigit >>=
             \t -> sat isDigit >>=
             \u -> return (t:u:[])) "31+5"
[("31","+5")]

? parse (sat isDigit >>=
             \t -> sat isDigit >>=
             \u -> return (t:u:[])) "3+5"
[]
```

- Reimplementation of >> with >>=

```
(>>) :: Parser a -> Parser b -> Parser b
p >> q = p >>= \_ -> q
```

More basic parsers and combinators

- **Chars** and **Strings** (including simpler definition of `sat`)

```
sat  :: (Char -> Bool) -> Parser Char
sat p = item >>= \x -> if p x then return x else failure

char  :: Char -> Parser Char
char x = sat (==x)

string       :: String -> Parser String
string ""     = return ""
string (x:xs) = char x >> string xs >> return (x:xs)
```

- **Repetition**

```
many :: Parser a -> Parser [a]    -- 0 or more repetitions of p
many p = many1 p ||| return []

many1 :: Parser a -> Parser [a]   -- 1 or more repetitions of p
many1 p = p >>= \t -> many p >>= \ts -> return (t:ts)
```

- What is the result of the following parse?

```
? parse (many1 (sat isDigit)) "31+5"
```

Improving readability

- Haskell supports a more readable use of >>=

```
many1 p = do t <- p
             ts <- many p
             return (t:ts)
```

This looks like an imperative program!

- Syntactic sugar

```
do t1 <- p1                          p1 >>= \t1 ->
   t2 <- p2                          p2 >>= \t2 ->
    :          abbreviates             :
   tn <- pn                          pn >>= \tn ->
   return (f t1 t2 ... tn)           return (f t1 t2 ... tn)
```

- More sugar: when `ti` unimportant

```
do t1 <- p1                          p1 >>= \t1 ->
      :                                 :
      pi          abbreviates       pi >>= \_ ->
      :                                 :
```

Improving readability (cont.)

```
many1 p = do t <- p
             ts <- many p
             return (t:ts)
```

- To use this sugar, we must put `Parser` into the class `Monad`

```
instance Monad Parser where
    return v  = ...
    p >>= g = ...
```

- We will have an another look at monads later in the course

Slightly more complex parsers

```
numPos :: Parser Int
numPos = do ts <- many1 (sat isDigit)
            return (read ts)   --- read maps numeric string to number

numNeg :: Parser Int
numNeg = do char '-'
            t <- numPos
            return (-t)

num :: Parser Int
num = numPos ||| numNeg     -- or: numPos +++ numNeg


? parse num "123"
[(123,""), (12,"3"), (1,"23")] :: [(Int,String)]

? parse num "-123"
[(-123,""), (-12,"3"), (-1,"23")] :: [(Int,String)]
```

Combinators — overview

```
data Parser a = Prs (String -> [(a,String)])

return      :: a -> Parser a
(>>=)       :: Parser a -> (a -> Parser b) -> Parser b

(>>)        :: Parser a -> Parser b -> Parser b
(|||)       :: Parser a -> Parser a -> Parser a
(+++)       :: Parser a -> Parser a -> Parser a

failure     :: Parser a
item        :: Parser Char

sat         :: (Char -> Bool) ->  Parser Char
char        :: Char -> Parser Char
string      :: String -> Parser String

many        :: Parser a -> Parser [a]
many1       :: Parser a -> Parser [a]

num         :: Parser Int       -- parse an integer
```

Parsing with combinators

- Combinators express general operations

  Sequence, selection, repetition, recognition of tokens, . . .

- It is now easy to recognize languages

  Result is pair $(t, [\,])$

- We shall demonstrate this idea with

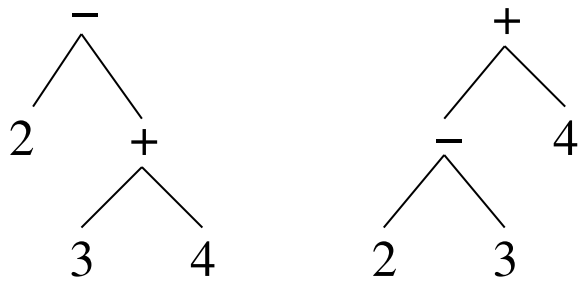
$$Expr \;::=\; Int \;\mid\; Expr\ \text{`}+\text{'}\ Expr \;\mid\; Expr\ \text{`}-\text{'}\ Expr$$

  But we must first solve a few small problems

Problem #1 — ambiguous grammars

$$Expr \ ::= \ Int \ \mid \ Expr \ \text{`}+\text{'} \ Expr \ \mid \ Expr \ \text{`}-\text{'} \ Expr$$

- How should $2 - 3 + 4$ be parsed?



- Solution is either
  - to use an unambiguous grammar (see compiler course) or
  - to give the user a means to eliminate ambiguity during parsing

$$Expr \ ::= \ Int \ \mid \ Expr \ \text{`}+\text{'} \ Expr \ \mid \ Expr \ \text{`}-\text{'} \ Expr \ \mid \ \text{`}(\text{'} \ Expr \ \text{`})\text{'}$$

Problem #2 — left recursion

$$Expr \ ::= \ Int \ \mid \ Expr \ \text{`+'} \ Expr \ \mid \ Expr \ \text{`}-\text{'} \ Expr \ \mid \ \text{`('} \ Expr \ \text{`)'}$$

- **Left-recursive grammars** produce problems!
  - Parsing an `Expr` requires first parsing an `Expr`.
  - This results in non-terminating recursion.
- **Key to solution**: each `Expr` begins with an `Int` or a '('

$$\begin{aligned} Atom &::= Int \mid \text{`('} \ Expr \ \text{`)'} \\ Expr &::= Atom \mid Atom \ \text{`+'} \ Expr \mid Atom \ \text{`}-\text{'} \ Expr \end{aligned}$$

- Use **concrete grammar** to build **abstract syntax tree** of type

```
data Expr = Lit Int | Add Expr Expr | Sub Expr Expr
```

The parser

Concrete grammar:

$$\begin{aligned} Atom &::= Int \mid \text{`(' } Expr \text{ `)'} \\ Expr &::= Atom \mid Atom \text{ `+' } Expr \mid Atom \text{ `$-$' } Expr \end{aligned}$$

Parser:

```
data Expr = Lit Int | Add Expr Expr | Sub Expr Expr
          deriving (Show, Eq)

atom = lit ||| pexpr
expr = atom ||| add ||| sub

lit = do x <- num
         return (Lit x)

pexpr = do string "("
           e <- expr
           string ")"
           return e

add = do a <- atom
         string "+"
         e <- expr
         return (Add a e)

sub = do a <- atom
         string "-"
         e <- expr
         return (Sub a e)
```

Examples

- Parse with success

```
? parse add "3+4"
[(Add (Lit 3) (Lit 4),"")] :: [(Expr,String)]

? parse add "3+(4-5)"
[(Add (Lit 3) (Sub (Lit 4) (Lit 5)),"")] :: [(Expr,String)]

? parse expr "5+(2--5)"
[(Lit 5,"+(2--5)"), (Add (Lit 5) (Sub (Lit 2) (Lit -5)),"")]
```

- Failure

```
? parse expr "--5"
[] :: [(Expr,String)]

? parse expr "3 + 4"
[(Lit 3," + 4")] :: [(Expr,String)]
```

- Exercise: extend to handle white space

A (mini-)calculator

- Program

```
str2expr :: String -> Expr
str2expr s = completeParse expr s

eval :: Expr -> Int
eval (Lit n)   = n
eval (Add x y) = eval x + eval y
eval (Sub x y) = eval x - eval y

calculate = eval . str2expr
```

- Examples

```
? str2expr "2+(5--2)"
Add (Lit 2) (Sub (Lit 5) (Lit -2)) :: Expr

? calculate "2+(5--2)"
9 :: Int
```

Summary

- Writing interpreters and compilers is fundamental
  - **read, evaluate, print**
  - Interpreters are everywhere!
- Many languages offer "program-generation tools"

  Opinion: None are so simple, elegant, and understandable as this
  Fact: Substantial time and code-savings
- Lazy evaluation plays an important role in (relatively) efficient generate and test

```
? parse num "123"
[(123,""), (12,"3"), (1,"23")] :: [(Int,String)]
```

Case Study 2: Mini-Haskell

Overview

- We have seen parsing combinators and a simple application
- Now, the λ-calculus
  - The same parsing combinators, but another language
  - Moreover, an interpreter for the λ-calculus
- Goals
  - A small but nontrivial example
  - A deeper understanding of Haskell itself

The λ-calculus

- Programs are terms (given a set of variable $\mathcal{V}$ and integers $\mathcal{Z}$)

$$\begin{array}{rcl} t & ::= & \mathcal{V} \mid (\lambda x.\, t) \mid (t_1\, t_2) \mid \\ & & \mathcal{Z} \mid (t_1 + t_2) \mid (t_1 \times t_2) \mid \\ & & \mathit{True} \mid \mathit{False} \mid (\textbf{iszero}\ t) \mid (\textbf{if}\ t_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2) \mid \\ & & (t_1, t_2) \mid (\textbf{fst}\ t) \mid (\textbf{snd}\ t) \end{array}$$

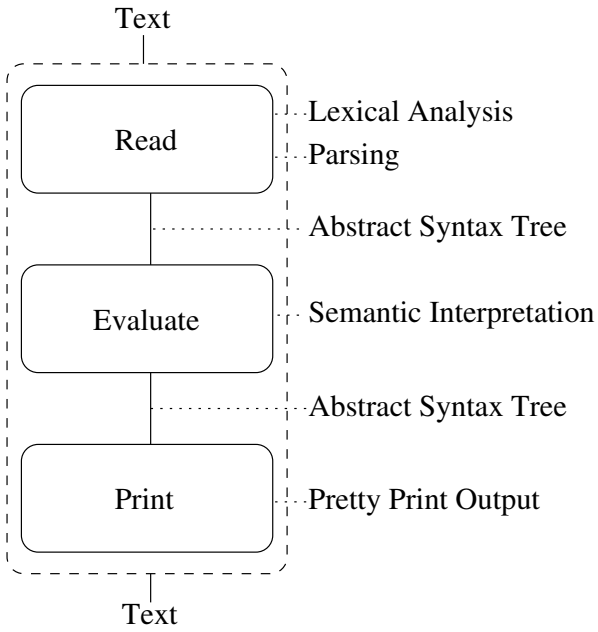
- Here we formalize the core

$$t \ ::= \ \mathcal{V} \mid \lambda x.\, t \mid t\ t'$$

  plus some syntactic sugar

- **Theorem (Church 1930s):** All computable functions can be represented within the (core) λ-calculus.

Overview



Read: parser for the λ-calculus.

Syntax trivial. We must however eliminate ambiguity and left-recursion. Deal with white-spaces, etc.

Evaluate: function from `Term` to `Term`

$$(\lambda x.\, x)(\lambda x\, y.\, x) \quad \hookrightarrow \quad \lambda x\, y.\, x$$

Print: simple recursion over data type

This is different than in ghci. How?

Parsing

- Same parser combinators as before

```
data Parser a = Prs (String -> [(a,String)])

return      :: a -> Parser a
(>>=)       :: Parser a -> (a -> Parser b) -> Parser b

(|||)       :: Parser a -> Parser a -> Parser a
many        :: Parser a -> Parser [a]
many1       :: Parser a -> Parser [a]

string      :: String -> Parser String
```

- Recall that parsers carry out complex operations by composing simple ones, like `p ||| q`

Auxiliary simple parsers

- Consuming spaces

```
spaces :: Parser String
spaces = many (sat (== ' '))
```

- Reading an identifier

```
identifier :: Parser String
identifier = do spaces
                id <- many1 (sat isLetter)
                spaces
                return id
```

- Reading a token

```
token :: String -> Parser String
token s = do spaces
             t <- string s
             spaces
             return t                  -- (i.e., s)
```

Parsing λ-terms

- Data type for λ-calculus terms

```
data Term = Id String | Ap Term Term | Lam String Term
            deriving (Show, Eq)
```

- Application $t_1\ t_2$ produces left recursion (prefix-syntax simpler!)
- Syntax without left-recursion

$$\begin{array}{rcl} \mathit{Term} & ::= & \mathit{Ident}\ \mathit{Term}^* \mid \mathit{Lamb}\ \mathit{Term}^* \mid \mathit{Paren}\ \mathit{Term}^* \\ \mathit{Lamb} & ::= & \text{`\%'}\ \mathit{Ident}^+\ \text{`.'}\ \mathit{Term} \\ \mathit{Paren} & ::= & \text{`('}\ \mathit{Term}\ \text{`)'} \end{array}$$

  - We use '%' and '.' instead of '\' and '->', respectively
  - Introduces parentheses
  - Every parsing starts with an identifier, or symbols '%' or '('

The parser

```
atom  = ident ||| lamb ||| paren

ident = do id <- identifier
           return (Id id)

term  = do t <- atom                    -- t
           ts <- many atom              -- [t1 t2 ... tn]
           return (foldl Ap t ts)       -- Ap(Ap(Ap(t t1) t2) ... tn)

lamb  = do token "%"
           ids <- many1 identifier      -- [x1, x2, ..., xn]
           token "."
           t <- term                    -- t
           return (foldr Lam t ids)     -- Lam x1 (Lam x2 (...(Lam xn t)))

paren = do token "("
           t <- term
           token ")"
           return t

str2term s = completeParse term s
```

Examples

```
? parse atom "x y z"
[(Id "x","y z"), (Id "x"," y z")] :: [(Term,String)]

? parse term "x y z"                    -- note left-associated output
[(Ap (Ap (Id "x") (Id "y")) (Id "z"),""), ...]

? parse term "(%x y. x y)"
[(Lam "x" (Lam "y" (Ap (Id "x") (Id "y"))),""), ...]

? parse term "(%x y. x y) z"
[(Ap (Lam "x" (Lam "y" (Ap (Id "x") (Id "y")))) (Id "z"),""), ...]
```

The pretty printer

```
prettyPrint (Id s)    = s
prettyPrint (Lam s t) = "(%" ++ s ++ ". " ++ (prettyPrint t) ++ ")"
prettyPrint (Ap s t)  = "(" ++ (prettyPrint s) ++ " "
                            ++ (prettyPrint t) ++ ")"
```

- Could also use type class `Show` for pretty printing
- Examples

```
? (prettyPrint . str2term) "%x y z. x z (y z)"
"(%x. (%y. (%z. ((x z) (y z)))))" :: String

? (prettyPrint . str2term) "%x y z. x (%r. r) (y ((z)))"
"(%x. (%y. (%z. ((x (%r. r)) (y z)))))" :: String
```

- Syntactic sugar supported in parsing (not pretty printing): nested abstraction and left-associative application

$$\begin{aligned}(\lambda x.\,(\lambda y.\,(\lambda z.\,((xz)(yz))))) &\equiv (\lambda x\,y\,z.\,((xz)(yz)))\\ &\equiv \lambda x\,y\,z.\,xz(yz)\end{aligned}$$

Evaluation in the λ-calculus

	standard		λ-calculus
Declaration:	$f\,x = g\,x\,3$	$\rightsquigarrow$	$f = \lambda x.\, g\,x\,3$
Application:	$f\,5$		$(\lambda x.\, g\,x\,3)(5)$
Evaluation:	$(g\,x\,3)[x \leftarrow 5]$		$(g\,x\,3)[x \leftarrow 5]$
Result:	$g\,5\,3$		$g\,5\,3$

To formalize evaluation, we must:

1. Formalize reduction based on substitution, i.e., $t[x \leftarrow s]$.
2. Fix an evaluation strategy,
   e.g., given $(\lambda x.\, x)\ ((\lambda y.\, y)2)$, what do we reduce first?

λ-calculus — substitution

- Evaluation is based on substitution

$$(\lambda x.\, g\, x\, 3)(5) = (g\, x\, 3)[x \leftarrow 5] = g\, 5\, 3$$

- But not all variables have the same role.
  - In $\lambda y.\, x\, y$, the variable y is **bound** and x is **free**
  - Analog from mathematics: $x^2 + \int_c^d x \cdot y\, dy$ or $\sum_{i=0}^{5} x \cdot i$

- Renaming and substitution are tricky: must avoid capture!

  Correct: $\int_a^b [x^2 + \int_c^d xy\, dy]\, dx = \int_a^b [y^2 + \int_c^d yz\, dz]\, dy$

  False: $\int_a^b [x^2 + \int_c^d xy\, dy]\, dx = \int_a^b [y^2 + \int_c^d yy\, dy]\, dy$

Substitution (cont.)

- Substitution must respect free and bound variables

$$\lambda x.\,(x\,(y\,x))$$

$$\lambda *.\,(*\,(y\,*))$$

$$\begin{array}{lcl} \mathit{free}\,(x) & = & \{\,x\,\} \\ \mathit{free}\,(M\,N) & = & \mathit{free}\,(M) \cup \mathit{free}\,(N) \\ \mathit{free}\,(\lambda x.\,M) & = & \mathit{free}\,(M) \setminus \{\,x\,\} \end{array}$$

- Examples

$$\begin{array}{rcl} \mathit{free}\,(\lambda x\,y.\,x\,\mathbf{z}) & = & \{\,z\,\} \\ \mathit{free}\,(\mathbf{x}\,\mathbf{y}\,(\lambda y\,z.\,\mathbf{x}\,y\,z)) & = & \{\,x, y\,\} \end{array}$$

Haskell implementation

```
free :: Term -> [String]
free (Id v)    = sing v
free (Ap s t)  = union (free s) (free t)
free (Lam v t) = diff (free t) (sing v)
```

$$
\begin{aligned}
\mathbf{free}(x) &= \{ x \} \\
\mathbf{free}(M\,N) &= \mathbf{free}(M) \cup \mathbf{free}(N) \\
\mathbf{free}(\lambda x.\, M) &= \mathbf{free}(M) \setminus \{ x \}
\end{aligned}
$$

```
empty  = []
sing a = [a]
```

Alternative: use Haskell library `Data.Set` to implement `free`

```
member [] _    = False
member (x:xs) a
  | x < a      = member xs a
  | x == a     = True
  | otherwise = False
```

```
union []     ys     = ys
union xs     []     = xs
union (x:xs) (x:ys)
  | x < y     = x : union xs (y:ys)
  | x == y    = x : union xs     ys
  | otherwise = y : union (x:xs) ys
```

```
diff []     _      = []
diff xs     []     = xs
diff (x:xs) (y:ys)
  | x < y     = x : diff xs (y:ys)
  | x == y    = diff xs ys
  | otherwise = diff (x:xs) ys
```

Free variables (cont.)

- Incremental testing supported by function composition

```
? str2term "%x y. x z"
Lam "x" (Lam "y" (Ap (Id "x") (Id "z"))) :: Term

? (free . str2term) "%x y. x z"
["z"] :: [String]

? (free . str2term) "x y (%y z. x y z)"
["x", "y"] :: [String]
```

- Now we can properly implement substitution

  We rename as necessary to avoid capturing free variables

λ-calculus — substitution

$M[x \leftarrow N]$ means that x is replaced by N in M

1. $x[x \leftarrow N] = N$
2. $y[x \leftarrow N] = y$ if $y \neq x$
3. $(P\,Q)[x \leftarrow N] = (P[x \leftarrow N])\,(Q[x \leftarrow N])$
4. $(\lambda x.\, P)[x \leftarrow N] = \lambda x.\, P$
5. $(\lambda y.\, P)[x \leftarrow N] = \lambda y.\, (P[x \leftarrow N])$ if $y \neq x$ and $y \notin \mathit{free}(N)$
6. $(\lambda y.\, P)[x \leftarrow N] = \lambda z.\, ((P[y \leftarrow z])[x \leftarrow N])$
   if $y \neq x$ and $y \in \mathit{free}(N)$, where $z \notin \mathit{free}(N\,P)$

Intuitive but easy to get wrong!

Substitution (cont.)

1. $x[x \leftarrow N] = N$
2. $y[x \leftarrow N] = y$, **if** $y \neq x$
3. $(P\,Q)[x \leftarrow N] = (P[x \leftarrow N]Q[x \leftarrow N])$
4. $(\lambda x.\, P)[x \leftarrow N] = \lambda x.\, P$
5. $(\lambda y.\, P)[x \leftarrow N] = \lambda y.\, (P[x \leftarrow N])$, **if** $y \neq x$ **and** $y \notin \mathit{free}(N)$
6. $(\lambda y.\, P)[x \leftarrow N] = \lambda z.\, ((P[y \leftarrow z])[x \leftarrow N])$ **if** $y \neq x$ **and** $y \in \mathit{free}(N)$, **where** $z \notin \mathit{free}(NP)$

- An example

$$\begin{aligned}(x(\lambda x.\, x\,y))[x \leftarrow \lambda z.\, z] &= x[x \leftarrow \lambda z.\, z](\lambda x.\, x\,y)[x \leftarrow \lambda z.\, z] \\ &= (\lambda z.\, z)\lambda x.\, x\,y\end{aligned}$$

- An example with renaming

$$\begin{aligned}(\lambda y.\, y\,x)[x \leftarrow y] &= \lambda z.\, ((y\,x)[y \leftarrow z][x \leftarrow y]) \\ &= \lambda z.\, (z\,x[x \leftarrow y]) \\ &= \lambda z.\, z\,y\end{aligned}$$

Case 6 avoids variable capture, i.e., $\lambda y.\, y\,y$

Implementation

1. $x[x \leftarrow N] = N$
2. $y[x \leftarrow N] = y$, **if** $y \neq x$
3. $(P\,Q)[x \leftarrow N] = (P[x \leftarrow N]Q[x \leftarrow N])$
4. $(\lambda x.\, P)[x \leftarrow N] = \lambda x.\, P$
5. $(\lambda y.\, P)[x \leftarrow N] = \lambda y.\, (P[x \leftarrow N])$, **if** $y \neq x$ **and** $y \notin \mathit{free}(N)$
6. $(\lambda y.\, P)[x \leftarrow N] = \lambda z.\, ((P[y \leftarrow z])[x \leftarrow N])$ **if** $y \neq x$ **and** $y \in \mathit{free}(N)$, **where** $z \notin \mathit{free}(NP)$

```
-- subst t v s = t[v <- s]
subst :: Term -> String -> Term -> Term
subst (Id x) v s     = if x == v then s else Id x
subst (Ap t1 t2) v s = Ap (subst t1 v s) (subst t2 v s)
subst (Lam x t) v s
  | x == v                    = Lam x t
  | not (member (free s) x) = Lam x (subst t v s)
  | otherwise                 = Lam z (subst (subst t x (Id z)) v s)
  where z = fresh (union (free t) (free s))
        fresh m = (foldr max "" m)  ++ "'"    -- returns id not in m


? prettyPrint (subst (str2term "x (%x. x y)") "x" (str2term "%z. z"))
"((%z. z) (%x. (x y)))" :: String

? prettyPrint (subst (str2term "x (%x. x y)") "y" (str2term "f x"))
"(x (%y'. (y' (f x))))" :: String
```

The essence of evaluation is β-reduction

- A term $(\lambda x.\, M)\, N$ is called a **redex**
- **β-reduction** is the rule for simplifying redeces

$$(\lambda x.\, M) N \hookrightarrow M[x \leftarrow N]$$

- The result $M[x \leftarrow N]$ is called the **contractum**
- **Example**

$$(\lambda x.\, f\,(x\,x))\, N \hookrightarrow f\,(N\,N)$$

- **Evaluation** fixes a strategy for reducing redeces

Evaluation strategy ($t \hookrightarrow r$)

- $t_1\, t_2$ represents the first application of a function to an argument
  - First evaluate t_1: $t_1 \hookrightarrow r_1$
    If $r_1 \neq \lambda x.\, r$ then throw an exception (or return application)
  - Then apply β-reduction to $r_1\, t_2$

$$(\lambda x.\, r) t_2 \hookrightarrow r[x \leftarrow t_2]$$

  - and the result is then further evaluated
- Evaluation strategy 1: **Eager**
  - t_2 evaluated prior to β-reduction
  - Evaluation carried out under an abstraction $(\lambda x.\, t)$
- Evaluation strategy 2: **Lazy**
  - t_2 substituted without evaluation
  - No evaluation under an abstraction

Implementing eager and lazy evaluation

```
beta (Lam x t) t' = subst t x t'

eager :: Term -> Term
eager (Id x)    = (Id x)
eager (Ap t t') = case r of
                    (Lam _ _) -> eager (beta r r')
                    _         -> Ap r r'
    where r  = eager t
          r' = eager t'
eager (Lam x t) = Lam x (eager t)


lazy :: Term -> Term
lazy (Id x)    = (Id x)
lazy (Ap t t') = case r of
                   (Lam _ _) -> lazy (beta r t')
                   _         -> Ap r t'
    where r = lazy t
lazy t         = t      -- no evaluation under a lambda abstraction
```

Two interpreters for the λ-calculus

```
evalE = prettyPrint . eager . str2term

evalL = prettyPrint . lazy . str2term
```

Examples

```
? evalE "(%x y. x) a b"
"a" :: String

? evalL "(%x y. x) a b"
"a" :: String

? evalE "(%x. (%y. y) x)"
"(%x. x)" :: String

? evalL "(%x. (%y. y) x)"
"(%x. ((%y. y) x))" :: String
```

Examples (cont.)

```
s = "(%x y z -> x z (y z))"
k = "(%x y -> x)"
i = "(%x -> x)"

? evalE (s++k++k)
"(%z. z)" :: String

? evalL (s++k++k)
"(%z. (((%x. (%y. x)) z) ((%x. (%y. x)) z)))" :: String

? evalL ((evalL (s++k++k))++"a")
"a" :: String

? evalL "(%x. x x) (%x. x x)"
"{Interrupted!}

? evalL "(%x y. x y) (x y z)"
"(%z'. (((x y) z) z'))" :: String
```

Coding in the λ-calculus

- Despite its simplicity, we can represent arithmetic functions in the λ-calculus.
- To do this, we first give a total, computable, injective function $\overline{\cdot}$, mapping $\mathcal{N}$ to λ-terms. For example,

$$\overline{n} = \lambda s\, z.\, s^n(z) \quad \text{e.g.} \quad \overline{2} = \lambda s\, z.\, s(sz)$$

  This representation is called the **Church numerals**.
- A λ-term $\overline{f}$ represents a function $f : \mathcal{N}^k \to \mathcal{N}$ iff for all $n_1, \ldots, n_k \in \mathcal{N}$

$$\overline{f}\ \overline{n_1}, \ldots,\ \overline{n_k} = \overline{f(n_1, \ldots, n_k)}$$

Coding in the λ-calculus (cont.)

- Representing the natural numbers (here n represents $\overline{n}$)

$$n = \lambda s\, z.\, s^n(z) \quad \text{e.g.} \quad 2 = \lambda s\, z.\, s(sz))$$

- Representing addition: $add = \lambda x\, y\, s\, z.\, xs(ysz)$

$$\begin{aligned} add\ n\ m &= (\lambda x\, y\, s\, z.\, xs(ysz))\ (\lambda s\, z.\, s^n(z))\ \lambda s\, z.\, s^m z \\ &\hookrightarrow \lambda s\, z.\, ((\lambda s\, z.\, s^n z)s)((\lambda s\, z.\, s^m z)sz)) \\ &\hookrightarrow \lambda s\, z.\, ((\lambda z.\, s^n z)(s^m z)) \\ &\hookrightarrow \lambda s\, z.\, s^n(s^m z) \\ &\hookrightarrow \lambda s\, z.\, s^{n+m} z \\ &= n + m \end{aligned}$$

- Idea can be extended to all (partial recursive) functions

Coding (with interpreter)

- Addition with our interpreter

```
add = "(%x y s z. x s (y s z))"

? evalE (add ++ "(%s z. s z)" ++ "(%s z. s (s z))")
"(%s. (%z. (s (s (s z)))))" :: String
```

- Addition in Haskell

```
hadd = \x y s z -> x s (y s z)

? hadd (\s z -> s (s z)) (\s z -> s z)
<<function>> :: (a -> a) -> a -> a

data Nat = Zero | Succ Nat
           deriving (Show,Ord,Eq)

? hadd (\s z -> s (s z)) (\s z -> s z) Succ Zero
Succ (Succ (Succ Zero)) :: Nat
```

Relationship to Haskell

- Haskell interpreter extends `evalL`
  - Constructors like ':' are interpreted lazily
  - Considerably more syntactic sugar
- Functions are not printed (`show`)

```
? \x y -> x y
<<function>> :: (a -> b) -> a -> b
```

- Types protect against runtime errors
- But `evalL` is the basis!

Summary

- Complicated concepts can be simply formalized
  - Parsing using primitive parsers + combinators
  - Evaluation strategies
- Haskell supports a clean modeling
  - Data types for abstract syntax trees
  - Higher-order functions for combinators
  - Lazy evaluation for efficiency
- Interpreter provides considerable insight into Haskell itself