

The Culture of Scheme Programming

Panicz Maciej Godek

`godek.maciek@gmail.com`

Functional Tricity#2, 30.06.2016

What is a computer program?

What is a computer program?

- a set of instructions for a computer to execute under some specified circumstances

What is a computer program?

What is a computer program?

- a set of instructions for a computer to execute under some specified circumstances

Computer program as a set of instructions

Implications:

- instruct computer how to perform a computation
- if certain patterns repeat, extract them to separate procedures

Computer program as a set of instructions

Implications:

- instruct computer how to perform a computation
- if certain patterns repeat, extract them to separate procedures

Computer program as a set of instructions

Implications:

- instruct computer how to perform a computation
- if certain patterns repeat, extract them to separate procedures



DRY SPOT!

WET WET
WETTT?

What is a computer program?

What is a computer program?

- a set of instructions for a computer to execute in some specified circumstances
- a way of expressing ideas, communicating them to other people

What is a computer program?

What is a computer program?

- a set of instructions for a computer to execute in some specified circumstances
- a way of expressing ideas, communicating them to other people

The anatomy of ideas

What is an idea?

idea – gr. ἰδεῖν – to see; εἶδος – form, essence, type, species

What can we do with ideas?

The anatomy of ideas

What is an idea?

idea – gr. ἰδεῖν – to see; εἶδος – form, essence, type, species

What can we do with ideas?

The anatomy of ideas

What is an idea?

idea – gr. ἰδεῖν – to see; εἶδος – form, essence, type, species

What can we do with ideas?

The anatomy of ideas

The acts of the mind (...) are chiefly these three:

- 1 *Combining several simple ideas into one compound one, and thus all **complex ideas** are made.*
- 2 *(...) bringing two ideas, whether simple or complex, together, [...], without uniting them into one, by which it gets all its **ideas of relations**.*
- 3 *(...) separating them from all other ideas that accompany them in their real existence: this is called abstraction, and thus all its **general ideas** are made.*

— John Locke, *An Essay Concerning Human Understanding*
(1690)

The anatomy of ideas

The acts of the mind (...) are chiefly these three:

- 1 *Combining several simple ideas into one compound one, and thus all **complex ideas** are made.*
- 2 *(...) bringing two ideas, whether simple or complex, together, [...], without uniting them into one, by which it gets all its **ideas of relations**.*
- 3 *(...) separating them from all other ideas that accompany them in their real existence: this is called abstraction, and thus all its **general ideas** are made.*

— John Locke, *An Essay Concerning Human Understanding*
(1690)

The anatomy of ideas

The acts of the mind (...) are chiefly these three:

- 1 *Combining several simple ideas into one compound one, and thus all **complex ideas** are made.*
- 2 *(...) bringing two ideas, whether simple or complex, together, [...], without uniting them into one, by which it gets all its **ideas of relations**.*
- 3 *(...) separating them from all other ideas that accompany them in their real existence: this is called abstraction, and thus all its **general ideas** are made.*

— John Locke, *An Essay Concerning Human Understanding*
(1690)

The anatomy of ideas

The acts of the mind (...) are chiefly these three:

- 1 Combining several simple ideas into one compound one, and thus all **complex ideas** are made.*
- 2 (...) bringing two ideas, whether simple or complex, together, [...], without uniting them into one, by which it gets all its **ideas of relations**.*
- 3 (...) separating them from all other ideas that accompany them in their real existence: this is called abstraction, and thus all its **general ideas** are made.*

— John Locke, *An Essay Concerning Human Understanding*
(1690)

The anatomy of ideas

Inductive definition. An *idea* is either:

- one of the simple ideas
- a complex idea – a combination of simpler ideas
- a general idea – an abstraction over combinations of ideas

The anatomy of ideas

Inductive definition. An *idea* is either:

- one of the simple ideas
- a complex idea – a combination of simpler ideas
- a general idea – an abstraction over combinations of ideas

The anatomy of ideas

Inductive definition. An *idea* is either:

- one of the simple ideas
- a complex idea – a combination of simpler ideas
- a general idea – an abstraction over combinations of ideas

The anatomy of ideas

Inductive definition. An *idea* is either:

- one of the simple ideas
- a complex idea – a combination of simpler ideas
- a general idea – an abstraction over combinations of ideas

Understanding understanding

What does it mean to understand an idea?

- reduce a complex idea to the simpler ones that can be considered obvious
- being able to map the ideas onto our experience and imagination
- being able to draw practical conclusions and apply the idea to one's own existential situation

Understanding understanding

What does it mean to understand an idea?

- reduce a complex idea to the simpler ones that can be considered obvious
- being able to map the ideas onto our experience and imagination
- being able to draw practical conclusions and apply the idea to one's own existential situation

Understanding understanding

What does it mean to understand an idea?

- reduce a complex idea to the simpler ones that can be considered obvious
- being able to map the ideas onto our experience and imagination
- being able to draw practical conclusions and apply the idea to one's own existential situation

Understanding understanding

What does it mean to understand an idea?

- reduce a complex idea to the simpler ones that can be considered obvious
- being able to map the ideas onto our experience and imagination
- being able to draw practical conclusions and apply the idea to one's own existential situation

Understanding understanding

Programming is understanding

— Kristen Nygaard

You think you know when you can learn, are more sure when you can write, even more when you can teach, but certain when you can program.

— Alan Perlis

Understanding understanding

Programming is understanding

— Kristen Nygaard

You think you know when you can learn, are more sure when you can write, even more when you can teach, but certain when you can program.

— Alan Perlis

Understanding understanding

Programming is understanding

— Kristen Nygaard

You think you know when you can learn, are more sure when you can write, even more when you can teach, but certain when you can program.

— Alan Perlis

Understanding understanding

Why do we need ideas?

Why do we need complex ideas?

What are the means by which we create new ideas?

Understanding understanding

Why do we need ideas?

Why do we need complex ideas?

What are the means by which we create new ideas?

Understanding understanding

Why do we need ideas?

Why do we need complex ideas?

What are the means by which we create new ideas?

Definition of a definition

definition – a linguistic expression whose purpose is to introduce a new term based on some terms that are already known.

Definition of a definition

definiendum
definition – ^{definiens} a linguistic expression...

Definition of a definition

definition – a linguistic expression ^{genus} whose purpose... _{differentia}

More sophisticated example

$M = \langle Q, \Gamma, \delta, q_0, F \rangle$, where

Q – finite, non-empty set of states,

Γ – finite, non-empty set of *alphabet symbols*

$\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{L, R\}$ – state transition function

$q_0 \in Q$ – initial state

$F \subseteq Q$ – a set of final states

Principle of compositionality

the meaning of a complex expression is determined by the meanings of its constituent expressions and the rules used to combine them
(Gottlob Frege)

Hypothesis: in a complex expression, one can distinguish a ruling word which determines the meaning of the whole expression.

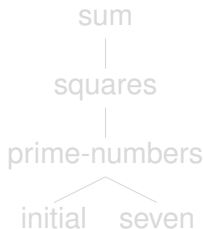
Principle of compositionality

the meaning of a complex expression is determined by the meanings of its constituent expressions and the rules used to combine them
(Gottlob Frege)

Hypothesis: in a complex expression, one can distinguish a ruling word which determines the meaning of the whole expression.

Principle of compositionality

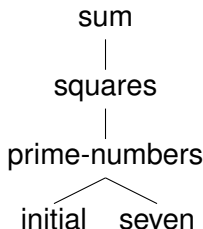
Conclusion: the structure of a compound expression (and its corresponding notion) can be represented as a tree.



Consequence: Languages have syntactic (and semantic) structures.

Principle of compositionality

Conclusion: the structure of a compound expression (and its corresponding notion) can be represented as a tree.



Consequence: Languages have syntactic (and semantic) structures.

Human and language

Who we are?

ζῶον λόγον ἔχον

And out of the ground the LORD God formed every beast of the field, and every fowl of the air; and brought them unto Adam to see what he would call them: and whatsoever Adam called every living creature, that was the name thereof.

— Genesis 2:19

Human and language

Who we are?

ζῶον λόγον ἔχον

And out of the ground the LORD God formed every beast of the field, and every fowl of the air; and brought them unto Adam to see what he would call them: and whatsoever Adam called every living creature, that was the name thereof.

— Genesis 2:19

Human and language

Who we are?

ζωον λογον εχων

*And out of the ground the LORD God formed every
beast of the field, and every fowl of the air; and
brought them unto Adam to see what he would call
them: and whatsoever Adam called every living
creature, that was the name thereof.*

— Genesis 2:19

Human and language

How can we know?

The limits of my language mean the limits of my world.

— Ludwig Wittgenstein

We dissect nature along lines laid down by our native language. The categories and types that we isolate from the world of phenomena we do not find there because they stare every observer in the face; on the contrary, the world is presented in a kaleidoscope flux of impressions which has to be organized by our minds—and this means largely by the linguistic systems of our minds.

— Benjamin Whorf

Human and language

How can we know?

The limits of my language mean the limits of my world.

— Ludwig Wittgenstein

We dissect nature along lines laid down by our native language. The categories and types that we isolate from the world of phenomena we do not find there because they stare every observer in the face; on the contrary, the world is presented in a kaleidoscope flux of impressions which has to be organized by our minds—and this means largely by the linguistic systems of our minds.

— Benjamin Whorf

Human and language

How can we know?

The limits of my language mean the limits of my world.

— Ludwig Wittgenstein

We dissect nature along lines laid down by our native language. The categories and types that we isolate from the world of phenomena we do not find there because they stare every observer in the face; on the contrary, the world is presented in a kaleidoscope flux of impressions which has to be organized by our minds—and this means largely by the linguistic systems of our minds.

— Benjamin Whorf

What does it mean to mean?

How is communication possible?

- ιδιος κοσμος
“When I use a word,” Humpty Dumpty said, in rather a scornful tone, “it means just what I choose it to mean—neither more nor less.”
— Lewis Carroll, *Through the Looking Glass*
- κοινος κοσμος

What does it mean to mean?

How is communication possible?

- ιδιος κοσμος

“When I use a word,” Humpty Dumpty said, in rather a scornful tone, “it means just what I choose it to mean—neither more nor less.”

— Lewis Carroll, *Through the Looking Glass*

- κοινος κοσμος

What does it mean to mean?

How is communication possible?

- ιδιος κοσμος
“When I use a word,” Humpty Dumpty said, in rather a scornful tone, “it means just what I choose it to mean—neither more nor less.”
— Lewis Carroll, *Through the Looking Glass*
- κοινος κοσμος

What does it mean to mean?

How is communication possible?

- ιδιος κοσμος
“When I use a word,” Humpty Dumpty said, in rather a scornful tone, “it means just what I choose it to mean—neither more nor less.”
— Lewis Carroll, *Through the Looking Glass*
- κοινος κοσμος

How natural is natural language?

How about using natural language to communicate ideas?

How natural is natural language?

Imprecision and ambiguity:

- Iraqi Head Seeks Arms
- Prostitutes Appeal to Pope
- Soviet Virgin Lands Short of Goal Again
- Regan Wins on Budget, but More Lies Ahead

How natural is natural language?

Imprecision and ambiguity:

- Iraqi Head Seeks Arms
- Prostitutes Appeal to Pope
- Soviet Virgin Lands Short of Goal Again
- Regan Wins on Budget, but More Lies Ahead

How natural is natural language?

Imprecision and ambiguity:

- Iraqi Head Seeks Arms
- Prostitutes Appeal to Pope
- Soviet Virgin Lands Short of Goal Again
- Regan Wins on Budget, but More Lies Ahead

How natural is natural language?

Imprecision and ambiguity:

- Iraqi Head Seeks Arms
- Prostitutes Appeal to Pope
- Soviet Virgin Lands Short of Goal Again
- Reagan Wins on Budget, but More Lies Ahead

How natural is natural language?

Imprecision and ambiguity:

- Iraqi Head Seeks Arms
- Prostitutes Appeal to Pope
- Soviet Virgin Lands Short of Goal Again
- Regan Wins on Budget, but More Lies Ahead

How natural is natural language?

Complicated structure

I suspect that machines to be programmed in our native tongues —be it Dutch, English, American, French, German, or Swahili— are as damned difficult to make as they would be to use.

— Edsger Dijkstra, *On the Foolishness of “Natural Language Programming”*

How natural is natural language?

Complicated structure

I suspect that machines to be programmed in our native tongues —be it Dutch, English, American, French, German, or Swahili— are as damned difficult to make as they would be to use.

— Edsger Dijkstra, *On the Foolishness of “Natural Language Programming”*

How natural is natural language?

Hanoi is a room.

A disk is a kind of supporter.

A post is a kind of supporter. A post is always fixed in place.

The left post, the middle post, and the right post are posts in Hanoi.

A disk is a kind of supporter.

The red disk is a disk on the left post.

The orange disk is a disk on the red disk.

The yellow disk is a disk on the orange disk.

The green disk is a disk on the yellow disk.

Definition: a disk is topmost if nothing is on it.

When play begins:

move 4 disks from the left post to the right post via the middle post.

To move (N - number) disk/disks from (FP - post) to (TP - post) via (VP - post):

if N > 0:

move N - 1 disks from FP to VP via TP;

say ``Moving a disk from [FP] to [TP]...'';

let D be a random topmost disk enclosed by FP;

if a topmost disk (called TD) is enclosed by TP, now D is on TD;

otherwise now D is on TP;

move N - 1 disks from VP to TP via FP.

How natural is natural language?

Verbosity

By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race.

— Alfred N. Whitehead, *An Introduction to Mathematics*

In the twelfth century, the Hindu mathematician Bhaskara said, “The root of the root of the quotient of the greater irrational divided by the lesser one being increased by one; the sum being squared and multiplied by the smaller irrational quantity is the sum of the two surd roots.”

$$\sqrt{(\sqrt{\frac{n}{k}} + 1)^2 k} = \sqrt{k} + \sqrt{n}$$

How natural is natural language?

Verbosity

By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race.

— Alfred N. Whitehead, *An Introduction to Mathematics*

In the twelfth century, the Hindu mathematician Bhaskara said, “The root of the root of the quotient of the greater irrational divided by the lesser one being increased by one; the sum being squared and multiplied by the smaller irrational quantity is the sum of the two surd roots.”

$$\sqrt{(\sqrt{\frac{n}{k}} + 1)^2 k} = \sqrt{k} + \sqrt{n}$$

How natural is natural language?

Verbosity

By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race.

— Alfred N. Whitehead, *An Introduction to Mathematics*

In the twelfth century, the Hindu mathematician Bhaskara said, “The root of the root of the quotient of the greater irrational divided by the lesser one being increased by one; the sum being squared and multiplied by the smaller irrational quantity is the sum of the two surd roots.”

$$\sqrt{(\sqrt{\frac{n}{k}} + 1)^2 k} = \sqrt{k} + \sqrt{n}$$

How natural is natural language?

Verbosity

By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race.

— Alfred N. Whitehead, *An Introduction to Mathematics*

In the twelfth century, the Hindu mathematician Bhaskara said, “The root of the root of the quotient of the greater irrational divided by the lesser one being increased by one; the sum being squared and multiplied by the smaller irrational quantity is the sum of the two surd roots.”

$$\sqrt{(\sqrt{\frac{n}{k}} + 1)^2 k} = \sqrt{k} + \sqrt{n}$$

Is mathematics the language of nature?

How about using mathematics to express ideas?

- compact notation
- “algebra of programs”
- funny symbols (Uncle Bob wouldn’t like it)

Is mathematics the language of nature?

How about using mathematics to express ideas?

- compact notation
- “algebra of programs”
- funny symbols (Uncle Bob wouldn’t like it)

Is mathematics the language of nature?

How about using mathematics to express ideas?

- compact notation
- “algebra of programs”
- funny symbols (Uncle Bob wouldn't like it)

Is mathematics the language of nature?

How about using mathematics to express ideas?

- compact notation
- “algebra of programs”
- funny symbols (Uncle Bob wouldn't like it)

Why functional programming?

What is so great about functional programming?

- easier to reason about
- no type errors
- referential transparency
- ...
- no side effects

Why functional programming?

What is so great about functional programming?

- easier to reason about
- no type errors
- referential transparency
- ...
- no side effects

Why functional programming?

What is so great about functional programming?

- easier to reason about
- no type errors
- referential transparency
- ...
- no side effects

Why functional programming?

What is so great about functional programming?

- easier to reason about
- no type errors
- referential transparency
- ...
- no side effects

Why functional programming?

What is so great about functional programming?

- easier to reason about
- no type errors
- referential transparency
- ...
- no side effects

Why functional programming?

What is so great about functional programming?

- easier to reason about
- no type errors
- referential transparency
- ...
- no side effects



CONDOMS

Use them.

Why functional programming?

In the beginning was the Word, and the Word was with God, and the Word was God.

— John 1:1

- substitution model of computation (conotation)
- program is just a name of some compound object (denotation)

Functional programming is like speaking language

Why functional programming?

In the beginning was the Word, and the Word was with God, and the Word was God.

— John 1:1

- substitution model of computation (conotation)
- program is just a name of some compound object (denotation)

Functional programming is like speaking language

Why functional programming?

In the beginning was the Word, and the Word was with God, and the Word was God.

— John 1:1

- substitution model of computation (conotation)
- program is just a name of some compound object (denotation)

Functional programming is like speaking language

Why functional programming?

In the beginning was the Word, and the Word was with God, and the Word was God.

— John 1:1

- substitution model of computation (conotation)
- program is just a name of some compound object (denotation)

Functional programming is like speaking language

Intermission

And now...

Intermission

I hope the field of computer science never loses its sense of fun. Above all, I hope we don't become missionaries. Don't feel as if you're Bible salesmen. The world has too many of those already.

— Alan Perlis, foreword to *Structure and Interpretation of Computer Programs*

Introduction

The philosophy of Scheme

The Crops

Showcase/Controversies/Books

What is a computer program?

The anatomy of ideas

Human and language

Advantages of functional programming

The philosophy of Scheme

The Culture of Scheme Programming

The philosophy of Scheme

Perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away.

— Antoine de Saint-Exupery

Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary.

— Revised⁵ Report on the Algorithmic Language Scheme
What are the necessary components of an expressive programming language?

The philosophy of Scheme

Perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away.

— Antoine de Saint-Exupery

Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary.

— Revised⁵ Report on the Algorithmic Language Scheme
What are the necessary components of an expressive programming language?

The philosophy of Scheme

Perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away.

— Antoine de Saint-Exupery

Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary.

— Revised⁵ Report on the Algorithmic Language Scheme
What are the necessary components of an expressive programming language?

The syntax of Scheme

Recall the Principle of Compositionality:

the meaning of a complex expression is determined by the meanings of its constituent expressions and the rules used to combine them

Apply the following conventions:

- write the ruling word (or the name of the combining rule) as the first element
- write the constituent expressions to the right of the ruling word; the order in which they are written determines their role in a given rule
- surround the combination in parentheses
- *et voila!*

The syntax of Scheme

Recall the Principle of Compositionality:

the meaning of a complex expression is determined by the meanings of its constituent expressions and the rules used to combine them

Apply the following conventions:

- write the ruling word (or the name of the combining rule) as the first element
- write the constituent expressions to the right of the ruling word; the order in which they are written determines their role in a given rule
- surround the combination in parentheses
- *et voila!*

The syntax of Scheme

Recall the Principle of Compositionality:

the meaning of a complex expression is determined by the meanings of its constituent expressions and the rules used to combine them

Apply the following conventions:

- write the ruling word (or the name of the combining rule) as the first element
- write the constituent expressions to the right of the ruling word; the order in which they are written determines their role in a given rule
- surround the combination in parentheses
- *et voila!*

The syntax of Scheme

Recall the Principle of Compositionality:

the meaning of a complex expression is determined by the meanings of its constituent expressions and the rules used to combine them

Apply the following conventions:

- write the ruling word (or the name of the combining rule) as the first element
- write the constituent expressions to the right of the ruling word; the order in which they are written determines their role in a given rule
- surround the combination in parentheses
- *et voila!*

The syntax of Scheme

Recall the Principle of Compositionality:

the meaning of a complex expression is determined by the meanings of its constituent expressions and the rules used to combine them

Apply the following conventions:

- write the ruling word (or the name of the combining rule) as the first element
- write the constituent expressions to the right of the ruling word; the order in which they are written determines their role in a given rule
- surround the combination in parentheses
- *et voila!*

The syntax of Scheme

Recall the Principle of Compositionality:

the meaning of a complex expression is determined by the meanings of its constituent expressions and the rules used to combine them

Apply the following conventions:

- write the ruling word (or the name of the combining rule) as the first element
- write the constituent expressions to the right of the ruling word; the order in which they are written determines their role in a given rule
- surround the combination in parentheses
- *et voila!*

The syntax of Scheme

Recall the Principle of Compositionality:

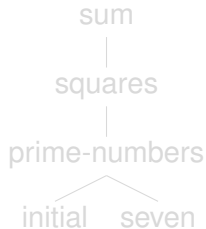
the meaning of a complex expression is determined by the meanings of its constituent expressions and the rules used to combine them

Apply the following conventions:

- write the ruling word (or the name of the combining rule) as the first element
- write the constituent expressions to the right of the ruling word; the order in which they are written determines their role in a given rule
- surround the combination in parentheses
- *et voila!*

The syntax of Scheme

“Sum of squares of initial seven prime numbers”

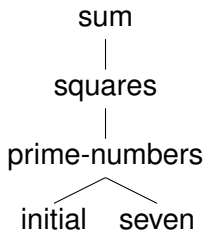


```
(sum (squares (prime-numbers initial seven)))
```

Note: the latter is more versatile!

The syntax of Scheme

“Sum of squares of initial seven prime numbers”

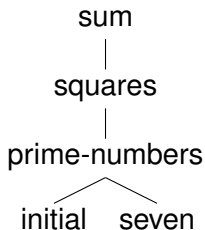


```
(sum (squares (prime-numbers initial seven)))
```

Note: the latter is more versatile!

The syntax of Scheme

“Sum of squares of initial seven prime numbers”

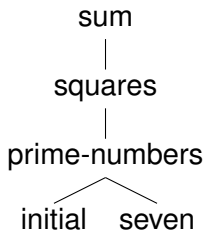


```
(sum (squares (prime-numbers initial seven)))
```

Note: the latter is more versatile!

The syntax of Scheme

“Sum of squares of initial seven prime numbers”



```
(sum (squares (prime-numbers initial seven)))
```

Note: the latter is more versatile!

The syntax of Scheme

A Quiz: which syntax is better?

```
int factorial(int n) {  
    int result = 1;  
    for (int i = 1; i <=  
n; ++i)  
        result *= i;  
    return result;  
}
```

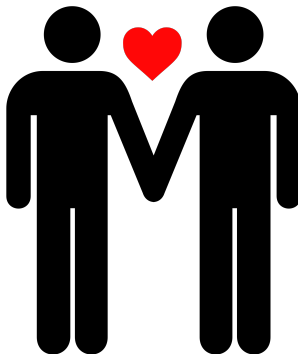
```
function factorial(n:  
integer): integer;  
var  
    i, result: integer;  
begin  
    result := 1;  
    for i := 2 to n do  
        result := result * i;  
    factorial := result  
end;
```

The syntax of Scheme

Homoiconicity

The syntax of Scheme

Homoiconicity



The semantics of Scheme

λ -calculus

- substitution model of computation
 $\lambda x.$ **This is x !**
 $(\lambda x.$ **This is x !)**it**
 \Rightarrow **This is it!****
- lexical scoping, applicative order of evaluation

The semantics of Scheme

λ -calculus

- substitution model of computation
 $\lambda x.$ **This is x !**
 $(\lambda x.$ **This is x !**)**it**
 \Rightarrow **This is it!**
- lexical scoping, applicative order of evaluation

The semantics of Scheme

λ -calculus

- substitution model of computation
 $\lambda x.$ **This is x !**
 $(\lambda x.$ **This is x !** $)it$
 \Rightarrow **This is it!**
- lexical scoping, applicative order of evaluation

The semantics of Scheme

λ -calculus

- substitution model of computation
 $\lambda x.$ **This is x !**
 $(\lambda x.$ **This is x !)it
 \Rightarrow **This is it!****
- lexical scoping, applicative order of evaluation

The semantics of Scheme

λ -calculus

- substitution model of computation
 $\lambda x.$ **This is x !**
 $(\lambda x.$ **This is x !**)**it**
 \Rightarrow **This is it!**
- lexical scoping, applicative order of evaluation

The semantics of Scheme

λ -calculus

- substitution model of computation
 $\lambda x.$ **This is x !**
 $(\lambda x.$ **This is x !**)**it**
 \Rightarrow **This is it!**
- lexical scoping, applicative order of evaluation

The semantics of Scheme

Special forms:

- `(define definiendum definiens)`
- `(lambda (arguments ...) body)`
- `(if then else)`
- `(quote s-exp)`
- `(set! variable value)`
- `(begin expression ...)`

The semantics of Scheme

Special forms:

- (**define** *definiendum definiens*)
- (**lambda** (*arguments ...*) *body*)
- (**if** *then else*)
- (**quote** *s-exp*)
- (**set!** *variable value*)
- (**begin** *expression ...*)

The semantics of Scheme

Special forms:

- (**define** *definiendum definiens*)
- (**lambda** (*arguments ...*) *body*)
- (**if** *then else*)
- (**quote** *s-exp*)
- (**set!** *variable value*)
- (**begin** *expression ...*)

The semantics of Scheme

Special forms:

- `(define definiendum definiens)`
- `(lambda (arguments ...) body)`
- `(if then else)`
- `(quote s-exp)`
- `(set! variable value)`
- `(begin expression ...)`

The semantics of Scheme

Special forms:

- `(define definiendum definiens)`
- `(lambda (arguments ...) body)`
- `(if then else)`
- `(quote s-exp)`
- `(set! variable value)`
- `(begin expression ...)`

The semantics of Scheme

Special forms:

- `(define definiendum definiens)`
- `(lambda (arguments ...) body)`
- `(if then else)`
- `(quote s-exp)`
- `(set! variable value)`
- `(begin expression ...)`

The semantics of Scheme

Special forms:

- (**define** *definiendum definiens*)
- (**lambda** (*arguments ...*) *body*)
- (**if** *then else*)
- (**quote** *s-exp*)
- (**set!** *variable value*)
- (**begin** *expression ...*)

The semantics of Scheme

Control structures:

- `goto`
- `break`
- `continue`
- `return`
- `throw/catch`
- `yield`

The semantics of Scheme

Control structures:

- goto
- break
- continue
- return
- throw/catch
- yield

The semantics of Scheme

Control structures:

- goto
- break
- `continue`
- `return`
- `throw/catch`
- `yield`

The semantics of Scheme

Control structures:

- `goto`
- `break`
- `continue`
- `return`
- `throw/catch`
- `yield`

The semantics of Scheme

Control structures:

- `goto`
- `break`
- `continue`
- `return`
- `throw/catch`
- `yield`

The semantics of Scheme

Control structures:

- `goto`
- `break`
- `continue`
- `return`
- `throw/catch`
- `yield`

The semantics of Scheme

Control structures:

- `goto`
- `break`
- `continue`
- `return`
- `throw/catch`
- `yield`

The semantics of Scheme

Control structures:

- `goto`
- `break`
- `continue`
- `return`
- `throw/catch`
- `yield`
- `call-with-current-continuation`

The semantics of Scheme

Self-describability:

The evaluator, which determines the meaning of expressions in a programming language, is just another program.

— the most fundamental idea in programming

The semantics of Scheme

Self-describability:

The evaluator, which determines the meaning of expressions in a programming language, is just another program.

— the most fundamental idea in programming

The conventions of Scheme

A Quiz: which notation is better?

`under_score_notation`

`camelCaseNotation`

`why-even-bother?`

The conventions of Scheme

A Quiz: which notation is better?

`under_score_notation`

`camelCaseNotation`

`why-even-bother?`

The conventions of Scheme

Minimal number of rules?

```
K(SII(S(K(S(S(KS)
(S(K(S(KS))
(S(K(S(KK))
(S(K(S(K(S(K(S(SI(K(S(K(S(S(KS)K)I))
(S(S(KS)K)(SII(S(S(KS)K)I))))))K))))
(S(K(S(K(S(SI(K(S(K(S(SI(K(S(K(S(S(KS)K)I))
(S(S(KS)K)(SII(S(S(KS)K)I))
(S(S(KS)K))(S(SII)I(S(S(KS)K)I))))))
K))))))
(S(S(KS)K)(K(S(S(KS)K))))))
(K(S(K(S(S(KS)K))K))) (SII) II)
```

Learnable

The story of 7 year old Zora Ball:

<https://www.youtube.com/watch?v=9oEZjpEqCwM>

Easy to implement

Multitude of implementations:

- Chez
- Racket
- Kawa
- Gambit (+termite)
- Guile
- Biwa
- Iron Scheme
- ... and more (approx. 80):

<http://www.schemers.org/Implementations/>

Easy to implement

Multitude of implementations:

- Chez
- Racket
- Kawa
- Gambit (+termite)
- Guile
- Biwa
- Iron Scheme
- ... and more (approx. 80):

<http://www.schemers.org/Implementations/>

Easy to implement

Multitude of implementations:

- Chez
- Racket
- Kawa
- Gambit (+termite)
- Guile
- Biwa
- Iron Scheme
- ... and more (approx. 80):

<http://www.schemers.org/Implementations/>

Easy to implement

Multitude of implementations:

- Chez
- Racket
- Kawa
- Gambit (+termite)
- Guile
- Biwa
- Iron Scheme
- ... and more (approx. 80):

<http://www.schemers.org/Implementations/>

Easy to implement

Multitude of implementations:

- Chez
- Racket
- Kawa
- Gambit (+termite)
- Guile
- Biwa
- Iron Scheme
- ... and more (approx. 80):

<http://www.schemers.org/Implementations/>

Easy to implement

Multitude of implementations:

- Chez
- Racket
- Kawa
- Gambit (+termite)
- Guile
- Biwa
- Iron Scheme
- ... and more (approx. 80):

<http://www.schemers.org/Implementations/>

Easy to implement

Multitude of implementations:

- Chez
- Racket
- Kawa
- Gambit (+termite)
- Guile
- Biwa

• Iron Scheme

• ... and more (approx. 80):

<http://www.schemers.org/Implementations/>

Easy to implement

Multitude of implementations:

- Chez
- Racket
- Kawa
- Gambit (+termite)
- Guile
- Biwa
- Iron Scheme

• ... and more (approx. 80):

<http://www.schemers.org/Implementations/>

Easy to implement

Multitude of implementations:

- Chez
- Racket
- Kawa
- Gambit (+termite)
- Guile
- Biwa
- Iron Scheme
- ... and more (approx. 80):

<http://www.schemers.org/Implementations/>

Managable

Easy to manipulate syntax tree

- syntax transformers (macros)

```
(define-syntax (let ((name value) ...)
                body . *)
  ((lambda (name ...) body . *) value))
```

- single-expression comments

```
(i can #(comment out (nested)
expressions!) and you?)
```

- possible to evaluate single expressions on the fly (e.g. geiser)

Managable

Easy to manipulate syntax tree

- syntax transformers (macros)

```
(define-syntax (let ((name value) ...)
                body . *)
  ((lambda (name ...) body . *) value))
```

- single-expression comments

```
(i can #(comment out (nested)
expressions!) and you?)
```

- possible to evaluate single expressions on the fly (e.g. geiser)

Managable

Easy to manipulate syntax tree

- syntax transformers (macros)

```
(define-syntax (let ((name value) ...)
                body . *)
  ((lambda (name ...) body . *) value))
```

- single-expression comments

```
(i can #;(comment out (nested)
expressions!) and you?)
```

- possible to evaluate single expressions on the fly (e.g. geiser)

Managable

Easy to manipulate syntax tree

- syntax transformers (macros)

```
(define-syntax (let ((name value) ...)
                body . *)
  ((lambda (name ...) body . *) value))
```

- single-expression comments

```
(i can #;(comment out (nested)
expressions!) and you?)
```

- possible to evaluate single expressions on the fly (e.g. geiser)

Managable

Easy to manipulate syntax tree

- syntax transformers (macros)

```
(define-syntax (let ((name value) ...)
                body . *)
  ((lambda (name ...) body . *) value))
```

- single-expression comments

```
(i can #;(comment out (nested)
expressions!) and you?)
```

- possible to evaluate single expressions on the fly (e.g. geiser)

Fixable

Possible to override core bindings

- curried definitions

```
(define ((f x) y) body)  
=== (define f (lambda (y) (lambda (x) body)))
```

- pattern-matching lambdas

```
(lambda ((x . y)) x) === car
```

Fixable

Possible to override core bindings

- curried definitions

```
(define ((f x) y) body)  
=== (define f (lambda (y) (lambda (x) body)))
```

- pattern-matching lambdas

```
(lambda ((x . y)) x) === car
```

Fixable

Possible to override core bindings

- curried definitions

```
(define ((f x) y) body)  
=== (define f (lambda (y) (lambda (x) body)))
```

- pattern-matching lambdas

```
(lambda ((x . y)) x) === car
```

Original ideas

Some ideas unthinkable in other languages:

- running evaluator backwards:
<https://www.youtube.com/watch?v=eQL48qYDwp4>
- ferns/engines <http://projects.csail.mit.edu/wiki/pub/JoeNear/FernMonad/frons.pdf>
- lazy streams (“Cons Should Not Evaluate Its Arguments”)

Original ideas

Some ideas unthinkable in other languages:

- running evaluator backwards:
`https://www.youtube.com/watch?v=eQL48qYDwp4`
- ferns/engines `http://projects.csail.mit.edu/wiki/pub/JoeNear/FernMonad/frons.pdf`
- lazy streams (“Cons Should Not Evaluate Its Arguments”)

Original ideas

Some ideas unthinkable in other languages:

- running evaluator backwards:
<https://www.youtube.com/watch?v=eQL48qYDwp4>
- **ferns/engines** <http://projects.csail.mit.edu/wiki/pub/JoeNear/FernMonad/frons.pdf>
- lazy streams (“Cons Should Not Evaluate Its Arguments”)

Original ideas

Some ideas unthinkable in other languages:

- running evaluator backwards:
`https://www.youtube.com/watch?v=eQL48qYDwp4`
- ferns/engines `http://projects.csail.mit.edu/wiki/pub/JoeNear/FernMonad/frons.pdf`
- lazy streams (“Cons Should Not Evaluate Its Arguments”)

What does the “of” mean?

How do you memoize the order of arguments to a function?

```
(define (shortest-cycle #;from first . #;through rest)

  (define (shortest-path #;from node #;to target #;through nodes)
    (if (null? nodes)
        '((,node ,target) , (distance #;from node #;to target))
        (let* ((subpaths (map (lambda (node)
                                (let ((other-nodes (delete node #;from nodes)))
                                  (shortest-path #;from node #;to target
                                                    #;through other-nodes)))
                                nodes))
              ((path _) increased-cost
               (argmin (λ ((next . _) cost))
                        (+ cost (distance #;from node #;to next)))
              subpaths)))
        '((,node . ,path) ,increased-cost))))

  (apply values (shortest-path #;from first #;to first #;through rest)))
```

What does the “of” mean?

How do you memoize the order of arguments to a function?

```
(define (shortest-cycle #;from first . #;through rest)

  (define (shortest-path #;from node #;to target #;through nodes)
    (if (null? nodes)
        '((,node ,target) ,(distance #;from node #;to target))
        (let* ((subpaths (map (lambda (node)
                                (let ((other-nodes (delete node #;from nodes)))
                                  (shortest-path #;from node #;to target
                                                    #;through other-nodes)))
                                nodes))
              ((path _) increased-cost
               (argmin (λ ((next . _) cost)
                        (+ cost (distance #;from node #;to next)))
                       subpaths)))
          '((,node . ,path) ,increased-cost))))

  (apply values (shortest-path #;from first #;to first #;through rest)))
```

Showing and telling

Better explain than instruct; better show than tell

• • •




Showing and telling

Better explain than instruct; better show than tell

```
(define-chess-rules
 (initial-board:
  ( (♔ ♙ _ _ _ _ ♖ ♕)
    (♘ ♚ _ _ _ _ ♗ ♞)
    (♙ ♛ _ _ _ _ ♖ ♙)
    (♚ ♜ _ _ _ _ ♗ ♔)
    (♙ ♙ _ _ _ _ ♖ ♙)
    (♙ ♙ _ _ _ _ ♖ ♙)
    (♙ ♙ _ _ _ _ ♖ ♙)
    (♙ ♙ _ _ _ _ ♖ ♙)
    (♙ ♙ _ _ _ _ ♖ ♙)
    (♙ ♙ _ _ _ _ ♖ ♙) )
```

• • •

Showing and telling

```
...
(
  ((? ? ... □/_ )
    (: : ↗ : )
    (? _ ... ? )
    ( ? ... ? ) )
; ; =====
((? ? ...  )
  (: : ↗ : )
  (? _ ... ? )
  (_ ? ... ? ) )
(symmetries:  all-rotations))
...
```

Showing and telling

What does the following code do?

```
(define (upper-left-corner rect w h)
  (map (lambda (row)
        (take row w))
       (take rect h)))

(e.g. (upper-left-corner '((a b c d)
                           (e f g h)
                           (i j k l)) 3 2)

==> ((a b c)
      (e f g)))
```

Showing and telling

What does the following code do?

```
(define (upper-left-corner rect w h)
  (map (lambda (row)
        (take row w))
       (take rect h)))
```

```
(e.g. (upper-left-corner '((a b c d)
                           (e f g h)
                           (i j k l)) 3 2)

====> ((a b c)
        (e f g)))
```

Showing and telling

What does the following code do?

```
(define (upper-left-corner rect w h)
  (map (lambda (row)
        (take row w))
       (take rect h)))

(e.g. (upper-left-corner ' ((a b c d)
                             (e f g h)
                             (i j k l)) 3 2)

==> ((a b c)
      (e f g)))
```


Showing and telling

Is the purity really that important?

```
(define (cross-over daddy mommy)
  (assert (= (length daddy) (length mommy)))
  (let* ((position (random (length daddy)))
         (sperm (take daddy position))
         (ovum (drop mommy position)))
    `(,@sperm ,@ovum)))
```

Controversies (and responses)

Criticism:

- Lots of Irritating Superfluous Parentheses!
- Phil Wadler, “Why Calculating is Better Than Scheming”
 - no pattern matching
 - unfamiliar syntax
 - no types
 - no lazy evaluation
- ungooglable name
- no libraries
- R6RS controversy

Controversies (and responses)

Criticism:

- Lots of Irritating Superfluous Parentheses!
- Phil Wadler, “Why Calculating is Better Than Scheming”
 - no pattern matching
 - unfamiliar syntax
 - no types
 - no lazy evaluation
- ungooglable name
- no libraries
- R6RS controversy

Controversies (and responses)

Criticism:

- Lots of Irritating Superfluous Parentheses!
- Phil Wadler, “Why Calculating is Better Than Scheming”
 - no pattern matching
 - unfamiliar syntax
 - no types
 - no lazy evaluation
- ungooglable name
- no libraries
- R6RS controversy

Controversies (and responses)

Criticism:

- Lots of Irritating Superfluous Parentheses!
- Phil Wadler, “Why Calculating is Better Than Scheming”
 - no pattern matching
 - unfamiliar syntax
 - no types
 - no lazy evaluation
- ungooglable name
- no libraries
- R6RS controversy

Controversies (and responses)

Criticism:

- Lots of Irritating Superfluous Parentheses!
- Phil Wadler, “Why Calculating is Better Than Scheming”
 - no pattern matching
 - unfamiliar syntax
 - no types
 - no lazy evaluation
- ungooglable name
- no libraries
- R6RS controversy

Controversies (and responses)

Criticism:

- Lots of Irritating Superfluous Parentheses!
- Phil Wadler, “Why Calculating is Better Than Scheming”
 - no pattern matching
 - unfamiliar syntax
 - no types
 - no lazy evaluation
- ungooglable name
- no libraries
- R6RS controversy

Controversies (and responses)

Criticism:

- Lots of Irritating Superfluous Parentheses!
- Phil Wadler, “Why Calculating is Better Than Scheming”
 - no pattern matching
 - unfamiliar syntax
 - no types
 - no lazy evaluation
- ungooglable name
- no libraries
- R6RS controversy

Controversies (and responses)

Criticism:

- Lots of Irritating Superfluous Parentheses!
- Phil Wadler, “Why Calculating is Better Than Scheming”
 - no pattern matching
 - unfamiliar syntax
 - no types
 - no lazy evaluation
- ungooglable name
- no libraries
- R6RS controversy

Controversies (and responses)

Criticism:

- Lots of Irritating Superfluous Parentheses!
- Phil Wadler, “Why Calculating is Better Than Scheming”
 - no pattern matching
 - unfamiliar syntax
 - no types
 - no lazy evaluation
- ungooglable name
- no libraries
- R6RS controversy

Controversies (and responses)

Criticism:

- Lots of Irritating Superfluous Parentheses!
- Phil Wadler, “Why Calculating is Better Than Scheming”
 - no pattern matching
 - unfamiliar syntax
 - no types
 - no lazy evaluation
- ungooglable name
- no libraries
- R6RS controversy

Controversies (and responses)

The newcomer's perspective

Why is it that you consider the R5RS the last “true” Scheme? (...)

Is it a particular feature set? A limit on page count?

Utter minimalism to the point of being virtually useless without implementation-dependent extensions?

What? (...)

*It seems to me that many Schemers have some ideal in their minds about what Scheme *really* is, but that language is imaginary, existing only in the R5RS document. (...)*

— Internet user, comp.lang.scheme

Controversies (and responses)

Scheme would still be useful as a notation for expressing computational methods even in the absence of a mechanical implementation.

— Revised⁵ Report on the Algorithmic Language Scheme

Important books

Books:

- Hal Abelson, Gerald Sussman, “Structure and Interpretation of Computer Programs”,
<https://mitpress.mit.edu/sicp/>
- Daniel Friedman, Mitchell Wand, “Essentials of Programming Languages”, <http://www.eopl3.com/>
- Mathias Felleisen, Robert Findler, Matthew Flat, Shiram Krishnamurthi, “How to Design Programs”,
<http://www.htdp.org/>
- Daniel Friedman et al., “The Little Schemer”, “The Seasoned Schemer”, “The Reasoned Schemer”, “The Little Prover”

Important books

Books:

- Hal Abelson, Gerald Sussman, “Structure and Interpretation of Computer Programs”,
<https://mitpress.mit.edu/sicp/>
- Daniel Friedman, Mitchell Wand, “Essentials of Programming Languages”, <http://www.eopl3.com/>
- Mathias Felleisen, Robert Findler, Matthew Flat, Shiram Krishnamurthi, “How to Design Programs”,
<http://www.htdp.org/>
- Daniel Friedman et al., “The Little Schemer”, “The Seasoned Schemer”, “The Reasoned Schemer”, “The Little Prover”

Important books

Books:

- Hal Abelson, Gerald Sussman, “Structure and Interpretation of Computer Programs”,
<https://mitpress.mit.edu/sicp/>
- Daniel Friedman, Mitchell Wand, “Essentials of Programming Languages”, <http://www.eopl3.com/>
- Mathias Felleisen, Robert Findler, Matthew Flat, Shiram Krishnamurthi, “How to Design Programs”,
<http://www.htdp.org/>
- Daniel Friedman et al., “The Little Schemer”, “The Seasoned Schemer”, “The Reasoned Schemer”, “The Little Prover”

Important books

Books:

- Hal Abelson, Gerald Sussman, “Structure and Interpretation of Computer Programs”,
<https://mitpress.mit.edu/sicp/>
- Daniel Friedman, Mitchell Wand, “Essentials of Programming Languages”, <http://www.eopl3.com/>
- Mathias Felleisen, Robert Findler, Matthew Flat, Shiram Krishnamurthi, “How to Design Programs”,
<http://www.htdp.org/>
- Daniel Friedman et al., “The Little Schemer”, “The Seasoned Schemer”, “The Reasoned Schemer”, “The Little Prover”

Important books

Books:

- Hal Abelson, Gerald Sussman, “Structure and Interpretation of Computer Programs”,
<https://mitpress.mit.edu/sicp/>
- Daniel Friedman, Mitchell Wand, “Essentials of Programming Languages”, <http://www.eopl3.com/>
- Mathias Felleisen, Robert Findler, Matthew Flat, Shiram Krishnamurthi, “How to Design Programs”,
<http://www.htdp.org/>
- Daniel Friedman et al., “The Little Schemer”, “The Seasoned Schemer”, “The Reasoned Schemer”, “The Little Prover”

Unimportant books

Books – continued:

- Panicz Maciej Godek, “A Pamphlet Against R”,
<http://panicz.github.io/pamphlet>

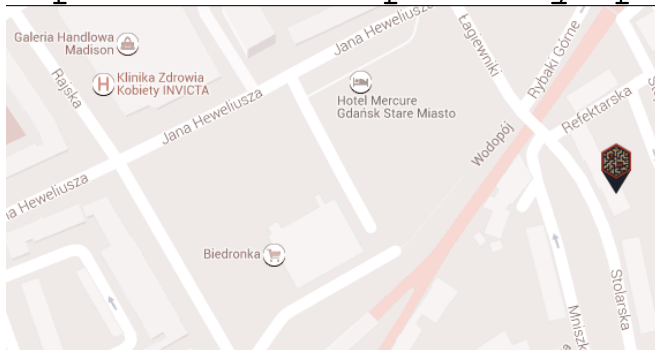
Unimportant books

Books – continued:

- Panicz Maciej Godek, “A Pamphlet Against R”,
<http://panicz.github.io/pamphlet>

Ads

<http://roomofplenty.pl/>



The End

Questions?
godek.maciek@gmail.com
@PaniczGodek