

Compiler Construction

Mayer Goldberg \ Ben-Gurion University

2025-02-14

Chapter 1

Goals

- ▶ Establishing common language & vocabulary
- ▶ Understanding the “big picture”

Agenda

- ▶ Some background in **programming languages**
 - ▶ *Abstraction*
 - ▶ *Dynamic vs Static*
 - ▶ *Functional vs Imperative* languages
- ▶ Introduction to **compiler construction**
- ▶ Introduction to the **ocaml** programming language

Abstraction

- ▶ Abstraction is a way of moving from a **particular** to the **general**
- ▶ Abstraction appears in mathematics, logic, and in computer science
- ▶ Abstraction is a force-multiplier, and a great time-saver

Abstraction (*continued*)

- Abstraction in logic: going from propositions to **quantified propositions**. For example:

$Hx \quad x \text{ is hungry}$

$Ex \quad x \text{ goes to eat}$

- Specific: $Hx_0 \rightarrow Ex_0$ means that if [the specific] x_0 is hungry, then [the specific] x_0 goes to eat
- General:

$$\exists x(Hx \rightarrow Ex)$$

$$\forall x(Hx \rightarrow Ex)$$

Abstraction (*continued*)

- ▶ Abstraction in mathematics: going from expression to function
 - ▶ Consider the expression $x \cdot \sin^2(1 + x)$
 - ▶ This expression denotes a **number**; not a **function**
 - ▶ Writing $(x \cdot \sin^2(1 + x))'$ is actually an abuse of notation, because only functions have derivatives
 - ▶ When we write $(x \cdot \sin^2(1 + x))'$ what we **intend** is the derivative of a function, the argument of which is x , and the body of which is $x \cdot \sin^2(1 + x)$:
 - ▶ $f(x) = x \cdot \sin^2(1 + x)$
 - ▶ $f'(x) = \sin^2(1 + x) + 2x \cdot \sin(1 + x) \cdot \cos(1 + x)$

Abstraction (*continued*)

- ▶ Abstraction in mathematics: going from expression to function
 - ▶ The only thing “wrong” here is that we gave the function a name — f
 - ▶ This is “wrong” because the choice of the name is arbitrary
 - ▶ Functions can be written anonymously using λ -notation:
 - ▶ The symbol λ (“lambda”) just means “anonymous function”
 - ▶ In theory (the λ -calculus): $\lambda x.(x \cdot \sin^2(1 + x))$
 - ▶ In programming (Scheme): `(lambda (x) (* x (square (sin (+ 1 x)))))`
 - ▶ The expression $x \cdot \sin^2(1 + x)$ is concrete, and for a specific x
 - ▶ The expression $\lambda x.(x \cdot \sin^2(1 + x))$ is an abstraction: We say that it abstracts the variable x over the concrete expression

Abstraction (*continued*)

- ▶ Abstraction in programming
 - ▶ Functional programming: Similar to abstraction mathematics
 - ▶ Expressions are abstracted into functions
 - ▶ Functions are abstracted into higher order functions
 - ▶ Collections of functions are abstracted into modules
 - ▶ Modules are abstracted into functors
 - ▶ Modules are abstracted into signatures
 - ▶ Procedural programming
 - ▶ Statements are abstracted into procedures

Abstraction (*continued*)

- ▶ Abstraction in programming (*continued*)
 - ▶ Object-oriented programming
 - ▶ Objects are abstracted into classes
 - ▶ Classes are abstracted into generics & templates
 - ▶ Classes are abstracted into packages
 - ▶ Logic programming
 - ▶ Similar to abstraction in logic
 - ▶ Propositions are abstracted into predicates
 - ▶ Textual abstraction
 - ▶ Text is abstracted into templates

What textual abstraction looks like

Concrete

September 12, 2001

Dear John:

I would like to interest you in our insurance policies. With a wife and three children to look after, I am sure you desire the extra sense of security afforded by the extended coverage of our life-insurance policies.

Abstract

`$(DATE)`

Dear `$(FIRST-NAME)`:

I would like to interest you in our insurance policies. With `$(DEPENDENTS)` to look after, I am sure you desire the extra sense of security afforded by the extended coverage of our life-insurance policies.

What textual abstraction looks like (*cont*)

- ▶ Notice the text variables that are embedded within the template:
\$(VARIABLE_1), \$(VARIABLE_2), \$(VARIABLE_3)
- ▶ All *junk mail*, whether printed or electronic, is generated in this way
- ▶ In word-processing, this functionality is known as **mail merge**:
You are **merging** templates and tables from a database or a spreadsheet...

Further Reading

-  The Structure and Interpretation of Computer Programs (SICP), chapters 1.3, 2.1, 2.3.1, 2.3.2
-  The M4 textual macro language
-  Paul Graham's article *Beating the Averages*

Dynamic vs Static

- **Dynamic** & **Static** are used in many areas of computer science
- In *programming languages theory*, these terms have a very specific meaning:

Dynamic

To say that something is **dynamic**, means that it can be —

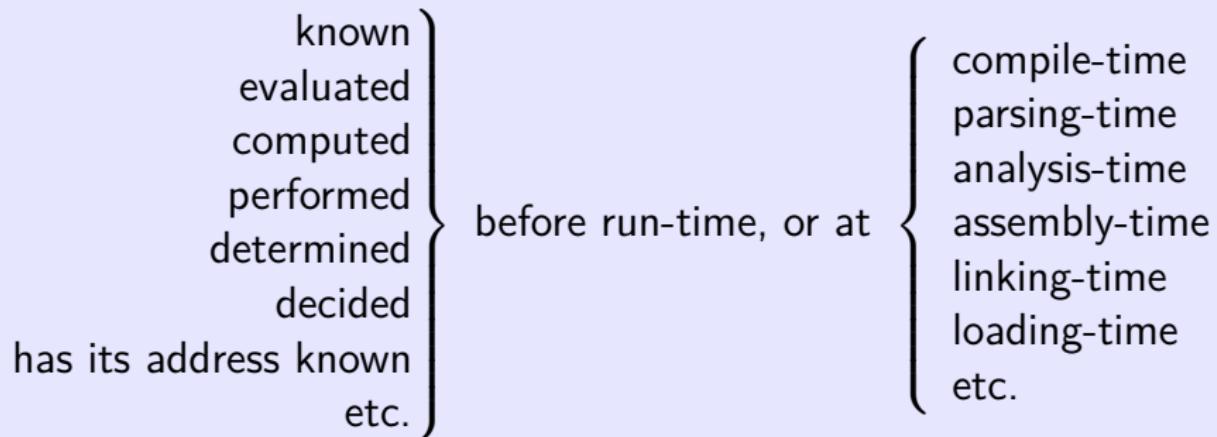
known
evaluated
computed
performed
determined
decided
has its address known
etc.

} no sooner than run-time

Dynamic vs Static (*continued*)

Static (aka *Lexical*)

To say that something is **static**, means that it can be —



Dynamic vs Static (*continued*)

From a type-theoretic point of view, the distinction of dynamic vs static is called the **phase distinction**:

- ▶ Programming languages where types exist at compile-time only are said to observe the **phase distinction**.



This means that:

- ☞ In languages that observe the **phase distinction**:

- ▶ Variables have types
- ▶ Data have no type

- ☞ In languages that fail to observe the **phase distinction**:

- ▶ Variables have no type
- ▶ Data have types

Dynamic vs Static (*continued*)

What is meant by compile-time type-information?

- ▶ Types are **annotations** that exist during compile-time
 - ▶ Variables, expressions, sub-expressions are all assigned types
 - ▶ Type checking is performed at compile-time, verifying that data is used according to its type
 - ▶ **Example:** The function $f: \tau \rightarrow \sigma$ can be applied to an argument of type τ , and returns a result of type σ
- ▶ Types are not represented during run-time
 - ▶ **Example:** A 64-bit integer is represented in **two's complement**, with bit #63 denoting the **sign**

Dynamic vs Static (*continued*)

What is meant by run-time type-information (RTTI)?

- ▶ Much less type-information is available at compile-time
- ▶ Much less type-checking can be done at compile-time
 - ▶ Example: The function `(lambda (x) x)` can be applied to anything, including itself. In general, nothing is known at compile-time about the type of `x`
- ▶ The type is a part of the binary encoding of the data at run-time
 - ▶ Example: An integer in Scheme is represented using a n -bits, some of those bits encode the type

Dynamic vs Static (*continued*)

What is meant by run-time type-information (RTTI)?

- ▶ Example: In our compiler we encode the RTTI as follows:

```
%define T_void          0
%define T_nil           1
%define T_char          2
%define T_string         3
%define T_closure         4
%define T_undefined        5
%define T_boolean          8
%define T_boolean_false   (T_boolean | 1)
%define T_boolean_true    (T_boolean | 2)
%define T_number           16
%define T_integer          (T_number | 1)
%define T_fraction          (T_number | 2)
%define T_real              (T_number | 3)
...
...
```

Dynamic vs Static (*continued*)

Open Question

Can you explain the motivation/rationale behind the way we encode **Booleans** and **numbers** in our compiler?

```
%define T_boolean          8
%define T_boolean_false    (T_boolean | 1)
%define T_boolean_true     (T_boolean | 2)
%define T_number           16
%define T_integer          (T_number | 1)
%define T_fraction         (T_number | 2)
%define T_real              (T_number | 3)
```

Dynamic vs Static (*continued*)

Question

Pick the feature that is strictly dynamic:

-  The address of a variable in Java (within the JVM)
-  Whether a C function is called with the syntactically-correct number of arguments
-  Whether a variable is initialized in Java
-  How much memory is taken up by an object (as in C++ or Java)
-  Whether an *array reference* in C is within bounds

Dynamic vs Static (*continued*)

Question

Pick the feature that is strictly static:

-  Whether an expression in the source code is positive
-  Whether a function shall be called
-  How many times a loop shall be performed
-  Whether an array de-reference is valid
-  Whether all parentheses are balanced.

Dynamic vs Static (*continued*)

Question

Pick the feature that is strictly static:

-  Whether a while-loop terminates in C
-  Whether an address on the stack is a frame pointer
-  Whether a for-loop iterates more than n times in C
-  Whether the value of an expression is greater than M
-  The type of a variable in C

Dynamic vs Static (*continued*)

Question

Pick the feature that is strictly dynamic:

-  What exceptions appear in the code
-  The type of a method
-  Whether all variables have been initialized before use
-  Whether a function-call is in tail-position
-  Whether a program de-allocates all dynamically-allocated resources

Dynamic vs Static (*continued*)

Question

Pick the correct statement:

-  The terms **static** and **dynamic** only describe the way programming languages handle types
-  There is no feature that is checked dynamically that could have been checked statically
-  It is impossible to check/enforce/validate dynamic behavior/features
-  No feature is both static and dynamic
-  The success of opening a file cannot be determined statically

Further Reading

-  The Dragon Book (2nd edition), page 25 (subsection 1.6.1)
-  Phase distinction in type theory (paper)
-  Phase distinction in the Wikipedia

Imperative vs functional languages

- ▶ You've already been exposed to functional programming in your PPL course
- ▶ Functional languages are often described as languages lacking in **side effects**
- ▶ Imperative languages have **side effects**

Defining a language in terms of what it is *not* does not teach us much.

- ▶ What can we say *positively* about functional programming?

Imperative vs functional languages (*continued*)

- ▶ Computer science is the illegitimate child of **mathematics** and **electrical engineering** 
- 👉 Electrical engineering gave us the *hardware*, the actual machine
- 👉 Nearly all **ideas** come from mathematics 

Computers

- ▶ Digital electronics: Gates, flip-flops, latches, memory, etc
- ▶ Boolean functions that *read* their inputs from memory & *write* their outputs to memory
- ▶ Reading & writing are synchronized using a clock (with a frequency measured in Hz, KHz, MHz, GHz...)
- ▶ A finite-state machine

Imperative vs functional languages (*continued*)

- ▶ We cannot design large software-systems while thinking at the level of digital electronics
- ▶ We need some theoretical foundations for programming & computation

Imperative vs functional languages (*continued*)

What is mathematics?

- ▶ A language
- ▶ A set of ideas, notions, definitions, techniques, results, all expressed in this language

What computer science takes from mathematics?

- ▶ Programming is based on **computable mathematics**
- ▶ **Theoretical** computer science uses **all** of mathematics
- 👉 In short: Everything!

Imperative vs functional languages (*continued*)

What programming languages take from mathematics?

- ▶ A language
- ▶ Notions such as functions, recursion, iteration, operators, variables, expressions, evaluation, numbers, types, sets, relations, ordered n -tuples, structures, graphs, etc
- ▶ Operations such as arithmetic, Boolean, structural (e.g., on n -tuples, sets, etc), abstraction, mapping, folding, etc.

Nearly all of the ideas in computer science come from mathematics!

Imperative vs functional languages (*continued*)

Mathematical objects are free of the limitations that beset computers:

Real-world compromises

- ▶ Computers can only **approximate** real numbers
- ▶ Computers cannot implement infinite tape (**Turing machines**)
- ▶ Mathematical objects are **cheaper** than objects created on a physical computer:
 - ▶ Functions are mappings; They take no time to compute or evaluate!
 - ▶ Knowing that an object exists is often all we need!
 -  We often don't need to know its value
 - ▶  We often don't need to know how to compute it
 - ▶ Bad things **cannot** happen:
 - ▶ No **exceptions, errors, incorrect results**
 - ▶ Nothing is lost, nothing is “too big” or “too much”

Imperative vs functional languages (*continued*)

Functional programming languages

- ▶ Closer to mathematics
- ▶ Easier to reason about
- ▶ Easier to transform
- ▶ Easier to generate automatically

Imperative programming languages

- ▶ Farther from mathematics
- ▶ Harder to reason about
- ▶ Harder to transform
- ▶ Harder to generate automatically

Imperative vs functional languages (*continued*)

Example of non-mathematical ideas: Side effects

Imagine having to teach C programming to a logician

- ▶ To simplify matters, let's pretend there is a `string` type in C (rather than `char *`)
- ▶ You teach them a simplified version of `printf`:
 - ▶ Only takes a single string argument
 - ▶ Returns an `int`: the number of characters in the string
- ▶  **Remember:** `stdio.h` defines the prototype for `printf` to return `int` (the number of characters printed)
- ▶ **Roughly:** `printf : string -> int`

Imperative vs functional languages (*continued*)

Example of non-mathematical ideas: Side effects

- ▶ But the logician objects: He already knows of a function from `string -> int` that returns the number of characters in the string
 - ☞ `strlen : string -> int`
 - ☜ He wants to know the difference between `printf` and `strlen`

Imperative vs functional languages (*continued*)

Side-effects: The Dialogue

- ▶ You: "Simple, printf **also** prints the string to the screen!"
- ▶ Logician: "What does it mean **to print??**"
- ▶ You: "Seriously?? The printf function **prints** its argument to the screen & also returns the number of characters it printed!"
- ▶ Logician: "But you said the domain of printf is **string** -> int, right?"
- ▶ You: "Yes, so?"
- ▶ Logician: "Then where's the screen??"
- ▶ You: "In front of you!"
- ▶ Logician: "Where's the screen **in the domain of the function printf!**"

Imperative vs functional languages (*continued*)

Side-effects: The Dialogue (*continued*)

- ▶ You: “It isn’t in the domain. You can think of the screen as a **global variable**. ”
- ▶ Logician: “I have no idea what you mean: How can the screen be a variable when it’s not passed as a parameter, and its type is not expressed in the domain of the function??”
- ▶ You: “But that’s the whole point of this printing being a **side effect**: It is not expressed in the type!”
- ▶ Logician: “Well, then `printf` isn’t a function!”
- ▶ You: “Ummm...”
-  Logician (having a *Eureka!*-moment): “I get it now! You got the domain of `printf` all wrong!”

Imperative vs functional languages (*continued*)

Side-effects from a logical point of view

- ▶ The **real** type of `printf` is `string × screen → int × screen`
- ▶ The underlined parts of the type are **implicit**, i.e., they are not written explicitly in the original type given for `printf`
- ▶ The implicit parts of the type form the **environment**
- ▶ The function-call mentions only the explicit arguments
- ▶ Leaving out the implicit arguments in the function call creates an **illusion of change**, as if the environment has changed: An illusory notion of **time** has been created; We have the environment **before** the function call, and the environment **after** the function call
- ▶ In fact, nothing has changed: The screen in the domain has been **mapped** into another screen in the range.

Imperative vs functional languages (*continued*)

Side-effects from a logical point of view (*continued*)

- ▶ Introducing side-effects introduces **discrete time**
- ▶ Having introduced time, we must now introduce **sequencing**:

```
{  
    printf("Hello ");  
    printf("world!\n");  
}
```

Imperative vs functional languages (*continued*)

Side-effects from a logical point of view (*continued*)

The notion of **sequencing**, like the notion of **time**, is illusory:

- ▶ The screen object in the **range** of `printf("Hello ")`; is the screen object in the **domain** of `printf("world!\n")`;
 - ▶ So the two `printf` expressions are **nested**, and this is why their “ordering” matters!
-  For the very same reason that $f(g(x))$ generally does not the same as $g(f(x))$...

Imperative vs functional languages (*continued*)

Functional Rendition

Imperative C

```
{  
    printf("Hello ");  
    printf("World!\n");  
}
```

$\langle "Hello ", screen \rangle \Rightarrow$
 $\langle 6, screen \oplus "Hello " \rangle$

$\langle "World!\n", screen \oplus "Hello " \rangle \Rightarrow$
 $\langle 7, screen \oplus "Hello World!\n" \rangle$

- The **return value** of the second call to `printf`
- The **screen** after the second call to `printf`

Imperative vs functional languages (*continued*)

Functional programming languages

- ▶ Closer to the mathematical notions of function, variable, evaluation
- ▶ Explicit about **types**
- ▶ Does not include notions that are not native to mathematics, such as **time**, **side-effect**, **environment**, etc
- ▶ Offers many other advantages

Imperative vs functional languages (*continued*)

Imperative programming languages

- ▶ Farther away from the mathematical notions such as function, variable, evaluation
- ▶ Hides information through the use of implicit arguments (for convenience)
- ▶ Harder to reason about: Contains notions such as **side effects**, **sequences**, **environment**, etc., that require translation before we can reason about them
- ▶ Abstraction is harder, prone to errors
- ▶ Side-effects create implicit, knotty inter-dependencies between various parts of a software system, making it harder to debug

Imperative vs functional languages (*continued*)

Abstraction in functional programming languages

- ▶ Values \Rightarrow Expressions \Rightarrow Functions
- ▶ Higher-order functions
- ▶ Mathematical operators: mapping, folding, filtering, partitioning, etc
- ▶ The interpreter evaluates expressions

Abstraction in imperative programming languages

- ▶ State \Rightarrow Change \Rightarrow Commands \Rightarrow Procedures
- ▶ Object-oriented programming
- ▶ Imperative \equiv Based upon commands (*imperare* means *to command* in Latin)
- ▶ The interpreter performs commands

Programming paradigms: A reality check

- ▶ There are very few strictly functional languages, i.e., languages in which side-effects cannot be expressed
- ▶ Most functional languages are really **quasi-functional**: They don't make it impossible to use side-effects, but they don't make these easy or fun to use
- ▶ Most new imperative languages do include features from functional languages
 - ▶ anonymous functions (`lambda`)
 - ▶ higher-order functions
 - ▶ modules/namespaces/functors

Imperative vs functional languages (*continued*)

Question

Languages based on the functional paradigm —

- 👎 do not allow us to perform side-effects
- 👎 do not support object-oriented programming
- 👎 are not suited to model and solve “real” problems
- 👎 are slow and inefficient
- 👍 support high order functions

Further Reading

- 🔗 Comparing programming paradigms
- 🔗 *What is functional programming* (blog post)



Introduction to compiler construction

- ▶ L_1 : Language
- ▶ L_2 : Language
- ▶ P_L : A program in language L
- ▶ Values: A set of values
- ▶ $\llbracket \cdot \rrbracket$: Semantic brackets

Interpreters evaluate/perform

The functional picture (evaluate)

- ▶ An interpreter is a function $\text{Int}_L : L \rightarrow \text{Values}$
- ▶ Interpreters map **expressions** to their **values**
- ▶ For example: In a functional subset of Scheme, we have
 $\text{Int}_{\text{funcScheme}} \llbracket (+ 3 5) \rrbracket = 8$

Introduction to compiler construction

Interpreters evaluate/perform

The imperative picture (perform)

- ▶ An interpreter is a function
 $\text{Int}_L : L \times \text{Environment} \rightarrow \text{Values} \times \text{Environment}$
- ▶ Interpreters map the **product** of an **expression** and an **environment** to the product of a **value** and an **environment**
- ▶ The environments are **implicit** in the imperative language
- ▶ When the environment in the domain is not equal to the environment in the range, an **illusion of change** has been created: “Something changed in the environment”
- ▶ For example: In the full, imperative Scheme, we have
 $\text{Int}_{\text{Scheme}} \llbracket \langle (\text{define } x 3), \{x \mapsto \text{undefined}\} \rangle \rrbracket = \langle \#\langle \text{void} \rangle, \{x \mapsto 3\} \rangle$

Introduction to compiler construction

Compilers translate

- ▶ A compiler is a function $\text{Comp}_{L_1}^{L_2} : L_1 \rightarrow L_2$
- ▶ Compilers **translate** programs from one language to another
- ▶ Let $P_{L_1} \in L_1$, then $\text{Comp}_{L_1}^{L_2} [P_{L_1}] \in L_2$
- ▶ The **correctness** of the translation is established using interpreters for both the source and target language:

$$\text{Int}_{L_1} [P_{L_1}] = \text{Int}_{L_2} [\text{Comp}_{L_1}^{L_2} [P_{L_1}]]$$

Introduction to compiler construction

Compilers translate (*continued*)

- We may chain any number of compilers together:

$$\begin{aligned}\text{Int}_{L_1}[\![P_{L_1}]\!] &= \text{Int}_{L_2}[\!\![\text{Comp}_{L_1}^{L_2}[\![P_{L_1}]\!]]\!] \\ &= \text{Int}_{L_3}[\!\![\text{Comp}_{L_2}^{L_3}[\!\![\text{Comp}_{L_1}^{L_2}[\![P_{L_1}]\!]]\!]]\!] \\ &= \text{Int}_{L_4}[\!\![\text{Comp}_{L_3}^{L_4}[\!\![\text{Comp}_{L_2}^{L_3}[\!\![\text{Comp}_{L_1}^{L_2}[\![P_{L_1}]\!]]\!]]\!]]\!] \\ &= \dots \text{etc.}\end{aligned}$$

Introduction to compiler construction

Question

Compiled code is generally faster than interpreted code. So why do we need interpreters? Why not stick with compiled code?

-  It's easier to program in interpreted languages
-  It's easier to debug interpreted code
-  Interpreters are more flexible than compilers
-  The difference is pretty much a matter of personal taste
-  Interpreters are the only way to reach values

Introduction to compiler construction

Question

When we program directly in machine language, where's the interpreter?

- 👎 The operating system is the interpreter
- 👎 The **shell** is the interpreter
- 👎 There is no interpreter
- 👎 The process of translating code to machine language is the underlying interpretation
- 👍 The microprocessor is a hardware implementation of an interpreter for the given micro-architecture

Introduction to compiler construction

Question

If interpreters are a logical necessity, why do we need a compiler?

-  For generality
-  For ease
-  For flexibility
-  For portability
-  For optimizations

Introduction to compiler construction

Another way of looking at the question

A compiler from language L to language L (itself!) is...
...just an optimizer!

What is an optimization

A compiler optimization is a **semantics-preserving** transformation that results in **more efficient code**

Semantics-preserving (\equiv meaning-preserving)

- Returns the same value (as the original code) under interpretation

More efficient code

- The interpreter takes less resources to evaluate the code.
Resources include:
 - 👉 Execution time
 - Computer memory
 - Network traffic
 - Microprocessor registers

or any other resource consumed by the code

What is an optimization (*continued*)

Special-purpose compilers may optimize for non-standard “resources”:

- ▶ Compilers to g-code, a language for CNCs, will try to minimize motion
- ▶ Compilers of graphs to visual presentations will try to minimize crossovers of edges
- ▶ Compilers of document-layout languages will try to minimize “widows”, “orphans”, hyphenations, and other typographically problematic conditions

What is an optimization (*continued*)

Question

Why do programmers write less-than-optimized code? What is the rationale for having compilers be in charge of optimizations? Are good programmers **really** that rare??

- ▶ Good code is hard to write
- ▶ Some programmers are incompetent
- ▶ Compilers are faster at detecting opportunities for optimizations
- ▶ Consistent output: Once debugged, compilers make no mistakes

All these reasons are true, but irrelevant!



Can you think of a reason why a programmer might **intentionally** write less-efficient code?

What is an optimization (*continued*)

Which code is better:

Code I

```
for (i = 0; i < 200; ++i) { A[i] = 0; }
for (i = 0; i < 200; ++i) { B[i] = 0; }
```

Code II

```
for (i = 0; i < 200; ++i) { A[i] = 0; B[i] = 0; }
```

Code III

```
#define M 200
#define N 200
...
for (i = 0; i < M; ++i) { A[i] = 0; }
for (i = 0; i < N; ++i) { B[i] = 0; }
```

What is an optimization (*continued*)

Analysis

► Code I

- ▶ less general than Code III
- ▶ less efficient than Code II
- ▶ less maintainable than Code III

► Code II

- ▶ less general than Code I
- ▶ more efficient than Code I, Code III
- ▶ less maintainable than Code I

► Code III

- ▶ more general than Code I, Code II
- ▶ less efficient than Code II
- ▶ more maintainable than Code I, Code II

What is an optimization (*continued*)

So which code is better?

FACT: Optimizing the loops in **Code III**, converting it to **Code II**, is performed by most compilers today

- ▶ Clearly **Code III** is better!

Conclusion

People write less-than-optimal code because:

- ▶ It is more general
- ▶ It is more maintainable
- ▶ It is more flexible
- ▶ Most compilers optimize away such inefficiencies

What is an optimization (*continued*)

How people use compilers

- ▶ Want to program in a more general, maintainable, flexible way
- ▶ Compilation is a point-in-time where generality is traded for efficiency:
 - ▶ Compilers are opportunistic
 - ▶ Compilers identify opportunities to trade generality for efficiency
 - ▶ The resulting code is unreadable, unmaintainable, machine-specific, and **fast**
 - ▶ We [normally] don't touch the compiled code: For programmers, it is only the quality of the **source code** that matters
-  Without optimizing compilers, we would be forced to write unmaintainable, overly-specific, machine-dependent code in order to obtain efficient code.

Composing & combining processors, interpreters, compilers, & programs

We can compose

- ▶ Processors running programs written in language L
- ▶ Interpreters written in L' , running programs written in language L
- ▶ Compilers written in L' , running on L , compile programs written in L'' into equivalent programs written in L''' (we are talking about 4 languages!)
- ▶ Programs written in L' , running on L

Remember Lego?



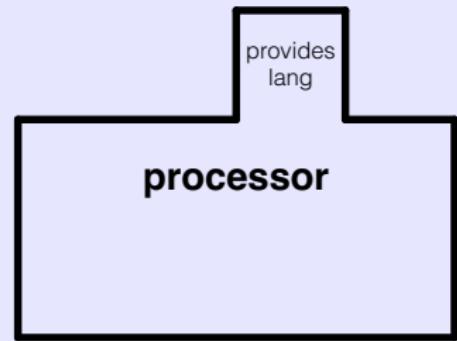
We have 5 new blocks for you!

Composing & combining processors, interpreters, compilers, & programs

A processor

- ▶ Hardware-implementations of interpreters for some language
- ▶ They have only 1 docking point:
The languages they provide

What it looks like

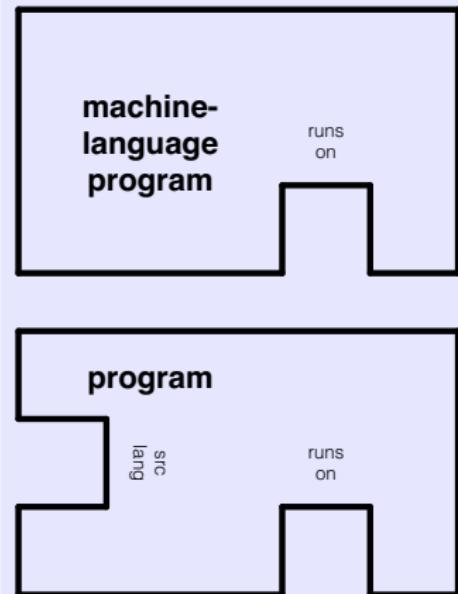


Composing & combining processors, interpreters, compilers, & programs

A program

- ▶ Programs have at least 1 docking point: The language they run on
- ▶ Mach Lang programs are hand-written directly in hex or binary. This is hardly ever done these days
- ▶ Other programs are written in a **source language** and **compiled** to some other language using a **compiler**. Such programs have an additional docking point

What it looks like

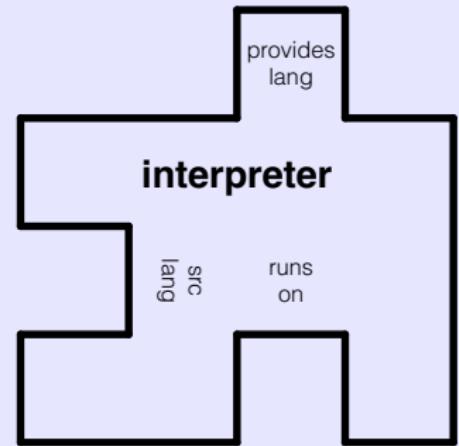


Composing & combining processors, interpreters, compilers, & programs

An interpreter

- ▶ Interpreters come with 3 docking points:
 - ▶ The language they provide
 - ▶ The language [interpreter] on which they run
 - ▶ The [source] language in which they were written

What it looks like

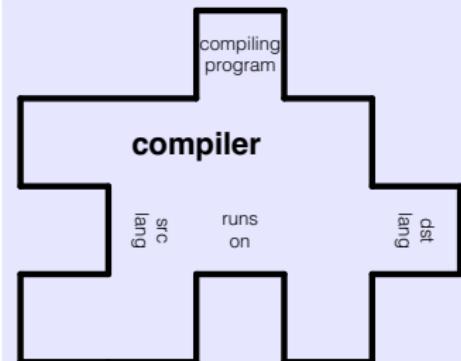


Composing & combining processors, interpreters, compilers, & programs

A compiler

- ▶ Compilers come with 4 docking points:
 - ▶ The language they compile **from**
 - ▶ The language they compile **to**
 - ▶ The language in which they were written
 - ▶ The language [interpreter] on which they run

What it looks like



Composing & combining processors, interpreters, compilers, & programs

Why bother with these pics??

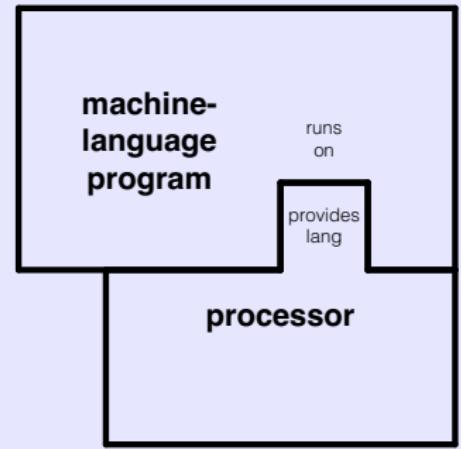
- ▶ Interpreters & compilers are often composed in complex ways
- ▶ Diagrams provide a simple, visual way to make sure that the compositions are correct

Composing & combining processors, interpreters, compilers, & programs

A machine language program running

- ▶ The program must be written in the same machine-language interpreted by the processor
- ▶ The two blocks join naturally

What it looks like

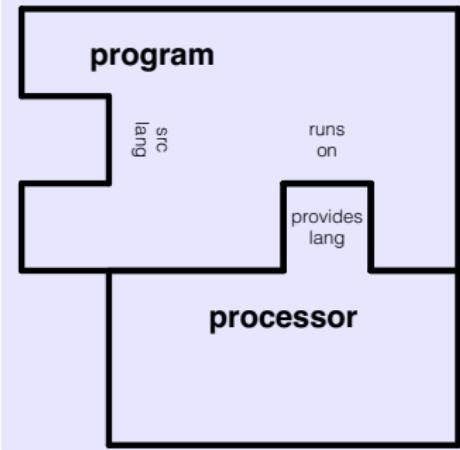


Composing & combining processors, interpreters, compilers, & programs

A compiled program running

- ▶ The program must have been compiled into the same machine-language interpreted by the processor
- ▶ The two blocks join naturally
- ▶ We are not saying anything about the language the program was written in (though we could!)

What it looks like

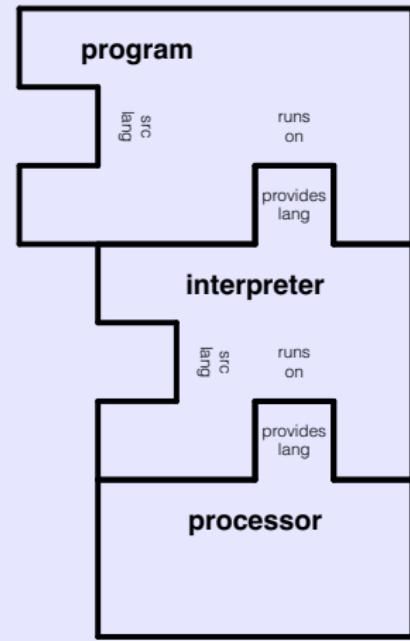


Composing & combining processors, interpreters, compilers, & programs

An interpreter running a compiled program (I)

- ▶ Interpreters are similar to processors
 - ▶ They execute/evaluate programs in a given language
- ▶ Interpreters are programs too!
 - ▶ Written in some source language
 - ▶ Compiled into some target language
 - ▶ They run on an interpreter:
 - ▶ a processor (hardware)
 - ▶ a program (another interpreter)

What it looks like

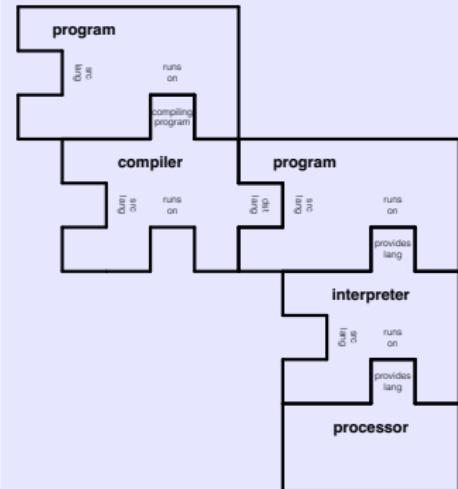


Composing & combining processors, interpreters, compilers, & programs

An interpreter running a compiled program (II)

- ▶ Interpreters can execute/evaluate programs that were compiled from another language. Note that the compiler
 - ▶ takes the program (on top) as source, and
 - ▶ outputs the program (on the right) as target
- ▶ It is the target program that is executed

What it looks like

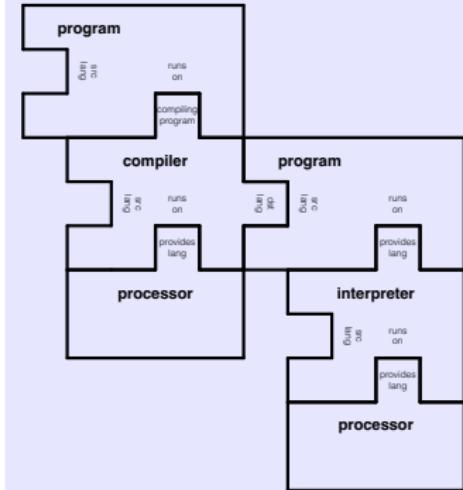


Composing & combining processors, interpreters, compilers, & programs

An interpreter running a compiled program (III)

- ▶ We may add additional details
 - ▶ the processor/interpreter on which the compiler executes.
- ▶ We are still missing details
 - ▶ the compiler that compiled the interpreter
 - ▶ the compiler that compiled the compiler
 - ▶ ...this can go on!

What it looks like



Composing & combining processors, interpreters, compilers, & programs

Cross-compilers

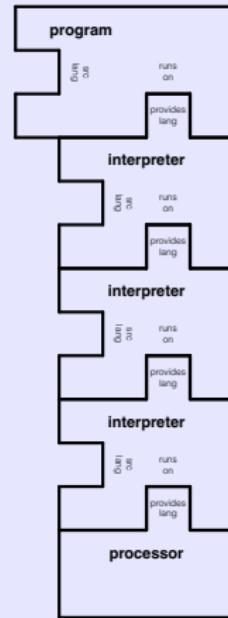
- ▶ The processor on which the compiler runs is different from the one on which the [compiled] program runs
- ▶ It **crosses** the boundaries of architectures.
- ▶ Java compiler `javac` is an example of a cross-compiler:
 - ▶ It runs on [e.g.,] x86
 - ▶ It generates Java-byte-code that runs on the JVM
 - ▶ The JVM is an interpreter (`java`) running on [e.g.,] x86

Composing & combining processors, interpreters, compilers, & programs

Towers of interpreters

- ▶ Interpreters may be stacked up in a **tower**
- ▶ Towers of interpreters consume resources that are **exponential** to the **height** of the tower
 - ▶ Unless there is a marked slowdown, this is not really a tower!
- ▶ Virtual machines (VMs) can be stacked up as towers of interpreters
 - ▶ IBM mainframe architecture actually does this!

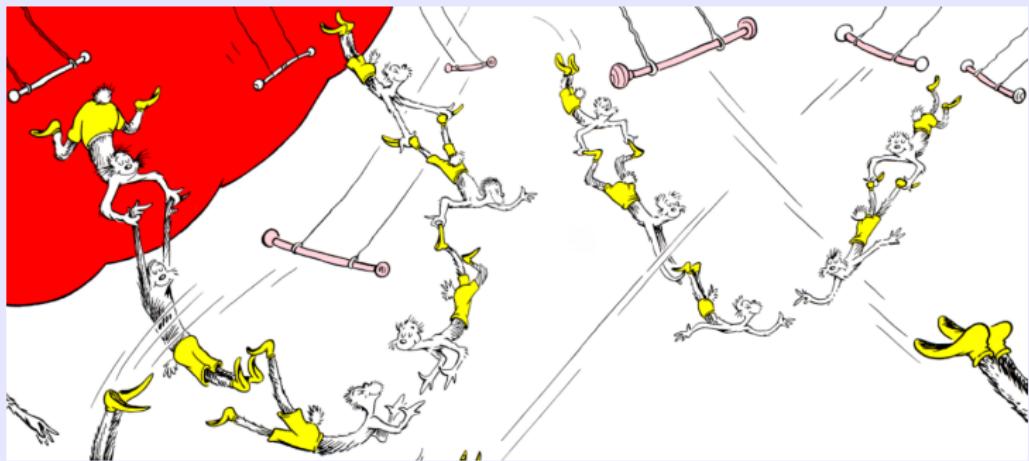
What it looks like



Composing & combining processors, interpreters, compilers, & programs

How are compilers and interpreters made?

- ▶ Using previously-written compilers and interpreters, of course!
- ▶ This process is known as **bootstrapping**, and it is a specific form of composing & combining compilers and interpreters...



Bootstrapping (I)

compiler	written in	compiled by	runs on	compiles	outputs
c ₁	pascal	ibm pascalvs	ibm 370	x86 asm	x86 exe
c ₂	x86 asm	c ₁	x86	x86 asm	x86 exe
c ₃	x86 asm	c ₂	x86	x86 asm	x86 exe

- ▶ c₁ is an assembler acting as a cross-compiler
- ▶ c₂ already runs on our PC, but it was created on an IBM mainframe
 - ▶ All the effort of writing an assembler (in Pascal) has to be re-done from scratch in x86 assembly language if we are to detach from the mainframe!
 - ▶ Any updates, upgrades, bug fixes, changes, require that we re-compile c₂ on the mainframe!
- ▶ c₃ is essentially c₂ compiling itself
 - ▶ With c₃, we are free from our old environment: Pascal on IBM mainframe!

Bootstrapping (II)

compiler	written in	compiled by	runs on	compiles	outputs
c ₃	x86 asm	c ₂	x86	x86 asm	x86 exe
c ₄	x86 asm	c ₃	x86	C (v. 0.1)	x86 exe
c ₅	C (v. 0.1)	c ₄	x86	C (v. 0.2)	x86 exe
c ₆	C (v. 0.2)	c ₅	x86	C (v. 0.2)	x86 exe

- ▶ With c₄ we're diverging:
 - ▶ c₄ is a C compiler
 - ▶ We don't yet support many features
- ▶ c₅ is a C compiler written in C! Notice that
 - ▶ it is written in an older version of C (v. 0.1)
 - ▶ it supports a newer version of C (v. 0.2)
- ▶ In writing c₆, we finally get to use all the language features our compiler supports!

Why study compiler construction

- ▶ Almost all of you shall **use** compilers
- ▶ Most of you shall never work **on** another compiler after this course
- 👉 So why study an entire course on writing compilers??

Why study compiler construction (*cont*)

There are many benefits to studying compilers

- ▶ Better understanding of programming languages
- ▶ Reduce the learning-curve for learning a new programming language
- ▶ Better understanding of what compilers can & cannot do
- ▶ Demystify the compilation process
- 👉 Compilers are examples of **code that writes code**

Code that writes code

- ▶ Like having a team of programmers working for you
 - ▶ Save time; Write code quickly
 - ▶ Gain consistency
- ▶ Spread your bugs **everywhere** 
 - ▶ A bug in the code-generator will spread bugs to many places
 - ▶ This actually makes the bugs **easier** to find!
 - ▶ Once debugged, the code is generated again, and the same bugs **never** re-appear

Why study compiler construction (*cont*)

Bottom line

- ▶ Knowledge of how compilers work will make you a more effective programmer
- ▶ Learning to write **code that writes code** is a force-multiplying technique you can use **anywhere** (think DSLs!)

Further Reading

-  Modern compiler design (2nd edition), Page 1 (Section 1: Introduction)
-  Self hosting
-  Bootstrapping
-  Interpreters

Chapter 1

Goals

- ✓ Establishing common language & vocabulary
- ✓ Understanding the “big picture”

Agenda

- ✓ Some background in **programming languages**
 - ▶ Abstraction
 - ▶ Dynamic vs Static
 - ▶ Functional vs Imperative languages
- ✓ Introduction to **compiler construction**
- 👉 Introduction to the **ocaml** programming language

Languages used in this course

In this course, we shall be working with 3 languages:

- ▶ The language in which **to write the compiler**: ocaml
- ▶ The language we shall **be compiling**: Scheme
- ▶ The language we shall **be compiling to**: x86/64 assembly language

Introduction to ocaml (1)

- ▶ ML is a family of statically-typed, quasi-functional programming languages
- ▶ The main members of ML are
 - ▶ SML (Standard ML)
 - ▶ ocaml
 - ▶ In Microsoftese, ocaml is called F#...

What kind of language is ocaml

Ocaml —

- ▶ is used all over the world
- ▶ is used in commercial and open source projects
- ▶ is powerful, efficient, convenient, modern, elegant, and has a rich toolset
- ▶ supports both functional and object-oriented programming
 - ▶ The ocaml object system is very powerful!
- ▶ makes it very difficult to have run-time errors!

The advantages of learning ocaml

- ▶ Very rich language
- ▶ Great support for abstractions of various kinds
- ▶ Great library support: dbms, networking, web programming, system programming, etc
- ▶ Compiles efficiently, either to bytecode or native

Why we are using ocaml in the course

- ▶ Pattern-matching, modules, object-orientation, and types make programming sophisticated, abstract, clean, easy, elegant, re-usable, safe
- ▶ Easy to enforce an API

Getting, installing & using ocaml

🔗 The OCaml Homepage, or



Getting, installing & using ocaml

- ▶ I run ocaml under GNU Emacs
(<https://www.gnu.org/software/emacs/>)



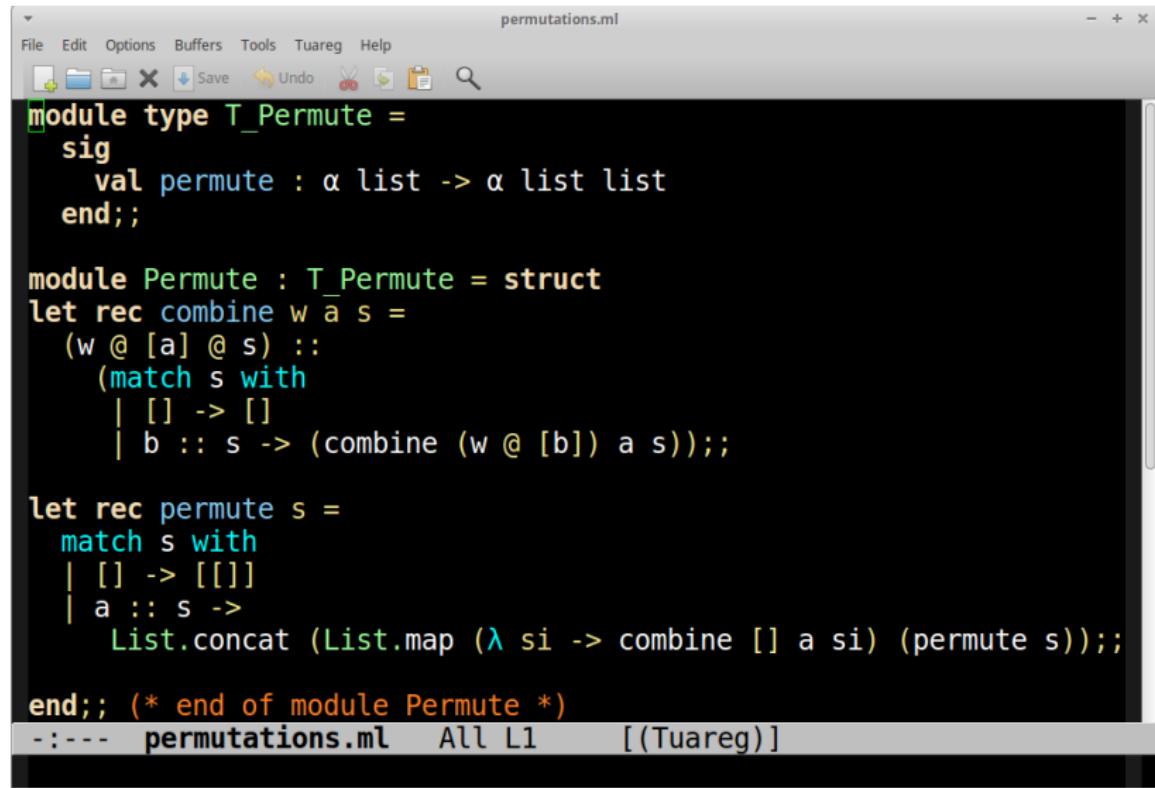
which is the best text editor. Period.

- ▶ Create the file `.ocamlinit` in your home directory, and place in it the line:

```
#use "topfind";;
```

This will load some basic tools you will want to use.

Getting, installing & using ocaml



The screenshot shows a window titled "permutations.ml" containing OCaml code. The code defines a module type T_Permute with a signature (sig) containing a single value permute : α list -> α list list. It then defines a module Permute that implements this type. The implementation uses recursive functions combine and permute. The combine function takes a list w and a list a, and adds a to the front of each element in w. The permute function takes a list s and returns all possible permutations by mapping combine over pairs of lists (a, s) where a is a prefix of s.

```
permutations.ml
File Edit Options Buffers Tools Tuareg Help
Save Undo ⌘S ⌘U ⌘O ⌘P ⌘F ⌘L ⌘S ⌘F
module type T_Permute =
  sig
    val permute : α list -> α list list
  end;;

module Permute : T_Permute = struct
let rec combine w a s =
  (w @ [a] @ s) :::
  (match s with
   | [] -> []
   | b :: s -> (combine (w @ [b]) a s));;

let rec permute s =
  match s with
  | [] -> []
  | a :: s ->
    List.concat (List.map (λ si -> combine [] a si) (permute s));;

end;; (* end of module Permute *)
- :---- permutations.ml  All L1      [(Tuareg)]
```

Getting, installing & using ocaml

- ▶ You are free to use ocaml under any editor/environment you like
- ▶ For example, for ocaml under Eclipse, try OcalDE at <http://www.algo-prog.info/ocaide/>, or



Further Reading

-  OCaml from the Very Beginning, by *John Whitington*
-  More OCaml: Algorithms, Methods & Diversions, by *John Whitington*
-  Real World OCaml: Functional programming for the masses, by *Yaron Minsky & Anil Madhavapeddy*
-  Practical OCaml, by *Joshua B. Smith*
-  The online manual at
<http://caml.inria.fr/pub/docs/manual-ocaml/>, or



Expressions & types

- ▶ Ocaml is a functional programming language
- ▶ Ocaml is *interactive*
- ▶ You enter **expressions** at the **prompt**, and get their **values** and their **types**
- ▶ Expressions are ended with `;;`

Expressions & types

An interaction at the ocaml prompt:

```
# 3;;
- : int = 3
# "asdf";;
- : string = "asdf"
# 'm';;
- : char = 'm'
# 3.1415;;
- : float = 3.1415
# [1; 2; 3; 5; 8; 13];;
- : int list = [1; 2; 3; 5; 8; 13]
# [[1]; [2; 3]; [4; 5; 6]];;
- : int list list = [[1]; [2; 3]; [4; 5; 6]]
```

Modules

- We shall learn about modules later on, as part of the module system
- In the meantime, modules are ways of aggregating functions & variables, while controlling their visibility
- Functionality in ocaml is managed via loading and using modules.

What's available??

Directives

- ▶ Commands that start with #
- ▶ Non-programmable
- ▶ Tell you about the run-time system
- ▶ Change the run-time system

Some useful directives

- ▶ `#list;;` to list available modules
- ▶ `#cd <string>;` to change to a directory
- ▶ `#require <string>;` to specify that a module is required
- ▶ `#show_module <module>;` to see the *signature* of the module
- ▶ `#trace <function>;` to trace a function
- ▶ `#untrace <function>;` to untrace a function

What's available??

What directives are available?

```
Hashtbl.iter
```

```
(fun k _v -> print_endline k)  
Toploop.directive_table;;
```

This is nasty! We can define a function to do that:

```
let directives () =  
  Hashtbl.iter  
    (fun k _v -> print_endline k)  
  Toploop.directive_table;;
```

and now we can just run `directives();` to see the list of directives.

What's available?

Pervasives

- ▶ The module Pervasives contains all the “builtin” procedures in ocaml.
- ▶ Try executing `#show_module Pervasives;;` and see what you get!

```
module Pervasives :  
  sig  
    external raise : exn -> 'a = "%raise"  
    external raise_notrace :  
      exn -> 'a = "%raise_notrace"  
    val invalid_arg : string -> 'a  
    val failwith : string -> 'a  
    exception Exit  
    ...
```

Integers in ocaml

```
# 1 + 2;;
- : int = 3
# 3 * 4;;
- : int = 12
# 8 / 3;;
- : int = 2
# 8 mod 3;;
- : int = 2
```

Read your error messages!

```
# 1.2 + 3.4;;
```

Characters 0-3:

```
1.2 + 3.4;;
```

^^^

Error: This expression has type float but an expression
was expected of type int

```
# cos(3);;
```

Characters 3-6:

```
cos(3);;
```

^^^

Error: This expression has type int but an expression
was expected of type float

Floating-point numbers in ocaml

Operators take a . after them to denote floating-point ops:

```
# 3.4 +. 4.5;;
- : float = 7.9
# 3.2 *. 5.1;;
- : float = 16.32
# cos(2.0);;
- : float = -0.416146836547142407
```

Booleans in ocaml

```
# true;;
- : bool = true
# false;;
- : bool = false
# true && false;;
- : bool = false
# false || false;;
- : bool = false
# 3 = 3;;
- : bool = true
# 3 = 4;;
- : bool = false
# 3 != 4;;
- : bool = true
```

Read your error messages!

In ocaml, unlike in Scheme, the `then`-clause and `else`-clause of if-expressions must be of the same type!

```
# if 3 = 3 then 123 else 456;;
- : int = 123
# if 3 = 4 then "moshe" else "yosi";;
- : string = "yosi"
# if 4 = 4 then 123 else "moshe";;
Characters 23-30:
  if 4 = 4 then 123 else "moshe";;
                           ^~~~~~
```

Error: This expression has type string but an
expression
was expected of type int

Bitwise Boolean functions over the integers

```
# 5 land 3;;
- : int = 1
# 8 lor 3;;
- : int = 11
# 5 lxor 3;;
- : int = 6
```

Characters in ocaml

```
# '*';;
- : char = '*'
# '\t';;
- : char = '\t'
# '\n';;
- : char = '\n'
# '\\';;
- : char = '\\'
# '\"';;
- : char = '\"'
# '\065';;
- : char = 'A'
```

Strings in ocaml

```
# "moshe!";;
- : string = "moshe!"
# "moshe\n";;
- : string = "moshe\n"
# "hello" ^ " " ^ "world";;
- : string = "hello world"
# "moshe".[3];;
- : char = 'h'
```

#show_module String;; will show you what string functions are available

Conversion & coercion of types

```
# char_of_int 97;;
- : char = 'a'
# int_of_char 'A';;
- : int = 65
```

- ▶ Chars in ocaml are encoded as single bytes in ASCII, not as Unicode!

```
# char_of_int 1488;;
Exception: Invalid_argument "char_of_int".
```

- 👉 There is Unicode support in ocaml (later!)
 - ▶ The **only** characters supported in F# are Unicode...

Conversion & coercion of types

```
# int_of_string "1234";;
- : int = 1234
# int_of_string "12+34";;
Exception: Failure "int_of_string".
# string_of_int 496351;;
- : string = "496351"
# float_of_string "123.456";;
- : float = 123.456
# string_of_float 3.1415927;;
- : string = "3.1415927"
```

Tuples in ocaml

Ocaml supports ordered n -tuples. If $e_1 : \tau_1, \dots e_n : \tau_n$, then
 $\langle e_1, \dots, e_n \rangle : (\tau_1 \times \dots \times \tau_n)$.

```
# (3, 4, 5);;
- : int * int * int = (3, 4, 5)
# (3, "blind", "mice");;
- : int * string * string = (3, "blind", "mice")
```

The ordered 0-tuple is possible too, and its type is unit:

```
# ();;
- : unit = ()
```

Lists in ocaml

For a type α , A list of type α list is either **empty**, or it contains something of type α , followed by an α list.

Ocaml supports lists as a builtin data type. Lists are lists of *some type*:

- ▶ lists of integers
- ▶ lists of strings
- ▶ lists of user-defined data-types

etc.

```
# [2; 3; 5; 8; 13];;
- : int list = [2; 3; 5; 8; 13]
# [true; false; false; false; true];;
- : bool list = [true; false; false; false; true]
```

Lists in ocaml

Elements in a list must all belong to the same type:

```
# [true; 234; "moshe!"];;
```

Characters 7-10:

```
[true; 234; "moshe!"];;  
~~~~~
```

Error: This expression has type int but an
expression
 was expected of type bool

This is different from lists in *dynamic languages* (Scheme, Racket, Prolog, Python, etc).

Lists in ocaml

The empty list has an interesting type:

```
# [];;
- : 'a list = []
```

In ocaml,

- ▶ '`a`' is called *alpha* and is often written using α
- ▶ '`b`' is called *beta* and is often written using β
- ▶ '`c`' is called *gamma* and is often written using γ
- ▶ ... etc.

The expressions '`a`', '`b`', '`c`', etc., are known as **type variables**

Lists in ocaml

- ▶ With non-empty lists, ocaml can figure out the type of the list based on the type of the elements in the list.
- ▶ With empty lists, ocaml is unable to figure out the type of the list. This is why you see the type variable unresolved in the type of [] .
- ▶ You may specify the type of α :

```
# ([] : int list);;
- : int list = []
# ([] : string list);;
- : string list = []
# ([] : int list list list);;
- : int list list list = []
```

Read your error messages!

💡 Specifying the type is **not the same as casting** in C/C++/Java:

```
# (2.345 : float);;  
- : float = 2.345  
# (2 : float);;  
Characters 1-2:  
  (2 : float);;  
    ^
```

Error: This expression has type int but an
expression was
expected of type float

There is no **casting** in ocaml, but there are procedures you can call to
generate corresponding data in another type.

Lists in ocaml

Working with lists:

Adding an element to a list:

```
# 3 :: [4; 5; 6];;
- : int list = [3; 4; 5; 6]
```

Appending elements to a list:

```
# [2; 3] @ [5; 8; 13];;
- : int list = [2; 3; 5; 8; 13]
```

`#show_module List;;` will show you what more is available

Functions in ocaml

Overview

- ▶ Syntax for named functions
- ▶ Syntax for anonymous functions
- ▶ Syntax for functions with pattern-matching
- ▶ Syntax for recursive functions
- ▶ Syntax for mutually recursive functions

Functions in ocaml

To define the function `square`, that takes an integer argument and returns its *square*, we define:

```
# let square n = n * n;;
val square : int -> int = <fun>
```

We can now use `square` as any builtin function:

```
# square;;
- : int -> int = <fun>
# square 234;;
- : int = 54756
# square(34);;
- : int = 1156
# square 0;;
- : int = 0
```

Read your error messages!

- ▶ You don't ordinarily need parenthesis
- ▶ It's not an error to have unneeded parenthesis
- ▶ Sometimes it's really needed!

```
# square (((((123)))));;
```

```
- : int = 15129
```

```
# square -234;;
```

```
Characters 0-6:
```

```
    square -234;;
```

```
~~~~~
```

Error: This expression has type int → int
but an expression was expected of type int

```
# square (-234);;
```

```
- : int = 54756
```

Read your error messages!



What is the ocaml type-checker thinking??

```
# square -5;;
Line 1, characters 0-6:
1 | square -5;;
           ^~~~~~
Error: This expression has type int -> int
      but an expression was expected of type int
# square - 5;;
Line 1, characters 0-6:
1 | square - 5;;
           ^~~~~~
Error: This expression has type int -> int
      but an expression was expected of type int
# (square) - (5);;
Line 1, characters 0-8:
1 | (square) - (5);;
           ^~~~~~
Error: This expression has type int -> int
      but an expression was expected of type int
```

Overview

- ✓ Syntax for named functions
- ▶ Syntax for anonymous functions
- ▶ Syntax for functions with pattern-matching
- ▶ Syntax for recursive functions
- ▶ Syntax for mutually recursive functions

Functions in ocaml

```
# fun n -> n * n;;
- : int -> int = <fun>
# ((fun n -> n * n) 123);;
- : int = 15129
# List.map;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
# List.map (fun n -> n * n) [1; 2; 3; 4; 5];;
- : int list = [1; 4; 9; 16; 25]
```

Functions in ocaml

Overview

- ✓ Syntax for named functions
- ✓ Syntax for anonymous functions
- Syntax for functions with pattern-matching
- Syntax for recursive functions
- Syntax for mutually recursive functions

Functions in ocaml

```
# function 0 -> "zero"
| 1 -> "one"
| 2 -> "two"
| n -> (string_of_int n) ^ " is too much!";;
- : int -> string = <fun>
# (function 0 -> "zero"
| 1 -> "one"
| 2 -> "two"
| n -> (string_of_int n) ^ " is too much!") (1);;
- : string = "one"
# (function 0 -> "zero"
| 1 -> "one"
| 2 -> "two"
| n -> (string_of_int n) ^ " is too much!") (3);;
- : string = "3 is too much!"
```

Functions in ocaml

Overview

- ✓ Syntax for named functions
- ✓ Syntax for anonymous functions
- ✓ Syntax for functions with pattern-matching
- Syntax for recursive functions
- Syntax for mutually recursive functions

Functions in ocaml (*continued*)

```
# let rec fact n =
  if (n = 0) then 1
  else n * (fact (n - 1));;
  val fact : int -> int = <fun>
# fact 5;;
- : int = 120
```

Functions in ocaml (*continued*)

Computing logarithms

It is straightforward to compute logarithms in any *base*:

$$\log_a(b) = \begin{cases} 1 + \log_a\left(\frac{b}{a}\right) & a < b \\ 1 & a = b \\ \frac{1}{\log_b(a)} & a > b \end{cases}$$

- ▶ You can carry this algorithm to any number of steps. The logarithm is then given as a **continued fraction**
- ▶ This algorithm can be implemented using a simple recursive function, that iterates over the number of times the **third** condition is met

Functions in ocaml (*continued*)

```
let rec logarithm a b n =
  if n = 0 then 1.0
  else if a < b
  then 1.0 +. logarithm a (b /. a) n
  else 1.0 /. logarithm b a (n - 1);;
```

Testing the function to compute $\log_{10}(5)$:

```
# logarithm 10. 5. 4;;
- : float = 0.692307692307692291
# logarithm 10. 5. 14;;
- : float = 0.698970004331193
# log(5.) /. log(10.);;
- : float = 0.698970004336018746
```

Functions in ocaml (*continued*)

Overview

- ✓ Syntax for named functions
- ✓ Syntax for anonymous functions
- ✓ Syntax for functions with pattern-matching
- ✓ Syntax for recursive functions
- Syntax for mutually recursive functions

Functions in ocaml

```
let epsilon = 1.0e-6

let square x = x *. x;;

let rec our_sin x =
  if Float.abs x < epsilon
  then x
  else 2. *. our_sin(x /. 2.)
        *. our_cos(x /. 2.)

and our_cos x =
  if Float.abs x < epsilon
  then sqrt(1. -. square(our_sin x))
  else square(our_cos(x /. 2.))
        -. square(our_sin(x /. 2.));;
```

Functions in ocaml

```
# our_cos 0.7;;
- : float = 0.764842187303379606
# cos 0.7;;
- : float = 0.764842187284488495
# our_cos (- 0.7);;
- : float = 0.764842187303379606
# Float.cos (- 0.7);;
- : float = 0.764842187284488495
# our_sin 0.123;;
- : float = 0.122690090024220544
# our_sin (- 0.123);;
- : float = -0.122690090024220544
# sin 0.123;;
- : float = 0.122690090024315329
```

Functions in ocaml

Overview

- ✓ Syntax for named functions
- ✓ Syntax for anonymous functions
- ✓ Syntax for functions with pattern-matching
- ✓ Syntax for recursive functions
- ✓ Syntax for mutually recursive functions

Currying arguments

Functions in ocaml are Curried...

Not this kind of Curry



Currying arguments

So what is Currying

- ▶ A function is a subset of the Cartesian product of a domain-set times a range-set: $f \subseteq \mathfrak{D} \times \mathfrak{R}$.
- ▶ Suppose the domain is itself a Cartesian product:
 $\mathfrak{D} = \mathfrak{D}_1 \times \cdots \times \mathfrak{D}_n$.
- ▶ Then $f \subseteq ((\mathfrak{D}_1 \times \cdots \times \mathfrak{D}_n) \times \mathfrak{R})$.
- ▶ The structure $((\mathfrak{D}_1 \times \cdots \times \mathfrak{D}_n) \times \mathfrak{R})$ is isomorphic to
 $\mathfrak{D}_1 \times (\mathfrak{D}_2 \times \cdots \times (\mathfrak{D}_n \times \mathfrak{R}) \cdots)$.
- ▶ The structure $\mathfrak{D}_1 \times (\mathfrak{D}_2 \times \cdots \times (\mathfrak{D}_n \times \mathfrak{R}) \cdots)$ is the domain of a function of 1 argument, that returns a function of 1 argument,
... that returns a function of 1 argument that returns something in \mathfrak{R} . This is a **Curried function**.

Currying arguments

The American logician, Haskell B Curry is responsible for the idea of Currying.

Haskell B Curry



Curried functions

So for any function of several variables f , we can define a **Curried function** f_{Curry} that Curries over its arguments, i.e., takes one at a time.

Currying arguments

- ▶ In applications, Ocaml Curries arguments naturally. This means that the **application** `f x y z w t` really means `(((((f x) y) z) w) t`.
- ▶ Parameters to **named functions** Curry naturally. For example:

```
# let f x y z = x + y + z;;
val f : int -> int -> int -> int = <fun>
# f 2;;
- : int -> int -> int = <fun>
# f 2 3;;
- : int -> int = <fun>
# f 2 3 4;;
- : int = 9
```

- ▶ To avoid Currying, you may pass tuples: In C, `f(x, y, z)` is a procedure call with 3 arguments. In ocaml, this same code is a procedure call with a single argument: an ordered triple.

Roadmap

- 👉 Motivation
- ▶ Boolean Connectives
- ▶ Predicates
- ▶ Quantifiers
- ▶ Syllogisms
- ▶ Formalization using FOPL



Goals

- ▶ The goal of this tutorial is to make you proficient in the topic of formalizing statements in **first-order predicate-logic** (FOPL)
- ▶ You've studied some basic FOPL in your freshman course **Logic & Set-Theory**, but the emphasis was mainly on set-theory
- ▶ Traditionally, formalization of sentences from a natural language to FOPL is a subject taught in logic courses in the *Faculty of the Humanities*
- ▶ By the end of this tutorial —
 - ▶ You should be able to read sentences in FOPL and grasp their meaning
 - ▶ You should be able to translate sentences from a natural language into FOPL, capturing & preserving their meaning



Motivation

- ▶ FOPL is the language of the exact sciences (mathematics & computer science)
 - ▶ It is precise
 - ▶ It is concise
 - ▶ It is unambiguous
 - ▶ It is language-neutral
 - ▶ It is easy to check



Motivation (*cont*)

- ▶ FOPL is used in your courses on calculus, discrete mathematics, algorithms, etc
- ▶ FOPL is also used in the **Compiler-Construction Course**
 - ▶ In recent years, FOPL emerged as a great vehicle for testing student-knowledge on various topics
 - ▶ This does not mean that questions that use FOPL are logic-questions
 - ▶ We only use the **language** of FOPL as a way of expressing knowledge about various problem-domains



Motivation (*cont*)

- ▶ In principle, most of this tutorial should be familiar to you, if you remember your freshman-courses
 - ▶ Which is why this is a **self-study** tutorial!
- ▶ You may think of this tutorial as a **refresher**
- ▶ The aim of this tutorial is to ensure that all students have sufficient experience with FOPL so as to enable them to use it effectively during tests

Roadmap

- ✓ Motivation
- 👉 Boolean Connectives
- ▶ Predicates
- ▶ Quantifiers
- ▶ Syllogisms
- ▶ Formalization using FOPL



Boolean Connectives: Negation

- ▶ If α is a proposition, then $\neg\alpha$ is a proposition
- ▶ If α is true, then $\neg\alpha$ is false, and vice versa
- ▶ The truth-table for negation is given by

α	$\neg\alpha$
F	T
T	F

- ▶ The property of double negation states that for any proposition α , we have $\neg\neg\alpha \Leftrightarrow \alpha$



Boolean Connectives: Conjunction

- If α, β are propositions, then $\alpha \wedge \beta$ is a proposition
- For $\alpha \wedge \beta$ to be true, **both** α, β must be true
- The truth-table for conjunction is given by:

α	β	$\alpha \wedge \beta$
F	F	F
F	T	F
T	F	F
T	T	T



Boolean Connectives: Disjunction

- ▶ If α, β are propositions, then $\alpha \vee \beta$ is a proposition
- ▶ For $\alpha \wedge \beta$ to be true, either α, β must be true
- ▶ The truth-table for disjunction is given by:

α	β	$\alpha \vee \beta$
F	F	F
F	T	T
T	F	T
T	T	T

Boolean Connectives (*cont*)

Conjunction and Disjunction are **commutative** and **associative**

- ▶ $\alpha \wedge \beta \Leftrightarrow \beta \wedge \alpha$
- ▶ $\alpha \vee \beta \Leftrightarrow \beta \vee \alpha$
- ▶ $(\alpha \wedge (\beta \wedge \gamma)) \Leftrightarrow ((\alpha \wedge \beta) \wedge \gamma)$
- ▶ $(\alpha \vee (\beta \vee \gamma)) \Leftrightarrow ((\alpha \vee \beta) \vee \gamma)$

Boolean Connectives (*cont*)

Conjunction and Disjunction satisfy two **distributive** laws:

- ▶ $(\alpha \wedge (\beta \vee \gamma)) \Leftrightarrow ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$
- ▶ $(\alpha \vee (\beta \wedge \gamma)) \Leftrightarrow ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$

Boolean Connectives — de Morgan's Laws

- $(\neg(\alpha \vee \beta)) \Leftrightarrow ((\neg\alpha) \wedge (\neg\beta))$
- $(\neg(\alpha \wedge \beta)) \Leftrightarrow ((\neg\alpha) \vee (\neg\beta))$



Boolean Connectives — Material-Implication

- ▶ If α, β are propositions, then $\alpha \rightarrow \beta$ is a proposition
 - ▶ Material-Implication captures the idea of if-then-else:
 - ▶ Read $\alpha \rightarrow \beta$ as
 - ▶ If α then β
 - ▶ If α is true, then β is true
 - ▶ α entails β
 - ▶ From α being true it follows that β is true
 - ▶ In $\alpha \rightarrow \beta$, α is called the antecedent of the implication, and β is called the conclusion of the implication
 - ▶ Material-Implication is sometimes written as $\alpha \supset \beta$
- “We learned this as just ‘implication’; Why do you call it ‘material implication’?
- ▶ We’ll get to that soon!

Boolean Connectives — Material-Implication (*cont*)

- ▶ Material-implication **fails** to hold only when the **antecedent** is true and the **conclusion** is false
- ▶ The truth-table for Material-Implication is given by:

α	β	$\alpha \rightarrow \beta$
F	F	T
F	T	T
T	F	F
T	T	T

Boolean Connectives — Material-Implication (*cont*)

Consider the following example of material-implication:

- ▶ “If dogs are green, then cats are green too”
 - ▶ This is a true statement
 - ▶ Neither the antecedent nor the conclusion hold true
 - ▶ Therefore the implication does hold true!
- ▶ In natural language, when we use conditional statements
 - ▶ We hardly ever use them **vacuously**
 - ▶ When we do, it’s only for comic effect
 - ▶ We normally intend to underscore some deep, often **causal** relationship between the antecedent and the conclusion:
 - ▶ “If you drop the clock, then it surely shall break”
- ▶ None of these are captured by the material-implication!

Boolean Connectives — Material-Implication (*cont*)

Here's another example of material-implication:

- ▶ “If $1 + 2 = 3$ then $n! = \Gamma(n + 1)$ ”
 - ▶ Both the antecedent and the conclusion hold true
 - ▶ Therefore the implication does hold true!
 - ▶ There is no obvious way of getting from the antecedent to the conclusion:
 - ▶ These are two, unrelated, true mathematical propositions
 - ▶ This antecedent cannot serve as **evidence** in establishing the truth of this conclusion

Boolean Connectives — Material-Implication (*cont*)

- ▶ There is something unnatural and superficial about the relationship implied by material-implication
- ▶ Philosophers have concerned themselves with many other kinds of implications, that attempt to capture some deeper relationship, possibly causal, between the antecedent and the conclusion
 - ▶ Modal
 - ▶ Counterfactual
 - ▶ Temporal
 - ▶ Causal
- etc
- ▶ The word “material” in the term “material-implication” is meant to express the superficiality of this relationship

Boolean Connectives — Material-Implication (*cont*)

- ▶ Material-implication has one advantage though: It is the **only** notion of implication that is **truth-functional**
 - ▶ The **truth-functionality** of material-implication means that the truth-value of $\alpha \rightarrow \beta$ is a **function** of the truth-value of α and the truth-value of β
- ▶ Any deeper implicative relation between antecedent and conclusion requires more information about the antecedent and the conclusion than merely their truth values!

Boolean Connectives — Material-Implication (*cont*)

- Material-implication can be written in terms of negation and disjunction, or negation and conjunction:

$$\begin{aligned} (\alpha \rightarrow \beta) &= (\neg\alpha) \vee \beta \\ &= \neg(\alpha \wedge \neg\beta) \end{aligned}$$

- Material implication naturally **associates to the right**:
 $\alpha \rightarrow \beta \rightarrow \gamma$ should be interpreted to mean $\alpha \rightarrow (\beta \rightarrow \gamma)$

Boolean Connectives — Bi-implication (*cont*)

- ▶ If α, β are propositions, then $\alpha \leftrightarrow \beta$ is a proposition
- ▶ Bi-implication captures the idea of if-and-only-if:
 - ▶ Read $\alpha \leftrightarrow \beta$ as
 - ▶ α if-and-only-if (iff) β
 - ▶ If α is true, then β is true, and vice versa
 - ▶ Either α, β are both true, or both false
 - ▶ α is equivalent to β
 - ▶ Bi-implication is also known by the names equivalence (\equiv), and bi-conditional

Boolean Connectives — Bi-implication (*cont*)

- ▶ Bi-implication captures the idea of two propositions having the same truth-value, whether both true or both false
- ▶ Here are two ways to define bi-implication:
 - ▶ $(\alpha \leftrightarrow \beta) \Leftrightarrow ((\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha))$
 - 👉 This is why it's called bi-implication or bi-conditional
 - ▶ $(\alpha \leftrightarrow \beta) \Leftrightarrow ((\alpha \wedge \beta) \vee ((\neg\alpha) \wedge (\neg\beta)))$
 - 👉 "either both true or both false"

Boolean Connectives — Functional Completeness

- ▶ Let $\text{Bool} = \{F, T\}$ be the set of Boolean values
- ▶ The set of all Boolean functions is the set of all functions in $\text{Bool}^n \rightarrow \text{Bool}^m$, for all natural numbers $m, n \in \mathbb{N}$
- ▶ The set of Boolean functions $\{\neg, \wedge\}$ can be used to express any Boolean function
 - ▶ The property of being able to express any function is called **functional completeness**
 - ▶ The set $\{\neg, \wedge\}$ are said to be **functionally complete**
- ▶ Another set of functionally-complete Boolean functions is $\{\neg, \vee\}$

Boolean Connectives — Functional Completeness (*cont*)

- The functions **nand**, **nor** are also functionally-complete:

$$\alpha \text{ \& } \beta = \neg(\alpha \wedge \beta)$$

$$\alpha \text{ \vee\!& } \beta = \neg(\alpha \vee \beta)$$

- **Hint:** Try to define negation, conjunction, disjunction using only **nand** or only **nor**
- Nand is also known as the **Sheffer stroke**, and is written as $\alpha \mid \beta$, after the American logician **Henry Maurice Sheffer**

Roadmap

- ✓ Motivation
- ✓ Boolean Connectives
- 👉 Predicates
- ▶ Quantifiers
- ▶ Syllogisms
- ▶ Formalization using FOPL

Predicates

- ▶ You may think of predicates as functions from some type α to the set of Bool = $\{F, T\}$ of Boolean values
 - ▶ The number of arguments taken by a predicate is its **arity**
 - ▶ We speak of
 - ▶ 1-place, or unary predicates
 - ▶ 2-place, or binary predicates
 - ▶ n -place, or n -ary predicates



Predicates (*cont*)

- ▶ Predicates extend our language to express **properties** and **relations**
- ▶ When α is the type of an object in our domain of discourse, we say that the predicate denotes a **property**
- ▶ When α is a **product-type**, populated by tuples of objects in our domain of discourse, we say that the predicate denotes a **relation** among those objects
- ▶ Predicates are written as Px or $P(x)$, where $x : \alpha$

Predicates (*cont*)

- ▶ Example:
 - ▶ We define the following predicates:
 - ▶ Let Bx denote that x is a boy
 - ▶ Let Gx denote that x is a girl
 - ▶ Let Lxy denote that x loves y
 - ▶ We can now express simple propositions using these predicates:
 - ▶ “Tarzan is a boy”: $B(\text{Tarzan})$
 - ▶ “Jane is a girl”: $G(\text{Jane})$
 - ▶ “Tarzan loves Jane”: $L(\text{Tarzan}, \text{Jane})$
 - ▶ “Jane does not love Tarzan”: $\neg L(\text{Jane}, \text{Tarzan})$
- 👉 When the predicate arguments are not single-letter variables, it's often more convenient to write them in the format $\text{Pred}(\text{arg}_1, \text{arg}_2, \dots)$

Roadmap

Quantifiers

- ▶ Quantifiers extend our language so we can talk about the **quantities** of objects that satisfy some proposition:
 - ▶ The **universal quantifier** \forall is used to assert that some proposition holds for all objects
 - ▶ the **existential quantifier** \exists is used to assert that some proposition holds for [at least] one object

Quantifiers (*cont*)

- ▶ Example:
 - ▶ Continuing our previous predicates:
 - ▶ Let Bx denote that x is a boy
 - ▶ Let Gx denote that x is a girl
 - ▶ Let Lxy denote that x loves y
 - ▶ How would we formalize the following:
 - ▶ “There exist at least two boys”
 - ▶ **Answer:** $\exists x \exists y (Bx \wedge By \wedge x \neq y)$
-  Had we not added the caveat that $x \neq y$, this proposition would have been true for a universe consisting of one boy!

Quantifiers (*cont*)

- ▶ Continuing the example with B , G , L :
 - ▶ How would we formalize the following:
 - ▶ “Not all girls love themselves”
 - ▶ **Answer:** $\neg\forall x(Gx \rightarrow Lxx)$
 - ▶ Later on, we shall see other ways of encoding this proposition
 - ▶ How would we formalize the following:
 - ▶ “All boys like Mary”
 - ▶ **Answer:** $\forall x(Bx \rightarrow L(x, Mary))$

Quantifiers (*cont*)

We can abbreviate sequences of same quantifiers:

- ▶ Rather than writing $\forall x \forall y (\alpha)$, we can write $\forall x, y (\alpha)$
- ▶ Rather than writing $\exists x \exists y (\alpha)$, we can write $\exists x, y (\alpha)$
- ▶ The order of quantifiers can be switched among their own kind:
 - ▶ **Universal**: $\forall x, y (\alpha)$ is equivalent to $\forall y, x (\alpha)$
 - ▶ **Existential**: $\exists x, y (\alpha)$ is equivalent to $\exists y, x (\alpha)$
- ▶ But the order **cannot be swapped** when the quantifiers are different:
 - ▶ $\forall x \exists y (\alpha)$ is not equivalent to $\exists y \forall x (\alpha)$
 - ▶ **Example**: “For every person, there exists a sandwich, such that this person eats that sandwich” is not equivalent to “There exists a sandwich, that is eaten by every person”!

Roadmap

- ✓ Motivation
- ✓ Boolean Connectives
- ✓ Predicates
- ✓ Quantifiers
- 👉 Syllogisms
- ▶ Formalization using FOPL

Syllogisms

- ▶ A **syllogism** is an argument-form; A way of reasoning deductively
- ▶ Many of the syllogisms we present were discovered by the Greek, Stoic Philosopher, **Chrysippus of Soli**, of the 3rd century BC
- ▶ Although this is a refresher tutorial with the aim of concentrating on formalization, we shall mention in passing some of the important syllogisms
- ▶ We present syllogisms in the form:

$$\frac{\text{assumption}_1, \quad \text{assumption}_2, \dots}{\therefore \text{conclusion}}$$

- ▶ The symbol \therefore should be read as “therefore”

Syllogisms — Modus Ponendo Ponens

$$\frac{\alpha, \quad \alpha \rightarrow \beta}{\therefore \beta}$$

- ▶ It's raining; If it's raining, John carries an umbrella \implies John is currently carrying an umbrella



Syllogisms — Modus Tollendo Tollens

$$\frac{\neg\beta, \quad \alpha \rightarrow \beta}{\therefore \neg\alpha}$$

- ▶ John is not currently carrying an umbrella; If it's raining, John carries an umbrella \Rightarrow It is currently not raining

Syllogisms — Modus Tollendo Ponens

- Modus Tollendo Ponens is also known as **Disjunctive Syllogism**

$$\frac{\neg\alpha, \quad \alpha \vee \beta}{\therefore \beta}$$

- The coffee is not black; Coffee is either black, or with milk \implies
The coffee has milk

Syllogisms — Modus Ponendo Tollens

$$\frac{\alpha, \quad \neg(\alpha \wedge \beta)}{\therefore \neg\beta}$$

- ▶ This food contains sugar; A food cannot both contain sugar and be dietetic \implies This food is not dietetic



Syllogisms — Hypothetical Syllogism

$$\frac{\alpha \rightarrow \beta}{\therefore \neg\beta \rightarrow \neg\alpha}$$

- If it's raining, John carries an umbrella \implies If John is not carrying an umbrella, it is not raining

Syllogisms — Double Negation

$$\frac{\neg\neg\alpha}{\therefore \alpha}$$

- ▶ It is not the case that this is not complicated \implies It is complicated 😊

Syllogisms (*cont*)

We state some useful, yet unnamed equivalences in sentential logic and FOPL

 You may verify them using truth-tables, or any proof-procedure you know

$$(\alpha \wedge \beta) \rightarrow \gamma \Leftrightarrow \alpha \rightarrow \beta \rightarrow \gamma$$

$$\neg(\alpha \rightarrow \beta) \Leftrightarrow (\alpha \wedge \neg\beta)$$

$$\exists x(\alpha) \Leftrightarrow \neg\forall x\neg\alpha$$

Roadmap

- ✓ Motivation
- ✓ Boolean Connectives
- ✓ Predicates
- ✓ Quantifiers
- ✓ Syllogisms
- 👉 Formalization using FOPL

Formalization using FOPL

- ▶ We have now gone through the preliminaries, and we are ready to tackle the main topic of this tutorial: **Formalization**
- ▶ **Formalization** is the subject of translating faithfully propositions in the natural language onto the formal language of logic
- ▶ You have seen some examples of formalization in your calculus classes:
- ▶ For example, the **limit of a sequence** $\{a_n\}_{n=0}^{\infty}$ is defined as a number L that a_n approaches arbitrarily close as n grows
- ▶ This description contains many ideas that are not rigorous enough for careful mathematical treatment. For example, what is meant by “approaches” or “arbitrarily”?



Formalization using FOPL (*cont*)

- ▶ What we mean to say is that the distance between a_n and L can be made as small as we like, and that we do this by selecting a sufficiently large value for n ...
- ▶ But this description is still problematic:
 - ▶ What does it mean “as small as we like”?
 - ▶ How large is “sufficiently large”??
- ▶ In encoding the idea of the limit of a sequence, we must clarify and refine the terms we use until we can express the idea of a limit in FOPL

Formalization using FOPL (*cont*)

- ▶ The distance between a_n and L is given by $|a_n - L|$
- ▶ To say that a_n can be made arbitrarily close to L means that $|a_n - L|$ can be made arbitrarily small
- ▶ To say that something can be made arbitrarily small means that it can be made smaller than any number we choose. For example, if we chose the positive, real number ε , we could make $|a_n - L| < \varepsilon$
- ▶ We now need to relate this closeness to n

Formalization using FOPL (*cont*)

- ▶ What we are trying to say is that from a certain value of n , depending on the choice of ε , the value of $|a_n - L|$ can be made less than ε
- ▶ The way of saying this precisely is to say that there exists an integer N that depends on our choice of ε , such that when $n > N$, we have $|a_n - L| < \varepsilon$
 - ▶ “When” means “for all”
 - ▶ Because the specific value of N depends upon the choice of ε , it is often written as a **function** of ε : $N(\varepsilon)$

Formalization using FOPL (*cont*)

Putting together all these elements, we can now formalize the idea of a limit of a sequence using the language of FOPL:

- ▶ The real-valued sequence $\{a_n\}_{n=0}^{\infty}$ has a limit L , if $\forall \varepsilon > 0, \exists N(\varepsilon) \in \mathbb{N}$, such that $\forall n > N(\varepsilon)$, we have $|a_n - L| < \varepsilon$
- 👉 Vast mathematical territory was discovered and explored without such levels of precision, **however**
 - ▶ It was harder to teach, learn, and understand
 - ▶ It was harder to verify proofs
 - ▶ It was harder to detect subtle flaws in arguments
 - ▶ It was harder to see the limitations of various results
 - ▶ It was hard to see the possibilities, let alone advance into certain domains of mathematics & their applications



Formalization using FOPL (*cont*)

- ▶ The language of FOPL, like any other language, takes time and practice to master
- ▶ Our goal for the remainder of this tutorial is to take you through progressively harder and trickier examples of formalization, so that you can gain some practice and facility in the use of FOPL
- ▶ Some of these examples will be taken from mathematics, some from natural language, and some from computer science



Formalization using FOPL — Example: Relations

- ▶ The 2-place predicate P is called **reflexive** if it holds for all elements: $\forall x Pxx$
 - ▶ The 2-place predicate P is called **symmetric** if the order of its arguments is immaterial: $\forall x, y (Pxy \leftrightarrow Pyx)$
 - ▶ The 2-place predicate P is called **antisymmetric** if whenever the order doesn't matter, both arguments are the same:
 $\forall x, y ((Pxy \wedge Pyx) \rightarrow x = y)$
 - ▶ The 2-place predicate P is called **transitive** if it composes:
 $\forall x, y, z ((Pxy \wedge Pyz) \rightarrow Pxz)$
- Notice that we assume an equality symbol ($=$).
 This is a kind of predicate! You can name it $\text{Eq}(x, y)$ if you like

Formalization using FOPL — Example: Sets

- ▶ We assume a membership predicate (\in)
 - ▶ You can always name it $\text{Member}(x, y)$
 - ▶ The predicate \notin can be defined as
$$x \notin y \equiv \text{NotIn}(x, y) \equiv \neg(x \in y)$$
- ▶ The existence of the empty set: $\exists \emptyset \forall x (x \notin \emptyset)$
- ▶ Subset of x, y (\subseteq): $\forall z (z \in x \rightarrow z \in y)$
- ▶ Set-equality of x, y : $\forall z (z \in x \leftrightarrow z \in y)$
- ▶ Intersection ($z = x \cap y$): $\forall u (u \in z \leftrightarrow (u \in x \wedge u \in y))$
- ▶ Union ($z = x \cup y$): $\forall u (u \in z \leftrightarrow (u \in x \vee u \in y))$
- ▶ Power-set ($y = \wp(x)$):
 - ▶ Using the definition of \subseteq : $\forall z (z \in y \rightarrow z \subseteq x)$
 - ▶ Expanding the definition of \subseteq : $\forall z (z \in y \rightarrow \forall u (u \in z \rightarrow u \in x))$

Formalization using FOPL — Example: Counting

You are given the 1-place predicate P :

- ▶ “**Nothing** satisfies P ”
 - ▶ **Answer:** $\neg\exists x Px$, or its equivalent:
 - ▶ **Answer:** $\forall x \neg Px$
- ▶ “There is **something** that satisfies P ”
 - ▶ **Answer:** $\exists x Px$
 - ▶ There may be more than one thing!
- ▶ “There is **exactly one thing** that satisfies P ”
 - ▶ **Answer:** $\exists x(Px \wedge \forall y(Py \rightarrow x = y))$
 - ▶ **Reading:** “Something exists, that satisfies P , and we shall call it x , and anything else that satisfies P is identical to x ”

Formalization using FOPL — Example: Counting (*cont*)

- ▶ “There is **at most one thing** that satisfies P ”
 - ▶ **Approach 1:** “Either nothing satisfies P or one thing only satisfies P ” (use the above)
 - ▶ **Approach 2:** “It is not the case that **at least two things** satisfy P ” (negate the answer below!)
- ▶ “There are **at least two distinct things** that satisfy P ”
 - ▶ **Answer:** $\exists x, y (x \neq y \wedge Px \wedge Py)$
 - ▶ **Reading:** “There exist two things x, y that are **distinct**, that both satisfy P ”

Formalization using FOPL — Example: Counting (*cont*)

- ▶ “There are **exactly two things** that satisfy P ”
 - ▶ **Answer:** $\exists x, y (x \neq y \wedge Px \wedge Py \wedge \forall z (Pz \rightarrow (z = x \vee z = y)))$
 - ▶ **Reading:** “There exist two things x, y that are **distinct**, that both satisfy P , and anything that satisfies P is either identical to x or identical to y ”
 - ▶ “There are **at most two things** that satisfy P ”
 - ▶ **Approach 1:** “Either nothing satisfies P , or exactly one thing satisfies P , or exactly two distinct things satisfy P ”
 - ▶ **Approach 2:** “It is not the case that **at least three things** satisfy P ”
- 👉 You should now be able to express statements about **quantity**

Formalization using FOPL — Example: Natural Language

Jonathan Swift
(1667—1745)



A Saying by *Swift*

“No man is thoroughly miserable unless he be condemned to live in Ireland.”

—*Jonathan Swift*



How might we formalize this quote in FOPL?

☞ The kind/level/depth of our encoding is really up to us!

Formalization using FOPL — Example: Natural Language

A first attempt:

- ▶ Let
 - ▶ Ux denote: x is thoroughly miserable and has not been condemned to live in Ireland
- ▶ The encoding of *Swit's* saying is $\neg\exists x Ux$



This is not very expressive!

Formalization using FOPL — Example: Natural Language

A second attempt:

- ▶ Let
 - ▶ Mx denote: x is thoroughly miserable
 - ▶ Cx denote: x has been condemned to live in Ireland
- ▶ The encoding of *Swift's* saying is $\neg\exists x(Mx \wedge \neg Cx)$, which after some cleanup becomes $\forall x(Mx \rightarrow Cx)$



This conveys a bit more:

- ☞ Notice how some of the logical structure of the original sentence is conveyed using our formalization

Formalization using FOPL — Example: Natural Language

A third attempt:

- ▶ Let
 - ▶ Mxt denote: x is miserable at time t
 - ▶ This lets us formalize “thoroughly miserable” as “miserable at all times”
 - ▶ Cxy denote: x has condemned y to live in Ireland
- ▶ The encoding of *Swift's* saying is $\forall x, t (Mxt \rightarrow \exists y Cyz)$



Formalization using FOPL — Example: Natural Language



We managed to express a lot more structure using FOPL, but this is **our interpretation imposed on the original text**, possibly distorting it:

- ▶ The meaning of “thoroughly miserable” has been reduced to just the temporal, and we lost the sense of “very miserable” or “totally miserable”
- ▶ The meaning of “condemned” has been reduced to “condemned by someone”, rather than “condemned by nature”, “condemned by misfortune”, “condemned by one’s own vices”, etc
- ▶ We may have lost more meaning than we managed to express!



Formalization using FOPL — Example: Pairs in Scheme

- ▶ The language of numbers, pairs, and the empty list in Scheme:
 - ▶ $\text{Nil}(x)$ denotes that x is the empty list ()
 - ▶ $\text{Num}(x)$ denotes that x is a number
 - ▶ $\text{Pair}(x, y, z)$ denotes that x is a **pair** with **car**-field y and **cdr**-field z
 - ▶ $\text{Sum}(x, y, z)$ denotes that $x = y + z$



Formalization using FOPL — Example: Pairs in Scheme

- ▶ Using this language, here are some problems to formalize:
- ▶ $P_1(x)$ denotes that x is a proper-list of **length** 2
- ▶ **Answer:** $P_1(x) \equiv \exists y, z, t, w (\text{Pair}(x, y, z) \wedge \text{Pair}(z, t, w) \wedge \text{Nil}(w))$
- ▶ **Reading:** x is a **pair** consisting of the **car** y , and the **cdr** z ; z is a **pair** consisting of the **car** t and the **cdr** w , and w is the empty list



Formalization using FOPL — Example: Pairs in Scheme

- ▶ $P_2(x, n)$ denotes that x is a list of 3 numbers, the sum of which is n
- ▶ Answer:

$$\begin{aligned} P_2(x, n) \equiv & \exists y, z, w, t, p, q, m (\text{Pair}(x, y, z) \wedge \text{Num}(y) \\ & \quad \text{Pair}(z, w, t) \wedge \text{Num}(w) \\ & \quad \text{Pair}(t, p, q) \wedge \text{Num}(p) \wedge \text{Nil}(q) \wedge \\ & \quad \text{Sum}(m, y, w) \wedge \text{Sum}(n, m, p)) \end{aligned}$$

- ▶ Reading: x consists of 3 nested pairs, the innermost ending with the empty list
 - ▶ The car-fields of the 3 nested pairs are y, w, p
 - ▶ $n = y + w + p$

Formalization using FOPL — Example: Pairs in Scheme

- ▶ $P_3(x)$ denotes that x is a proper list, i.e., a list the rightmost **cdr**-field of which is the empty list ()
- ▶ **Answer:**
 - ▶ $P_3(x) \equiv \text{Nil}(x) \vee \exists y, z (\text{Pair}(x, y, z) \wedge P_3(z))$
- ▶ **Reading:**
 - ▶ x is a proper list if it is either the empty list, or of it a pair the **cdr**-field of which is a proper list
- 👉 Notice that P_3 is defined **recursively**
 - 👉 **Question:** If we added the 2-place predicate $\text{Sub}(x, y)$, that holds when x occurs as a sub-S-expression in y , would we be able to define P_3 **without** recursion?

Formalization using FOPL — Example: Pairs in Scheme

- ▶ $P_4(x, n)$ denotes that x has n parentheses in its **canonical form**
 - ▶ Remember that the canonical form
 - ▶ ...of $(1 \ . \ (2 \ . \ (3 \ . \ ()))))$ is $(1 \ 2 \ 3)$
 - ▶ ...or $(1 \ . \ (2 \ . \ 3))$ is $(1 \ 2 \ . \ 3)$
 - ▶ **Answer:** We introduce the auxiliary predicate P_5 :

$$P_4(x, n) \equiv \exists a, d, m ((\text{Nil}(x) \wedge n = 2) \vee (\text{Pair}(x, a, d) \wedge P_5(x, m) \wedge n = m + 2) \vee (\neg \text{Nil}(x) \wedge \neg \text{Pair}(x, a, d) \wedge n = 0))$$

$$P_5(x, n) \equiv \exists a, d, m, p ((\text{Nil}(x) \wedge n = 0) \vee (\text{Pair}(x, a, d) \wedge P_4(a, m) \wedge P_5(d, p) \wedge n = m + p) \vee (\neg \text{Nil}(x) \wedge \neg \text{Pair}(x, a, d) \wedge n = 0))$$

Formalization using FOPL — Example: Pairs in Scheme

► Reading:

- The number of parentheses in the **empty** list is 2
- The number of parentheses in a **number** is 0
- The number of parentheses in a **pair** with **car**-field y and **cdr**-field z is the **sum** of
 - The number of parentheses in the **car**-field, and
 - The number of parentheses in the **cdr**-field

👉 Notice that P_4 is defined **recursively**

💡 Why don't we need to add 2 to the number of parentheses each time the argument to P_4 is a **pair**?



Conclusion

- ▶ The language of FOPL can be used to formalize many problem domains
- ▶ You should be able to recognize and use the various Boolean connectives
- ▶ You should be familiar with the most commonly-used syllogisms
- ▶ You should be familiar with the language of **predicates** & **quantifiers**, and know how to use them to express **quantity**
- ▶ We have seen examples from set theory, natural language, computer science
- ▶ We are ready to use the language of FOPL in the compilers course! 

References

-  Symbolic Logic, 5th edition, by *Irving M Copi*
-  Introduction to Logic, 14th edition, by *Irving M Copi, Carl Cohen, & Kenneth McMahon*
-  List of valid argument forms

Roadmap

- ✓ Motivation
- ✓ Boolean Connectives
- ✓ Predicates
- ✓ Quantifiers
- ✓ Syllogisms
- ✓ Formalization using FOPL

Roadmap

- 👉 Motivation
- ▶ Multiple-choice tests
- ▶ Appendix-based questions
- ▶ Domain-coverage questions
- ▶ FOPL-based questions
- ▶ Curving

History

- ▶ In 2015, exams in the course started moving away from **open questions**
- ▶ In 2017, exams in the course were streamlined to include **only** multiple-choice questions



Motivation

- ▶ Multiple-choice tests offer many advantages:
 - ▶ Ease of reading students' answers
 - ▶ Ease & speed of grading
 - ▶ Answers are unambiguous
 - ▶ Consistent grading
 - ▶ Ease, consistency, & fairness of remediation in case errors are discovered on an exam-question
 - 💡 We work extra hard, and use many tools to reduce such occurrences, but exams are complex documents, and sometimes 💩 happens!
 - ▶ Ease of analysis of the test results, per question & per item
 - 👉 And much more!



Motivation (*cont*)

- ▶ Since 2017, all exams in this course have adhered to the exact same structure:
 - ① No material is allowed beyond the examination booklet
 - ② Multiple-choice questions with 5 items
 - ③ Exactly one correct item
 - ④ Roughly around 30 q'tions per 2-hour exam, 40 q'tions per 3-hour exam, or around 50 q'tions per 4-hour exam
 - ⑤ Answers are written on a single answer sheet that appears at the end of the exam booklet
 - ⑥ Any additional, supporting information is given in **the appendices** to the exam
 - 👉 Questions that refer to appendices do so by item & page number
 - ⑦ You are provided with notebooks to be used as scrap-paper

Motivation (*cont*)

- ▶ Since 2017, all exams in this course have adhered to the exact same structure:
 - ⑧ You get to keep the ~~exam booklet~~ & scrap-paper
 - ⑨ You are provided with special sheets on which you may submit inquiries during the exam
 - ⑩ Should the course staff determine that an inquiry merits an answer, a reply is formulated, and posted in each examination room.
 - ⑪ Answers, when given, are only given to the entire class.
 - 👉 No individual answers are given!

Motivation (*cont*)

- ▶ Multiple-choice exams in this course have a peculiar structure, with which you may not be familiar
- ▶ The point of this tutorial is to familiarize yourself with the structure of tests in this course, so that —
 - ▶ You are not surprised
 - ▶ You know how to approach the exam optimally
 - ▶ You can maximize your effectiveness

Roadmap

- ✓ Motivation
- 👉 Multiple-choice tests
- ▶ Appendix-based questions
- ▶ Domain-coverage questions
- ▶ FOPL-based questions
- ▶ Curving

Multiple-choice tests

- ▶ A multiple-choice test consists of Multiple-Choice questions
- ▶ A multiple-choice question consists of two parts:
 - ▶ A stem
 - ▶ Items

A Stem

The stem provides the **context** in which an item should be selected:

- ▶ A story
- ▶ A question
- ▶ A description

...etc

Items

Each multiple-choice question comes with an enumerated set of items, one of which should be selected:

- ▶ It answers the question posed in the stem
- ▶ It is the **most likely** given the information in the stem
- ▶ It follows from the details in the stem
- ▶ It is the **most relevant** to the stem

...etc



Items (*cont*)

- ▶ The expected/correct item is known as the **key** or **answer**
- ▶ The incorrect items are known as **distractors**

Items (*cont*)

- ▶ There are different ways to formulate multiple-choice questions
- ▶ In this course, all multiple-choice questions are formulated in the same way:
 - ▶ You are given a stem
 - ▶ You are given 5 items
 - ▶ One key
 - ▶ Four distractors
 - ▶ When the stems of several questions refer to the same information, this common information factored out and placed in an **appendix**, and referenced via page-number and appendix-number

Here is an example of a multiple-choice question:

57. נתון קטע הקוד הבא:

```
movbe rax, qword [M]
cmp rax, qword [M]
je A
jmp B
```

מהו מוגעימ ל-label A ? כאשר M מצביע על –

- (1) תוו ASCII של פלינדרום.
- (2) פלינדרום בבסיס 10, שמיוצג כמספר 8-ספרתי בבסיס 256.
- (3) פלינדרום 8-ספרתי בbasis 16.
- (4) **רַק הַשְׁאֵלָה נִכּוֹן**
- (5) פלינדרום בbasis 10, שמיוצג כמספר 8-ספרתי בbasis 16.

- Item ④ means that none of the other items are true. It is a way of saying "none of the above", when the other items need not be above. This enables us to permute the items in each question, to create different **versions** of the same exam.

Roadmap

- ✓ Motivation
- ✓ Multiple-choice tests
- 👉 Appendix-based questions
- ▶ Domain-coverage questions
- ▶ FOPL-based questions
- ▶ Curving

Appendix-based questions

We mentioned previously that when the stems of several questions refer to the same information, this information is factored-out and placed in an appendix

- 👉 A common example of an appendix-based question is a single open question that has been split-up/rewritten as several multiple-choice questions:
 - ▶ An appendix may contain a computer program, with various “holes”, or indexed parts that have been left out
 - ▶ Each such part is handled in a different multiple-choice question
 - ▶ So rather than writing code to answer an open question, you fill-in the missing parts in a skeletal code, by selecting each part from among the items of a multiple-choice question
- 👉 This gives a consistent & reasonable model for partial credit!

Appendix-based questions (*cont*)

Here is an example of an appendix:

נحوונה שלדמת הקוד של הפורצדרה הprimיטיבית `not`:

```
not:
    push rbp
    mov rbp, rsp

    cmp [בובי], [בובי]
    jne .E_bad_arg_count

    mov rax, [בובי]
    cmp rax, [בובי]
    je .returnTrue
    mov rax, [בובי]
    jmp .exit

.returnTrue:
    mov [בובי], SOB_TRUE
.exit:
    pop rbp
    ret
```

השאלות הקשורות בנספח זהה עוסקות בהשלמה החלקיים החסרים בקוד. נקודות בקוד (*labels*) הקשורות לדיווח על שגיאות והושטטו במקוון, ואין דרשוות לשאלת.

Appendix-based questions (*cont*)

And here is a related question:

738. השאלה מחייבת להגדיר הנקודות בפסקה 2.84 בעמוד 356. השלימו את **כפוי ני**.

- dword [ebp + 0x18] (1)
- qword [rbp + 0x28] (2)
- qword [rbp + 0x18] (3)
- qword [rbp + 0x20] (4)
- dword [ebp + 0x0c] (5)

Roadmap

- ✓ Motivation
- ✓ Multiple-choice tests
- ✓ Appendix-based questions
- 👉 Domain-coverage questions
- ▶ FOPL-based questions
- ▶ Curving



SELF-STUDY Taking tests in this course

Domain-coverage questions

Roadmap

- ✓ Motivation
- ✓ Multiple-choice tests
- ✓ Appendix-based questions
- ✓ Domain-coverage questions
- 👉 FOPL-based questions
- ▶ Curving



SELF-STUDY Taking tests in this course

FOPL-based questions

Roadmap

- ✓ Motivation
- ✓ Multiple-choice tests
- ✓ Appendix-based questions
- ✓ Domain-coverage questions
- ✓ FOPL-based questions
- 👉 Curving



SELF-STUDY Taking tests in this course

Curving

Roadmap

- ✓ Motivation
- ✓ Multiple-choice tests
- ✓ Appendix-based questions
- ✓ Domain-coverage questions
- ✓ FOPL-based questions
- ✓ Curving

Chapter 2

Goals

- ▶ The pipeline of the compiler
- ▶ Introduction to **syntactic analysis**
- ▶ Further steps in OCaml

Agenda

- ▶ The pipeline
 - ▶ Syntactic analysis
 - ▶ Semantic analysis
 - ▶ Code generation
- ▶ The compiler for the course
- ▶ The language of S-expressions
- ▶ More OCaml

Refresher

Last week, we discussed

- ▶ The interpreter as an **evaluation function**
- ▶ The compiler as a **translator & optimizer**
- ▶ We explored the relations between interpretation & compilation

This was a rather high-level view of the area

We now wish to consider compilation as a **large software-project**

Compilation as translation

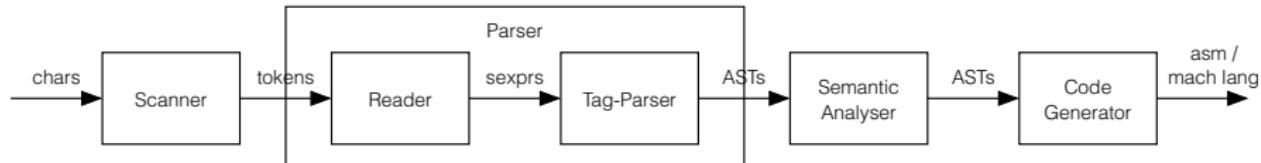
A compiler translates between languages:

- ▶ Understanding the **syntax** of the program
 - ▶ What kinds of statements & expressions there are
 - ▶ What are the various parts of these statements & expressions
 - ▶ Are they syntactically correct
- ▶ Understanding the **meaning** of the program
 - ▶ Do the operations make sense?
 - ▶ What are their types?
 - ▶ Are they used in accordance with their types?
 - ▶ On what data is the program acting?
 - ▶ What is returned?
- ▶ Once we understand the syntax and meaning of a sentence, we can render it in another language

The pipeline of the compiler

Since the 1950's, the standard architecture for compilers has been a pipeline:

- ▶ Syntactic analysis
 - ▶ Scanning
 - ▶ Parsing
 - ▶ Reading
 - ▶ Tag-Parsing
- ▶ Semantic analysis
- ▶ Code generation



The pipeline of the compiler

The stages in the compiler pipeline are distinguished by

- ▶ Function: What they do
- ▶ Dependencies: Which stages depend on which other
- ▶ Complexity: How difficult it is to perform a stage

In programming languages:

- ▶ Understanding **syntax** is relatively straightforward (unlike in natural languages)
- ▶ Understanding **meaning** is much harder than understanding **syntax**
- ▶ Meaning is built **upon syntax** (in natural languages, syntax & meaning can be inter-dependent)
- ▶ Code generation is relatively straightforward (template-based)

The pipeline of the compiler

Optimizations

How optimizations fit into the pipeline of the compiler:

- ▶ We distinguish [at least] two levels of optimizations:
 - ▶ **High-level** optimizations (closer to the source language) would go into the **semantic analysis** phase
 - ▶ **Low-level** optimizations (closer to assembly language) would go into the **code generation** phase

This distinction can be fuzzy. Some make it fuzzier with **intermediate-level optimizations**

An example of a high-level optimization

Suppose the compiler can know that the value of `n` is 0 when reaching the following statement:

```
if (n == 0) {  
    foo();  
}  
else {  
    goo(n);  
}
```

Then an obvious optimization to perform would be to eliminate the if-statement, replacing it with:

```
foo();
```

How has the code improved:

Before

```
if (n == 0) {  
    foo();  
}  
else {  
    goo(n);  
}
```

After

```
foo();
```

What was gained

- ▶ The test during run-time has been eliminated
- ▶ The code is shorter
- ▶ Possibly lead to further, cascading optimizations

An example of a low-level optimization

Before:

```
mov rax, 1  
mov rax, 2
```

After:

```
mov rax, 2
```

How has the code improved:

Before

```
mov rax, 1  
mov rax, 2
```

After

```
mov rax, 2
```

What was gained

- ▶ Saved 1 cycle
- ▶ Made the code smaller
- ▶ If this code appears within a loop, gains shall be multiplied...

The pipeline of the compiler

Basic concepts

- ▶ Concrete syntax
- ▶ Abstract syntax
- ▶ Abstract Syntax-Tree (AST)
- ▶ Token
- ▶ Delimiter
- ▶ Whitespace

Concrete syntax (*continued*)

The concrete syntax of a programming language is a specification of the syntax of programs in that language in terms of a stream of characters:

- ▶ It's one-dimensional
- ▶ Lacking in structure
 - ▶ No nesting
 - ▶ No sub-expressions
- ▶ Difficult to work with
 - ▶ Difficult to access parts
 - ▶ Difficult to determine correctness
- ▶ Contains redundancies (spaces, comments, etc)

```
(define fact ((lambda (n) (if (zero? n) 1  
(* n (fact (- n 1)))))))
```

Think of

- ▶ A text file
- ▶ Characters typed at the prompt

The pipeline of the compiler

Basic concepts

- ✓ Concrete syntax
- ▶ Abstract syntax
- ▶ Abstract Syntax-Tree (AST)
- ▶ Token
- ▶ Delimiter
- ▶ Whitespace

Abstract syntax

The abstract syntax of a programming language is a set of mutually-recursive definitions of abstract data-structures:

- ▶ Multi-dimensional
- ▶ Conveys structure
 - ▶ Nested
 - ▶ Recursive (following the inductive definition of the grammar)
- ▶ Easier to work with than the concrete syntax
 - ▶ Easier to access parts
 - ▶ Easier to verify correctness
 - ▶ Some syntactic correctness issues have already been decided

The pipeline of the compiler

Basic concepts

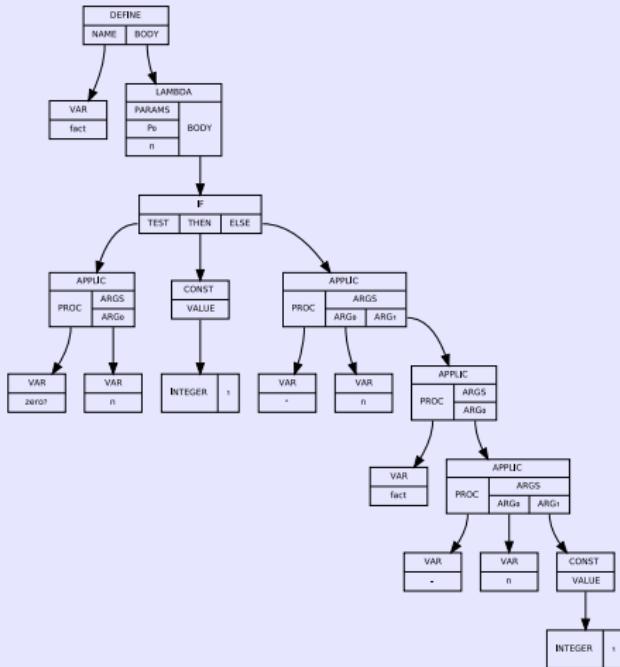
- ✓ Concrete syntax
- ✓ Abstract syntax
- Abstract Syntax-Tree (AST)
- Token
- Delimiter
- Whitespace

Abstract Syntax-Tree (AST)

Notice

- ▶ The AST is a tree
- ▶ A data-structure that represents code
- ▶ Follows the abstract syntax of the language
- ▶ No text, parenthesis, spaces, tabs, newlines
- ▶ The structure is evident
- ▶ Easy to find sub-expressions
- ▶ Easier to analyze, transform, and compile

The AST of fact



Concrete vs Abstract Syntax

- ▶ **Parsing**: going from concrete syntax to abstract syntax
- ▶ **Parser**: the tool that performs parsing, constructing an AST

Concrete Syntax

- ▶ Lacks structure
- ▶ Prone to errors
- ▶ Hard to delimit sub-expressions
- ▶ Inefficient to work with
- ▶ Concrete Syntax can be avoided
 - ▶ Visual languages
 - ▶ Structure/syntax editors

Abstract Syntax

- ▶ Has structure
- ▶ Many kinds of errors are avoided
- ▶ Sub-Expressions are readily accessible
- ▶ Efficient to work with

The pipeline of the compiler

Basic concepts

- ✓ Concrete syntax
- ✓ Abstract syntax
- ✓ Abstract Syntax-Tree (AST)
- Token
- Delimiter
- Whitespace

Tokens

- ▶ The smallest, meaningful, lexical unit in a language
- ▶ Described using **regular expressions**
- ▶ Identified using DFA (a very simple model of computation)
- ▶ Examples
 - ▶ Numbers
 - ▶ [Non-nested] Strings
 - ▶ Names (variables, functions)
 - ▶ Punctuation
- ▶ Cannot handle nesting of any kind:
 - ▶ Parenthesized expressions
 - ▶ Nested comments
 - ▶ etc.

Tokens (*continued*)

- ▶ Scanning: going from **characters** into **tokens**
- ▶ Scanner: the **tool** that performs scanning
- ▶ Scanner-generator: the **tool** that takes definitions for tokens, using regular expressions (and callback functions), and **returns a scanner**
- ▶ Examples of scanner-generators: *lex*, *flex*

The pipeline of the compiler

Basic concepts

- ✓ Concrete syntax
- ✓ Abstract syntax
- ✓ Abstract Syntax-Tree (AST)
- ✓ Token
- Delimiter
- Whitespace

Delimiters

- ▶ Delimiters are characters that separate tokens
- ▶ In most languages spaces, parentheses, commas, semicolons, etc., are all delimiters
- ▶ Some tokens must be separated by delimiters
 - ▶ Two consecutive numbers, two consecutive symbols, etc.
- ▶ Some tokens do not need to be separated by delimiters
 - ▶ Two consecutive strings, an open parenthesis followed by a number, etc.
- ▶ Delimiters are language-dependent

The pipeline of the compiler

Basic concepts

- ✓ Concrete syntax
- ✓ Abstract syntax
- ✓ Abstract Syntax-Tree (AST)
- ✓ Token
- ✓ Delimiter
- Whitespace

Whitespace

- ▶ Whitespace refers to characters that
 - ▶ Have no graphical representation
 - ▶ Occur before or after tokens
 - ▶ Spaces within strings are not whitespaces...
 - ▶ Serve no syntactic purpose other than as **delimiters** and for **indentation**
- ▶ Whitespace is language-dependent

Delimiters in various languages

C & Scheme

Spaces, tab, newlines, carriage returns, form feeds are examples of whitespaces

Java

Literal newline characters may not occur inside a literal string (must use \n). Otherwise, similar to C & Scheme.

Python

Leading tabs are not whitespaces because they have a clear syntactic function: They denote nesting level.

Concrete vs Abstract syntax

Artifacts of the Concrete Syntax

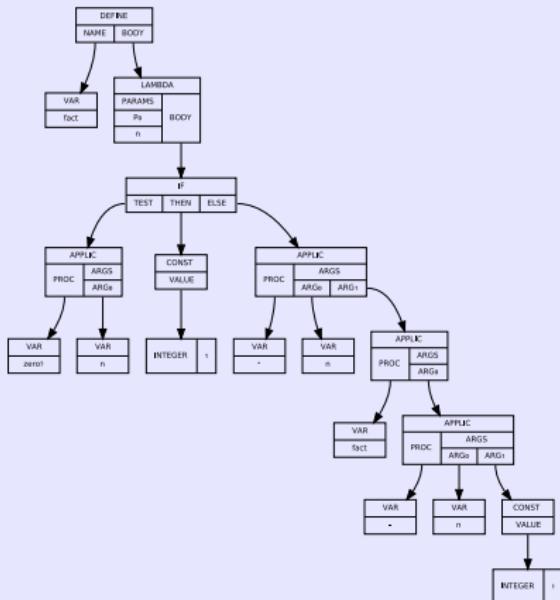
- ▶ Delimiters & whitespaces
- ▶ Parentheses, brackets, braces, and other grouping, nesting, and structring mechanisms (e.g., begin...end)
- 👉 Re-examine the concrete and abstract syntax for the factorial function, and notice what's gone!

Concrete vs Abstract syntax (*continued*)

The concrete syntax

```
(define fact ((lambda (n)
  (if (zero? n) 1
    (* n (fact (- n 1)))))))
```

The abstract syntax



The pipeline of the compiler (*continued*)

Basic concepts

- ✓ Concrete syntax
- ✓ Abstract syntax
- ✓ Abstract Syntax-Tree (AST)
- ✓ Token
- ✓ Delimiter
- ✓ Whitespace

The pipeline of the compiler (*continued*)

Question

Which of the following statements is correct?

-  Every token becomes a vertex in the AST
-  Every AST is a binary tree
-  ASTs can contain cycles
-  Comments are a part of the abstract syntax
-  ASTs contain type tags

More on parsing

To parse computer programs in a given language, we rely on:

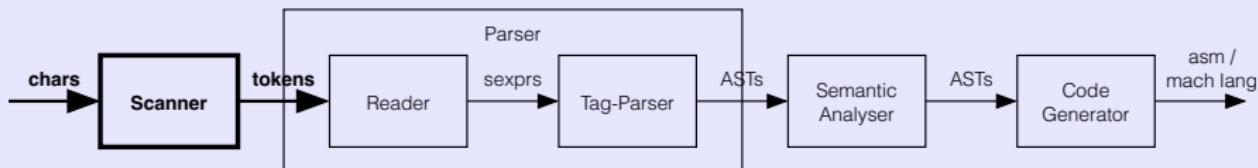
- ▶ **Grammars** with which to express the syntax of the language
 - ▶ There are different kinds of grammars (CFG, CSG, two-level, etc)
 - ▶ There are different languages for expressing the grammar (e.g., BNF, EBNF, etc)
- ▶ **Algorithms** for parsing programs as per kind of grammar
- ▶ **Techniques** (e.g., parsing-combinators, DCGs)

Parser generator: Takes a description of the grammar for a language L, and generates a parser for L. For example, *yacc*, *bison*, *nearly*, etc.

The pipeline of the compiler (*continued*)

Scanning

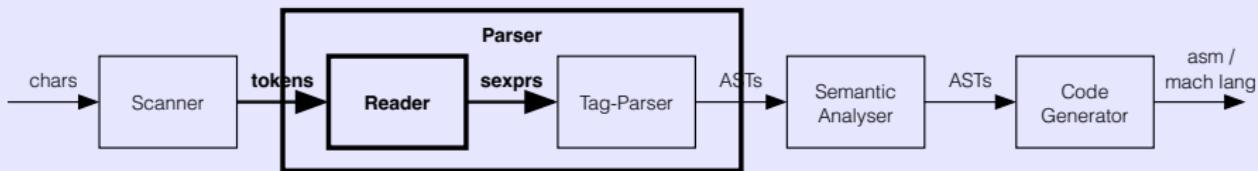
- ▶ Going from characters to tokens
- ▶ Identifying & grouping characters into tokens for words, numbers, strings, etc.
- ▶ Parsing over tokens is more efficient than parsing over characters
 - ☞ As the parser examines various ways to parse the code, the parser can avoid re-identifying and re-building complex tokens such as numbers, strings, etc



The pipeline of the compiler (*continued*)

Reading

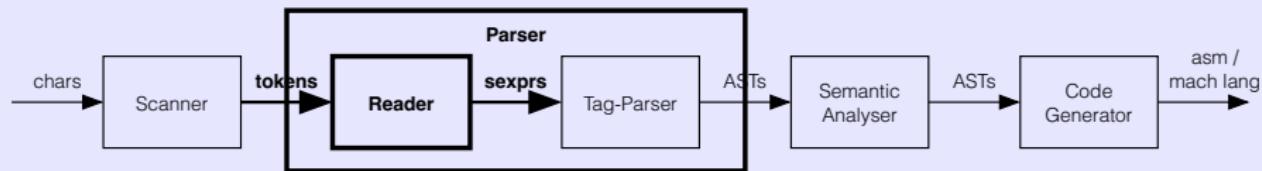
- ▶ In LISP/Prolog, the **parser** is split into two components:
 - ▶ The **reader**, or the parser for the data language
 - ▶ The **tag-parser**, or the parser for the source code
- ▶ In LISP/Scheme/Racket/Clojure/etc, the **abstract syntax** for the data is the **concrete syntax** for the code
- ▶ In Prolog, the **abstract syntax** for the data is the **abstract syntax** for the code
 - ▶ Prolog is the programming language with the most powerful capabilities of **reflection**, i.e., code examining and working with itself.



The pipeline of the compiler (*continued*)

Reading — Summary

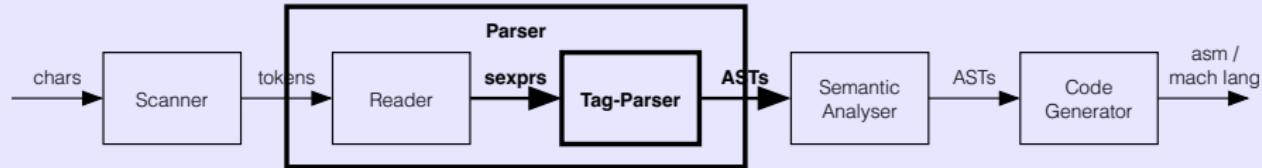
- ▶ In programming languages in which the syntax of code is not a part of the syntax of data, concrete syntax is given as a stream of characters
- ▶ In programming languages in which the syntax of code is part of the syntax of data, things are a bit more complex:
 - ▶ The concrete syntax of data is a stream of characters
 - ▶ The concrete language of code is the abstract syntax of the data
- ▶ In Scheme, the language of data is called S-expressions (sexprs, more on this, later)



The pipeline of the compiler (*continued*)

Tag-Parsing

- ▶ The tag-parser takes sexprs and returns [ASTs for] exprs
- ▶ Languages other than from the LISP & Prolog families do not split parsing into a **reader** & **tag-parser**
 - ▶ In such languages, parsing goes directly from tokens to [ASTs for] expressions
- 👉 Every valid program “used to be” [i.e., before tag-parsing] a valid sexpr
- 👉 Not every valid sexpr is a valid program!



The pipeline of the compiler (*continued*)

Question

A parser should:

- 👎 Perform optimizations
- 👎 Evaluate expressions
- 👎 Raise type-mismatch errors
- 👎 Find potential runtime errors (null-pointer dereferences, array-index errors, etc.)
- 👍 Validate the **structure** of input programs against a syntactic specification

The pipeline of the compiler (*continued*)

Question

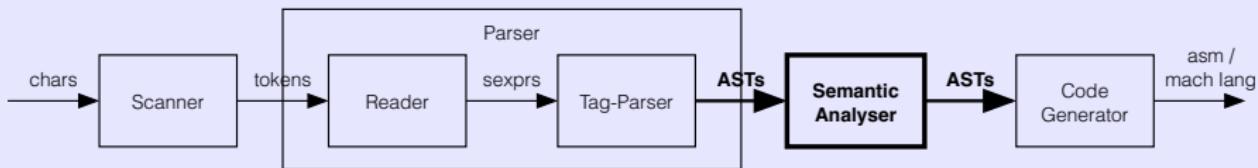
Using an AST, it is **impossible** to:

- 👎 Perform code reformatting/beautification/style-checking
- 👎 Perform optimizations
- 👎 Output a new program which is semantically equivalent to the input program (code generation)
- 👎 Refactor the input program
- 👍 Generate a list of all the comments in the code

The pipeline of the compiler (*continued*)

Semantic Analysis

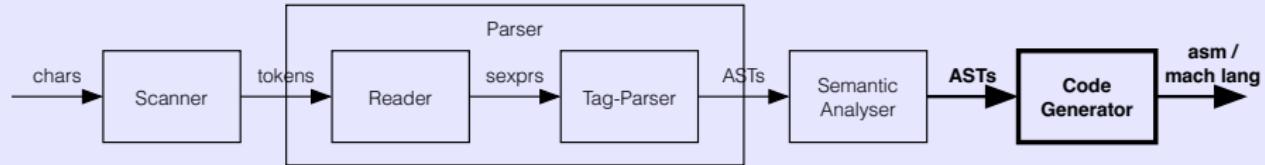
- ▶ Annotate the ASTs
- ▶ Compute addresses
- ▶ Annotate tail-calls
- ▶ Type-check code
- ▶ Perform optimizations



The pipeline of the compiler (*continued*)

Code Generation

- ▶ Generate a stream of instructions in
 - ▶ assembly language
 - ▶ machine language
 - ▶ Build executable
 - ▶ some other target language...
- ▶ Perform low-level optimizations



The compiler for the course

Our compiler project

- ▶ Written in OCaml
- ▶ Supports a subset of Scheme + extensions
- ▶ Supports two, simple optimizations
- ▶ Compiles to x86/64
- ▶ Runs on linux

What our project shall lack

- ▶ Support for the full language of Scheme
- ▶ Support for garbage collection
- ▶ The ability to compile itself

S-expressions

- ▶ We're going to learn about syntax by studying the syntax of Scheme
 - ▶ After all, we're writing a Scheme compiler...
 - ▶ It's relatively simple, compared to the syntax of C, Java, Python, and many other languages
 - ▶ It comes with some interesting twists
- ▶ Scheme comes with two languages:
 - ▶ A language for code
 - ▶ A language for data

and there's a tricky relationship between the two.
- ▶ The key to understanding the syntax of Scheme, is to think about **data**

The Language of Data

What is a language of data? — A language in which to

- ▶ Describe arbitrarily-complex data
 - ▶ Possibly multi-dimensional, deeply nested
 - ▶ Polymorphic
 - ▶ Possibly circular
- ▶ Access components easily and efficiently

The Language of Data (*continued*)

Today many languages of data are known:

- ▶ S-expressions (the first: 1959)
- ▶ Functors (1972)
- ▶ Datalog (1977)
- ▶ SGML (1986)
- ▶ MS DDE (1987)
- ▶ CORBA (1991)
- ▶ MS COM (1993)
- ▶ MS DCOM (1996)
- ▶ XML (1996)
- ▶ JSON (2001)

The Language of Data (*continued*)

What makes S-expressions and Functors unique?

- ▶ They're the first... 😊
- ▶ They're supported natively, as part of specific programming languages
 - ▶ S-expressions are supported by LISP-based languages, including Scheme & Racket
 - ▶ Functors are supported by Prolog-based languages
- 👉 The language of code is a [strict] subset of the language of data

The Language of Data (*continued*)

Think for a moment about the language of XML:

<something>...</something>, etc

- ▶ It's not supported natively by almost any programming language
- ▶ Most modern languages (Java, Python, etc) support it via libraries
- ▶ Almost no programming language has XML for its concrete syntax:

```
<package name="Foo">
    <class name="Foo">
        <method name="goo">
            ...
        </method>
    </class>
</package>
```

This would be cumbersome, and weird!

The Language of Data (*continued*)

However, if some programming language both

- ▶ Supported XML as its data language
- ▶ Were itself written in XML

Then a parser for XML could also read programs written in that language:

- ▶ Writing interpreters, compilers, and other language-tools would have been much simpler!
- ▶ Reflection (code examining code) would be simple

The Language of Data (*continued*)

This is the case with S-expressions:

- ▶ They are the data language for LISP-based languages, including Scheme
- ▶ LISP-based languages are written using S-expressions
- ▶ Writing interpreters and compilers in LISP-based languages is much simpler than in other languages
- ▶ Computational reflection was **invented** in LISP!
- ▶ This is the real reason behind all these parentheses in Scheme:
 - ▶ A very simple language
 - ▶ Supports core types: *pairs*, *vectors*, *symbols*, *strings*, *numbers*, *booleans*, the *empty list*, etc.
 - ▶ A syntactic compromise that is great for expressing **both** code and data

S-expressions (*continued*)

Back to S-expressions

- ▶ S-expressions were invented along with LISP, in 1959
- ▶ S-expressions stand for **Symbolic Expressions**
- ▶ The term is intended to distinguish itself from **numerical expressions**
 - ▶ Before LISP (and long after it was invented), most computation concerned itself with numbers
 - ▶ Computer languages were great at “crunching numbers”, but working with non-numeric data-types was difficult
 - ▶ String libraries were non-standard and uncommon
 - ▶ Polymorphic data was unheard of
 - ▶ Nested data-structures needed to be implemented from scratch, usually with arrays of characters and/or integers...

S-expressions (*continued*)

Back to S-expressions

Then S-expressions were invented as part of a very dynamic programming language (LISP):

- ▶ Working with data-structures became considerably simpler
 - ▶ Trivially allocated (no pointer-arithmetic)
 - ▶ Polymorphic (lists of lists of numbers and strings and vectors of booleans and...)
 - ▶ Easy to access sub-structures (no pointer-arithmetic)
 - ▶ Easy to modify (in an easy-going, functional style)
 - ▶ Easy to examine (they're just made up of primitive types)
 - ▶ Easy to redefine
 - ▶ Automatically deallocated (**garbage collection**)
- ▶ Treating code as data became considerably simpler

S-expressions (*continued*)

Several **fields** were invented using LISP and its tools:

- ▶ Symbolic Mathematics (*Macsyma*, a precursor to *Wolfram Mathematica*)
- ▶ Artificial Intelligence
- ▶ Computer adventure-game generation-languages (MDL, ZIL)

S-expressions (*continued*)

Definition: S-expressions

The language is made up of

- ▶ The empty list: ()
- ▶ Booleans: #f, #t
- ▶ Characters: #\a, #\Z, #\space, #\return, #\x05d0, etc
- ▶ Strings: "abc", "Hello\nWorld\t\x05d0;hi!", etc
- ▶ Numbers: -23, #x41, 2/3, 2-3i, 2.34, -2.34+3.5i
- ▶ Symbols: abc, lambda, define, fact, list->string
- ▶ Pairs: (a . b), (a b c), (a (2 . #f) "moshe")
- ▶ Vectors: #(), #(a b ((1 . 2) #f) "moshe")

Traditionally, non-pairs are known as atoms.

S-expressions (*continued*)

Proper & improper lists

- ▶ The name LISP comes from LISt Processing.
- ▶ In fact, LISP has no direct support for lists.
- ▶ LISP has ordered pairs
 - ▶ Ordered pairs are created using cons
 - ▶ The first and second projections over ordered pairs are car and cdr. For all x, y:
 - ▶ $(\text{car } (\text{cons } x \ y)) \equiv x$
 - ▶ $(\text{cdr } (\text{cons } x \ y)) \equiv y$
 - ▶ The ordered pair of x and y can be written as $(x \ . \ y)$

S-expressions (*continued*)

The dot-rules

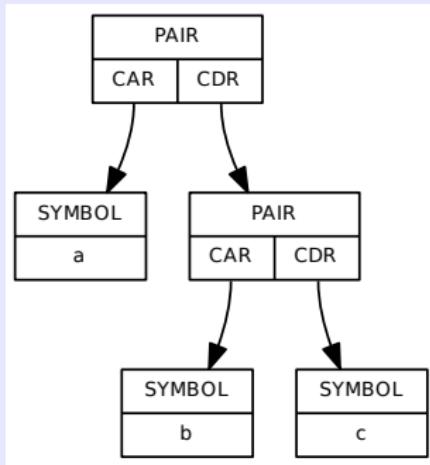
Two rules govern how ordered pairs are **printed**:

- ▶ **Rule 1:** For any S-expression \mathcal{E} , the ordered pair $(\mathcal{E} . \text{ } ())$ is printed as (\mathcal{E}) , which looks like a list of 1 element.
- ▶ **Rule 2:** For any $\mathcal{E}_1, \mathcal{E}_2, \dots$, the ordered pair $(\mathcal{E}_1 . \text{ } (\mathcal{E}_2 \dots))$ is printed as $(\mathcal{E}_1 \text{ } \mathcal{E}_2 \dots)$
- ▶ These rules just effect **how** pairs are printed
- ▶ These rules do not effect the internal representation of the pairs
- ▶ These rules give us a **canonical representation** for pairs

S-expressions (*continued*)

Example

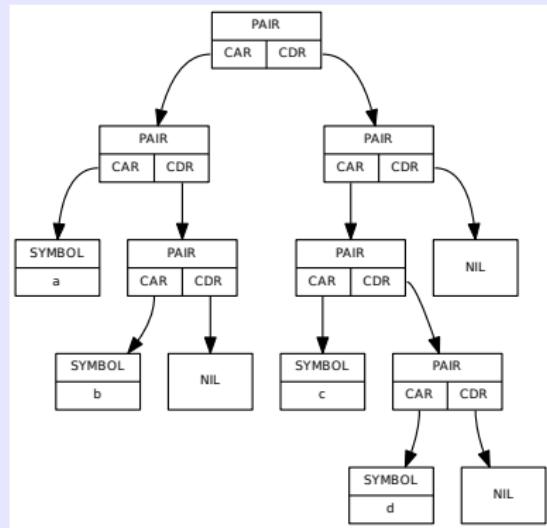
- ▶ The pair $(a \ . \ (b \ . \ c))$ is printed as $(a \ b \ . \ c)$



S-expressions (*continued*)

Example

- ▶ The pair $((a . (b . ())) . ((c . (d . ())))))$ is printed as $((a\ b)\ (c\ d))$



S-expressions (*continued*)

- ▶ Lists in Scheme can come in two forms, **proper lists** and **improper lists**.
- ▶ When we just speak of lists, we usually mean **proper lists**.
- ▶ Most of the list processing functions (length, map, etc) take only **proper lists**:

```
> (length '(a b . c))
```

Exception in length: (a b . c) is not a proper list
Type (debug) to enter the debugger.

S-expressions (*continued*)

Proper lists

- ▶ Proper lists are nested, ordered-pairs the rightmost cdr of which is the empty list (aka `nil`)
- ▶ Testing for pairs is **cheap**, and is done by means of the builtin predicate `pair?`, which tests the Run-Time Type-Information (RTTI) of its argument (which involves checking a **data-field** in the argument)
- ▶ Testing for [proper] lists is **expensive**, since it traverses nested, ordered pairs, until it reaches their rightmost cdr. This is done by means of the builtin predicate `list?`

S-expressions (*continued*)

Proper lists

Here's a definition for list?:

```
(define list?
  (lambda (e)
    (or (null? e)
        (and (pair? e)
              (list? (cdr e)))))))
```

S-expressions (*continued*)

Improper lists

- ▶ Pairs that are not proper lists are **improper lists**.
- ▶ Improper lists end with a rightmost cdr that is **not** the empty list (**nil**)
- ▶ List-processing procedures such as `length`, `map`, etc, fail over improper lists
- ▶ There is no builtin procedure for testing improper lists, but it could be written as follows:

```
(define improper-list?
  (lambda (e)
    (and (pair? e)
         (not (list? (cdr e))))))
```

S-expressions (*continued*)

Self-evaluating forms

Booleans, numbers, characters, strings are **self-evaluating forms**. You can evaluate them directly at the prompt:

```
> 123  
123  
> "abc"  
"abc"  
> #t  
#t  
> #\m  
#\m
```

S-expressions (*continued*)

Other forms

The empty list, pairs, and vectors cannot be evaluated directly at the prompt:

- ▶ Entering an empty list or a vector or an improper list at the prompt generates a run-time error.
- ▶ Entering a symbol at the prompt causes Scheme to attempt to evaluate a **variable** by the same name
- ▶ Entering a proper list, that is not the empty list, at the prompt causes Scheme to attempt to evaluate an **application**:
`> (a b c)`

Exception: variable b is not bound
Type (debug) to enter the debugger.

S-expressions: quote & friends

To evaluate S-expressions that are not self-evaluating, we must use the form `quote`:

- ▶ The special form `quote` can be written in two ways:
 - ▶ '`<expr>`'
 - ▶ `(quote <expr>)`
- Both forms are **equivalent**, but the **reader** will convert the first into the second
- ▶ When you type `abc` at the Scheme prompt, you're evaluating the variable `abc`
- ▶ When you type '`abc`' at the Scheme prompt, you're evaluating the literal **symbol** `abc`
- ▶ The value of the literal symbol `abc` is just itself, which is why when you type '`abc`' at the Scheme prompt, you get back `abc`

S-expressions: quote & friends

- ▶ When you type () at the Scheme prompt, you're evaluating an application with no function and no arguments! This is a syntax-error!
- ▶ When you type '() at the Scheme prompt, you're evaluating a literal empty list
- ▶ The value of the literal empty list is just itself, which is why when you type '() at the Scheme prompt, you get back ()

S-expressions: quote & friends

- ▶ When you type (a b c) at the Scheme prompt, you're evaluating the application of the procedure a to the arguments b and c, which are variables
- ▶ When you type '(a b c) at the Scheme prompt, you're evaluating the literal list (a b c)
- ▶ The value of the literal list (a b c) is just (a b c), which is why when you type '(a b c) at the Scheme prompt, you get back (a b c).
- ▶ Quoting a self-evaluating S-expression is possible, and redundant:

```
> '2
```

```
2
```

```
> (+ '2 '3)
```

```
5
```

S-expressions: quote & friends

So what does quote do?

- 💡 The quote form does **nothing**
 - ▶ It is not a procedure
 - ▶ It doesn't take an argument
 - ▶ It delimits a constant, literal S-expressions
- ▶ The syntactic function of quote in Scheme is the same as the syntactic function of braces { ... } in C in defining literal data:

```
const int A[] = {4, 9, 6, 3, 5, 1};
```

S-expressions: quote & friends

Meet quasiquote

- ▶ Similarly to quote, the form quasiquote can be written in two ways:
 - ▶ `<expr>
 - ▶ (quasiquote <expr>)
- Both forms are **equivalent**, but the **reader** will convert the first into the second
- ▶ quasiquote is also used to define data:
 - ▶ `abc is the same as 'abc
 - ▶ `(a b c) is the same as '(a b c)
- ▶ But quasiquote has two neat tricks!

S-expressions: quote & friends

Meet quasiquote (*cont*)

- ▶ The following two forms may occur within a quasiquote-expression:
 - ▶ The unquote form:
 - ▶ ,<expr>
 - ▶ (unquote <expr>)
 - Both forms are equivalent, but the reader will convert the first into the second
 - ▶ The unquote-splicing form:
 - ▶ ,@<expr>
 - ▶ (unquote-splicing <expr>)
 - Both forms are equivalent, but the reader will convert the first into the second
- ▶ Both unquote & unquote-splicing are used within quasiquote-expressions, to mix-in dynamic and static data

S-expressions: quote & friends

Meet quasiquote (*cont*)

```
> '(a (+ 1 2 3) b)
(a (+ 1 2 3) b)
> '(a ,(+ 1 2 3) b)
(a ,(+ 1 2 3) b)
> `(a (+ 1 2 3) b)
(a (+ 1 2 3) b)
> `(a ,(+ 1 2 3) b)
(a 6 b)
> `(a ,(append '(x y) '(z w)) b)
(a (x y z w) b)
> `(a ,@(append '(x y) '(z w)) b)
(a x y z w b)
```

S-expressions: quote & friends

Meet quasiquote (*cont*)

- ▶ The expression ``(a ,(append '(x y) '(z w)) b)` is equivalent to
`(cons 'a (cons (append '(x y) '(z w)) '(b)))`
- ▶ The expression ``(a ,@(append '(x y) '(z w)) b)` is equivalent to
`(cons 'a (append (append '(x y) '(z w)) '(b)))`
- ▶ The difference between unquote & unquote-splicing is that
 - ▶ unquote mixes-in an expression into a list using `cons`
 - ▶ unquote-splicing mixes-in an expression into a list using `append`

S-expressions: quote & friends

Meet quasiquote (*cont*)

- ▶ Together, quasiquote, unquote, & unquote-splicing are known as the **quasiquote-mechanism** or the **backquote-mechanism**
- ▶ The **quasiquote-mechanism** allows us to create data by **template**, that is, by specifying the **shape** of the data
- ▶ In Scheme, convenient ways to create data translate immediately into convenient ways to create code
 - ▶ Therefore we expect the **quasiquote-mechanism** to have useful applications within programming languages
 - ▶ We can turn code that computes something into code that shows us a computation...

S-expressions: quote & friends

Consider the familiar factorial function:

```
(define fact
  (lambda (n)
    (if (zero? n)
        1
        (* n (fact (- n 1))))))
```

S-expressions: quote & friends

We use the **quasiquote-mechanism** to convert the application `(* n (fact (- n 1)))` into code that **describes** what factorial does:

```
(define fact
  (lambda (n)
    (if (zero? n)
        1
        `(* ,n ,(fact (- n 1))))))
```

Running `(fact 5)` now gives:

```
> (fact 5)
(* 5 (* 4 (* 3 (* 2 (* 1 1)))))
```

As you can see, factorial now returns a **trace** of the computation.

S-expressions: quote & friends

We are now going to use the **quasiquote-mechanism** to get Scheme to teach us about the structure of S-expressions.

Consider the following code:

```
(define foo
  (lambda (e)
    (cond ((pair? e)
            (cons (foo (car e))
                  (foo (cdr e))))
          ((or (null? e) (symbol? e)) e)
          (else e))))
```

What does this program do?

S-expressions: quote & friends

Let's call foo with some arguments:

```
> (foo 'a)
a
> (foo 123)
123
> (foo '())
()
> (foo '(a b c))
(a b c)
```

S-expressions: quote & friends

Looking over the code again

```
(define foo
  (lambda (e)
    (cond ((pair? e)
            (cons (foo (car e))
                  (foo (cdr e))))
          ((or (null? e) (symbol? e)) e)
          (else e))))
```

we notice that:

- ▶ The 2nd and 3rd ribs of the cond overlap [we could have removed the 2nd]
- ▶ All **atoms** are left unchanged
- ▶ All pairs are duplicated, while recursing over the car and cdr of the pair

So foo does nothing, though it does it recursively! 😊

S-expressions: quote & friends

We use the **quasiquote-mechanism** to cause `foo` to generate a **trace**:

```
(define foo
  (lambda (e)
    (cond ((pair? e)
            `(cons ,(foo (car e))
                  ,(foo (cdr e))))
          ((or (null? e) (symbol? e)) `',e)
          (else e))))
```

S-expressions: quote & friends

Running foo now gives us some interesting data:

```
> (foo 'a)
'a
> (foo '(a b c))
(cons 'a (cons 'b (cons 'c '())))
> (foo '(a 1 b 2))
(cons 'a (cons 1 (cons 'b (cons 2 '()))))
> (foo 123)
123
> (foo '((a b) (c d)))
(cons
  (cons 'a (cons 'b '()))
  (cons (cons 'c (cons 'd '())) '()))
```

S-expressions: quote & friends

- ▶ Using the **quasiquote-mechanism**, we got `foo` to **describe** how S-expressions are created using the most basic API
- ▶ We should really add support for proper lists and vectors!
- ▶ In fact, the name `describe` is far more appropriate than `foo`...

Let's rewrite `foo`...

S-expressions: quote & friends

```
(define describe
  (lambda (e)
    (cond ((list? e)
            `'(list ,@(map describe e)))
          ((pair? e)
            `'(cons ,(describe (car e))
                     ,(describe (cdr e))))
          ((vector? e)
            `'(vector
                ,@(map describe
                        (vector->list e))))
          ((or (null? e) (symbol? e)) `',e)
          (else e))))
```

S-expressions: quote & friends

Running `describe` on various S-expressions is very instructive:

```
> (describe '(a b c))
(list 'a 'b 'c)
> (describe '#(a b c))
(vector 'a 'b 'c)
> (describe '(a b . c))
(cons 'a (cons 'b 'c))
> (describe ''a)
(list 'quote 'a)
```

Wait! What's with the last example?!

S-expressions: quote & friends

Recall what we said about quote, quasiquote, unquote, & unquote-splicing:

- ▶ '*<expr>* ≡ (quote *<expr>*)
- ▶ `<expr> ≡ (quasiquote *<expr>*)
- ▶ ,*<expr>* ≡ (unquote *<expr>*)
- ▶ ,@*<expr>* ≡ (unquote-splicing *<expr>*)

Now we get to see this happen...

S-expressions: quote & friends

Now we get to see this happen:

```
> (describe ''<expr>)
(list 'quote '<expr>)
> (describe '`<expr>')
(list 'quasiquote '<expr>)
> (describe ',<expr>)
(list 'unquote '<expr>)
> (describe ',@<expr>)
(list 'unquote-splicing '<expr>)
```

Rule: Every Scheme expression used to be an S-expression when it was little! 

S-expressions: quote & friends

Question

What is (length '!!!!!!moshe) ?

-  17
-  16
-  Generates an error message!
-  1
-  2

S-expressions: quote & friends

Explanation

`(length '.....'moshe)` is the same as `(length '(quote <something>))`, where `<something>` is
`'.....'moshe`, but that really doesn't matter! We are still computing the length of a list of size 2:

- ▶ The first element of the list is the symbol `quote`
- ▶ The second element of the list is `'.....'moshe`

S-expressions: quote & friends (*continued*)

Question

The **structure** of the S-expression ''a in Scheme is:

-  Just the symbol a
-  The proper list (quote . (a . ()))
-  The proper list (quote . (quote . (a . ())))
-  An invalid S-expression
-  The nested proper list (quote . ((quote . (a . ())). ()))

Tag-Parsing (*continued*)

- ▶ In a previous slide, we made the claims that in all descendants of LISP (including Scheme):
 - ☞ Every valid program “used to be” [i.e., before tag-parsing] a valid sexpr
 - ☞ Not every valid sexpr is a valid program!
- ▶ We can now show you some examples

As data (S-expressions)

- ▶ (if if if if) is a list of size 4
- ▶ (if (zero? n) 'zero 'non-zero) is also a list of size 4

As code

- ▶ (if if if if) is not a valid if-expression
- ▶ (if (zero? n) 'zero 'non-zero) is a valid if-expression

Further reading

-  The Dragon Book (2nd edition): Chapter 1.2 - The structure of a compiler, pages 4–11
-  Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I (by John McCarthy, 1960)
-  Apache Jelly: A scripting engine that uses a mixture of XML & Java that turns XML into executable code

Chapter 2

Goals

- ✓ The pipeline of the compiler
- ✓ Introduction to **syntactic analysis**
- 👉 Further steps in OCaml

Agenda

OCaml

- ▶ Types
- ▶ References
- ▶ Modules & signatures
- ▶ Functional programming in OCaml

Introduction to OCaml (2)

Still need to cover

To program in OCaml effectively in this course , we still need to learn some additional topics:

- ▶ Defining new data types
- ▶ Assignments, side-effects,

What we shan't cover

Object Orientation: Once you're comfortable with the OCaml, you might like to pick up the object-oriented layer. As object-orientation goes, you should find it to be sophisticated and expressive.

Types

New types are defined using the type statement:

```
type fraction = {numerator : int; denominator : int};;
```

The above statement defines a new type `fraction` as a **record** consisting of two fields: `numerator` & `denominator`, both of type `int`.

Types (*continued*)

Once fraction has been defined, the underlying system recognizes it for all records with these fields & types:

```
# {numerator = 2; denominator = 3};;  
- : fraction = {numerator = 2; denominator = 3}  
# {denominator = 3; numerator = 2};;  
- : fraction = {numerator = 2; denominator = 3}
```

Notice that the **order** of the fields in a record is immaterial, because the fields are accessed through their names, which are converted consistently into **offsets**.

Types (*continued*)

The type-inference engine in OCaml will correctly infer newly-defined types:

```
let add_fractions f1 f2 =
  match f1, f2 with
  | {numerator = n1; denominator = d1},
    {numerator = n2; denominator = d2} ->
    {numerator = n1 * d2 + n2 * d1;
     denominator = d1 * d2};;
```

And of course:

```
# add_fractions
  {numerator = 2; denominator = 3}
  {numerator = 4; denominator = 5};;
- : fraction = {numerator = 22; denominator = 15}
```

Types (*continued*)

We can define **disjoint types** as follows:

```
type number =
| Int of int
| Frac of fraction
| Float of float;;
```

Think of the | as **disjunction**. The initial | is **optional** in OCaml.

Types (*continued*)

We can now define a **list of numbers** as follows:

```
# [Int 3;
  Frac {numerator = 3;
         denominator = 4};
  Float (4.0 *. atan(1.0))];;
- : number list =
[Int 3;
  Frac {numerator = 3;
         denominator = 4};
  Float 3.14159265358979312]
```

Notice that OCaml had no trouble identifying each of the three elements of the list as belonging to type `number`.

Types (*continued*)

Working with disjoint types

Use `match` to dispatch over the corresponding type constructor, and make sure you handle each and every possibility!

```
let string_of_number = function
| Int n -> Format.sprintf "%d" n
| Frac {numerator = num; denominator = den} ->
  Format.sprintf "%d/%d" num den
| Float x -> Format.sprintf "%f" x;;
```

Types (*continued*)

Working with disjoint types (*continued*)

And here's how it looks:

```
# number_to_string (Int 234);;
- : string = "234"
# number_to_string (Frac {numerator = 2; denominator
    = 5});;
- : string = "2/5"
# number_to_string (Float 234.234);;
- : string = "234.234000"
```

References

Let us take another look at the record-type. Recall the definition of fraction:

```
# type fraction = {numerator : int; denominator : int};  
type fraction = { numerator : int; denominator : int; }
```

In the function `add_fractions` we used pattern-matching to access the record-fields.

References (*continued*)

OCaml lets you access fields directly, using the **dot-notation** that is familiar from **object-oriented programming**:

```
# {numerator = 3; denominator = 5}.numerator;;
- : int = 3
# {numerator = 3; denominator = 5}.denominator;;
- : int = 5
```

References (*continued*)

OCaml offers a special record-type known as a **reference**.

- ▶ References are **derived types**. For any type $\underline{\alpha}$, we can have a type $\underline{\alpha} \text{ ref}$.
- ▶ References are records with a single field contents
- ▶ References have a special syntax ! to dereference the field:

```
# {contents = 1234};;
- : int ref = {contents = 1234}
# {contents = 1234}.contents;;
- : int = 1234
# ! {contents = 1234};;
- : int = 1234
```

References (*continued*)

- ▶ References have a special syntax := for assignment
- ▶ This is how assignments are managed in OCaml

```
# let x = ref 1234;;
val x : int ref = {contents = 1234}
# x;;
- : int ref = {contents = 1234}
# !x;;
- : int = 1234
# x := 4567;;
- : unit = ()
# x;;
- : int ref = {contents = 4567}
# !x;;
- : int = 4567
```

References (*continued*)

- ▶ It is **not possible** to perform assignments on variables
- ▶ It is **only possible** to change the fields of reference types

```
# let x = "abc";;
val x : string = "abc"
# x := "def";;
Characters 0-1:
  x := "def";;
  ^
```

Error: This expression has type string
but an expression was expected of type
 'a ref

References (*continued*)

- You can define a reference type of any other type, including other reference types:

```
# let x = ref (ref 1234);;
val x : int ref ref = {contents = {contents = 1234}}
# x := ref 5678;;
- : unit = ()
# x;;
- : int ref ref = {contents = {contents = 5678}}
# !x := 9876;;
- : unit = ()
# x;;
- : int ref ref = {contents = {contents = 9876}}
```

Modules, signatures, functors

Modules

- ▶ A **module** is a way of packaging functions, classes, variables, & types
- ▶ A **signature** is the type of a module
 - ▶ Visibility of a module can be **restricted** through the signature
- ▶ **Functors** are functions from functors/modules to functors/modules

Goals

- ▶ Learn to work with existing modules
- ▶ Learn to write your own modules

Modules, signatures, functors (*continued*)

We define the function `hyp` to compute the hypotenuse of a triangle, given two sides and the angle between them (*cosine law*). We use the auxiliary function `square`:

```
# module M = struct
    let square x = x *. x
    let hyp a b theta =
        sqrt((square a) +. (square b) -. 2.0 *. a *. b *. (cos theta))
end;;
module M : sig
    val square : float -> float
    val hyp : float -> float -> float -> float
end
```

Modules, signatures, functors (*continued*)

Both M.square and M.hyp are **visible**:

```
# M.hyp;;
- : float -> float -> float -> float = <fun>
# M.square;;
- : float -> float = <fun>
# M.square 2.0;;
- : float = 4.
# M.hyp 3.5 5.6 0.645771823239;;
- : float = 3.50763282088818817
```

Modules, signatures, functors (*continued*)

We define the module type based on the returned signature of M,
but with the square function removed:

```
# module type SigHyp = sig
    val hyp : float -> float -> float -> float
end;;
module type SigHyp = sig
    val hyp : float -> float -> float -> float
end
# module M : SigHyp = struct
    let square x = x *. x
    let hyp a b theta =
        sqrt((square a) +. (square b) -.
              2.0 *. a *. b *. (cos theta))
end;;
module M : SigHyp
```

Modules, signatures, functors (*continued*)

Visibility is now **restricted**:

- ▶ M.hyp is visible from outside M
- ▶ M.square is not visible from outside M
- ▶ Functions visible **from outside** may use functions visible **from inside**

```
# M.hyp;;
- : float -> float -> float -> float = <fun>
```

```
# M.square;;

```

Characters 0-8:

```
M.square;;
^~~~~~
```

Error: Unbound value M.square

```
# M.hyp 3.5 5.6 0.645771823239;;
- : float = 3.50763282088818817
```

Modules, signatures, functors (*continued*)

Summary

- ▶ Modules & signatures are the way to package functions & control visibility
 - ▶ Convenient, super-efficient, safe
 - ▶ No need to use local, nested functions to manage visibility
 - ▶ Always use signatures to control visibility!

Learn on your own

- ▶ Modules can contain types too, and be used to parameterize code with types
 - ▶ Simpler & better than generics & templates
- ▶ Functors map modules/functors \Rightarrow modules/functors

Further reading

-  The Objective Caml Programming Language, Chapter 12
-  An online tutorial on OCaml modules

Parsing Techniques

Dozens of parsing algorithms are known:

- ▶ Parsing algorithms are tailored to a specific kind of grammar
 - ▶ Different kinds of grammars can be parsed by different algorithms
 - ▶ Different kinds of grammars have different levels of complexity on the **Chomsky Hierarchy**
- ▶ Most programming languages can be described using **context-free grammars**
- ▶ Some older languages can only be described using **context-sensitive grammars**

Parsing Techniques (*continued*)

Context-free Grammars (CFGs)

A CFG is a structure of the form $G = \langle V, \Sigma, R, S \rangle$:

- ▶ V is a set of **non-terminals**
- ▶ Σ is a set of **terminals**, or **tokens**
- ▶ R is a **relation** in $V \times (V \cup \Sigma)^+$
 - ▶ Members of R are called **production rules** or **rewrite rules**
- ▶ S is the **an initial non-terminal**

Parsing Techniques (*continued*)

Context-free Grammars (conveniences)

- ▶ We abbreviate the two productions $\langle A, X \rangle, \langle A, Y \rangle \in R$ with $\langle A, X \mid Y \rangle$ (disjunction)
- ▶ We abbreviate the three productions $\langle A, X \rangle, \langle X, \varepsilon \rangle, \langle X, BX \rangle \in R$, where X has no other productions, with $\langle A, B^* \rangle$, (Kleene-star)
- ▶ We abbreviate the three productions $\langle A, X \rangle, \langle X, B \rangle, \langle X, BX \rangle \in R$, where X has no other productions, with $\langle A, B^+ \rangle$, (Kleene-plus)
- ▶ We abbreviate the two productions $\langle A, \varepsilon \rangle, \langle A, B \rangle \in R$, with $\langle A, B? \rangle$ (maybe)

Parsing Techniques (*continued*)

The two basic approaches to parsing CFG are **top-down** & **bottom-up**:

Top-down algorithms

- ▶ Start with the initial non-terminal
- ▶ Rewrite the LHS of a non-terminal with its RHS, matching the input stream of tokens
- ▶ Keep rewriting until the entire input stream is matched

Parsing Techniques (*continued*)

The two basic approaches to parsing CFG are **top-down** & **bottom-up**:

Bottom-up algorithms

- ▶ Start with the input stream of tokens
- ▶ Find a rewrite rule where the RHS matches sequences in the input, and rewrite them to the LHS, **reducing** several items to a single non-terminal
- ▶ Keep rewriting until the entire input stream has been reduced to the initial non-terminal

Parsing Techniques (*continued*)

How most parsing algorithms are used

- ▶ Describe the grammar of the language using a DSL for some restricted CFG
 - ▶ Example: Backus-Naur Form (BNF)
- ▶ Associate **actions** with each production rule:
 - ▶ How to build the AST when a specific rule is matched
- ▶ A parser generator (e.g., *yacc*, *bison*, *antlr*, etc) compiles the grammar:
 - ▶ Performing various optimizations
 - ▶ Generating code in some language (C, Java, OCaml, etc)
 - ▶ This code is the parser
- ▶ Calling the parser on some input returns an AST

Parsing Techniques (*continued*)

Goals of parsing algorithms

- ▶ Minimal restrictions on the grammar
- ▶ Avoid backtracking as much as possible
- ▶ Maximum optimizations of the parser

Parsing Combinators

A technique for **embedding** a specification of a grammar into a programming language:

- ▶ Parsers for larger languages are **composed** from parsers for smaller languages
- ▶ The grammar can be written & debugged **bottom-up**
- ▶ The parsers are **first-class objects**:
 - ▶ We get to use abstraction to create complex parsers quickly & simply
 - ▶ Re-use effectively common sub-languages
- ▶ Simple to understand & implement
- ▶ Very rapid development

Parsing Combinators (*continued*)

Parsing combinators do have some disadvantages:

- ▶ The grammar is embedded **as-is**:
 - ▶ As much backtracking as implied by the grammar: Rewrite rules that have large common prefixes are going to require plenty of backtracking:

$$A \rightarrow xByCzDt$$

$$A \rightarrow xByCzDw$$

- ▶ No optimizations or transformations are performed on it!
- ▶ You can perform optimizations manually, by rewriting the grammar:

$$A \rightarrow xByCzDE$$

$$E \rightarrow t \mid w$$

Parsing Combinators (*continued*)

Parsing combinators do have some disadvantages:

- ▶ ε -productions & **left-recursion** result in infinite loops
 - ▶ We need to eliminate these manually!
- ▶ Can produce inefficient parsers rather efficiently! 😊

Parsing Combinators (*continued*)

Nevertheless:

- ▶ Parsing combinators are very simple to learn about grammars:
 - ▶ No complex algorithms are necessary!
 - ▶ The easiest way to design complex grammars & their parsers:
Abstraction —
 - ▶ shortens & simplifies the code
 - ▶ encourages re-use & consistency
- ▶ Optimizations can always be done manually!

Parsing Combinators (*continued*)

- 👉 Our parsers take a string and an **index** (int) whence to start reading the string, and return an AST.
- 👉 Upon success, our parsers return a **parsing_result**, which is a record defined as follows:

```
# type α parsing_result = {  
    index_from : int;  
    index_to : int;  
    found : α  
};;  
type α parsing_result = { index_from : int;  
    index_to : int; found : α; }
```

- 👉 Why do we return an object of [polymorphic] type α ??

Parsing Combinators (*continued*)

Sometimes our parsers must fail on their input. When this happens, we **raise** an **exception** (which in other languages is called **throwing** an exception).

We should therefore define an exception:

```
exception X_no_match;;
```

Parsing Combinators (*continued*)

Parsing combinators are **compositional**. This means

- ▶ We build parsers of large languages by **combining** parsers for smaller [sub-]languages
- ▶ The procedures that combine parsers are called **parsing combinators** (PCs)
- ▶ **The base case** for this inductive composition of parsers requires that we be able to parse single characters
 - ▶ All other parsers are built on top of such simple parsers for single characters

Parsing Combinators (*continued*)

The const PC takes a **predicate** (char → bool), and return a parser that recognizes this character:

```
# let const pred =
  fun str index ->
  if (index < String.length str)
    && (pred str.[index])
  then {
    index_from = index;
    index_to = index + 1;
    found = str.[index]
  }
  else raise X_no_match;;
val const : (char -> bool) -> string -> int ->
            char PC.parsing_result = <fun>
```

Parsing Combinators (*continued*)

We define the non-terminal that recognizes the capital letter 'A' by calling const with a predicate that returns true if its argument is equal to 'A':

```
# let ntA = const (fun ch -> ch = 'A');;
val ntA : string -> int ->
            char PC.parsing_result = <fun>
```

Parsing Combinators (*continued*)

Using ntA

```
# ntA "ABC" 0;;
- : char PC.parsing_result = {PC.index_from =
  0; index_to = 1; found = 'A'}
# ntA "" 0;;
Exception: PC.X_no_match.
# ntA "aA" 0;;
Exception: PC.X_no_match.
# ntA "ABCA" 3;;
- : char PC.parsing_result =
  {PC.index_from = 3; index_to = 4; found
  = 'A'}
```

- We only match the **head** of the input
- Obviously, ntA fails on an empty string

Parsing Combinators (*continued*)

- We hide the interface to our parsers behind the procedure

```
test_string:
```

```
let test_string nt str index =
    nt str index;;
```

Parsing Combinators (*continued*)

We can now test as follows:

```
# test_string ntA "" 0;;
Exception: PC.X_no_match.
# test_string ntA "Abc" 0;;
- : char PC.parsing_result
= {PC.index_from = 0; index_to = 1; found = 'A'}
```

- ▶ This is only for testing! When we deploy our parser, **we'll call it directly**.
- ▶ We'll have other, nicer procedures for testing the parsers

Parsing Combinators (*continued*)

Constant parsers are not very useful! Let's consider **catenation**:

```
let caten nt_1 nt_2 str index =
  let {index_from = index_from_1;
       index_to = index_to_1;
       found = e_1} = (nt_1 str index) in
  let {index_from = index_from_2;
       index_to = index_to_2;
       found = e_2} = (nt_2 str index_to_1) in
  {index_from = index_from_1;
   index_to = index_to_2;
   found = (e_1, e_2)};;
```

- ▶ We try to parse the head of s using nt1
 - ▶ If we succeed, we get e1 and the remaining chars s
 - ▶ We try to parse the head of s (what remained after nt1) using nt2
 - ▶ If we succeed, we get e2 and the remaining chars s
 - ▶ We return the **pair** of e1 & e2, as well as the remaining chars

Parsing Combinators (*continued*)

We define and test the parser for A followed by B:

```
# let ntAB =
    caten (const (fun ch -> ch = 'A'))
           (const (fun ch -> ch = 'B'));;
    val ntAB : string -> int -> (char * char)
PC.parsing_result = <fun>
# test_string ntAB "ABC" 0;;
- : (char * char) PC.parsing_result =
{PC.index_from = 0; index_to = 2; found = ('A', 'B')}
# test_string ntAB "abc" 0;;
Exception: PC.X_no_match.
# test_string ntAB "Abc" 0;;
Exception: PC.X_no_match.
# test_string ntAB "AB" 0;;
- : (char * char) PC.parsing_result =
{PC.index_from = 0; index_to = 2; found = ('A', 'B')}
# test_string ntAB "A\u00d7Bcdef" 0;;
Exception: PC.X_no_match.
```

Parsing Combinators (*continued*)

We now consider **disjunction** of two parsers:

```
# let disj nt1 nt2 str index =
  try (nt1 str index)
    with X_no_match -> (nt2 str index);;
val disj : ( $\alpha \rightarrow \beta \rightarrow \gamma$ ) -> ( $\alpha \rightarrow \beta \rightarrow \gamma$ ) ->  $\alpha \rightarrow \beta$ 
->  $\gamma$  = <fun>
```

- ▶ We try to parse the head of s using $nt1$
 - ▶ If we succeed, then the call to $nt1$ returns normally
 - ▶ If we fail we try to parse the head of s using $nt2$
- ▶ Notice that $disj$ is very abstract, in that OCaml can know very little about what we are passing it, and hence the polymorphic types α, β, γ that are part of the [polymorphic] type of $disj$!

Parsing Combinators (*continued*)



Can we do better? How can we inform OCaml that nt1 and nt2 are parsers?

👉 Using a **type-annotation!**

```
# let disj (nt1 : string -> int -> 'a
PC.parsing_result) nt2 str index =
  try (nt1 str index)
  with X_no_match -> (nt2 str index);;
val disj :
(string -> int -> α PC.parsing_result) ->
(string -> int -> α PC.parsing_result) ->
string -> int -> α PC.parsing_result = <fun>
```

👉 All we did was annotate nt1...



👉 How did OCaml deduce the type of nt2 from the type of nt1??

Parsing Combinators (*continued*)

We define and test the parser for either A or a:

```
# let ntA_or_a =
    disj (const (fun ch -> ch = 'A'))
          (const (fun ch -> ch = 'a'));;
val ntA_or_a : string -> int -> char
PC.parsing_result = <fun>
# test_string ntA_or_a "" 0;;
Exception: PC.X_no_match.
# test_string ntA_or_a "this\u00a9won't\u00a9work\u00a9either" 0;;
Exception: PC.X_no_match.
# test_string ntA_or_a "A\u00a9nice\u00a9example" 0;;
- : char PC.parsing_result = {PC.index_from = 0;
                                index_to = 1; found = 'A'}
# test_string ntA_or_a "a\u00a9nice\u00a9example" 0;;
- : char PC.parsing_result = {PC.index_from = 0;
                                index_to = 1; found = 'a'}
```

Parsing Combinators (*continued*)

What next?

- ▶ Some simple parsers
- ▶ Learn about the algebra of PCs
- ▶ Learn of new PC operators
- ▶ Learn how to use abstraction to make our life simpler
- ▶ The Self-Study Tutorial

Some simple parsers

```
let nt_epsilon str index =
  {index_from = index; index_to = index; found = []};;

let nt_none _str _index = raise X_no_match;;

let nt_end_of_input str index =
  if (index < String.length str)
  then raise X_no_match
  else {index_from = index; index_to = index; found =
    []};;
```

- ▶ nt_epsilon is the parser that recognizes ε -productions
- ▶ nt_none is the parser that always fails
- ▶ nt_end_of_input is the parser that recognizes the end of the input stream (and fails otherwise)

Parsing Combinators (*continued*)

What next?

- ✓ Some simple parsers
- ▶ Learn about the algebra of PCs
- ▶ Learn of new PC operators
- ▶ Learn how to use abstraction to make our life simpler
- ▶ The Self-Study Tutorial

The Algebra of PCs

Why do `nt_epsilon` & `nt_end_of_input` match with the **empty list** `[]`?

This has to do with the Algebra of parsing combinators:

- ▶ What is the **unit element** of catenation?
 - ▶ Answer: $r = \varepsilon$
 - ▶ We're looking for a non-terminal r such that for any s , we have $rs = sr = s\dots$
 - ▶ This means that `nt_epsilon` is the unit element for caten:
 - ▶ $\text{caten } \text{nt_epsilon} \text{ nt} \equiv \text{caten } \text{nt} \text{ nt_epsilon} \equiv \text{nt}$
 - ▶ Both `nt_epsilon` & `nt_end_of_input` are used '**til the end of something**'
 - ▶ The natural operation is to create a list of all things until ε or the end-of-input are reached
 - ▶ The unit element for **append** on lists is the empty list
 - ▶ Ergo, it is natural to match `[]` when either condition is encountered

The Algebra of PCs (*continued*)

Similarly, `nt_none` is the unit element in the algebra of **disjunction**:

$$\text{disj } \text{nt } \text{nt_none} \equiv \text{disj } \text{nt_none } \text{nt} \equiv \text{nt}$$

- 👉 Later on, we shall use the algebra of PCs together with **folding** operations to create complex parsers easily

Parsing Combinators (*continued*)

What next?

- ✓ Some simple parsers
- ✓ Learn about the algebra of PCs
- ▶ Learn of new PC operators
- ▶ Learn how to use abstraction to make our life simpler
- ▶ The Self-Study Tutorial

New PC Operators

Identifying the characters, or pairs of characters, etc, that match a grammar is often not enough:

- ▶ We want to be able to create an AST for that piece of syntax
- ▶ We do this by specifying **postprocessing** or **callback** functions over the expression that was matched.
- ▶ In our package, the PC that performs this is called pack

```
# let pack nt f str index =
    let {index_from; index_to; found} = (nt str
    index) in
    {index_from; index_to; found = (f found)};;
val pack :
  ( $\alpha \rightarrow \beta \rightarrow \gamma$  parsing_result) ->
  ( $\gamma \rightarrow \delta$ ) ->  $\alpha \rightarrow \beta \rightarrow$ 
   $\delta$  parsing_result = <fun>
```

- ▶ pack takes a non-terminal nt and a function f
 - ▶ returns a parser that recognizes the same language as nt
 - ▶ ...but which applies f to whatever was matched

New PC Operators (*continued*)

Adding type-annotations:

```
# let pack (nt : α parser) (f : α -> β) str index =
  let {index_from; index_to; found} = (nt str
  index) in
  {index_from; index_to; found = (f found)};;
val pack : α PC.parser -> (α -> β) -> string -> int
-> β PC.parsing_result = <fun>
```

- ▶ This is actually much better! 😊
 - ▶ You can now **clearly see** that `disj` takes two parsers (curried), and returns a parser

Parsing combinators (*continued*)

Example: Identifying digits

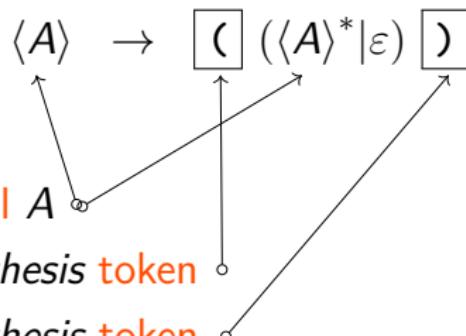
```
# let nt_digit_0_to_9 =
  const (fun ch -> '0' <= ch && ch <= '9');;
  val nt_digit_0_to_9 : string -> int -> char PC.
  parsing_result = <fun>
# test_string nt_digit_0_to_9 "234" 0;;
- : char PC.parsing_result = {index_from = 0; index_to =
  1; found = '2'}
# let nt_digit_0_to_9 =
  pack (const (fun ch -> '0' <= ch && ch <= '9'))
    (let ascii_0 = int_of_char '0' in
     (fun ch -> (int_of_char ch) - ascii_0));;
  val nt_digit_0_to_9 : string -> int -> int PC.
  parsing_result = <fun>
# test_string nt_digit_0_to_9 "234" 0;;
- : int PC.parsing_result = {index_from = 0; index_to =
  1; found = 2}
```

Recursive productions

- ▶ Grammars are often **recursive** or **mutually-recursive**:
 - ▶ The non-terminal on the LHS of a production often appears on the RHS (**recursion**)
 - ▶ The non-terminal on the LHS of a production often appears in one of the RHSs of the transitive-reflexive closure of the relation (**mutual recursion**)
- ▶ Currently, we are unable to describe recursive rules using PCs

Recursive productions (*continued*)

We are unable to describe recursive rules using PCs:



- ▶ The **non-terminal** A
- ▶ The **open-parenthesis token**
- ▶ The **close-parenthesis token**
- ▶ Nevermind that we don't yet have star...
- ▶ We can't use nt_A before it's defined!

Recursive productions (*continued*)

We are unable to describe recursive rules using PCs:

```
let nt_A =
caten (const (fun ch -> ch = '('))
      (caten (disj (star nt_A)
                    nt_epsilon)
              (const (fun ch -> ch = ')'))));;
```

- ▶ Nevermind that we don't yet have star...
- ▶ We can't use nt_A before it's defined!

Recursive productions (*continued*)

We are unable to describe recursive rules using PCs:

- ▶ The problem is not specific to parsing combinators.
 - ▶ For example, you couldn't define in Scheme:
`(define f (g (h f)))`because you can't use something before it's defined! (Ok, in some languages you can!)
- ▶ So how are recursive definitions possible at all?
 - ▶ When you define a recursive function you are not using the function (the **closure** object) before it's defined
 - ▶ You are using **the address of the function**, rather than its **value**, before the function is defined
- ▶ Recursive functions are **circular data-structures**:
 - ▶ The language definition permits you to define **these particular circular structures statically**, rather than at **run-time**

Parsing combinators (*continued*)

To implement recursive parsers, we need to **delay** the evaluation of the recursive non-terminal

- ▶ “Wrap it in a lambda...”



Think of the function `cos`:

- ▶ The function `(fun x -> cos x)` computes the same thing as `cos`
- ▶ To pass around `cos`, the compiler must know the value of `cos` (the **closure** object)
- ▶ To pass around `(fun x -> cos x)`, the compiler needs to know the **address** of `cos`
- ▶ We shall have more to say about this later on

Recursive productions (*continued*)

Example: Identifying digits (*continued*)

```
# let rec nt_natural str =
  pack (caten nt_digit_0_to_9
        (disj nt_natural
              nt_epsilon))
        (function (a, s) -> a :: s) str;;
val nt_natural : string -> int ->
  int list PC.parsing_result = <fun>
```

► Notice the packing function (function (a, s) -> a :: s)

Recursive productions (*continued*)

Example: Identifying digits (*continued*)

```
# test_string nt_natural "1234" 0;;
- : int list PC.parsing_result =
{index_from = 0; index_to = 4; found = [1; 2; 3; 4]}
```

We are not done yet:

-  We got a list of digits, as opposed to a list of chars!
-  We want to **left-fold** these digits into a number in base 10

Parsers combinators (*continued*)

We pack the list of digits using a **left-fold**:

```
# let nt_natural =
let rec nt str =
  pack (caten nt_digit_0_to_9
        (disj nt
              nt_epsilon))
        (function (a, s) -> a :: s) str in
  pack nt
  (fun s ->
    (List.fold_left
      (fun a b -> 10 * a + b)
      0
      s));
val nt_natural : string -> int -> int PC.parsing_result =
<fun>
```

- ▶ Notice the type of the parser: `char list -> int * char list`

Recursive productions (*continued*)

Testing it:

```
# test_string nt_natural "1234" 0;;
- : int PC.parsing_result =
{index_from = 0; index_to = 4; found = 1234}
# test_string nt_natural "496351" 0;;
- : int PC.parsing_result =
{index_from = 0; index_to = 6; found = 496351}
```

Parsing combinators (*continued*)

- ▶ By now, our toolset of parsing combinators consists of
 - ▶ const
 - ▶ caten
 - ▶ disj
 - ▶ pack
- ▶ We can handle recursive grammars
- ▶ We can create ASTs
- ▶ In principle, we can implement parsers for any language
- 👉 We now wish to add additional PCs to simplify the task of writing parsers

New PC Operators (*continued*)

The Kleene-Star

The Kleene-star is a meta-production-rule, or a rule-schema, or a “macro” over production-rules.

- ▶ For any NT P , P^* stands for the rule $Pstar$ defined as follows:

$$Pstar \rightarrow PPstar \mid \varepsilon$$

- ▶ The point of the Kleene-star is to recognize the catenation of zero or more expressions in P .

Stephen Cole
Kleene



New PC Operators (*continued*)

Here is our support for the Kleene-star:

```
let rec star nt str index =
  try let {index_from = index_from_1;
           index_to = index_to_1;
           found = e} = (nt str index) in
    let {index_from = index_from_rest;
         index_to = index_to_rest;
         found = es} = (star nt str index_to_1)
         in
    {index_from = index_from_1;
     index_to = index_to_rest;
     found = (e :: es)}
  with X_no_match -> {index_from = index;
                        index_to = index; found = []};;
```

Notice how we match ε implicitly.

New PC Operators (*continued*)

The Kleene-plus

- ▶ For any NT P , P^+ stands for the rule $Pplus$ defined as follows:

$$Pplus \rightarrow PPplus \mid P$$

- ▶ The point of the Kleene-plus is to recognize the catenation of one or more expressions in P .
- ▶ Kleene didn't really invent the Kleene-plus
 - ▶ Rather, Kleene-plus is a natural extension of Kleene-star

New PC Operators (*continued*)

Here is our support for the Kleene-plus:

```
let plus nt =
  pack (caten nt (star nt))
    (fun (e, es) -> (e :: es));;
```

Notice how we define the Kleene-plus as the catenation of Kleene-star and the original NT.

New PC Operators (*continued*)

Let's test star and plus:

```
# let star_star = star (const (fun ch -> ch = '*'));;
val star_star : string -> int -> char list
    PC.parsing_result = <fun>
# let star_plus = plus (const (fun ch -> ch = '*'));;
val star_plus : string -> int -> char list
    PC.parsing_result = <fun>
# test_string star_star "****the\u00d7end!" 0;;
- : char list PC.parsing_result =
{index_from = 0; index_to = 4; found = ['*'; '*'; '*';
  '*']}
# test_string star_plus "****the\u00d7end!" 0;;
- : char list PC.parsing_result =
{index_from = 0; index_to = 4; found = ['*'; '*'; '*';
  '*']}
# test_string star_star "the\u00d7end!" 0;;
- : char list PC.parsing_result = {index_from = 0;
  index_to = 0; found = []}
# test_string star_plus "the\u00d7end!" 0;;
Exception: PC.X_no_match.
```

New PC Operators (*continued*)

OCaml provides the polymorphic type

α option = None | Some of α as a way of dealing with situations where a value may or may not exist.

We're going to use α option to implement **maybe**, which takes a parser r , and returns a parser $r^?$ that recognizes **zero or one occurrences** of whatever is recognized by r .

New PC Operators (*continued*)

```
let maybe nt str index =
  try let {index_from; index_to; found} = (nt str index)
  in
    {index_from; index_to; found = Some(found)}
  with X_no_match -> {index_from = index; index_to =
  index; found = None};;
```

New PC Operators (*continued*)

Assume you have the parser `nt_integer`, that recognizes integers.
Here is how we might use `maybe`:

```
# test_string nt_integer "1234" 0;;
- : int PC.parsing_result =
{index_from = 0; index_to = 4; found = 1234}
# test_string (maybe nt_integer) "1234" 0;;
- : int option PC.parsing_result =
{index_from = 0; index_to = 4; found = Some 1234}
# test_string (maybe nt_integer) "moshe" 0;;
- : int option PC.parsing_result =
{index_from = 0; index_to = 0; found = None}
```

You would use **pattern matching** (via `match`) to handle both cases
(`None`/`Some`)

New PC Operators (*continued*)

We might want to attach an arbitrary predicate to serve as a **guard** for a parser, so that the parser succeeds only if the matched object satisfies the guard. This is what the `only_if` PC does:

```
let only_if nt pred str index =
  let ({index_from; index_to; found} as result) = (nt
    str index) in
  if (pred found) then result
  else raise X_no_match;;
```

New PC Operators (*continued*)

Let's use guard to identify only even numbers:

```
# test_string (only_if nt_integer (fun n -> n  
  land 1 = 0))  
          "12345" 0;;  
Exception: PC.X_no_match.  
# test_string (only_if nt_integer (fun n -> n  
  land 1 = 0))  
          "123456" 0;;  
- : int PC.parsing_result =  
{index_from = 0; index_to = 6; found = 123456}
```

This exceeds the expressive power of CFGs!

Parsing Combinators (*continued*)

What next?

- ✓ Some simple parsers
- ✓ Learn about the algebra of PCs
- ✓ Learn of new PC operators
- ▶ Learn how to use abstraction to make our life simpler
- ▶ The Self-Study Tutorial

Functional abstraction in PCs

We now wish to demonstrate some examples of using **functional abstraction** to write parsers in a general, consistent, and convenient way.

Up to now we used to define single-character parsers using const:

```
let nt_A = const (fun ch -> ch = 'A');;
```

This is kind of clumsy. Let's see how we can do this better!

Functional abstraction in PCs (*continued*)

```
let lowercase_ascii =
  let delta = int_of_char 'A' - int_of_char 'a' in
  fun ch ->
  if ('A' <= ch && ch <= 'Z')
  then char_of_int ((int_of_char ch) - delta)
  else ch;;

let make_char equal ch1 = const (fun ch2 -> equal
  ch1 ch2);;

let char = make_char (fun ch1 ch2 -> ch1 = ch2);;

let char_ci =
  make_char
  (fun ch1 ch2 ->
    (lowercase_ascii ch1) =
    (lowercase_ascii ch2));;
```

The use of `make_char` allows us to define parser-generating functions for characters, in a case-sensitive or case-insensitive way.

Functional abstraction in PCs (*continued*)

```
# test_string (char 'a') "abc" 0;;
- : char PC.parsing_result = {index_from = 0;
    index_to = 1; found = 'a'}
# test_string (char 'a') "ABC" 0;;
Exception: PC.X_no_match.
# test_string (char_ci 'a') "abc" 0;;
- : char PC.parsing_result = {index_from = 0;
    index_to = 1; found = 'a'}
# test_string (char_ci 'a') "ABC" 0;;
- : char PC.parsing_result = {index_from = 0;
    index_to = 1; found = 'A'}
```

Functional abstraction in PCs (*continued*)

If we wish to recognize entire words, this is still very cumbersome.
We can put to a good use the algebra of **catenation** to do better: To identify a word, we —

- ▶ Take a string of chars, and convert it to a list
- ▶ Map over each character in the list, creating a parser that recognizes **that** character
- ▶ Perform a **right fold** over that list using the caten operation (with an appropriate pack)
 - ▶ The **unit element** is the unit element of catenation, namely **epsilon**

By abstracting over `char` we can get both case-sensitive and case-insensitive variants!

Functional abstraction in PCs (*continued*)

Here is the code:

```
let make_word char str =
  List.fold_right
    (fun nt1 nt2 -> pack (caten nt1 nt2) (fun (a,
          b) -> a :: b))
    (List.map char (string_to_list str))
  nt_epsilon;;
let word = make_word char;;
let word_ci = make_word char_ci;;
```

Functional abstraction in PCs (*continued*)

```
# test_string (word "moshe")
          "moshe is a nice guy!" 0;;
- : char list PC.parsing_result =
{index_from = 0; index_to = 5; found = ['m'; 'o';
  's'; 'h'; 'e']}
# test_string (word_ci "moshe")
          "Moshe is a nice guy!" 0;;
- : char list PC.parsing_result =
{index_from = 0; index_to = 5; found = ['M'; 'o';
  's'; 'h'; 'e']}
```

Functional abstraction in PCs (*continued*)

We might want to pick **any** single character in a string. Rather than specifying long disjunctions, we can use `one_of` to do this for us.

- ▶ Very similar to `word`:
 - ▶ We use `disj` rather than `caten`
 - ▶ The **unit element** for `disj` is `nt_none`

Such is the power of abstraction!

Functional abstraction in PCs (*continued*)

```
let make_one_of char str =
  List.fold_right
    disj
    (List.map char (string_to_list str))
  nt_none;;  
  
let one_of = make_one_of char;;  
  
let one_of_ci = make_one_of char_ci;;
```

As usual, we generate both the case-sensitive and case-insensitive versions!

Functional abstraction in PCs (*continued*)

Let's try out one_of:

```
# test_string (one_of "abcdef") "moshe!" 0;;
Exception: PC.X_no_match.
# test_string (one_of "abcdef") "be moshe!" 0;;
- : char PC.parsing_result =
{index_from = 0; index_to = 1; found = 'b'}
```

Functional abstraction in PCs (*continued*)

When we wanted to recognize a **range** of characters, we, once again, used the `const` PC. We can do better using abstraction:

```
let make_range leq ch1 ch2 str index =
  const (fun ch -> (leq ch1 ch) && (leq ch ch2))
        str index;;  
  
let range = make_range (fun ch1 ch2 -> ch1 <= ch2)
;;  
  
let range_ci =
  make_range
    (fun ch1 ch2 ->
      (lowercase_ascii ch1) <=
      (lowercase_ascii ch2));;
```

Functional abstraction in PCs (*continued*)

And here is how we can test range:

```
# test_string (star (range 'a' 'z')) "hello world!"  
0;;  
- : char list PC.parsing_result =  
{index_from = 0; index_to = 5; found = ['h'; 'e'; '  
l'; 'l'; 'o']}
```



```
# test_string (star (range 'a' 'z')) "HELLO WORLD!"  
0;;  
- : char list PC.parsing_result = {index_from = 0;  
index_to = 0; found = []}
```



```
# test_string (star (range_ci 'a' 'z')) "Hello  
World!" 0;;  
- : char list PC.parsing_result =  
{index_from = 0; index_to = 5; found = ['H'; 'e'; '  
l'; 'l'; 'o']}
```

Functional abstraction in PCs (*continued*)

- ▶ The PC `trace_pc` is a **wrapper** (using the **decorator pattern**) that can be used to trace any parser
- ▶ The `trace_pc` PC takes a **documentation string** and a parser, and returns a tracing parser.

Functional abstraction in PCs (*continued*)

```
# test_string
  (trace_pc
    "The word \"hi\""
    (word "hi"))
  "high" 0;;
;;; The word "hi" matched from char 0 to char 2,
  leaving 2 chars unread
- : char list PC.parsing_result =
{index_from = 0; index_to = 2; found = ['h'; 'i']}
```



```
# test_string
  (trace_pc
    "The word \"hi\""
    (word "hi"))
  "bye" 0;;
;;; The word "hi" failed
Exception: PC.X_no_match.
```

Parsing Combinators (*continued*)

What next?

- ✓ Some simple parsers
- ✓ Learn about the algebra of PCs
- ✓ Learn of new PC operators
- ✓ Learn how to use abstraction to make our life simpler
- The Self-Study Tutorial

Further reading



Parsing Combinators

SELF-STUDY Hacking grammars using parsing combinators

What do we mean by “hacking grammars”

- ▶ Hacking grammars is the skill of being able to write quickly and effectively parsers for complex and interesting languages
- ▶ There is a gap between the theory of grammars and the implementation of practical parsers
- ▶ Much of the theory is concerned with languages, and the computational complexity of various grammars for these languages

SELF-STUDY Hacking grammars using parsing combinators

What do we mean by “hacking grammars”

- ▶ The actual implementation of parsers, the “applied” part of the study of grammars has to do with tools: scanner and parser generators (lex, flex, yacc, bison, etc)
 - ▶ These tools are mainly concerned with **generating efficient parsers**
 - ▶ They are not concerned with **generating parsers efficiently**
 - ▶ Sometimes using these tools can be very cumbersome
 - ▶ Most of these tools do not support much abstraction
 - ▶ Most of these tools place severe restrictions on the languages they support, in order to support them efficiently

SELF-STUDY Hacking grammars using parsing combinators

What do we mean by “hacking grammars”

- ▶ Our goal in this tutorial is to bypass all these details, and get you extremely adept at implementing parsers for all sorts of complex and interesting languages
- ▶ This is a very useful skill that will serve you in years to come
- ▶ Even if you never write a full, optimizing compiler, like *gcc*, you may still have to work with languages
- ▶ By the end of this tutorial you should be able to design a flexible language, and implement a parser for it using parsing combinators.
- ▶ This skill should make it easier for you to
 - ▶ Use standard scanner and parser generators later on
 - ▶ Implement parsers quickly and effectively

SELF-STUDY Hacking grammars using parsing combinators

What do we mean by “hacking grammars”

- ▶ Not all of your data is going to come in the form of XML (so you won't be able to read it using SAX or DOM parsers)
- ▶ Some of your data may be too complex for processing using regular expressions
- ▶ If you're working on a problem that is sufficiently large and complex, being able to design a DSL (*Domain-Specific Language*) to express aspects of your problem-domain, and then translate expressions in this DSL into other languages (C, Java, Python, etc), can work wonders for you!

SELF-STUDY Hacking grammars using parsing combinators

Optimize first for the most important resource: Your time!

- ▶ Our goal is to optimize our [the programmers'] time
- ▶ Produce a working parser as quickly as possible
- ▶ Keep the grammar as flexible and abstract as possible, to maintain consistency, correctness, and extensibility
 - ▶ Otherwise, we'll have a hellish job modifying the parser when we need to extend or change the grammar
 - ▶ If we need to optimize the parser, we can do this later, after we have a working prototype

SELF-STUDY Hacking grammars using parsing combinators

Optimize first for the most important resource: Your time!

Parsing combinators are an approach to **embedding** a grammar into a programming language, as a **recursive-descent parsers** in a **compositional** way

SELF-STUDY Hacking grammars using parsing combinators

Optimize first for the most important resource: Your time!

- ▶ **Embedding** means that the grammar is implemented **as-is**
 - ▶ Scanner and parser generators generally perform various transformations and optimizations on the grammar to implement it efficiently
 - ▶ Parsing combinators merely implement the grammar **as-is**, without changing or transforming it
 - ▶ If the embedded grammar is [overly] inefficient or the parser goes into infinite loops, the grammar needs to be modified by the user (by factoring common sub-parts, or removing left-recursion, etc)
 - ▶ While it is certainly possible to write super-efficient parsers using parsing combinators, it is entirely up to the programmer

SELF-STUDY Hacking grammars using parsing combinators

Optimize first for the most important resource: Your time!

- ▶ **Recursive-Descent** means that each **non-terminal** in the grammar is implemented as a [possibly] recursive procedure that consumes the **terminals** from the input-stream, in the form of **characters** or **tokens**
- ▶ **Compositional** means that larger parsers are built by composing smaller parsers (using the parsing combinators). The opposite of compositional is **monolithic**, i.e., made as one-piece (**Latin: mono** — one, *litus* — stone, *monolithic* — “made of one stone”)

SELF-STUDY Hacking grammars using parsing combinators

Optimize first for the most important resource: Your time!

- ▶ Parsing combinators are extremely easy to test, bottom-up, that is, long before the entire language is supported
 - ▶ They work great with the TDD methodology

SELF-STUDY Hacking grammars using parsing combinators

The importance of abstraction in hacking grammars

- ▶ Abstraction is the force-multiplier in computer-science and mathematics
 - ▶ Abstraction allows us to mass-produce things, according to a pattern, or a template
 - ▶ This automation buys us many things:
 - ▶ Consistency
 - ▶ Similar things are handled similarly
 - ▶ Exceptions must be stated explicitly
 - ▶ Conciseness & Speed of development
 - ▶ Abstractions are **meta-rules**, or rules for making rules, and a small number of meta-rules can replace a huge number of rules

 SELF-STUDY Hacking grammars using parsing combinators

- ▶ Parsing combinations embed a grammar as a **first-class object** in the language
 - ▶ In functional languages, parsers are implemented as functions
 - ▶ In object-oriented languages, parsers are implemented as objects
- ▶ Because our grammars are implemented as first-class objects, we can use the abstraction paradigms that come with the language to abstract common features out of various non-terminals, thereby creating **generator functions/methods** for these non-terminals

SELF-STUDY Hacking grammars using parsing combinators

Example 1 for abstraction

- ▶ Different programming languages define various kinds of sequences: Things that are separated by separators:
 - ▶ Statements separated by semicolons (Ocaml, Pascal, but not C (think why!))
 - ▶ Expressions separated by commas (function arguments in C, Java, Pascal; Array indecies in Pascal)
- ▶ There is a special [sub-]grammar for a [sub-]language of things separated by other things...
- ▶ In standard scanner & parser generators there is no way to avoid having to specify time and again, a specialized version of this sub-grammar

SELF-STUDY Hacking grammars using parsing combinators

Example 1 for abstraction (*cont*)

- ▶ Using parsing combinators, it is simple to implement a function that takes a non-terminal $\langle E \rangle$, that recognizes the things we wish to separate, and another non-terminal $\langle \text{Sep} \rangle$, that recognizes the separator, and return a parser that recognizes a sequence of $\langle E \rangle$ separated by $\langle \text{Sep} \rangle$.
- ▶ In fact, we may even want to have **two** such functions: One for parsers of zero or more $\langle E \rangle$, and another for parsers of one or more $\langle E \rangle$...

SELF-STUDY Hacking grammars using parsing combinators

Example 1 for abstraction (*cont*)

- ▶ Once we have written and debugged such functions
 - ▶ We can use them in any grammar that recognizes sequences
 - ▶ We can easily recognize a **variety** of sequences, making the grammar more flexible and reducing the learning-curve for the language
 - ▶ Things separated by commas, colons, semicolons, horizontal bars, ampersands, etc
 - ▶ And there shall be no bugs in these sub-grammars!

SELF-STUDY Hacking grammars using parsing combinators

Example 2 for abstraction

- ▶ The programming language BASIC comes with many commands, e.g., PRINT, INPUT, etc
- ▶ Some dialects of BASIC permit any **initial prefix** of a command to be used as an abbreviation for that command, so for example, the PRINT command can be written as P, PR, PRI, PRIN, PRINT
- ▶ This means that each command has several forms. Recognizing each of these forms is tedious and prone to errors
- ▶ Ordinary scanner & parser generators do not provide any solution to this: If you want your parser to support such a multitude of abbreviations, you need to hand-code them one at-a-time

SELF-STUDY Hacking grammars using parsing combinators

Example 2 for abstraction (*cont*)

- ▶ If you're using parsing combinators, it is simple to write a procedure that takes the string PRINT, and returns a parsing combinator that recognizes this, and all of its abbreviations.
- ▶ Once you write such a procedure, you can use it to generate parsers for recognizing all the commands in the language
- ▶ This approach is quick and concise, and lends itself better to supporting future extensions

SELF-STUDY Hacking grammars using parsing combinators

Example 2 for abstraction (*cont*)

This guarantees correctness:

- ▶ If you have a bug in the function that generates a parser for a keyword and all its prefixes, then the bug will appear everywhere, so it is easier to detect!
- ▶ Once detected and removed, the bug disappears from all uses of this function
- ▶ Once removed, this bug is gone forever

SELF-STUDY Hacking grammars using parsing combinators

Playing with a full deck of cards

The parsing combinators package you receive in this course is very rich

- ▶ It's being improved continuously, getting more toys each year
- ▶ A lot of effort has been spent on improving tools for debugging & tracing
- ▶ It supports many extensions to the language of CFGs, and far exceeds it

SELF-STUDY Hacking grammars using parsing combinators

Playing with a full deck of cards (*cont*)

- ▶ It supports many higher-order functions that provide abstraction over parsing combinators
 - ▶ These generators of non-terminals can save you a lot of time you might have to spend otherwise, writing and debugging common forms
- 👉 You can save yourself a lot of time and trouble by going over the file `pc.scm` and its documentation, and learning what's available

SELF-STUDY Hacking grammars using parsing combinators

Packing early and frequently

You will see many examples of packing later on, when we examine extended examples

- ▶ If you study these examples, you shall notice that I try to pack as early as possible
- ▶ Another thing you should notice that is that I try to keep the packing functions as simple as possible, preferring multiple pack statements over a single, larger one
- ▶ You are advised to do the same: Pack early, pack often, and keep the packing functions as simple and as concise as possible

SELF-STUDY Hacking grammars using parsing combinators

Packing early and frequently (*cont*)

Examples:

- ▶ When you recognize a **digit character**, use pack pack immediately to replace it with the corresponding digit (integer)
- ▶ When you recognize the characters for a **mantissa**, use pack immediately to fold[-right] the digits, and generate the mantissa
- ▶ If you don't use pack early, you will end up processing nasty, deeply-nested, complex data structures with special cases and exceptions (lists of lists ... of lists of characters)
- ▶ When you pack early and frequently, your **callback functions** can be simpler and shorter, and you keep your work neat, understandable, and uncluttered!

SELF-STUDY Hacking grammars using parsing combinators

Testing frequently

- ▶ Testing frequently is important in any programming problem, and it is especially important when you're writing parsers!
- ▶ When we program in a general-purpose programming language, we have a pretty good **mental model** of our code:
 - ▶ We know what a **loop** is
 - ▶ We know what a **conditional statement** is
 - ▶ We know what a **variable** is
 - ▶ We know how to **define functions, methods, classes**, etc
 - ▶ We know how all these statements are **performed**, and how they **effect the computing environment** (memory, screen, network, database, etc)
 - ▶ We know how **arithmetic and Boolean expressions** are evaluated

SELF-STUDY Hacking grammars using parsing combinators

Testing frequently

👉 And yet, despite our clear mental model —

- ▶ we still mess up
- ▶ we still introduce bugs
- ▶ we still need to fix things
- ▶ and so we still need to test, and often!

👉 And we have **far more experience** with general-purpose programming than with designing grammars for languages...

SELF-STUDY Hacking grammars using parsing combinators

Testing frequently

- ▶ We are far less adept in the skill of describing the syntax of languages formally, in terms of a grammar
 - ▶ Some definitions, while correct, can be extremely inefficient
 - ▶ Some definitions, while correct, are not suitable, as-is, to be embedded, as recursive-descent parsers, into a programming language — If they have left-recursions and epsilon-steps, recursive-descent parsers shall go into infinite loops
 - ⚠ Such loops may be tricky to detect, because the left-recursion might affect only a small part of the language

SELF-STUDY Hacking grammars using parsing combinators

Testing frequently

- 👉 For all these reasons and more, it is important to test frequently
 - ▶ 😊 Fortunately, the parsing combinators package you receive in this course contains special facilities that make testing and debugging simple and straightforward
- 👉 You are encouraged to follow the extended examples that follow, learn how to design your grammar, how to test and debug quickly and easily, and become a grammar hacker!

SELF-STUDY An Extended Example of Parsing: Numbers

In this tutorial we would like to be able to recognize various kinds of numbers

- ▶ Integers
- ▶ Fractions
- ▶ Real numbers

SELF-STUDY An Extended Example of Parsing: Numbers (*cont*)

 One of the things we're going to need to do is figure out how to convert integers to their corresponding ASCII values, and vice versa. Let's take a peek in the Char module using the directive `#show_module`:

```
# #show_module Char;;
module Char = Char
module Char :
  sig
    external code : char -> int = "%identity"
    val chr : int -> char
    ...
  end
```

I deleted anything we don't need to see.

SELF-STUDY An Extended Example of Parsing: Numbers (*cont*)

Notice that we have two procedures code and chr, which seem to have both the right names and the right types. Let's give them a try:

```
# Char.code;;
- : char -> int = <fun>
# Char.code 'a';;
- : int = 97
# Char.code ' ';;
- : int = 32
```

SELF-STUDY An Extended Example of Parsing: Numbers (*cont*)

```
# Char.chr;;
- : int -> char = <fun>
# Char.chr 97;;
- : char = 'a'
# Char.chr 32;;
- : char = ' '
```

So far so good!

SELF-STUDY An Extended Example of Parsing: Numbers (*cont*)

We can now define a non-terminal for recognizing a digit:

```
# let nt_digit =
  let nt1 = range '0' '9' in
  nt1;;
val nt_digit : string -> int -> char PC.
  parsing_result = <fun>
```

SELF-STUDY An Extended Example of Parsing: Numbers (*cont*)

We try out the code:

```
# test_string nt_digit "234" 0;;
- : char PC.parsing_result = {index_from = 0;
    index_to = 1; found = '2'}
# test_string nt_digit "234" 1;;
- : char PC.parsing_result = {index_from = 1;
    index_to = 2; found = '3'}
# test_string nt_digit "234" 2;;
- : char PC.parsing_result = {index_from = 2;
    index_to = 3; found = '4'}
```

SELF-STUDY An Extended Example of Parsing: Numbers (*cont*)

Conclusion

This is **almost** right:

- ▶ We can now recognize digits
- ▶ The object found, however, is a digit **character**
- ▶ We want the **numeric value** of the character
- ▶ We can get to it using pack and the Char.code procedure

SELF-STUDY An Extended Example of Parsing: Numbers (*cont*)

Using pack and Char.code to convert a digit char to the corresponding numeric value:

```
# let nt_digit =
  let nt1 = range '0' '9' in
  let nt1 = pack nt1
    (let delta = Char.code '0' in
     fun ch -> (Char.code ch) - delta) in
    nt1;;
val nt_digit : string -> int -> int PC.
  parsing_result = <fun>
```

SELF-STUDY An Extended Example of Parsing: Numbers (*cont*)

We try out the code:

```
# test_string nt_digit "234" 0;;
- : int PC.parsing_result = {index_from = 0;
    index_to = 1; found = 2}
# test_string nt_digit "234" 1;;
- : int PC.parsing_result = {index_from = 1;
    index_to = 2; found = 3}
# test_string nt_digit "234" 2;;
- : int PC.parsing_result = {index_from = 2;
    index_to = 3; found = 4}
```

SELF-STUDY An Extended Example of Parsing: Numbers (*cont*)

Getting from digits to a number, using folding

- ▶ Recursive functions fall into patterns
- ▶ Folding operators are two such patterns
- ▶ Functional programming languages encourage the use of such operators:
 - ▶ Very rapid coding
 - ▶ Shorter, clearer programs
 - ▶ Easy to prove correctness

SELF-STUDY An Extended Example of Parsing: Numbers (*cont*)

Getting from digits to a number, using folding

- **The left fold:** Given a binary function f , a list $[x_1; x_2; \dots; x_n]$ and a unit value u , the left-fold (`List.fold_left f u [x1; x2; ...; xn]`) computes $(f (\dots (f (f (f u x1) x2) x3) \dots) xn)$

SELF-STUDY An Extended Example of Parsing: Numbers (*cont*)

Getting from digits to a number, using folding

- **The right fold:** Given a binary function f , a list $[x_1; x_2; \dots; x_n]$ and a unit value u , the right-fold (`List.fold_left f [x1; x2; ...; xn] u`) computes $(f\ x_1\ (f\ x_2\ \dots\ (f\ x_n\ u)\ \dots))$

SELF-STUDY An Extended Example of Parsing: Numbers (*cont*)

Getting from digits to a number, using folding

- ⌚ Where might we want to use such folding operations??
 - ▶ As we shall soon see
 - ☞ To gather the digits to the **left of the decimal point** (the integer-part), we would use a **left-fold**
 - ☞ To gather the digits to the **right of the decimal point** (the **mantissa**), we would use a **right-fold**
 - ☞ It is trivial to change the folding functions to work with any counting base (e.g., **hexadecimal**)

SELF-STUDY An Extended Example of Parsing: Numbers (*cont*)

```
# List.fold_left;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# List.fold_left (fun a b -> 10 * a + b) 0 [4; 9;
  6; 3; 5; 1];;
- : int = 496351
```

- 👉 Using the left-fold, we can go from the list of digits to the integer value
- 👉 By replacing the number 10 with 16, in the folding operator, we can convert numbers in base 16 (and this generalizes to any counting base)
- 👉 If you've taken a course on **numerical analysis**, it should be apparent that this algorithm is essentially **Horner's Method**, where $x = 10$

SELF-STUDY An Extended Example of Parsing: Numbers (*cont*)

```
# test_string nt_nat "496351" 0;;
# let nt_nat =
  let nt1 = nt_digit in
  let nt1 = plus nt1 in
  let nt1 = pack nt1
    (fun digits ->
       List.fold_left
         (fun acc digit -> 10 * acc +
           digit)
         0
       digits) in
  nt1;;  
  
val nt_nat : string -> int -> int PC.parsing_result
= <fun>
# test_string nt_nat "496351" 0;;
- : int PC.parsing_result = {index_from = 0;
  index_to = 6; found = 496351}
```

Roadmap

- ▶ High-level optimizations
 - ▶ Constant-folding
 - ▶ Constant-propagation
- ▶ Low-level optimizations
 - ▶ Write-after-Write
 - ▶ Move-optimization
 - ▶ Jump-optimization

Roadmap

- 👉 High-level optimizations
 - ▶ Constant-folding
 - ▶ Constant-propagation
- ▶ Low-level optimizations
 - ▶ Write-after-Write
 - ▶ Move-optimization
 - ▶ Jump-optimization

Roadmap

- ▶ High-level optimizations
 - 👉 Constant-folding
 - ▶ Constant-propagation
- ▶ Low-level optimizations
 - ▶ Write-after-Write
 - ▶ Move-optimization
 - ▶ Jump-optimization

Roadmap

- ▶ High-level optimizations
 - ✓ Constant-folding
 - ☛ Constant-propagation
- ▶ Low-level optimizations
 - ▶ Write-after-Write
 - ▶ Move-optimization
 - ▶ Jump-optimization

SELF-STUDY Optimizations

Roadmap

- ✓ High-level optimizations
 - ✓ Constant-folding
 - ✓ Constant-propagation
- 👉 Low-level optimizations
 - ▶ Write-after-Write
 - ▶ Move-optimization
 - ▶ Jump-optimization

SELF-STUDY Optimizations

Roadmap

- ✓ High-level optimizations
 - ✓ Constant-folding
 - ✓ Constant-propagation
- ▶ Low-level optimizations
 - 👉 Write-after-Write
 - ▶ Move-optimization
 - ▶ Jump-optimization

Roadmap

- ✓ High-level optimizations
 - ✓ Constant-folding
 - ✓ Constant-propagation
- ▶ Low-level optimizations
 - ✓ Write-after-Write
 - 👉 Move-optimization
 - ▶ Jump-optimization

SELF-STUDY Optimizations

Roadmap

- ✓ High-level optimizations
 - ✓ Constant-folding
 - ✓ Constant-propagation
- ▶ Low-level optimizations
 - ✓ Write-after-Write
 - ✓ Move-optimization
 - 👉 Jump-optimization

SELF-STUDY Optimizations

Roadmap

- ✓ High-level optimizations
 - ✓ Constant-folding
 - ✓ Constant-propagation
- ✓ Low-level optimizations
 - ✓ Write-after-Write
 - ✓ Move-optimization
 - ✓ Jump-optimization

Roadmap

- ▶ Expressions in Scheme
 - ▶ The `expr` datatype
 - ▶ The Tag-Parser
 - ▶ Macros & special forms
 - ▶ Lexical hygiene

Expressions in Scheme

Recall that Scheme has two parsers

- ▶ A parser for **data**, known as a **reader**
- ▶ A parser for **code**, known as a **tag-parser**

Having completed our study of the **reader**, we now turn our attention to the **tag-parser**

Expressions in Scheme

The **abstract syntax** for code in Scheme can be described in terms of

- ▶ Constants
 - ▶ Variables
 - ▶ if-Expressions
 - ▶ or-Expressions
 - ▶ Sequences
 - ▶ Assignments
 - ▶ Definitions
 - ▶ Lambda-expressions of various kinds (more on that later)
 - ▶ Applications
- ☞ The above **forms** in the abstract syntax represent a choice that is reasonably complete, and easy to implement efficiently
- ☞ This is not the only possible choice

Expressions in Scheme

What about all the rest??

- ▶ All other forms (e.g., let, cond, etc) can be written in terms of these **core forms**
- ▶ The process of **translating** other forms (e.g., let-expressions) into the language of the **core forms** is called **macro expansion**
- ▶ The Scheme programming language provides facilities for **user-defined macros**
 - ▶ Users can invent new syntactic forms
 - ▶ Describe their translation into pre-existing forms using Scheme code
 - ▶ The Scheme system uses the user-code to translate **user-defined macros** into the core forms

The choice of the core forms

The choice of the core forms is guided by **minimalism & efficiency**:

Minimalism

- ▶ Supporting many forms in the compiler will make the compiler huge, complex, and harder to modify
- ▶ Allowing the compiler to focus on optimizing a small number of core forms results in smaller, simpler compilers
- ▶ Debugging macro-expanded code is very difficult

Minimalism

The choice of the core forms is guided by **minimalism & efficiency**:

Efficiency

There is a debate as to whether less forms would make for a better compiler:

- ▶ On the one hand, the minimalist compiler can focus more on less
- ▶ On the other hand, the minimalist compiler knows less about the programmer's intentions

The question of the advantages vs the disadvantages of minimalism is an age-old dilemma:

Archilochus πόλλ' οἶδ' ἀλώπηξ, ἀλλ' ἐχῖνος ἔν μέγα

Erasmus *Multa novit vulpes, verum echinus unum magnum*

The fox knows many things, but the hedgehog knows one great thing

Minimalism (*continued*)

For example, as we shall see later on, we're going to translate let-expressions into **applications**. But is this such a good idea?

- Pro** The semantic analysis & code generation will be smaller and simpler
- Con** The resulting code is going to be inefficient
- Pro** A good optimizing compiler can optimize away this inefficiency
- Con** But mapping the variable from memory to a register, for efficiency, would be harder
- Pro** ...

This debate can go on for a while!

Minimalism (*continued*)

For your project, we shall attempt to strike a balance:

- ▶ The compiler should not be difficult to write
- ▶ The code should not be overly inefficient 😊

The list of core forms suggested achieves this balance

Minimalism (*continued*)

Just a quick question

What is the absolute minimum number of core forms that are needed to be **Turing-complete**?

Minimalism (*continued*)

Answer

- ▶ In principle, applications and lambda-expressions are all you need
- ▶ Alternately, applications, and the following one, single constant are sufficient:

```
(define love
  (lambda (x)
    ((x (lambda (x)
              (lambda (y)
                (lambda (z)
                  (((x z) (y z)))))))
     (lambda (x) (lambda (y) x)))))
```

And “love is all you need!” (with apologies to the Beatles)...

Expressions in Scheme

We now define the core forms in terms of their **AST node types**, which are represented as **type-constructors** in the **disjoint type expr**:

```
type var =
  | Var of string;;
type lambda_kind =
  | Simple
  | Opt of string;;
type expr =
  | ScmConst of sexpr
  | ScmVarGet of var
  | ScmIf of expr * expr * expr
  | ScmSeq of expr list
  | ScmOr of expr list
  | ScmVarSet of var * expr
  | ScmVarDef of var * expr
  | ScmLambda of string list * lambda_kind * expr
  | ScmApplic of expr * expr list;;
```

Expressions in Scheme

Constants (type expr, type-constructor ScmConst)

- ▶ Type: ScmConst of sexpr
- ▶ The type-constructor ScmConst is used both for **self-evaluating** and **non-self-evaluating** constants



A **self-evaluating constant** is one you can type at the Scheme prompt and see it printed back at you: Numbers, chars, Booleans, strings

$$\begin{aligned} \llbracket \langle \text{self-evaluating sexpr} \rangle \rrbracket &= \text{ScmConst}(\langle \text{self-evaluating sexpr} \rangle) \\ \llbracket (\text{quote } \langle \text{sexpr} \rangle) \rrbracket &= \llbracket \text{Pair}(\text{Symbol}("quote"), \\ &\quad \text{Pair}(\langle \text{sexpr} \rangle, \text{Nil})) \rrbracket \\ &= \text{ScmConst}(\langle \text{sexpr} \rangle) \end{aligned}$$

Expressions in Scheme

Constants (type expr, type-constructor ScmConst)

⚠ A tricky example:

```
[(quote (quote <expr>))]  
= [Pair(Symbol("quote"),  
       Pair(Pair(Symbol("quote"),  
                  Pair(<expr>, Nil)),  
             Nil))]  
= ScmConst(Pair(Symbol("quote"),  
                  Pair(<expr>, Nil)))
```

Expressions in Scheme

Variables (type expr, type-constructor ScmVarGet)

- ▶ Type: ScmVarGet of var
- ▶ Variables are **literal symbols** that are not **reserved words**
 - ▶ The latest version of Scheme (R⁶RS) does not have many **reserved words**
 - ▶ Not having reserved words makes the parser more complex
 - ▶ We're going to ignore this, and assume that words that are used for **syntax** are reserved words. These include:
 - ▶ and, begin, cond, define, else, if, lambda, let, let*, letrec, or, quasiquote, quote, set!, unquote, unquote-splicing
 - ▶ There are additional reserved words, but we'll ignore those

Expressions in Scheme

Conditionals (type expr, type-constructor ScmIf)

- ▶ Type: ScmIf of expr * expr * expr
- ▶ Scheme supports **if-then** variant without an **else**-clause
 - ▶ These are used when the **then**-clause contains side-effects
 - ▶ The “missing”/implicit **else**-clause is defined to be ScmConst(Void)
 - ▶ We shall support the **if-then** variant, and tacitly add the implicit **else**-clause
- ▶ This is your first **recursive** case of the **expr** datatype: An **expr** that contains sub-**exprs**.
 - ▶ Obviously, the tag-parser will have to be recursive!

Expressions in Scheme

Sequences (type expr, type-constructor ScmSeq)

- ▶ Type: ScmSeq of expr list
- ▶ There are two types of sequences:
 - ▶ Explicit sequences (begin-expressions)
 - ▶ Implicit sequences
 - ▶ Body of lambda
 - ▶ In the Ribs of cond
 - ▶ In the body of let, let*, letrec
 - ▶ Other syntactic forms we shall not support
- ▶ Both **implicit** & **explicit** sequences are encoded as single expressions using the type-constructor ScmSeq
- ▶ All expressions within a sequence are generally executed from first to last

Expressions in Scheme

Sequences (type expr, type-constructor `ScmSeq`)

- ▶ The value of a sequence is generally the value of the last expression in the sequence
- ▶ Sequences have to do with side-effects: Expressions that have no side effects are redundant (can be removed) in all but the last position of a sequence
- ▶ Continuation-handling forms, such as `call/cc`, affect the behaviour of sequences
 - ▶ We shall not consider `call/cc` in this course
- ▶ It is possible to macro-expand sequences
 - ▶ We will learn the expansion later on
 - ▶ The expansion results in impractically-inefficient code
 - ▶ We support sequences directly for reasons of efficiency

Expressions in Scheme

Assignments (type expr, type-constructor ScmVarSet)

- ▶ Type: ScmVarSet of var * expr
- ▶ The AST node for set! (pronounced “set-bang”) expressions
- ⚠ The mother of all change; The essence of side-effects: While assignment is difficult to analyze, and does most of its damage at run-time, there's not much to say about it syntactically.

Expressions in Scheme

Definitions (type expr, type-constructor ScmVarDef)

- ▶ Type: ScmVarDef of var * expr
- ▶ The AST node for define-expressions
- ▶ Two syntaxes for define:

① (define <var> <expr>)

- ▶ Example:

```
(define pi (* 4 (atan 1)))
```

Expressions in Scheme

Definitions (type expr, type-constructor ScmVarDef)

- ▶ Two syntaxes for define:

② `(define (<var> . <arglist>) . (<expr>+))`

- ▶ Called **MIT-define**, because of its use in the MIT course (and textbook by the same name) *The Structure and Interpretation of Computer Programs* (see bibliography)
- ▶ This form is macro-expanded into
`(define <var> (lambda <arglist> . (<expr>+)))`
- ▶ Used to define functions without specifying the λ : This is **almost always** a bad idea!
- ▶ Note the implicit sequences!
- ▶ Example: `(define (square x) (* x x))`

Expressions in Scheme

Disjunctions (type expr, type-constructor ScmOr)

- ▶ Type: ScmOr of expr list
- ▶ $\llbracket (\text{or}) \rrbracket = \llbracket \#f \rrbracket$ (by definition)
- ▶ $\llbracket (\text{or} \langle \text{expr} \rangle) \rrbracket = \llbracket \langle \text{expr} \rangle \rrbracket$ ($\#f$ is the **unit element** of or)
- ▶ The real work is done here:

$$\begin{aligned}\llbracket (\text{or} \langle \text{expr}_1 \rangle \cdots \langle \text{expr}_n \rangle) \rrbracket \\ = \text{ScmOr}(\llbracket \langle \text{expr}_1 \rangle \rrbracket; \cdots; \llbracket \langle \text{expr}_n \rangle \rrbracket)\end{aligned}$$

- ▶ It is possible to macro-expand disjunctions
 - ▶ We will learn the expansion later on
 - ▶ The expansion results in impractically-inefficient code
 - ▶ We support disjunctions directly for reasons of efficiency

Expressions in Scheme

Applications (type expr, type-constructor ScmApplic)

- ▶ Type: ScmApplic of expr * (expr list)
- ▶ The AST node separates the expression in the **procedure position** from the list of arguments
- ▶ The tag-parser recurses over the procedure & the list of arguments:
$$[(\langle \text{expr} \rangle \langle \text{expr} \rangle_1 \cdots \langle \text{expr} \rangle_n)] = \\ \text{ScmApplic}([\langle \text{expr} \rangle], [\langle \text{expr} \rangle_1]; \cdots; [\langle \text{expr} \rangle_n])$$

Expressions in Scheme

Lambda (type expr, type-constructor ScmLambda)

- ▶ Types:
 - ▶ ScmLambda of string list * lambda_kind * expr where lambda_kind can be either
 - ▶ Simple
 - ▶ Opt of string
- ▶ Scheme has three lambda-forms, and we are going to represent these three forms using a combination of the list of **parameters names** and the **lambda_kind**

Expressions in Scheme

Lambda (type expr, type-constructor ScmLambda)

- ▶ The general form of lambda-expressions is
 $(\lambda \langle arglist \rangle . (\langle expr \rangle^+))$:
- ① If $\langle arglist \rangle$ is a **proper list** of unique variable names, then the lambda-expression is said to be **simple**, and we represent it using the AST node `ScmLambda` with the value for `lambda_kind` as `Simple`

Expressions in Scheme

Lambdas (type expr, type-constructor ScmLambda)

- ▶ The general form of lambda-expressions is
 $(\lambda \langle arglist \rangle . (\langle expr \rangle^+))$:
- ② If $\langle arglist \rangle$ is the **improper list** $(v_1 \dots v_n . vs)$, then the lambda-expression is said to take at least n arguments:
 - ▶ The first n arguments are mandatory, and are assigned to v_1 through v_n respectively (unique variable names)
 - ▶ The list of values of any additional arguments is going to be the value of the **optional** parameter vs
 - ▶ If precisely n arguments are given, then the value of vs is going to be the **empty list**
 - ▶ We represent lambda-expressions with optional arguments by using the AST node `ScmLambda` with the value for `lambda_kind` as `Opt var`, for the optional variable `var`

Expressions in Scheme

Lambdas (type expr, type-constructor ScmLambda)

- ▶ The general form of lambda-expressions is
`(lambda <arglist> . (<expr>+))`:
- ③ If $\langle \text{arglist} \rangle$ is the symbol vs , then the lambda-expression is said to be **variadic**, and may be applied to **any** number of arguments:
 - ▶ The list of values of the arguments is going to be the value of the **optional** parameter vs
 - ▶ If no arguments are given, then the value of vs is going to be the **empty list**
 - ▶ We represent variadic lambda-expressions using the AST node `ScmLambda` (with an empty list) with the value for `lambda_kind` as `Opt var` for the optional variable `var`

Expressions in Scheme

Lambda With Optional Arguments — Demonstration

```
> (define f
  (lambda (a b c . d)
    `((a ,a) (b ,b) (c ,c) (d ,d))))
> (f 1)
```

Exception: incorrect number of arguments to
#<procedure f>
Type (debug) to enter the debugger.

Expressions in Scheme

Lambda With Optional Arguments — Demonstration

```
> (f 1 2)
```

Exception: incorrect number of arguments to
#<procedure f>

Type (debug) to enter the debugger.

```
> (f 1 2 3)
```

```
((a 1) (b 2) (c 3) (d ()))
```

```
> (f 1 2 3 4 5)
```

```
((a 1) (b 2) (c 3) (d (4 5)))
```

Expressions in Scheme

Variadic Lambda — Demonstration

```
> (define g
  (lambda s
    `(s ,s)))
> (g)
(s ())
> (g 1 2 3)
(s (1 2 3))
```

Roadmap

- ▶ Expressions in Scheme
 - ✓ The `expr` datatype
 - ▶ The Tag-Parser
 - ▶ Macros & special forms
 - ▶ Lexical hygiene

The Tag-Parser

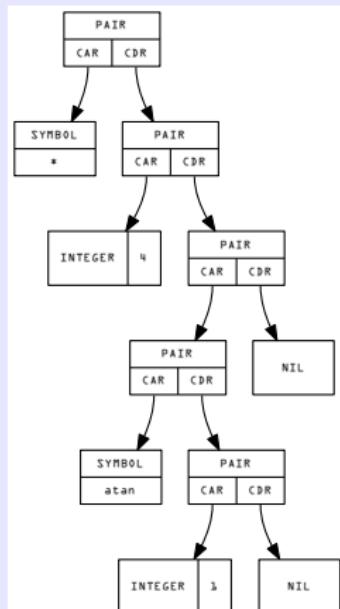
The tag-parser is a function, mapping from sexprs to exprs:

- ▶ Not all valid sexprs are valid exprs
- ▶ Some sexprs need to be disambiguated before they can be converted to an expr (most notably, the lambda-forms)
- ▶ Plenty of testing needs to be done to ensure that syntactically-incorrect forms are rejected

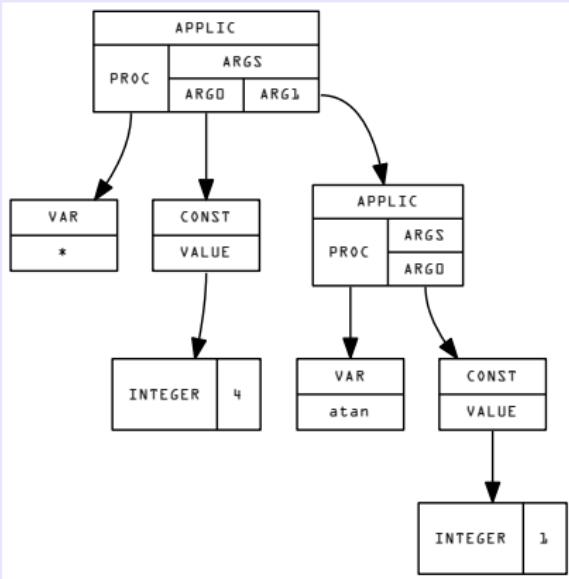
The Tag-Parser (*continued*)

Example: The syntax for $(\ast 4 (\text{atan} 1))$

Concrete syntax



Abstract syntax



The Tag-Parser (*continued*)

Some examples of syntactically-incorrect exprs

These are all valid sexprs and invalid expressions:

- ▶ (lambda (x) . x) (the body is not a [proper] list of exprs)
- ▶ (quote . a) (not a proper list)
- ▶ (quote he said he understood unquote) (length does not equal 2)
- ▶ (lambda (a b c a a b) (+ a b c)) (the param list contains duplicates)

The Tag-Parser (*continued*)

How to write a tag-parser

- ▶ A recursive function `tag_parse : sexpr -> expr`
 -  The concrete syntax for `expr` is the abstract syntax for `sexpr`
- ▶ For the core forms (we shall deal with macro-expanded forms later on)
 - ① Use pattern-matching to match over the **concrete syntax** of various syntactic forms
 - ▶ Check out the `expr` datatype so you know what you **must** support
 - ② Perform any additional testing necessary
 - ▶ For example, that argument-lists in lambda-expressions contain no duplicates!
 - ③ Call `tag_parse` recursively for sub-expressions
 - ④ Generate the corresponding AST

Roadmap

- ▶ Expressions in Scheme
 - ✓ The `expr` datatype
 - ✓ The Tag-Parser
 - ▶ Macros & special forms
 - ▶ Lexical hygiene

Macros & special forms

What are macros

- ▶ Transformers on source-code
 - ▶ They take source code, and rewrite it
 - ▶ They operate on the **concrete syntax**
- ▶ Generally execute at compile time
 - ▶ Some work on run-time macro-expansion has been done in the past, but is considered esoteric
- ▶ Used to provide **shallow** support for syntactic forms
 - ▶ Macros are **syntactic sugar** or notational conveniences
 - ▶ They are “expanded away”, and then they are **gone**
 - ▶ Macros are not supported deep within the compiler
 - ▶ The semantic analysis or code-generation stages of the compiler pipeline know nothing about macros

Macros & special forms (*continued*)

Illustrating “shallow support” vs “deep support”

- ▶ When you think of the word “home”, perhaps you think of:
 - ▶ stability, security, safety
 - ▶ family, relations
 - ▶ mortgage

etc.
- ▶ All this is part of the **meaning** of “home”
- ▶ **Meaning** in the compiler has to do with the **semantic analysis** phase of the compiler
- ▶ The **meaning** of “home” enriches anything you say and do that pertains to your home

Macros & special forms (*continued*)

Illustrating “shallow support” vs “deep support” (cont)

- ▶ Suppose home for you is just **shorthand** for a **location**
 - ▶ You could then translate the word “home” to USC 32.0260699N, 34.7580834E
 - ▶ This translation would take place **early on**, so that
 - ▶ “going home” would mean going to a specific coordinate
 - ▶ “longing for home” would mean longing to be at a specific coordinate
 - etc.
- ▶ Such sentences would carry none of the meaning, significance, feelings, associated with the word “home”
- ▶ The word “home” would be **disconnected from any meaning** other than a geographical location
- ▶ This would be insanity!

Macros & special forms (*continued*)

Illustrating “shallow support” vs “deep support” (cont)

Back to the compiler:

- ▶ When you have a notion of “loop” in your language, the compiler can associate with it many things:
 - ▶ a code fragment that gets executed over and over
 - ▶ termination conditions
 - ▶ branch prediction information
- etc.
- ▶ We can **macro-expand** a loop into a recursive function with all recursive calls in tail-position
- ▶ All intentions about the code (namely, its loop-like behaviour) are gone

Macros & special forms (*continued*)

Illustrating “shallow support” vs “deep support” (cont)

- ▶ Some information can be reconstructed through analysis during the **semantic analysis phase**
- ▶ Other information is lost
 - ▶ For example, we might want our compiler to keep the **index variables** of the loop in **registers**, or generate **branch-prediction hints**
 - ▶ These would be simple to do when considering loops
 - ▶ These would be difficult, and require a great deal of analysis with functions
- 👉 Keeping around the meaning of syntactic forms would make it easier to translate them efficiently
- 👉 This meaning having been lost in macro-expansion, it is difficult to recover completely

Macros & special forms (*continued*)

- ▶ We now consider some special forms in Scheme
 - ▶ Some of these will be implemented in our compiler
 - ▶ Some of these are of theoretical interest only
 - ▶ We want to understand what special forms can be dispensed with
 - ▶ We don't want our compiler to be overly inefficient

Scheme, Boolean values, Boolean operators

- ▶ Some languages (such as Java or ocaml) are **strict** about Booleans:
 - ▶ Conjunctions, disjunctions, conditionals, etc. only take expressions that evaluate to Boolean values
 - ▶ The distinction is between **false** and **true**
 - ▶ **Not false** is exactly **true**:

```
# not false;;
- : bool = true
# not true;;
- : bool = false
# 4 && 5;;
Characters 0-1:
  4 && 5;;
  ^
```

Error: This expression has type int but an expression
was expected of type bool

Scheme, Boolean values, Boolean operators

- ▶ Some languages (such as Java or ocaml) are **strict** about Booleans:

- ▶ Conjunctions, disjunctions, conditionals, etc. only take expressions that evaluate to Boolean values
- ▶ The distinction is between **false** and **true**
 - ▶ **Not false** is exactly **true**:

```
# 4 || 5;;  
Characters 0-1:  
 4 || 5;;  
^
```

Error: This expression has type int but an expression
 was expected of type bool

```
# if "moshe" then "then" else "else";;  
Characters 3-10:  
  if "moshe" then "then" else "else";;  
~~~~~
```

Error: This expression has type string but an expression
 was expected of type bool

Scheme, Boolean values, Boolean operators

- ▶ Some languages (such as C or Scheme) are **lenient** about Booleans:
 - ▶ Conjunctions, disjunctions, conditionals, etc. can take any expression
 - ▶ The distinction is between **false** and **not false**
 - ▶ **Not false** is not the same as **true**:

```
> (not #f)  
#t  
> (not #t)  
#f  
> (not 'moshe)  
#f  
> (not (not 'moshe))  
#t
```

Scheme, Boolean values, Boolean operators

- ▶ Some languages (such as C or Scheme) are lenient about Booleans:
 - ▶ Conjunctions, disjunctions, conditionals, etc. can take any expression
 - ▶ The distinction is between `false` and `not false`
 - ▶ `Not false` is not the same as `true`:

```
> (and 2 3 4)  
4  
> (or 2 3 4)  
2  
> (if 3 'then 'else)  
then  
> (if '() 'then 'else)  
then  
> (if #f 'then 'else)  
else
```

Macros & special forms (*continued*)

and

- ▶ Conjunctions are easily expanded into nested if-expressions:
 - ▶ $\llbracket (\text{and}) \rrbracket = \llbracket \#t \rrbracket$ (by definition)
 - ▶ $\llbracket (\text{and } \langle \text{expr} \rangle) \rrbracket = \llbracket \langle \text{expr} \rangle \rrbracket$ (#t is the **unit element** of and)
 - ▶ $\llbracket (\text{and } \langle \text{expr}_1 \rangle \langle \text{expr}_2 \rangle \cdots \langle \text{expr}_n \rangle) \rrbracket =$
 $(\text{if } \llbracket \langle \text{expr}_1 \rangle \rrbracket \llbracket (\text{and } \langle \text{expr}_2 \rangle \cdots \langle \text{expr}_n \rangle) \rrbracket \llbracket \#f \rrbracket)$
- ▶ The assembly code generated for and-expansions is no different from the assembly code that would have been generated had we supported and-expressions as a core syntactic form
- 👉 You should implement this macro-expansion in your tag-parser

Macros & special forms (*continued*)

or

Macro-expanding or-expressions is very different from macro-expanding and-expressions

- ▶ The first two clauses are similar to what we do with and-expressions:
 - ▶ $\llbracket (\text{or}) \rrbracket = \llbracket \#f \rrbracket$ (by definition)
 - ▶ $\llbracket (\text{or } \langle \text{expr} \rangle) \rrbracket = \llbracket \langle \text{expr} \rangle \rrbracket$ (because $\#f$ is the **unit** of or)
- ▶ For the third clause, you might consider something like:
$$\llbracket (\text{or } \langle \text{expr}_1 \rangle \langle \text{expr}_2 \rangle \cdots \langle \text{expr}_n \rangle) \rrbracket = \\ (\text{if } \llbracket \langle \text{expr}_1 \rangle \rrbracket \llbracket \#t \rrbracket \llbracket (\text{or } \langle \text{expr}_2 \rangle \cdots \langle \text{expr}_n \rangle) \rrbracket)$$
- ▶ This macro-expansion is, of course, **incorrect!** (think why)

Macros & special forms (*continued*)

or (*continued*)

- ▶ Take another look at $\llbracket (\text{or } \langle \text{expr}_1 \rangle \langle \text{expr}_2 \rangle \cdots \langle \text{expr}_n \rangle) \rrbracket = (\text{if } \llbracket \langle \text{expr}_1 \rangle \rrbracket \llbracket \#t \rrbracket \llbracket (\text{or } \langle \text{expr}_2 \rangle \cdots \langle \text{expr}_n \rangle) \rrbracket)$
- ▶ Suppose we implemented or-expressions in this way: What would be the **value** of (or 2 3) ?
 - ◀ It would be #t
 - ▶ Scheme returns 2

Macros & special forms (*continued*)

or (*continued*)

Let us consider a simpler version of our problem: How to macro-expand $(\text{or } \langle expr_1 \rangle \langle expr_2 \rangle)$

- ▶ This is fine, because or-expressions associate!
- ▶ What about this macro-expansion: $\llbracket (\text{or } \langle expr_1 \rangle \langle expr_2 \rangle) \rrbracket = (\text{if } \llbracket \langle expr_1 \rangle \rrbracket \llbracket \langle expr_1 \rangle \rrbracket \llbracket \langle expr_2 \rangle \rrbracket)$
- ▶ This macro-expansion is, of course, **incorrect!** (think why)

Macros & special forms (*continued*)

or (*continued*)

- ▶ Take another look at $\llbracket (\text{or } \langle \text{expr}_1 \rangle \langle \text{expr}_2 \rangle) \rrbracket = (\text{if } \llbracket \langle \text{expr}_1 \rangle \rrbracket \llbracket \langle \text{expr}_1 \rangle \rrbracket \llbracket \langle \text{expr}_2 \rangle \rrbracket)$
- ▶ Suppose we implemented or-expressions in this way: What would be the **output** of (or (begin (display "*\n") #t) 'moshe)
 - ⌚ It would print * **twice** and return #t
 - ▶ Scheme prints * **once** and returns #t
- ▶ We told you side-effects were tricky! ☺
- 👉 How might we make sure to evaluate $\langle \text{expr}_1 \rangle$ only once?

Macros & special forms (*continued*)

or (*continued*)

Suppose we used a let-expression to store the value of $\langle expr_1 \rangle$:

- ▶ What about the expansion: $\llbracket (\text{or} \langle expr_1 \rangle \langle expr_2 \rangle) \rrbracket = (\text{let } ((x \langle expr_1 \rangle)) (\text{if } x x \langle expr_2 \rangle))$
- ▶ This macro-expansion is, of course, **incorrect!** (think why)

Macros & special forms (*continued*)

or (*continued*)

- ▶ Take another look at $\llbracket (\text{or } \langle \text{expr}_1 \rangle \langle \text{expr}_2 \rangle) \rrbracket = (\text{let } ((\text{x } \langle \text{expr}_1 \rangle)) (\text{if } \text{x } \text{x } \langle \text{expr}_2 \rangle))$
- ▶ Suppose we implemented or-expressions in this way: What would be the **value** of `(let ((x 'ha-ha!)) (or #f x))`
 - ⌚ It would be `#f`
 - ▶ Scheme returns ha-ha!
- 👉 Why would this expansion evaluate to `#f`?

Macros & special forms (*continued*)

or (*continued*)

Look at how macro-expansion proceeds with our example:

```
[(let ((x 'ha-ha!)) (or #f x))]  
= (let ((x 'ha-ha!))  
    (let ((x #f))  
        (if x x x)))
```

- ▶ Notice how the macro-expansion **introduced a new** let between the original let and the or-expression
 - ▶ This new let introduced a variable binding that just so happens to use the same variable as in the outer let-binding
 - ▶ This new variable binding contaminated the code: The value of x was found in the wrong lexical environment!

Macros & special forms (*continued*)

or (*continued*)

Look at how macro-expansion proceeds with our example:

```
[(let ((x 'ha-ha!)) (or #f x))]  
= (let ((x 'ha-ha!))  
    (let ((x #f))  
        (if x x x)))
```

- ▶ Notice how the macro-expansion **introduced a new** let between the original let and the or-expression
 - ▶ This is known as a **variable-name capture**
 - ▶ Macro-expansions that result in variable-name captures are said to be **unhygienic**

Macros & special forms (*continued*)

or (*continued*)

A hygienic macro-expansion for `or` would require that no user-code may see any variables introduced by our macro-expansion

- ▶ This is often impossible to accomplish without resorting to tricks
- ▶ This often requires tricky, circuitous, counter-intuitive expansions that incur great performance penalties
 - ▶ This is the case with `or`
 - ▶ This is why we support disjunctions directly as a core form in our compiler
- 👉 You should not macro-expand `or`-expressions in your compiler!

Macros & special forms (*continued*)

or (*continued*)

This is how to macro-expand or-expressions (if someone were to hold a gun to your head and force you):

```
[(or <expr1> <expr2>)]
  = (let((x [<<expr1>>])
         (y (lambda () [<<expr2>>])))
    (if x x (y)))
  = ((lambda (x y) (if x x (y)))
      [<<expr1>>]
      (lambda () [<<expr2>>]))
```

- 👉 Notice that $\langle expr_1 \rangle, \langle expr_2 \rangle$ cannot access the variables x, y introduced by the expansion!

Macros & special forms (*continued*)

or (*continued*)

The cost of the hygienic expansion of or-expressions is high:

- ▶ Two lambda-expressions, and hence the creation of two closures
 - ▶ This is the same as allocating two objects in an OOPL
- ▶ Two applications

for each pair of two expressions

- 👉 For an or-expression that has $n + 1$ disjuncts, we would need:
 - ▶ To create/allocate $2n$ closures
 - ▶ To evaluate $2n$ applications
- 👉 By contrast, implementing the or-expression as a core form in our compiler requires **no allocation of closures** and **no applications**, regardless of the number of disjuncts!

Macros & special forms (*continued*)

begin

- ▶ The general form: `(begin <expr1> ... <exprn>)`
- ▶ Sequences are associative, so we need only consider the binary case: `(begin <expr1> <expr2>)`
- ▶ How might we possibly expand it? How about

$$\begin{aligned} & \llbracket (\text{begin } \langle \text{expr}_1 \rangle \langle \text{expr}_2 \rangle) \rrbracket \\ &= (\text{let } ((\text{x } \llbracket \langle \text{expr}_1 \rangle \rrbracket)) \\ & \quad \llbracket \langle \text{expr}_2 \rangle \rrbracket) \end{aligned}$$

- ▶ This macro-expansion is, of course, **incorrect!** (think why)

Macros & special forms (*continued*)

begin (*continued*)

- ▶ Take another look at

$$\begin{aligned} & \llbracket (\text{begin } \langle \text{expr}_1 \rangle \langle \text{expr}_2 \rangle) \rrbracket \\ &= (\text{let } ((\text{x } \llbracket \langle \text{expr}_1 \rangle \rrbracket)) \\ & \quad \llbracket \langle \text{expr}_2 \rangle \rrbracket) \end{aligned}$$

- ▶ This expansion introduces the variable x
 - ▶ Notice that $\llbracket \langle \text{expr}_2 \rangle \rrbracket$ can access this variable!
 - ▶ This means that this expansion is not hygienic!
- ▶ Suppose we implemented begin-expressions in this way: What would be the value of $(\text{let } ((\text{x } 3)) \text{ (begin } 2 \text{ x}))$
 - ⌚ It would be 2
 - ▶ Scheme returns 3

Macros & special forms (*continued*)

begin (*continued*)

How about

$$\begin{aligned} & \llbracket (\text{begin } \langle \textit{expr}_1 \rangle \langle \textit{expr}_2 \rangle) \rrbracket \\ &= (\text{if } \llbracket \langle \textit{expr}_1 \rangle \rrbracket \llbracket \langle \textit{expr}_2 \rangle \rrbracket \llbracket \langle \textit{expr}_2 \rangle \rrbracket) \end{aligned}$$

- ▶ We see that first $\llbracket \langle \textit{expr}_1 \rangle \rrbracket$ evaluates, and then $\llbracket \langle \textit{expr}_2 \rangle \rrbracket$, so the order is correct
- ▶ No variables are introduced, so there's no issue of lexical hygiene
- ▶ This expansion is actually correct, but bad! (think why)

Macros & special forms (*continued*)

begin (*continued*)

- ▶ Take another look at

$$\begin{aligned} & \llbracket (\text{begin } \langle \text{expr}_1 \rangle \langle \text{expr}_2 \rangle) \rrbracket \\ &= (\text{if } \llbracket \langle \text{expr}_1 \rangle \rrbracket \llbracket \langle \text{expr}_2 \rangle \rrbracket \llbracket \langle \text{expr}_2 \rangle \rrbracket) \end{aligned}$$

- ▶ The text of $\llbracket \langle \text{expr}_2 \rangle \rrbracket$ actually appears **twice** in the expanded form!
- ▶ This means that a begin with $n + 1$ expressions will expand to an expression of size $O(2^n)$
 - ▶ This is clearly not practical!

Macros & special forms (*continued*)

begin (*continued*)

How about

$$\begin{aligned} & \llbracket (\text{begin } \langle \text{expr}_1 \rangle \langle \text{expr}_2 \rangle) \rrbracket \\ &= (\text{and} \ (\text{or} \ \llbracket \langle \text{expr}_1 \rangle \rrbracket \ \llbracket \#t \rrbracket) \\ &\quad \llbracket \langle \text{expr}_2 \rangle \rrbracket) \end{aligned}$$

- ▶ We see that $(\text{or} \ \llbracket \langle \text{expr}_1 \rangle \rrbracket \ \llbracket \#t \rrbracket)$ evaluates first, and that its value is always `#t`
- ▶ So the `and` continues on to evaluate $\llbracket \langle \text{expr}_2 \rangle \rrbracket$
 - ▶ The ordering is correct!
- ▶ No variables are introduced so there's no issue of lexical hygiene
- ▶ No expression is duplicated
- ◀ This expansion actually works!

Macros & special forms (*continued*)

begin (*continued*)

- ▶ How about

$$\begin{aligned} & \llbracket (\text{begin } \langle \text{expr}_1 \rangle \langle \text{expr}_2 \rangle) \rrbracket \\ &= (\text{or} (\text{and} \llbracket \langle \text{expr}_1 \rangle \rrbracket \llbracket \#f \rrbracket) \\ &\quad \llbracket \langle \text{expr}_2 \rangle \rrbracket) \end{aligned}$$

- ▶ We see that first $(\text{and} \llbracket \langle \text{expr}_1 \rangle \rrbracket \llbracket \#f \rrbracket)$ evaluates, and then $\llbracket \langle \text{expr}_2 \rangle \rrbracket$, so the order is correct
- ▶ No variables are introduced, so there's no issue of lexical hygiene
- ◀ This expansion actually works!

Macros & special forms (*continued*)

begin (*continued*)

How about

```
[(begin <expr1> <expr2>)]  
= (let ((x [(<expr1>)])  
        (y (lambda () [<expr2>])))  
    (y))
```

- ▶ We introduced two variables x & y:
 - ▶ Neither [<expr₁>] nor [<expr₂>] can access these variables!
 - ▶ The solution is hygienic!
- ▶ [<expr₁>] evaluates in parallel with the creation of the closure
for (lambda () [<expr₂>])

Macros & special forms (*continued*)

begin (*continued*)

How about

```
[(begin <expr1> <expr2>)]  
= (let ((x [<expr1>])  
        (y (lambda () [<expr2>]))  
        (y))
```

- ▶ Evaluating `(lambda () [<expr2>])` **does not evaluate** `[(<expr2>)]`
- ▶ `[(<expr2>)]` is evaluated **only** when the closure is **applied!**
- ◀ This expansion actually works!

Macros & special forms (*continued*)

begin (*continued*)

How about

$$\begin{aligned} & \llbracket (\text{begin } \langle \text{expr}_1 \rangle \langle \text{expr}_2 \rangle) \rrbracket \\ &= (\text{let } ((\text{x } \llbracket \langle \text{expr}_1 \rangle \rrbracket) \\ &\quad (\text{y } (\text{lambda } () \llbracket \langle \text{expr}_2 \rangle \rrbracket))) \\ &\quad (\text{y})) \end{aligned}$$

- ▶ This expansion is actually more efficient than the previous two (which used and & or):
 - ▶ and & or with more than one expression always involve a test and a conditional jump
 - ▶ Sequencing does not logically require tests or conditional jumps
 - ▶ So this solution is less expensive

Macros & special forms (*continued*)

begin (*continued*)

How about

$$\begin{aligned} & \llbracket (\text{begin } \langle \text{expr}_1 \rangle \langle \text{expr}_2 \rangle) \rrbracket \\ &= (\text{let } ((\text{x } \llbracket \langle \text{expr}_1 \rangle \rrbracket) \\ &\quad (\text{y } (\text{lambda } () \llbracket \langle \text{expr}_2 \rangle \rrbracket)) \\ &\quad (\text{y})) \end{aligned}$$

- ▶ This expansion is actually more efficient than the previous two (which used and & or):
 - ▶ It's still pretty horrible: Two applications, and two closures created for every pair of expressions in a begin:
 - ▶ Sequences of $n + 1$ expressions require $2n$ applications and the creation of $2n$ closures, which are then garbage-collected

Macros & special forms (*continued*)

begin (*continued*)

How about

```
[(begin <expr1> <expr2>)]  
= (let ((x [(<expr1>)])  
        (y (lambda () [<expr2>])))  
    (y))
```

- ▶ This is why we support sequences **natively** within our compiler
- 👉 You should **not** implement this macro-expansion in your compilers!

Macros & special forms (*continued*)

let

- ▶ The let-expression is a way of defining any number of **local variables**, and assigning them **initial values**.
- ▶ Once the local variables have been initialized, they are accessible to an implicit sequence of expressions that are evaluated in their lexical scope.
- ▶ The syntax looks like this:

$$\begin{aligned} & (\text{let } ((v_1 \langle Expr_1 \rangle) \\ & \quad \dots \\ & \quad (v_n \langle Expr_n \rangle)) \\ & \quad \langle expr_1 \rangle \dots \langle expr_m \rangle) \end{aligned}$$

Macros & special forms (*continued*)

let (*continued*)

We wish to macro-expand let-expressions:

- ▶ Local variables are **parameters** of lambda-expressions
- ▶ Expressions that can access local variables come from the **bodies** of lambda-expressions
- ▶ The parameters of lambda-expressions get their values when lambda-expressions are **applied** to **arguments**
 - ▶ The values of the arguments are the **initial values** of the parameters

Macros & special forms (*continued*)

let (*continued*)

Putting it all together, we get the following macro-expansion:

$$\begin{aligned} & \llbracket (\text{let } ((\text{v}_1 \langle Expr_1 \rangle) \\ & \quad \dots \\ & \quad (\text{v}_n \langle Expr_n \rangle)) \\ & \quad \langle expr_1 \rangle \dots \langle expr_m \rangle) \rrbracket \\ = & \llbracket (\text{lambda } (\text{v}_1 \dots \text{v}_n) \\ & \quad \langle expr_1 \rangle \dots \langle expr_m \rangle) \\ & \quad \langle Expr_1^\circ \rangle \dots \langle Expr_n^\circ \rangle) \rrbracket \end{aligned}$$

- ▶ The expansion is hygienic (think why)
- ☞ You should implement this macro-expansion in your compiler!

Macros & special forms (*continued*)

let*

- ▶ Recall that the ordering of let-bindings is undefined, and that we may not assume they take place in any particular sequence
 - ▶ This follows from the fact that
- ▶ The asterisk in name of the let*-form is meant to suggest the Kleene-star
- ▶ A let*-expression denotes nested let-expressions. The following equations define the behaviour of the tag-parser on let*-expressions:
 - ① This is the first of the two base cases:

$$\begin{aligned} & \llbracket (\text{let*} () \langle \textit{expr}_1 \rangle \cdots \langle \textit{expr}_m \rangle) \rrbracket \\ &= \llbracket (\text{let} () \langle \textit{expr}_1 \rangle \cdots \langle \textit{expr}_m \rangle) \rrbracket \end{aligned}$$

Macros & special forms (*continued*)

let* (*continued*)

- ② This is the second base case:

$$\begin{aligned} & \llbracket (\text{let* } ((\text{v } \langle \textit{Expr} \rangle)) \langle \textit{expr}_1 \rangle \cdots \langle \textit{expr}_m \rangle) \rrbracket \\ &= \llbracket (\text{let } ((\text{v } \textit{Expr})) \langle \textit{expr}_1 \rangle \cdots \langle \textit{expr}_m \rangle) \rrbracket \end{aligned}$$

👉 Think why two base cases are needed here!

Macros & special forms (*continued*)

let* (*continued*)

- ③ This is the inductive case:

$$\begin{aligned} & \llbracket (\text{let* } ((v_1 \langle Expr_1 \rangle) \ (v_2 \langle Expr_2 \rangle) \ \cdots \ (v_n \langle Expr_n \rangle)) \\ & \quad \langle expr_1 \rangle \ \cdots \ \langle expr_m \rangle) \rrbracket \\ = & \ \llbracket (\text{let } ((v_1 \langle Expr_1 \rangle)) \\ & \quad (\text{let* } ((v_2 \langle Expr_2 \rangle) \ \cdots \ (v_n \langle Expr_n \rangle)) \\ & \quad \langle expr_1 \rangle \ \cdots \ \langle expr_m \rangle)) \rrbracket \end{aligned}$$

Macros & special forms (*continued*)

let* (*continued*)

The expansion for let*-expressions **seems** terribly inefficient:

- ▶ A nested let-expression for every **rib** in the original let*-expression
- ▶ This means
 - ▶ One more closure created
 - ▶ One application performed
 - ▶ One closure garbage-collected

for each **rib** in the original let*-expression

Macros & special forms (*continued*)

let* (*continued*)

In fact, **this inefficiency is illusory** because each call is performed in tail-position, so:

- ▶ The **call** is tail-call-optimized and replaced with a **branch**
- ▶ The allocation/creation of a new closure is avoided through a simple analysis in the **semantic analysis** phase
- ▶ Rather than allocating new frames, the old frames are being overwritten with the information of the new frames
- ▶ So in fact, this code is optimized into simple assignments
- 👉 **The bottom line:** The macro-expansion is efficient when compiled by a reasonably clever, optimizing compiler
- 👉 You should implement this macro-expansion in your compiler!

Macros & special forms (*continued*)

let* (*continued*)

- ▶ Because let*-expressions expand into nested let-expressions, let*-expressions tolerate repeated [re-]definitions of variables.
- 👉 In fact, this is partly how/why let*-expressions are used:

Macros & special forms (*continued*)

let* (*continued*)

An expression of the form

```
(let* ((x 1)
      (y 2)
      (x (+ x y))
      (y (* x y))
      (x (- x y))
      (y (expt x y)))
  (list x y))
```

is macro-expanded into:

```
(let ((x 1))
  (let ((y 2))
    (let ((x (+ x y)))
      (let ((y (* x y)))
        (let ((x (- x y)))
          (let ((y (expt x y)))
            (list x y)))))))
```

Macros & special forms (*continued*)

let* (*continued*)

And the expression

```
(let ((x 1))
  (let ((y 2))
    (let ((x (+ x y)))
      (let ((y (* x y)))
        (let ((x (- x y)))
          (let ((y (expt x y)))
            (list x y)))))))
```

is further [macro-]expanded into:

Macros & special forms (*continued*)

let* (*continued*)

is further [macro-]expanded into:

```
((lambda (x)
  ((lambda (y)
    ((lambda (x)
      ((lambda (y)
        ((lambda (x)
          ((lambda (y)
            ((list x y))
              (expt x y))))
            (- x y)))
          (* x y)))
        (+ x y)))
      2))
  1))
```

👉 The following calls are in tail-position

Macros & special forms (*continued*)

let* (*continued*)



We haven't yet studied the tail-position in this course, but you covered tail-position both in the courses

- ① Introduction to Computer Science With Java
- ② Principles of Programming Languages

- We shall cover the tail-position in detail in this course, especially in the context of the **control stack**, and the **tail-call optimization**
- But you should know that any reasonable optimizing compiler would convert this code into something equivalent to:

Macros & special forms (*continued*)

let* (*continued*)

- ▶ But you should know that any reasonable optimizing compiler would convert this code into something equivalent to:

```
(let ((x 1)
      (y 2))
  (set! x (+ x y))
  (set! y (* x y))
  (set! x (- x y))
  (set! y (expt x y))
  (list x y))
```

- 👉 We can use let*-expressions to write in a **purely functional way** code that would otherwise require the use of **side-effects to variables (assignments)**

Macros & special forms (*continued*)

letrec

Let's re-examine the special form let:

- ▶ We can use let to define local variables
- ▶ The value of these variables can be anything really, including functions:
 - ▶ Here's how one might define local procedures using let:

```
(let ((square
      (lambda (x)
        (* x x))))
  ;; here we can use the procedure square
  (sqrt
    (+ (square a)
        (square b)
        (* -2 a b (cos theta)))))
```

Macros & special forms (*continued*)

letrec (*continued*)

Nevertheless, let has one shortcoming when it comes to defining local procedures: Recursive procedures cannot be defined “as is”

- If we were to try to define and use the **factorial** function using let, it might look like:

```
(let ((fact
      (lambda (n)
        (if (zero? n)
            1
            (* n (fact (- n 1)))))))
  (fact 5))
```

Macros & special forms (*continued*)

letrec (*continued*)

Which expands to

```
((lambda (fact) (fact 5))
 (lambda (n)
  (if (zero? n)
      1
      (* n (fact (- n 1))))))
```

- ▶ Notice the body of fact is not able to access the parameter fact of the procedure (lambda (fact) (fact 5))
 - ▶ The parameter fact is only accessible in the **body** of this procedure
- ▶ The reference to fact within the text of the body of the factorial procedure is **free**, and refers to a **global variable** defined at the top-level.

Macros & special forms (*continued*)

letrec (*continued*)

To see things more clearly, we macro-expand the let. Note the parameter fact and whence it can be accessed:

```
(lambda (fact) (fact 5))  
  (lambda (n)  
    (if (zero? n)  
        1  
        (* n (fact (- n 1))))))
```

- :(This just looks like an example of recursion, but in fact :) it isn't!

Macros & special forms (*continued*)

letrec (*continued*)

An expansion that does work would be something like:

```
(let ((fact 'whatever))
  (set! fact
    (lambda (n)
      (if (zero? n)
          1
          (* n (fact (- n 1)))))))
  (fact 5))
```

- ▶ Can you see why it works?

Macros & special forms (*continued*)

letrec (*continued*)

Let's expand the let-expression and see why this expansion works:

```
(lambda (fact)
  (set! fact
    (lambda (n)
      (if (zero? n)
          1
          (* n (fact (- n 1)))))))
  (fact 5))
'whatever)
```

- ☞ The text of the body of the factorial procedure appears within the **body** of the `(lambda (fact) ...)` procedure, which is why it may access the `fact`: This **is** recursion!

Macros & special forms (*continued*)

letrec (*continued*)

The general macro-expansion implied by the last example is presented below:

```
[(letrec ((f1 <Expr1>)
          (f2 <Expr2>)
          ...
          (fn <Exprn>))
  <expr1> ... <exprm>)] = (let ((f1 'whatever)
                                         (f2 'whatever)
                                         ...
                                         (fn 'whatever))
  (set! f1 <Expr1>)
  (set! f2 <Expr2>)
  ...
  (set! fn <Exprn>)
  <expr1> ... <exprm>)
```

Macros & special forms (*continued*)

letrec (*continued*)

This expansion is **almost** right:

- ▶ The main problem with the expansion has to do with **nested** define-expressions
 - ▶ It is possible to use `define` to define a local function within a `let` or `lambda` form
 - ▶ Several of such definitions may appear at the top of the body
 - ▶ Several of such definitions may be grouped together within a `begin`-expression
 - ▶ All the nested `define`-expressions must appear at the top of the body even if they are grouped in different `begin`-expressions
 - ▶ It is a syntax error to have a nested `define` after a non-`define`-expression in the body of a `lambda` or `let`

Macros & special forms (*continued*)

letrec (*continued*)

This means that such definitions are possible:

```
(define f
  (lambda (a b c)
    (define g1 (lambda () ...)))
  (begin
    (define g2 (lambda (x) ...))
    (begin
      (define g3
        (lambda (x y) ...)))
      (define g4
        (lambda (z) ...))))
  ...))
```

Macros & special forms (*continued*)

letrec (*continued*)

And now we have a problem:

- If nested define-expressions appear within the body of a letrec-expression, and we macro-expand the letrec-expression into a let-expression with assignments at the top of its body, then the nested define-expressions will appear after the set!-expressions, and this would be syntactically illegal!
- In fact, we shall not support nested define-expressions in our compiler
 - ☞ You should implement this macro-expansion in your compilers!
- We still need to find a macro-expansion for letrec that can live with nested define-expressions...

Macros & special forms (*continued*)

letrec (*continued*)

This macro-expansion does the trick:

```
[(letrec ((f1 <Expr1>)
          (f2 <Expr2>)
          ...
          (fn <Exprn>))
  <expr1> ... <exprm>)] = [(let ((f1 'whatever)
          (f2 'whatever)
          ...
          (fn 'whatever))
  (set! f1 <Expr1>)
  (set! f2 <Expr2>)
  ...
  (set! fn <Exprn>)
  (let ()
    <expr1> ... <exprm>))]
```

letrec (*continued*)

Why does this macro-expansion work?

- ▶ Notice that the body of the original letrec-expression is now wrapped within a (in `(let () ...)`), and any nested define-expressions will appear at the top of **that** let, which is perfectly acceptable

Macros & special forms (*continued*)

letrec (*continued*)

There is something fundamentally unsavory about the last two macro-expansions for letrec:

- ▶ The letrec form has to do with defining locally-recursive procedures
- ▶ Recursion forms a cornerstone for **functional programming**
- ▶ In both cases, the macro-expanded code contains assignments, which are side-effects
- ▶ Side-effects are specifically excluded in pure functional programming
- 👉 It seems as if there is something about recursion that requires side effects, and this raises doubts about the entire functional programming agenda: **Is functional programming not powerful enough to express one of its most basic ideas?**

Macros & special forms (*continued*)

letrec (*continued*)

- ▶ The short answer is that **yes**, functional programming can express the idea of recursion in a way that is natural and native to functional programming, without any side-effects
- ▶ To understand this answer, we will need to
 - ▶ Study some **fixed-point theory**
 - ▶ Think harder about what recursion really means
- ▶ The full answer to this question is one of the most beautiful and exciting topics in the foundations of computer science and in programming languages theory
- ▶ For the time being, we move on to further topics that are necessary for you to work on your compiler projects
- ▶ We shall return to the topic in several weeks... Stay tuned!

Macros & special forms (*continued*)

cond

The cond form has the general form:

$$\begin{aligned} & (\text{cond} \langle rib_1 \rangle \\ & \quad \cdots \\ & \quad \langle rib_n \rangle) \end{aligned}$$

There are 3 kinds of cond-**ribs**:

- ① The common form $(\langle expr \rangle \langle expr_1 \rangle \cdots \langle expr_m \rangle)$, where $\langle expr \rangle$ is the **test-expression**: It is evaluated, and if not false, the rib is satisfied, all subsequent ribs are ignored, the corresponding implicit sequence is evaluated, and its final expression is returned.

Macros & special forms (*continued*)

cond (*continued*)

The cond form has the general form:

$$\begin{aligned} & (\text{cond } \langle rib_1 \rangle \\ & \quad \dots \\ & \quad \langle rib_n \rangle) \end{aligned}$$

There are 3 kinds of cond-**ribs**:

- ② The arrow form ($\langle expr \rangle \Rightarrow \langle expr_f \rangle$), where $\langle expr \rangle$ is evaluated:
If non-false, the rib is satisfied, and the return value is the application of $\langle expr_f \rangle$ to the value of $\langle expr \rangle$.

Macros & special forms (*continued*)

cond (*continued*)

The cond form has the general form:

$$\begin{aligned} & (\text{cond } \langle rib_1 \rangle \\ & \quad \dots \\ & \quad \langle rib_n \rangle) \end{aligned}$$

There are 3 kinds of cond-**ribs**:

- ③ The else-rib has the form $(\text{else } \langle expr_1 \rangle \dots \langle expr_m \rangle)$. It is satisfied immediately, and all subsequent ribs are ignored. The implicit sequence is evaluated, and the value of its final expression is returned.

Macros & special forms (*continued*)

cond (*continued*)

The cond-form macro-expands into nested if-expressions:

- ① The general form of the rib converts into an if-expression with a **condition** and an **explicit sequence** for the then-clause. The else-clause of the if-expression continues the expansion of the cond:

Macros & special forms (*continued*)

cond (*continued*)

The cond form macro-expands into nested if-expressions:

- ② The arrow-form of the rib converts into a let that captures the value of the test, and if not false, passes it onto the function. For test-expression $\langle expr \rangle$, and function-expression $\langle expr_f \rangle$, the following expansion would do:

```
(let ((value [[⟨expr⟩]])
      (f (lambda () [[⟨exprf⟩]]))
      (rest (lambda () [[⟨continue with cond-ribs⟩]])))
  (if value
      ((f) value)
      (rest)))
```

Macros & special forms (*continued*)

cond (*continued*)

The cond form macro-expands into nested if-expressions:

- ③ The else-form of the rib converts into a begin-expression, and subsequent ribs are ignored

An example of expanding cond

cond form

```
(cond ((zero? n) (f x) (g y))
      ((h? x) => (p q))
      (else (h x y) (g x)))
      ((q? y) (p x) (q y)))
```

Expanded form

```
(if (zero? n)
    (begin (f x) (g y))
    (let ((value (h? x))
          (f (lambda () (p q)))
          (rest (lambda () (begin (h x y) (g x)))))

        (if value
            ((f) value)
            (rest))))
```

Macros & special forms (*continued*)

The expansion of quasiquote-expressions

- ▶ Quasiquote-expressions are expanded **twice**:
 - ▶ Once in the reader, when the forms `⟨expr⟩, ,⟨expr⟩, ,@⟨expr⟩, and in fact '⟨expr⟩ too, are converted to their list forms: (quasiquote ⟨expr⟩), (unquote ⟨expr⟩), (unquote-splicing ⟨expr⟩), and (quote ⟨expr⟩), respectively
 - ▶ And a second time when quasiquote-expressions are expanded away in the tag-parser
 - 👉 This is what we're focusing on here!

Macros & special forms (*continued*)

The expansion of quasiquote-expressions (*cont*)

- ▶ Since R⁶RS, quasiquote-expressions can be nested, which means we can quasiquote quasiquoted expressions... This is complex, and not terribly useful, so we're not going to support it: We assume ordinary quasiquote-expressions **not** to include quasiquote-expressions
- ▶ We assume we have already received the form (quasiquote ⟨*sexpr*⟩), and are now going to reason about ⟨*sexpr*⟩

Macros & special forms (*continued*)

The expansion of quasiquote-expressions (cont)

- ① Upon receiving the expression (unquote $\langle sexpr \rangle$), we return $\langle sexpr \rangle$
- ② Upon receiving the expression (unquote-splicing $\langle sexpr \rangle$), we return the same expression preceded by a quote: (quote (unquote-splicing $\langle sexpr \rangle$))
- ③ Given either the empty list or a symbol, we wrap (quote ...) around it
- ④ Given a vector, we convert it to a list, expand the list using the quasiquote-expander, and convert it back to a vector using the built-in procedure list->vector

Macros & special forms (*continued*)

The expansion of quasiquote-expressions (*cont*)

This is the heart of the algorithm:

- ⑤ Given a pair, let A be the car, and let B be the cdr respectively.
 - ▶ If $A = (\text{unquote-splicing } \langle \text{sexpr} \rangle)$, then return $(\text{append } \langle \text{sexpr} \rangle \ [B])$
 - ▶ Otherwise, return $(\text{cons } [A] [B])$
- 👉 You should implement this quasiquote-expansion in your tag-parser!

Macros & special forms (*continued*)

The expansion of quasiquote-expressions (*cont*)

Some examples:

<i>sexpr</i>	<i>expansion</i>
,x	x
,@x	,@x
(a b)	(cons 'a (cons 'b '()))
(,a b)	(cons a (cons 'b '()))
(a ,b)	(cons 'a (cons b '()))
(,@a b)	(append a (cons 'b '()))
(a ,@b)	(cons 'a b)

Macros & special forms (*continued*)

The expansion of quasiquote-expressions (*cont*)

Some examples:

sexpr	expansion
(,a ,@b)	(cons a (append b '()))
(,@a ,@b)	(append a (append b '()))
(,@a . ,b)	(append a b)
(,a . ,b)	(cons a b)
(,a . ,@b)	(list a 'unquote-splicing 'b)
(((@a)))	(cons (cons (append a '()) '()) '())
#(a ,b c ,d)	(vector 'a b 'c d)

Macros & special forms (*continued*)

The expansion of quasiquote-expressions (*cont*)

- ▶ The expansion is a trivial recursive procedure
- ▶ The expansion is ~~hardly ever~~ isn't always the shortest
- ▶ The expansion can be made to compete with the best expanders, if you optimize the output using a pattern-based post-processor that iterates over the output until it reaches a fixed point

Macros & special forms (*continued*)

The expansion of quasiquote-expressions (cont)

- 👉 The real question is **what** to optimize:
 - ▶ Optimizing for time: The expanded form should call fewer functions
 - ▶ Optimizing for space: The expanded form should avoid, when possible, to recreate constants
 - ▶ **Example:** Consider ``(a ,b c d e f)`
 - ▶ Naïve output: `(cons 'a (cons b (cons 'c (cons 'd (cons 'e (cons 'f '()))))))`
 - ▶ Optimized for time: `(list 'a b 'c 'd 'e 'f)` (save on function calls)
 - ▶ Optimized for space: `(cons 'a (cons b '(c d e f)))` (save on pairs)

The Tag-Parser (*continued*)

How to write a tag-parser (*continued*)

- ▶ For macro-expanded forms

- ① Use pattern-matching to match over the **concrete syntax** of various syntactic forms
 - ▶ Consult the list of special forms that need to be macro-expanded
- ② Perform any additional testing necessary
- ③ Call macro-expanders, which are functions that are specific to each special form
 - ▶ For example, `expand_let`, `expand_cond`, etc
 - ▶ The macro-expanders should convert the concrete syntax of the special form to the concrete syntax of the expanded form
 - ▶ You may still use expandable special forms in the expansion
 - ▶ For example, expanding `letrec` may result in a `let-expression`, that will be expanded by the `expand_let` expander...

The Tag-Parser (*continued*)

How to write a tag-parser (*continued*)

- ▶ For macro-expanded forms (*continued*)
 - ④ Call the `tag_parser` recursively on what the macro-expanders return
 - ▶ For example, `tag_parse(expand_let e)`

Roadmap

- ▶ Expressions in Scheme
 - ✓ The `expr` datatype
 - ✓ The Tag-Parser
 - ✓ Macros & special forms
 - ▶ Lexical hygiene

Lexical hygiene

We had to deal with lexical hygiene issues in some of our macro-expansions. So at the end of the chapter on macro-expansion, this is worth recapitulating:

- ▶ Hygiene problems occur when variables that are introduced during macro-expansion are visible to user-code
 - ▶ When user-code refers to such variables, this is **always** an error
 - ▶ We refer to such an error **variable-name captures**
- ▶ We can avoid variable-name capture by
 - ▶ Using gensym to generate an **uninterned symbol**
 - ▶ By finding in the expansion a “safe” variable-name to use
 - ▶ By changing the macro-expansion so that administrative variables are invisible to user-code

Roadmap

- ✓ Expressions in Scheme
 - ✓ The `expr` datatype
 - ✓ The Tag-Parser
 - ✓ Macros & special forms
 - ✓ Lexical hygiene

Further reading

-  *The Structure and Interpretation of Computer Programs*, by Abelson and Sussman: A classical introductory text on programming, abstraction, interpretation. This is the textbook used for a legendary course at MIT by the same title.
-  *Scheme and the Art of Programming*, by Springer and Friedman: One of the best books on Scheme and functional programming ever written. Very thorough and systematic.

Roadmap

- ▶ Scope
- ▶ The lexical environment
- ▶ Boxing

Scope

- ▶ Scope has to do with **names** as **bindings**:
 - ▶ Variables
 - ▶ Functions
 - ▶ Methods
 - ▶ Modules
 - ▶ Packages
 - ▶ Namespaces
- etc.
- ▶ Scope has to do with where, and under what circumstances is a binding valid
 - 👉 Where, and under what circumstances is a **name** visible/accessible

Scope (*continued*)

Here are the kinds of questions we might ask:

- ▶ Point at a specific variable declaration in your code: What parts of your code can access this variable?
⚠ Not another variable by the same name!
- ▶ Point at a variable occurrence anywhere in the code: Where is it defined?

If you grew up on 1-2 programming languages, you might wonder whether these are serious questions: After all, how many different ways can there be to relate names and bindings?

- ▶ Well, we're going to see two very different ways: **Dynamic Scope**, and **Lexical (aka Static) Scope**

Scope (*continued*)

Dynamic Scope

- ▶ When you call a function, the function may take arguments. The values of these arguments are bound to special variables we call **function parameters** or just **parameters**.
- ▶ As we mentioned previously, the term **dynamic** means that something is happening or known or computed during run-time. It's another way of saying that it is not or cannot happen, be known, or be computed at compile-time.
- ▶ As we examine dynamic scope, you should ask yourself what are the dynamic properties of this particular scope, and what do they imply about the experience of programming in languages that make use of dynamic scope...

Scope (*continued*)

Dynamic Scope

Dynamic scope begins with a very natural intuition about variable name bindings:

- ▶ Upon a function call, **push** the bindings, i.e., the associations of names & values onto the run-time stack
- ▶ When evaluating/executing the body of the function, look up the values of any variable names on the run-time stack
- ▶ Upon returning from the function call, pop the parameter-value bindings off of the stack, so the top of the stack is where it was just before the call
- ⚠ Because this intuition is so natural, it took a couple of decades before people realized just how problematic it really is!

Scope (*continued*)

Dynamic Scope (*continued*)

Let us see what is implied by such a model

- ▶ Consider the following code, written in some dialect of Scheme that uses dynamic scope:

```
> (define foo
  (lambda (n)
    (if (zero? n)
        k
        (+ n (foo (- n 1))))))
> (let ((k 0))
  (foo 1000000))
500000500000
```

Scope (*continued*)

Dynamic Scope (*continued*)

How does the code `(let ((k 0)) (foo 1000000))` evaluate?

- ▶ The binding `k:0` is pushed onto the stack
- ▶ The binding `n:1000000` is pushed onto the stack
- ▶ The binding `n:999999` is pushed onto the stack
- ▶ The binding `n:999998` is pushed onto the stack
- ▶ ...
- ▶ The binding `n:0` is pushed onto the stack
- ▶ The value of `k` is searched from the **top** of the stack...
 - ▶ The **access time** for **variable lookup** is dynamic, and a function of run-time data
 - ▶ The **access time** is vastly different for different variables

Scope (*continued*)

Dynamic Scope (*continued*)

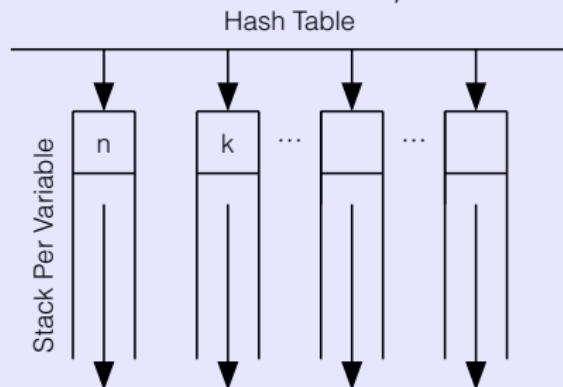
- ▶ This behaviour is very different from what you see in C/C++ or Java or Scheme...
- ▶ This is called the **deep binding** implementation of **dynamic scope**
 - ▶ It's not the ideas here that are very "deep", but rather the stack!
 - ▶ Deep binding was used to implement LISP way back in 1959, when LISP was invented
 - ▶ The terrible performance that resulted from deep binding gave LISP a bad reputation it had a hard time shaking off decades later...
 - ▶ Because the problem was understood in terms of **efficiency**, a solution was soon found that was sufficiently efficient: The **shallow binding** implementation of dynamic scope

Scope (*continued*)

Dynamic Scope (*continued*)

Shallow binding is a different implementation of dynamic scope:

- ▶ Rather than a single stack onto which bindings are pushed, a different data structure is used:
 - ▶ A hash-table of stacks, with one stack per variable name:



- ▶ The current value of a variable is **always** at the top of the stack associated with that variable name, so **access time is constant**

Scope (*continued*)

Dynamic Scope (*continued*)

- ▶ Variable-lookup under shallow bindings is much faster than under deep binding, although it's still not as efficient as the situation in C/C++/Java, etc.
- ▶ The performance of dynamically-scoped programming languages was no longer atrocious
- ▶ Shallow binding replaced deep binding as the implementation technique for dynamic scope

Dynamic Scope (*continued*)

The problem is that there's still something quite dynamic about shallow binding

- ▶ that has nothing to do with efficiency
- ▶ that poses a more significant problem than performance

Scope (*continued*)

Dynamic Scope (*continued*)

Imagine the following implementation of the `map` function:

```
(define map
  (lambda (f s)
    (if (null? s)
        '()
        (let ((x (f (car s))))
          (cons x (map f (cdr s)))))))
```

- ▶ **Problem:** Find a function `f` and a list `s`, such that:
 - ▶ `s` should be a finite list, without any circularities
 - ▶ `f` should terminate on all inputs

And yet, despite the reasonableness of `f` & `s`, `(map f s)` should go into an infinite loop...

Scope (*continued*)

Dynamic Scope (*continued*)

- ▶ Let f be (lambda (a) (set! s '(la la la la)) a)
- ▶ Let s be '(mary had a little lambda!)
- ▶ And the following code enters into an infinite loop:

```
> (map (lambda (a)
                (set! s '(la la la la))
                a)
         '(mary had a little lambda!))
```

Scope (*continued*)

The definition of map

```
(define map
  (lambda (f s)
    (if (null? s)
        '()
        (let ((x (f (car s))))
          (cons x (map f (cdr s)))))))
```

Enter an infinite loop...

```
> (map (lambda (a)
              (set! s '(la la la la))
              a)
         'mary had a little lambda!))
```

Scope (*continued*)

Dynamic Scope (*continued*)

So here's the real problem with dynamic scope, regardless of how it is implemented (whether deep binding or shallow binding):

- ▶ The variable-bindings are **visible across procedure boundaries**:
 - ▶ This means that if a procedure `f` has a **local** variable `x`, and `f` calls `g`, then `g` can access `x`, and any procedure called by `g` can access `x`
 - ▶ This means that as long as your code calls functions written elsewhere, these functions may access any of your local variables for read/write
 - ▶ This is a form of **code-injection** that cannot be sanitized away!

Scope (*continued*)

Dynamic Scope (*continued*)

So here's the real problem with dynamic scope, regardless of how it is implemented (whether deep binding or shallow binding):

- ▶ Or from the perspective of the callee:
 - ▶ If the callee relies on some **global** function, and some code along the call-chain defined a **local** variable by the same name, the callee is unable to access the global function, because it is overridden by a “closer”, local binding

Scope (*continued*)

Dynamic Scope (*continued*)

So here's the real problem with dynamic scope, regardless of how it is implemented (whether deep binding or shallow binding):

- ▶ Under dynamic scope, the whole concept of **correctness** of the code takes on a weird, unexpected meaning:
 - ▶ Correctness is no longer a feature of the source code, but of the specific circumstance in which it is used:
 - ▶ Under dynamic scope, the correctness of the code becomes a feature of a specific instance of running the code: It becomes situational —
 - ▶ One way of using the code will be correct, and yield correct results
 - ▶ Another way of using the code will be incorrect, and yield incorrect results
 - ▶ We cannot **prove correctness** and be assured the program will act correctly under all circumstances

Scope (*continued*)

Dynamic Scope (*continued*)

So here's the real problem with dynamic scope, regardless of how it is implemented (whether deep binding or shallow binding):

- ▶ Dynamic scope loses something that today we generally take for granted
 - ▶ There's a name for this: It's called **referential transparency**
 - ▶ Referential transparency means that we may replace an expression with its value in any context in which the original expression appears, without changing the meaning/behaviour of the program
- 👉 Dynamic scope creates complex relations between different parts of a program, that are difficult to trace or predict, resulting in bugs that are difficult to find and remove

Scope (*continued*)

Dynamic Scope (*continued*)

- ▶ The main objection to dynamic scope isn't efficiency:
 - ▶ The issue of efficiency merely underscored the problems with a particular implementation of dynamic scope, namely **deep binding**
- ▶ The main objection to dynamic scope is from a software-engineering perspective
 - ▶ Dynamically-scoped code is difficult to understand, and prone to errors that are difficult to trace
 - ▶ Loss of **referential transparency**

Scope (*continued*)

Dynamic Scope (*continued*)

- ▶ Since the late 1950's and up to the 1970's, dynamic scope was popular in many dynamic programming languages: Early dialects of LISP, early dialects of Smalltalk, Snobol, Logo, most dialects of APL, Mathematica, Macsyma, bash, and many other
- ▶ Dynamic scope is **optional** in some modern programming languages

Scope (*continued*)

Dynamic Scope (*continued*)

- ▶ Dynamic scope pretty much disappeared as a mechanism for managing the scope of variables
- ▶ Newly-created languages do not use dynamic scope for variables
- ▶ Dynamic scope has nevertheless not retired:
 - 👉 It is now used to manage the scope of exception-handler in most programming languages
 - ▶ Unlike the situation the management of scope of variables, the scope of exception-handlers **mandates** deep binding

Dynamic Scope (*continued*)

Exception handling

- ▶ Exception handling is a control mechanism that provides for a **non-local exit-point** where exceptional circumstances are handled
 - ▶ Exceptional circumstances include error conditions, the unavailability of resources, and other problems
 - ▶ Some exceptional circumstances warrant the termination of the program, in which case, action is taken to stabilize the computing environment: Closing files & network connections, de-allocating resources, etc.
 - ▶ Some exceptional circumstances can be handled by the program, after which execution proceeds as usual
 - ▶ Non-local exit-points are placed in the program where action is taken in response to exceptional circumstances

Exception handling (*continued*)

Non-locality

The significance of non-locality of the exception handler is that errors may be handled at very different places from where they occur:

- ▶ Consider an SQL query at one point in the program:
 - ▶ The query fails because of the exceptional circumstance of a failure of a DBMS connection
 - ▶ The code that sends the query might know nothing about the DBMS connection itself
 - ▶ The code that handles the exceptional situation might know nothing about the SQL query
- ▶ So the exception handler and the SQL query appear in different places of the program
 - ▶ Each having its own state
 - ▶ Each having its own concerns

Exception handling (*continued*)

Non-locality (*continued*)

There may be **more than a single handler** for an exception:

- ▶ A later handler may override an earlier handler, or
- ▶ A later handler may provide some handling of the situation, and then delegate control to an earlier handler, to continue handling
 - ▶ A later handler could save information on the query that had failed, so that it could be re-issued later
 - ▶ An earlier handler could add a line to the system log
 - ▶ Yet an earlier handler could attempt to restore the DBMS connection or try alternative servers elsewhere

And all these event handlers would each do something, and then raise the same exception, triggering a **cascading** sequence of events

Dynamic Scope (*continued*)

Exception handling (*continued*)

- ▶ From the description of how event handling is used, it follows that exception handlers must be arranged in a linear, LIFO structure: A **stack**
- ▶ Handlers must be picked in a LIFO manner, by their name/type
- ▶ When an exception handler is reached, all exception handlers **for any exception** that were placed on the stack **after** this handler are removed
- ▶ This implies **deep-binding dynamic scope**

Dynamic Scope (*continued*)

Exception handling (*continued*)

- ▶ If a program raises exceptions often, the handlers of which are far from the top of the stack, a heavy performance penalty will result
- ⚠ The situation in C++, and other languages that do not use **dynamic memory management**, is actually worse than in languages that do, because when an exception is raised, **all objects allocated on the stack after the target handler** must be de-allocated (as on function/method return).

Scope (*continued*)

Question

Code compiled to use dynamic scoping:

- 👎 Is always less efficient than code compiled to use lexical scoping
- 👎 Always uses more memory than code compiled to use lexical scoping
- 👎 Always behaves the same as code compiled to use lexical scoping
- 👎 Never contains pre-computed, static memory addresses
- 👍 Breaks our notion of correctness

Scope (*continued*)

Lexical Scope

- ▶ With **dynamic scope**, when a program calls some function or refers to some variable, and this function or variable is not a parameter of the function, it is often not possible to know **where** the function or variable is defined.
- ▶ The idea behind **lexical scope** (*aka static scope*) is that such questions should be answerable at compile-time, and known by the compiler
 - ▶ Static scope means that the scope of a name can be determined before the program is executed
 - ▶ Lexical scope means that the scope of a name is a lexical property of the code, namely, a property of the **syntax**
 - ▶ Both lexical scope & static scope mean exactly the same thing, seen from two different angles

Scope (*continued*)

Lexical Scope

- ▶ That we can know where a name is defined means that we can know its **address**
- ▶ The absolute, physical (64-bit) address is a **run-time artifact**
- ▶ What can be known statically is the **lexical address** of any name
 - ▶ The lexical address **abstracts** over the physical address
 - ▶ The lexical address can be used **to generate efficient assembly code**
 - ▶ The lexical address is **relative**

Lexical Scope

We recognize three kinds of variables in Scheme (and all lexically-scoped languages do something similar):

- ▶ Parameters
- ▶ Bound variables
- ▶ Free variables

Lexical Scope

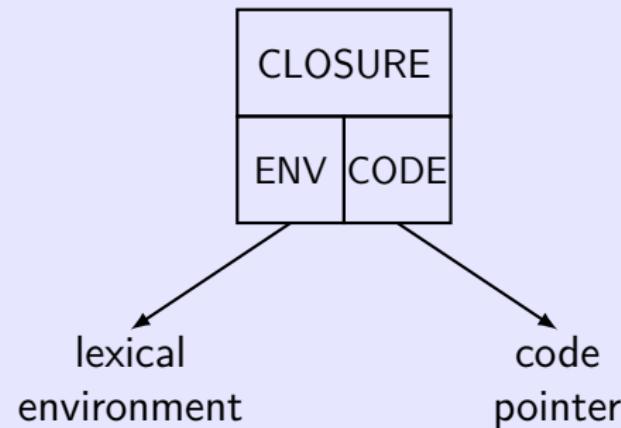
Parameters

- ▶ Parameters are the variables procedures use to access their arguments
- ▶ **Example:** The parameters of the procedure `(lambda (a b) (+ (* a a) (* b b)))` are a, b
 - ▶ The variables +, * are not parameters!
- ▶ The lexical address of a parameter is its 0-based index in the parameter-list of the lambda-expression in which it is defined
 - ▶ **Example:** In the above code, a would be *parameter-0*, and b would be *parameter-1*

Lexical Scope (*continued*)

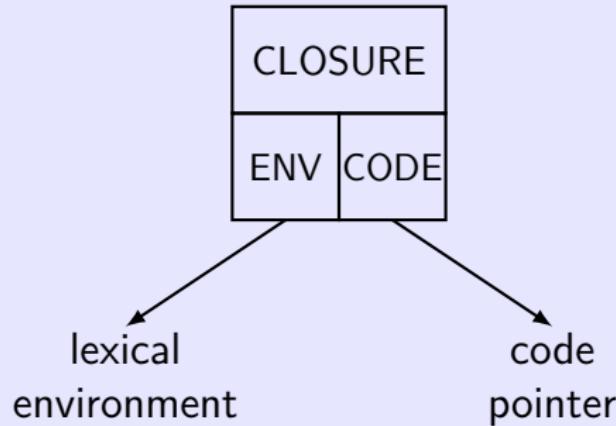
Bound variables

- ▶ The value of a lambda-expression is a **closure**
- ▶ A closure is a tagged data-structure that *encloses* a **lexical environment** and a **code pointer**:



Lexical Scope (*continued*)

Bound variables



- ▶ The **lexical environment** is similar to the **state** of an object
- ▶ The **code pointer** is similar to the address of a **method**
- 👉 Closures are like objects with a single method called **apply**

Example: Closure, bound variable

```
> (define count
  (let ((n 0))
    (lambda ()
      (set! n (+ n 1))
      n)))
> (count)
1
> (count)
2
```

- ▶ The variable `n` is in the **lexical environment** of `count`
 - ▶ `n` is a **bound variable** within the body of `count`
- ▶ The procedure `count` takes no parameters!

Lexical Scope

Bound variables

```
(lambda (a)
  (a (lambda (b)
    (a b (lambda (c)
      (a b c))))))
```

- ▶ Parameter occurrences \circledast (on the stack)
- ▶ Bound variable occurrences \circledast (in the lexical environment, stored in the heap)

Lexical Scope

The lexical environment

- ▶ We see that as the very same variable is accessed from within inner lambda-expressions, what used to be a parameter becomes a bound variable, and what used to reside on the stack now resides in the heap
- ▶ The process of copying values from the stack onto the heap **extends** the lexical environment of the outer lambda-expression, and the extended environment becomes the lexical environment of the inner lambda-expression
- ▶ The extended lexical environment is the environment used in the closure of the **inner** lambda-expression
- ▶ The maximal size of the lexical environment is the number of nested lambda-expressions minus 1

Lexical Scope

The lexical environment (*continued*)

- ▶ **Question:** What is the size of the largest lexical environment generated by the following code:

```
(define foo
  (lambda (q a z)
    (lambda (w s x)
      (lambda (e d c)
        (lambda (r f v)
          (lambda (t g b)
            (+ q a z w s x e d c r f v t g b)))))))
```

☞ **Answer:** 4

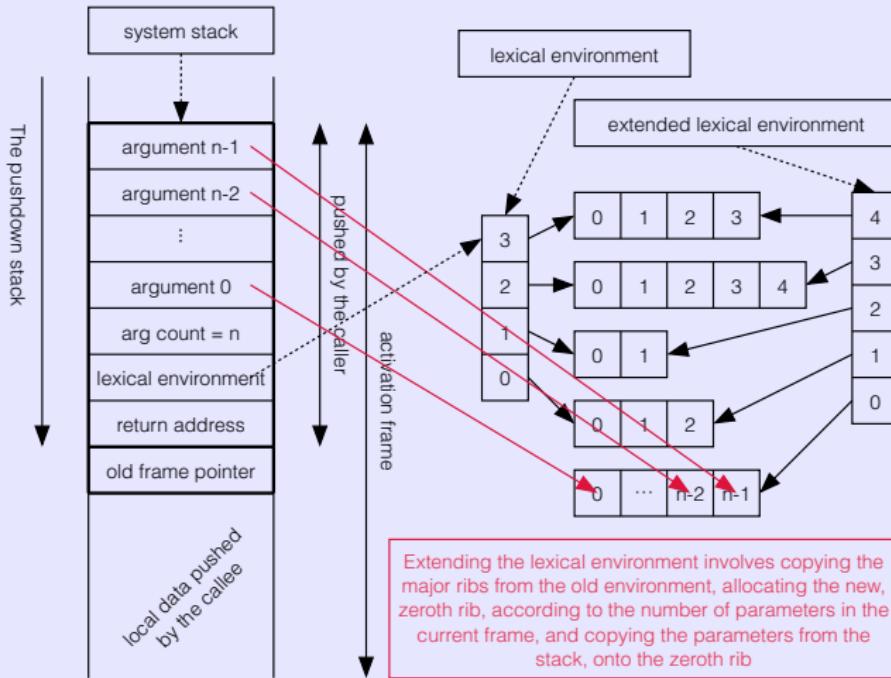
Lexical Scope

The lexical environment (*continued*)

- ▶ The **lexical addressing** for bound variables is tightly coupled with the representation of the environment
 - ▶ We can discuss lexical addressing abstractly, without committing to any specific data structures with which to represent the environment
 - ▶ To implement the lexical environment efficiently, we must delve into the nitty-gritty details of how and where things are represented, so we can **compute** the lexical address of a variable
- ▶ The diagram on the next slide describes
 - ▶ The structure of the system stack
 - ▶ The structure of the activation frame
 - ▶ The lexical environment: its structure & location on the stack
 - ▶ The extended environment: its structure & relation to the environment it extends

Lexical Scope

The lexical environment (*continued*)



Lexical Scope

The lexical environment

- 👉 You should commit the above diagram to memory, and understand each and every component in it!
- 👉 Our implementation of lexical scope is very different from the one you learned in PPL (Principles of Programming Language); We shall discuss both implementations later on, but in the meantime, know that our implementation is designed for efficiency
- 👉 Our implementation of lexical scope is the same as that of most functional and **classless** object-oriented programming languages, and *mutatis mutandis* similar to that of most **class-based** object-oriented programming languages

Lexical Scope

The lexical environment (*continued*)

- ▶ The lexical environment is a vector of vectors
 - ▶ This is not the same as a two-dimensional array: The inner vectors needn't be the same size
- ▶ The **lexical address of a bound variable** consists of two integers, indexing the major and minor vectors, respectively
- ▶ **Example:** What is the address of x in:

```
(lambda (x)
  (lambda (y z)
    (lambda (t)
      x)))
```

- ▶ **Answer:** *bound-1-0.*

Lexical Scope

The lexical environment (*continued*)

- ▶ Example: What are the addresses of x in:

```
(lambda (x)
  (x_ (lambda (y)
    (x_ y (lambda (z)
      (x_ y z))))))
```

► x

- ▶ parameter-0
- ▶ bound-0-0
- ▶ bound-1-0

Lexical Scope

The lexical environment (*continued*)

- ▶ Example: What are the addresses of *y* & *z* in:

```
(lambda (x)
  (x (lambda (y)
    (x y (lambda (z)
      (x y z))))))
```

The diagram illustrates the lexical environment for variables *y* and *z*. It shows the nested lambda expressions and the binding of variables to parameters or other variables.

- ▶ *y*
 - ▶ *parameter-0* (points to the first *y* in the innermost lambda)
 - ▶ *bound-0-0* (points to the second *y* in the innermost lambda)
- ▶ *z*
 - ▶ *parameter-0* (points to the *z* in the innermost lambda)

Scope (*continued*)

Lexical Scope (*continued*)

Summing up the above example:

- ▶ Notice that **different** variables can have the **same** lexical address
- ▶ Notice that the **same** variable can have **different** lexical addresses
- ▶ The lexical address is **relative**
 - ▶ Relative to the **lexical environment**
 - ▶ Relative to the **frame-pointer**

Scope (*continued*)

Lexical Scope (*continued*)

- ▶ An **empty lexical environment** is one that contains no variables
- ▶ The lexical environment is constructed **incrementally**, by **extending** an existing environment, starting all the way from the empty lexical environment



When is the lexical environment extended?

☞ **Answer:** There are two possibilities:

- ▶ During application (**PPL course**)
 - ▶ As part of the creation of a new closure (**compiler-construction course**)
-
- ▶ The **lexical address of free variables** is the address of their values in the **top-level**
 - ▶ The **top-level** & its structure shall be discussed later on, as we approach the topic of **code generation**

Lexical Scope (*continued*)

Extending the lexical environment

- ▶ Customarily all LISP/Scheme interpreters (ever since the first one in 1959) extend the environment during application: Recall that your Scheme interpreter —
 - ▶ did not use a **stack**
 - ▶ did not distinguish between parameters and bound variables
 - ▶ all variables were implemented in an "environment"
 - ▶ Including **free variables**, which made up the **initial environment**
 - ▶ was not very efficient 😊
- ▶ Customarily all **stack-based** LISP/Scheme compilers extend the environment during the creation of **closures**:
 - ▶ Parameters are distinguished from bound variables
 - ▶ Parameters live on the stack
 - ▶ Bound variables live in lexical environments
 - ▶ Free variables live in hash tables

Scope (*continued*)

Lexical Scope (*continued*)

What did you do in your PPL interpreters:

- ▶ You **extended** the environment on **applications**
- ▶ You **copied** the address of the environment on **closure-creation**

This means that

- ▶ Constructing new closures was very cheap
 - ▶ Applications were expensive
-  Which is more common?

Scope (*continued*)

Lexical Scope (*continued*)

What we shall do in our compilers:

- ▶ We **extend** the environment on **closure-creation**
- ▶ We **push** [the address of] the environment onto the stack on application

This means that

- ▶ Constructing new closures is expensive
- ▶ Applications are cheap



Which is more common?

Scope (*continued*)

Lexical Scope (*continued*)

In this course:

- ▶ By the **lexical environment** we mean **only** the vector of vectors that implements bound variables, and that is **part of the closure data-structure**
 - ▶ Lexical environments **do not maintain**:
 - ▶ Parameters — Those live on the stack
 - ▶ Global variables — Those live in the top-level environment, which is a hash-table, rather than a lexical-environment
- 👉 In PPL, the global environment was also called the **initial environment**, and all lexical environments were extensions of this initial environment
- 👉 In the **compiler-construction course**, unlike in **PPL**, we do not build lexical environments **on top** of the global, top-level environment

Scope (*continued*)

Lexical Scope (*continued*)

In this course:

- ▶ Variable **names** are not important in the compiler:
 - ▶ Names are used for writing code and for debugging the compiler
 - ▶ Names are **compiled away** into **lexical addresses**, which are symbolic representations for **locations**:
 - ▶ Parameters live on the stack
 - ▶ Bound variables live in lexical environments
 - ▶ Free variables live in the top-level
- 👉 By assigning to each class of variables its own access-mechanism, supported by appropriate data structures and addressing schemas, we achieve faster access-time!

Scope (*continued*)

Lexical Scope (*continued*)

Extending the lexical environment during the creation of closures isn't very inefficient either, however:

- ▶ The average number of nested λ -expressions is 1.2 (the size of the lexical environment is around 0.2). This means that
 - ▶ Most people don't write code with nested lambda-expressions
 - ▶ Most people don't rely on nested procedures for abstraction
- 👉 Moving the cost of extending the lexical environment to the creation of new closures pays off, because people create far less closures than they apply!
- ▶ The same is true in OOPs
 - 👉 How often have you seen code that uses nested classes, and how deep was the nesting??

Scope (*continued*)

What happens during closure creation

- ① A new environment is allocated
 - ▶ The size of the new env is $1 + \text{the size of the old env}$
- ② The addresses of the minor vectors are copied from the old env to the ext env
 - ▶ $\text{ExtEnv}_{j+1} \leftarrow \text{Env}_j, j = 0, 1, \dots, |\text{Env}|$
- ③ A new rib is allocated for ExtEnv_0 :
 - ▶ $\text{ExtEnv}_0[j] \leftarrow \text{Param}_j, j = 0, 1, \dots, \text{ParamCount}$

👉 The env is now extended!
- ④ A closure data-structure is allocated
- ⑤ The closure is set to point to the extended lexical environment, and to the code

Scope (*continued*)

Lexical Scope (*continued*)

To acquire some intuition as to why it is more efficient to extend the environment during the creation of closures rather than during function application, just think about the analogous situation in OOPLs:

- ▶ The creation of closures is very similar to the **creation of objects**
- ▶ The application of closures is very similar to the **application of methods**

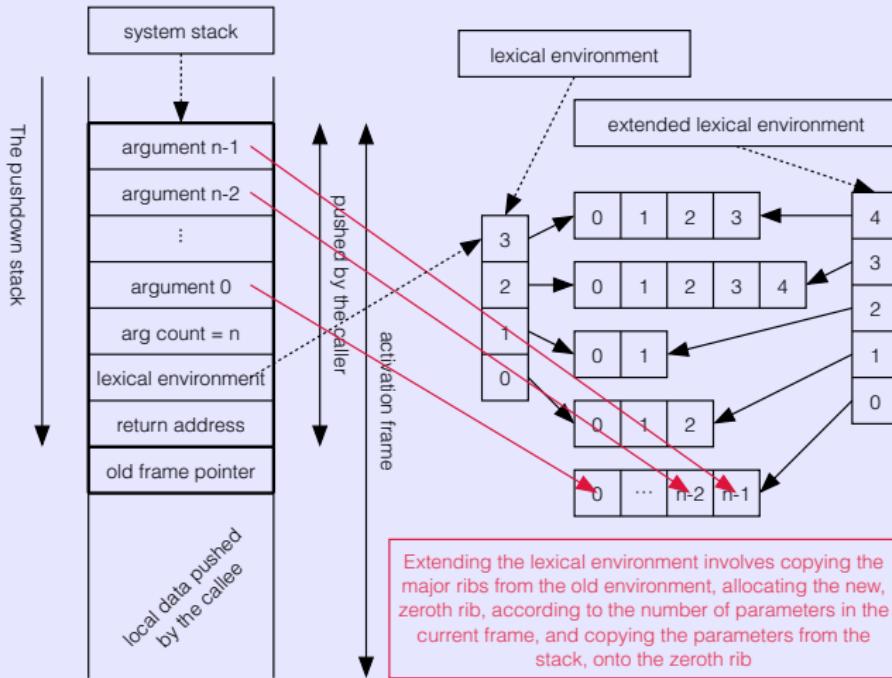
 Which implementation of an OOPL would you rather have:

- ▶ One that makes it cheap to create objects, and expensive to call their methods
- ▶ One that makes it expensive to create objects, and cheap to call their methods

 Which does your code do more: Create objects or call methods?

Scope (*continued*)

Lexical Scope (*continued*)



Scope (*continued*)

What happens during procedure calls

- ▶ Before the call
 - ① The arguments are evaluated and pushed from last to first
 - ② The number of arguments are pushed
 - ▶ This supports procedures with an indefinite number of arguments
 - ③ The procedure-expression is evaluated
 - ▶ Verify that the value is indeed a closure!
 - ④ The lexical environment of the closure is pushed
 - ⑤ Call the code-pointer of the closure
 - ▶ Calls in tail-position are handled differently! More on this later...
- ▶ After the call
 - ① The stack is restored to the state before the call
 - ▶ Again, tail-calls make this tricky! More on this later...

Chapter 4

Roadmap

- ✓ Scope
- ▶ The lexical environment
- ▶ Boxing

Lexical Scope (*continued*)

Sharing the lexical environment

Closures can **share** code-pointers, environments, and also **parts** of the lexical environment:

- ▶ Closures with different environments and the same code-pointer
- ▶ Closures with the same environment and different code-pointers
- ▶ Closures with partly-shared environments and different code-pointers

Lexical Scope (*continued*)

Sharing the lexical environment

Closures with **different environments** and the **same code-pointer**:

```
(define ^count
  (lambda ()
    (let ((n 0))
      (lambda ()
        (set! n (+ n 1))
        n))))
(define count-1 (^count))
(define count-2 (^count))
```

Lexical Scope (*continued*)

Sharing the lexical environment

Closures with **different environments** and the **same code-pointer**:

```
> (count-1)  
1  
> (count-1)  
2  
> (count-1)  
3  
> (count-2)  
1  
> (count-2)  
2  
> (count-1)  
4
```

Lexical Scope (*continued*)

Sharing the lexical environment

Closures with the **same environment** and **different code-pointers**:

```
(define count #f)
(define reset #f)
(let ((n 0))
  (set! count
        (lambda ()
          (set! n (+ 1 n))
          n)))
  (set! reset
        (lambda ()
          (set! n 0))))
```

Lexical Scope (*continued*)

Sharing the lexical environment

Closures with the **same environment** and **different code-pointers**:

```
> (count)  
1  
> (count)  
2  
> (count)  
3  
> (reset)  
> (count)  
1  
> (count)  
2
```

Lexical Scope (*continued*)

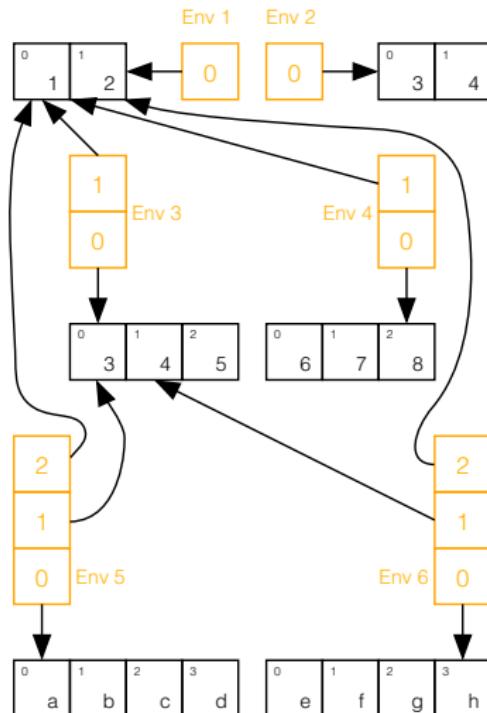
Sharing the lexical environment

Closures **sharing parts of their lexical environments** in a **tree-like** fashion:

```
(define f
  (lambda (q a)
    (lambda (z w s)
      (lambda (x e d c)
        ...
        ))))
(define f12 (f 1 2))
(define f34 (f 3 4))
(define f12345 (f12 3 4 5))
(define f12678 (f12 6 7 8))
(define f12345abcd (f12345 'a 'b 'c 'd))
(define f12345efgh (f12345 'e 'f 'g 'h))
```

Lexical Scope (*continued*)

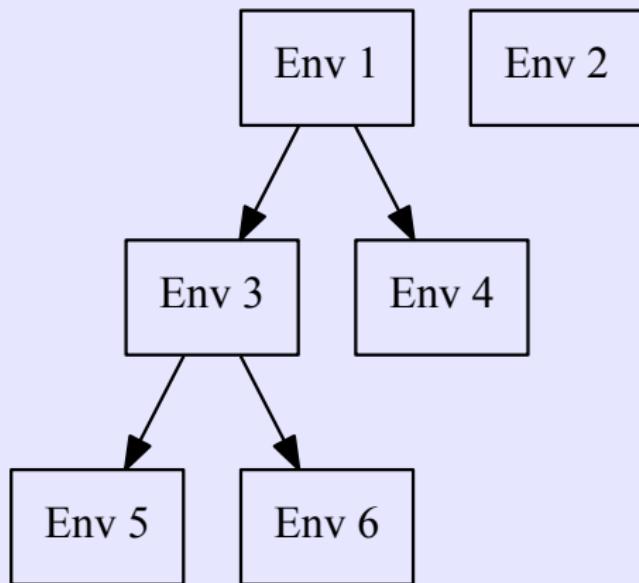
Closures sharing parts of their lexical environments in a tree-like fashion:



Lexical Scope (*continued*)

Sharing the lexical environment

Closures sharing parts of their lexical environments in a tree-like fashion: What parts of the code are shared:



Scope (*continued*)

Lexical addressing in Comp vs PPL

- ▶ In the **compiler-construction course**, we extend the lexical environment during the creation of new closures, and push it during application
 - ▶ As a result, we distinguish between **parameters & bound variables**
- ▶ In the **PPL course**, we extend the lexical environment during application, and save it during the creation of new closures
 - ▶ As a result, we **do not** distinguish between **parameters & bound variables**
 - ▶ What is referred to as parameters in the **compilers course** are simply the **zeroth rib** in the lexical environment in the **PPL course**
- ▶ This difference has an effect on the **computation of lexical addresses**

Scope (*continued*)

Example:

Lex Addr for Comp Const

```
(lambda (x)
  (xp0 (lambda (y)
    (xb00 yp0 (lambda (z)
      (xb10 yb00 zp0))))))
```

Lex Addr for PPL

```
(lambda (x)
  (xb00 (lambda (y)
    (xb10 yb00 (lambda (z)
      (xb20 yb10 zb00))))))
```

Scope (*continued*)

Distinguishing scope

Suppose we're working on some system for which you have no manual... **How can we tell the scope?**

- ▶ The trick is to refer to some **free variable** from within a procedure
- ▶ Define the variable **globally** with one value
- ▶ To define another variable by the same name **locally** within a second procedure, with a different value
- ▶ Call the second procedure
 - ▶ If we get the **local value**, we're running under **dynamic scope**
 - ▶ If we get the **global value**, we're running under **lexical scope**

Scope (*continued*)

Distinguishing scope

```
(define *free-variable* 'lexical-scope)
(define return-free-variable
  (lambda () *free-variable*))
(define get-scope
  (lambda ()
    (let ((*free-variable* 'dynamic-scope))
      (return-free-variable))))
```

Call the procedure `get-scope` to find whether you're running under lexical scope or dynamic scope...

Chapter 4

Roadmap

- ✓ Scope
- The lexical environment
- Boxing

The OOP world

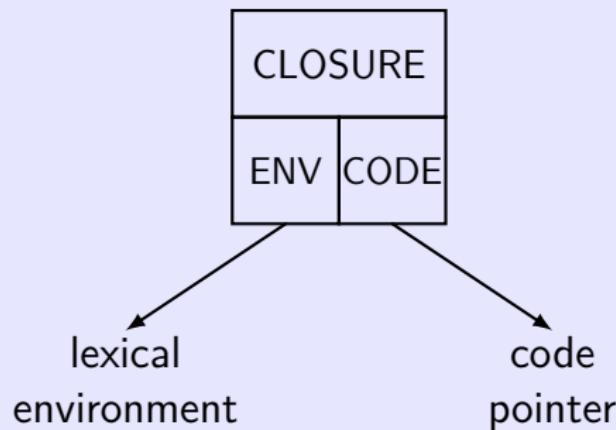
What we learned about lexical scope is not unique to LISP/Scheme or even to [quasi]-functional programming languages

- ▶ All modern programming languages use lexical scope
- ▶ Any language that supports higher-order procedures supports closures and the sharing parts of environments
- ▶ Compiling **methods** is very similar to compiling **closures**
- ▶ The run-time behaviour of **methods** is very similar to that of **closures**
- ▶ Objects are very similar to lexical environments
- 👉 We would like to explore these similarities
 - ▶ Learn how to compile OOPLs
 - ▶ Leverage our intuition about OOPLs
 - ▶ Leverage our intuition about functional programming

The OOP world (*continued*)

Closures & Objects

- ▶ A closure is a data structure that combines a lexical environment & some code:



- ▶ What if we wanted to have more than one code-pointer?

The OOP world (*continued*)

Closures & Objects (*continued*)

Closures with more than one code pointer:

- ① Have a function return a list/vector of functions
- ② Have a function take a function of several arguments and apply it to several functions

The OOP world (*continued*)

Simple, Object-Oriented-like count/reset

- ① Here's how to define it

```
(define make-counter
  (lambda ()
    (let ((n 0))
      (let ((count (lambda () (set! n (+ n 1)) n))
            (reset (lambda () (set! n 0))))
        (list count reset))))
```

The OOP world (*continued*)

Simple, Object-Oriented-like count/reset

- ① Here's how to use it

```
> (define c1 #f)
> (define c2 #f)
> (define r1 #f)
> (define r2 #f)
> (apply
  (lambda (_c1 _r1) (set! c1 _c1) (set! r1 _r1))
  (make-counter))
> (apply
  (lambda (_c2 _r2) (set! c2 _c2) (set! r2 _r2))
  (make-counter))
```

The OOP world (*continued*)

Simple, Object-Oriented-like count/reset

- ① Here's how to use it

```
> (c1)
```

```
1
```

```
> (c1)
```

```
2
```

```
> (c2)
```

```
1
```

```
> (c2)
```

```
2
```

```
> (r1)
```

```
> (c2)
```

```
3
```

```
> (c1)
```

```
1
```

The OOP world (*continued*)

Simple, Object-Oriented-like count/reset

- ② Here's how to define it

```
(define make-counter
  (lambda ()
    (let ((n 0))
      (let ((count
              (lambda ()
                (set! n (+ n 1)) n)))
        (reset (lambda () (set! n 0))))
      (lambda (u)
        (u count reset))))))
```

The OOP world (*continued*)

Simple, Object-Oriented-like count/reset

② Here's how to use it

```
> ((make-counter)
  (lambda (_c1 _r1) (set! c1 _c1) (set! r1 _r1)))
> ((make-counter)
  (lambda (_c2 _r2) (set! c2 _c2) (set! r2 _r2)))
```

The OOP world (*continued*)

Simple, Object-Oriented-like count/reset

- ② Here's how to use it

```
> (c1)
```

```
1
```

```
> (c1)
```

```
2
```

```
> (c2)
```

```
1
```

```
> (c2)
```

```
2
```

```
> (r1)
```

```
> (c2)
```

```
3
```

```
> (c1)
```

```
1
```

The OOP world (*continued*)

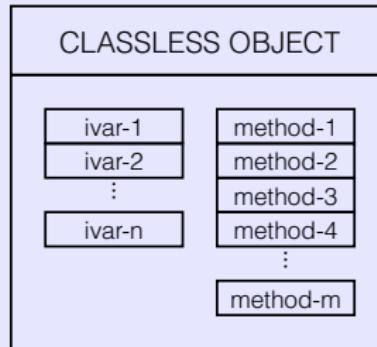
Simple, Object-Oriented-like count/reset

- ▶ As you can see, the two implementations behave identically
- ▶ We can associate any number of “methods” with the same lexical environment
- ▶ These “methods” are used similarly to how methods are used in OOPLs

The OOP world (*continued*)

Closures & Objects (*continued*)

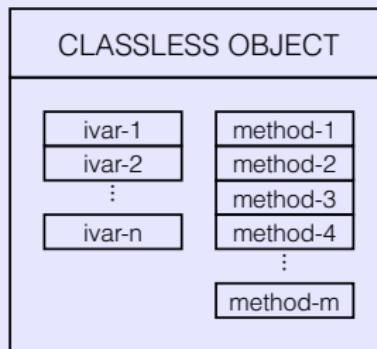
- ▶ A classless, prototype-based object (like in older versions of Javascript) is a structure very similar to a closure:
 - ▶ It contains **state**, in the form of **instance variables**
 - ▶ It contains pointers to code
- ☞ **Issue:** The size of the object can be very large



The OOP world (*continued*)

Closures & Objects (*continued*)

A classless object (Javascript-like):



- 👉 **Issue:** The size of the object can be very large
- ▶ **Observation:** While the ivars may change, the code pointers do not
- ▶ **Observation:** The code pointers are common to all objects of this kind

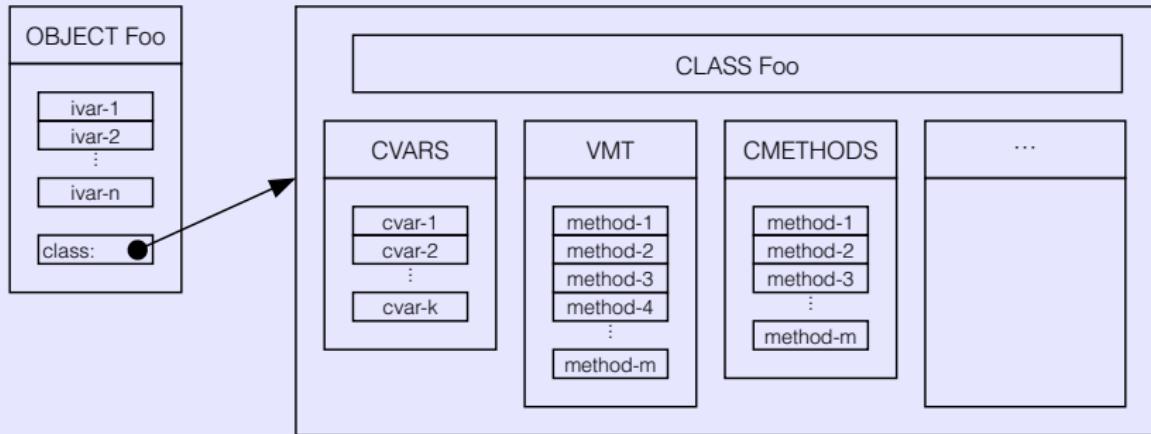
The OOP world (*continued*)

Introducing classes

- ▶ Contain all data that is shared by all objects of the same kind
 - ▶ Virtual-Method Table (VMT)
 - ▶ Static, class vars
 - ▶ Static, class methods
 - ▶ In Smalltalk: Collection of all instances of the class
 - ▶ Various data in support of administration & reflection
- ▶ Moving from instance-based OOP to class-based OOP can be thought of as an example of **refactoring** to the **flyweight pattern**
- ▶ After refactoring, the instance becomes much smaller, consisting of only the instance variables + a pointer to the corresponding class

The OOP world (*continued*)

Introducing classes



The OOP world (*continued*)

Object creation

- ▶ Allocate memory for object
- ▶ Initialize instance variables
- ▶ Link the object to its class
 - ▶ In Smalltalk: Add the instance to the instances-container in the class object

The OOP world (*continued*)

Virtual-method call

- ▶ Upon call
 - ▶ Evaluate method arguments, push values from last to first
 - ▶ **Optionally:** Push the number of arguments
 - ▶ Push `this/self`
 - ▶ De-reference `this` → `class` → `VMT[...]` to arrive at the address of the method
 - ▶ Call the method
 - ▶ For tail-calls, handle differently: More on this later, when we cover the tail-call optimization
- ▶ Upon return
 - ▶ Restore stack to its position before the call
 - ▶ Again, tail-calls make this tricky! More on this later...

The OOP world (*continued*)

Closures & Objects (*continued*)

Summary:

- ▶ Objects & closures are similar
- ▶ Calling a method & calling a closure are similar
- ▶ The lexical environment & `this/self` are similar
- ▶ Bound variables & instance variables are similar
- ▶ Functions are **constructors** for objects of the type of the body of the function:
 - ▶ `cos` is a constructor of floating-point numbers
 - ▶ `string-append` is a constructor of strings
- etc.
- ▶ Closures are objects with a single method, `apply`

Further reading

-  The Flyweight Pattern
-  Design Patterns: Elements of Reusable Object-Oriented Software
-  Refactoring to Patterns

Roadmap

- ✓ Scope
- ✓ The lexical environment
- Boxing

Lexical scope, sharing (*continued*)

We mentioned before that as we evaluate inner lambda-expressions, parameters are copied from the stack onto the **extended lexical environment** of the closure that is the value of the inner lambda-expression:

- ▶ Until control is returned from the outer lambda-expressions, the variables are **both** on the stack and in a lexical environment
 - ▶ That the value of a variable is duplicated and appears simultaneously at addresses A & B , raises the question of whether change to the object at A would/could/should be observable at location B ?
- 👉 **Solution:** Move a pointer away!

Lexical scope, sharing (*continued*)

Moving a pointer away

- ▶ The object appears in only one place: C
- ▶ Both A & B contain the **address** of the object, i.e., C
- ▶ “Changing the object at address A ” means de-referencing the pointer in A , which gives C , and changing the one and only occurrence of the object, which is in C . Such a change would be observable from anywhere that contains the address of C .

Lexical scope, sharing (*continued*)

```
interface I { void foo(int x); }
...
int z;
...
I obj1 = new I() {
    void foo(int x) {
        I obj2 = new I() {
            void foo(int y) {
                z = (++x) * (--y); }
        };
    ...
}
};
```

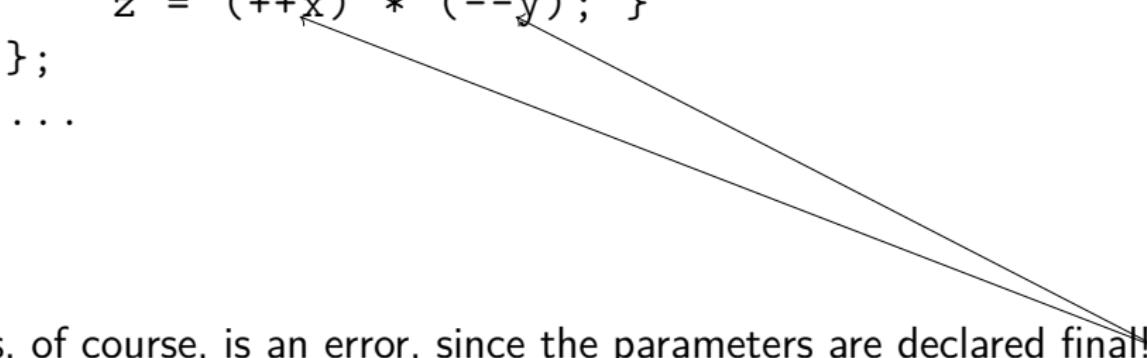
A parameter of a method of an outer class must be declared **final**.

Lexical scope, sharing (*continued*)

```
interface I { void foo(final int x); }

...
int z;

...
I obj1 = new I() {
    void foo(final int x) {
        I obj2 = new I() {
            void foo(final int y) {
                z = (++x) * (--y); }
        };
        ...
    }
};

...

```

This, of course, is an error, since the parameters are declared final! 

Lexical scope, sharing (*continued*)

```
interface I { void foo(final int [] x); }

...
int z;
...

I obj1 = new I() {
    void foo(final int [] x) {
        I obj2 = new I() {
            void foo(final int [] y) {
                z = (++x[0]) * (--y[0]);
            }
        };
    ...
}
};
```

This is one solution...

Lexical scope, sharing (*continued*)

The process of moving one-pointer away from an object is known as “boxing”:

- ▶ We can use a container object in Java, or an array of size 1
- ▶ For each variable being boxed, all references to it are replaced with de-references, either for `set` or for `get`
- ▶ The reference is indeed `final`, and does not change
 - ▶ This is why it can be copied any number of times!
- ▶ What does change is the `contents` of the de-referenced object

Lexical scope, sharing (*continued*)

- ▶ In Java, boxing has not been supported so far, but is scheduled for a future version
- ▶ Until Java supports boxing transparently, it must be done explicitly by the programmer, as shown in the above example
- ▶ Boxing is automatic & transparent in LISP/Scheme/Smalltalk
 - It is done when needed
- ▶ Boxing raises the access cost for variables, so we want to box as few variables as possible
- 👉 We shall add support for boxing variables in our compiler

Lexical scope, sharing (*continued*)

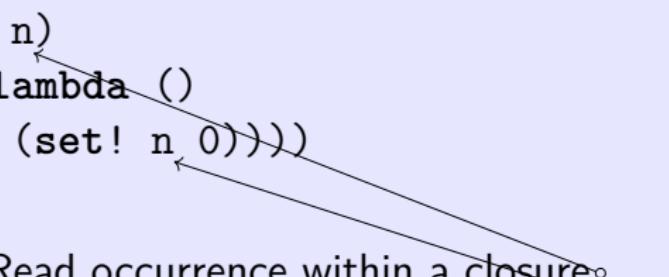
When a variable must be boxed

- ▶ We present a criterion that is sufficient, but not necessary
 - ▶ This means that sometimes we box variables in situations where boxing is not, in fact, necessary
 - ▶ Our criterion is conservative: It shall always box variables when necessary
- ▶ For our compiler, you should box a variable if:
 - ▶ The variable has [at least] one occurrence in for **read** within some closure, and [at least] one occurrence for **write** in **another** closure
 - ▶ Both occurrences do not **already** refer to the same **rib** in a lexical environment

Example of boxing

We **should** box

```
(lambda (n)
  (list
    (lambda ()
      (set! n (+ n 1)))
    n)
  (lambda ()
    (set! n 0))))
```



- ▶ Read occurrence within a closure ◦
- ▶ Write occurrence within another closure ◦
- ▶ Both occurrences **do not already** share a rib

Example of boxing

We should not box

```
(lambda (n)
  (lambda ()
    (list
      (lambda ()
        (set! n (+ n 1)))
      n)
    (lambda ()
      (set! n 0)))))
```

- ▶ Read occurrence within a closure
- ▶ Write occurrence within another closure
- ▶ Both occurrences already share a rib

Example of boxing

We should not box

```
(lambda (n)
  (set! n (+ n 1))
  n)
```



- ▶ The read/write occurrences are within the same closure.

Example of boxing

We should not box

```
(lambda (n)
  (lambda (u)
    (u (lambda ()
      (set! n (+ n 1))
      n)
    (lambda ()
      (set! n 0))))))
```

- Both the `set!` & `get` occurrences of `n`, though in two different closures, **do share** the same rib in their lexical environments

Example of boxing

We **should** box

```
(lambda (n)
  (list
    (begin
      (set! n (* n n))
      n)
    (lambda () n))))
```

- ▶ The set & get occurrences of n occur within two different closures (note that the set occurrence is to a parameter!)
- ▶ They **do not** share the same rib in their respective, lexical environments

Example of boxing

We **should not** box

```
(lambda (n)
  (list
    (lambda () (set! n 0))
    (lambda () (set! n 1))))
```

- ▶ There is no **get** occurrence for n

Chapter 4

Roadmap

- ✓ Scope
- ✓ The lexical environment
- ✓ Boxing

Further reading

Chapter 5

Agenda

- ▶ Intuition about the tail-calls, tail-position, & the tail-call-optimization
- ▶ The tail-position, tail-call
- ▶ The TCO
- ▶ Loops & tail-recursion
- ▶ Annotating the tail-call
- ▶ What TCO code looks like
- ▶ Implementing the TCO

The tail-call (*intuitively*)

- ▶ Two people are walking through a forest, and encounter a witch 
- ▶ The witch grants them three wishes...
- ▶ Here is what each person wished for:

First Person	Second Person
US\$1,000,000	3 more wishes
A grade of 100	US\$1,000,000
3 more wishes	A grade of 100

- ▶ What is the difference?

The tail-call (*continued*)

- Here is what each person wished for:

First Person	Second Person
US\$1,000,000	3 more wishes
A grade of 100	US\$1,000,000
3 more wishes	A grade of 100

- The first person's wishes are simple to grant
- The second person's wishes are annoying:
 - The witch must **remember** what to do once she **returned** from granting the 3 **new** wishes... She still has work to do!
 - Since this nonsense is going to go on for a while, the witch needs a **stack** of paper slips to manage the outstanding requests, in order...

This is the difference between a tail-call (First Person) & a non-tail-call (Second Person)

The tail-call (*continued*)

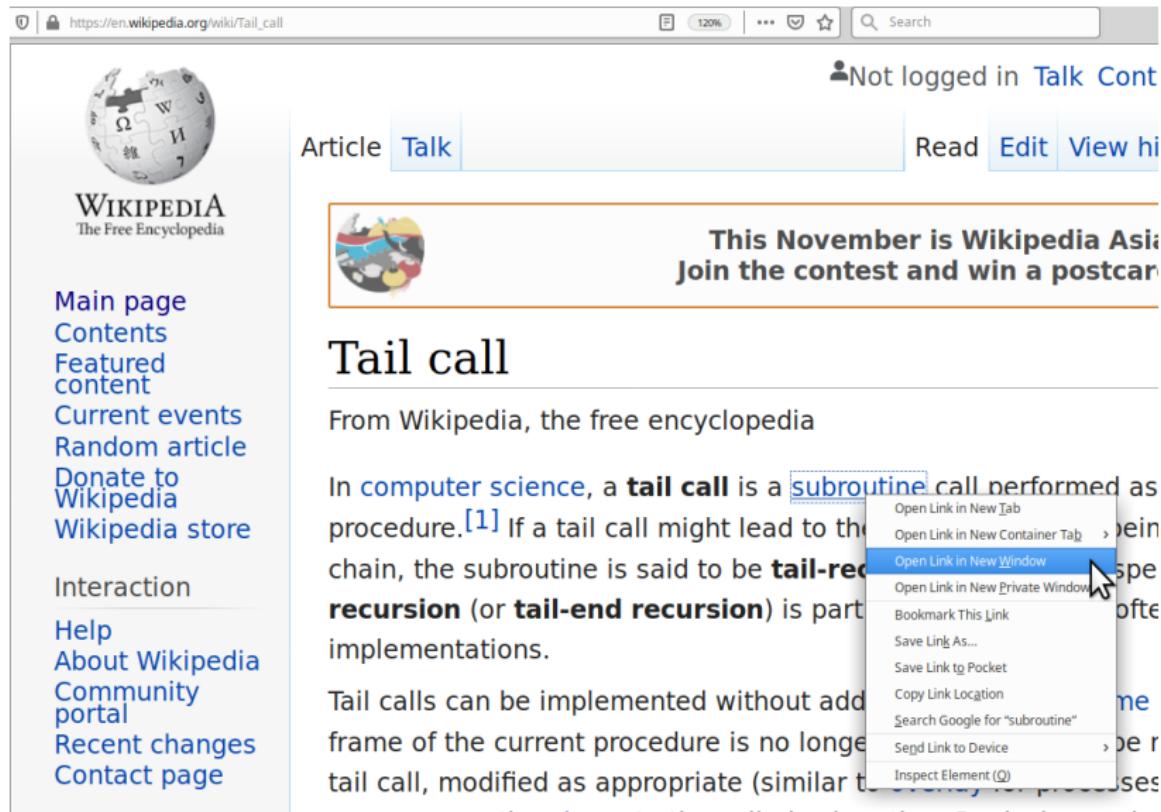
Back to reality:

- ▶ Non-tail-calls require additional stack-frames to manage arguments & return-addresses
 - ▶ The stack depth (in frames) is proportional to the number of non-tail-calls
- ▶ Tail-calls do not require additional stack-frames
 - ▶ The stack depth (in frames) is independent of the number of tail-calls

The tail-call optimization (*intuitively*)

- ▶ You are surfing a web browser
- ▶ The browser is broken: It has no **back-key** (◀)
 - ▶ Once you click on a link, you are unable to **return**
- ▶ To read a web-page, you therefore **right-click** on a link & select the option to **open in a new frame**
 - ▶ You read the page in the **new frame**, possibly opening links in additional frames
 - ▶ When you are done reading a page, you click on the **☒** button to remove the frame

The tail-call optimization (*intuitively, cont*)



A screenshot of a web browser displaying the Wikipedia article on tail-call optimization. The URL is https://en.wikipedia.org/wiki/Tail_call. The page title is "Tail call". The main content discusses tail calls and tail-recursion. A context menu is open over the word "tail", showing options like "Open Link in New Window" and "Open Link in New Container Tab".

Not logged in Talk Cont

Article Talk Read Edit View hi

This November is Wikipedia Asia
Join the contest and win a postcar

Tail call

From Wikipedia, the free encyclopedia

In computer science, a **tail call** is a [subroutine](#) call performed as procedure.^[1] If a tail call might lead to the chain, the subroutine is said to be **tail-recursion** (or **tail-end recursion**) is part implementations.

Tail calls can be implemented without adding frame of the current procedure is no longer tail call, modified as appropriate (similar to

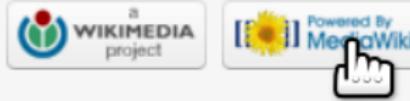
- Open Link in New Tab
- Open Link in New Container Tab >
- Open Link in New Window** →
- Open Link in New Private Window
- Bookmark This Link
- Save Link As...
- Save Link to Pocket
- Copy Link Location
- Search Google for "subroutine"
- Send Link to Device
- Inspect Element (Q)

The tail-call optimization (*intuitively, cont*)

- ▶ Some web-pages have special links:
 - ▶ These links are not just the last links on the page
 - ▶ These links are the **last thing** on the page
 - ▶ There's nothing to read past these links!
- ☞ Not all pages have such special links! This page does:

e License; additional terms
ivacy Policy. Wikipedia® is a
t organization.

Developers Cookie statement



The tail-call optimization (*intuitively, cont*)

- 👉 This page **does not** have such a special link. The last link on the page **is not the last thing** on the page, so returning from that link, there is yet something to read:



👉 Yeah? What's there left to read???

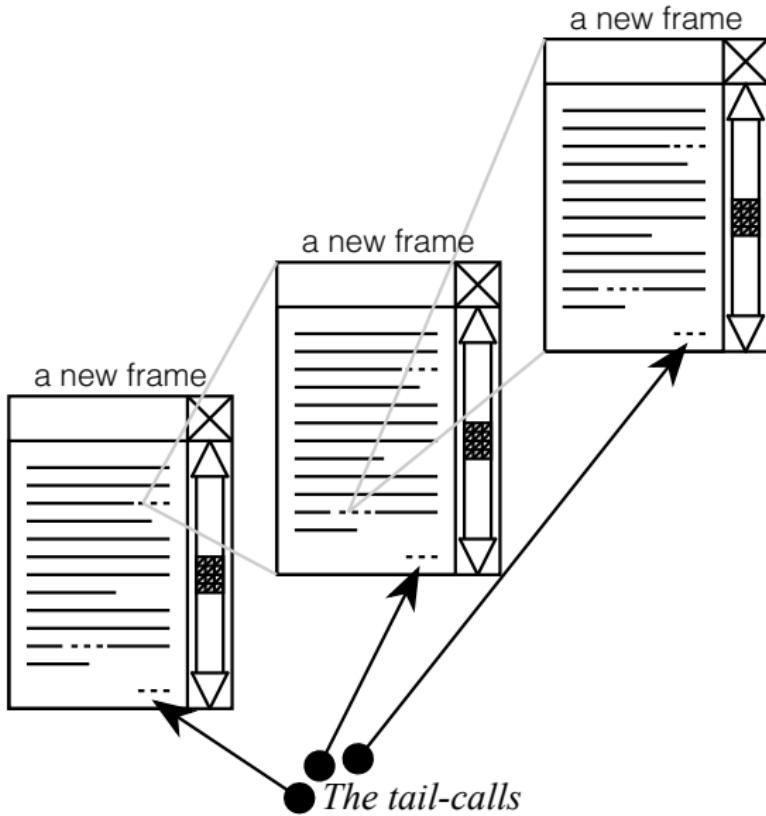
- ▶ Enjoy the logo; It is not a link! 😊



The tail-call optimization (*intuitively, cont*)

- ▶ Because there's nothing to read past these links, we need **neither to open** a new frame **nor to return** from it:
 - ▶ Rather than right-click & open the page in the new frame, we simply click on the link **in place**
 - ▶ The new contents shall overwrite the old contents
 - ▶ The new contents can be larger or smaller than the contents it replaces
 - ▶ The **size of the frame** can change
 - ▶ The **number of frames** shall not change
 - ▶ When we're done reading the page, we close it with 

The tail-call optimization (*intuitively, cont*)



Chapter 5

Agenda

- ✓ Intuition about the tail-calls, tail-position, & the tail-call-optimization
- ▶ The tail-position, tail-call
- ▶ The TCO
- ▶ Loops & tail-recursion
- ▶ Annotating the tail-call
- ▶ What TCO code looks like
- ▶ Implementing the TCO

The tail-position

- ▶ The **tail-position** is a point in the body of a function, procedure, method, subroutine, etc., **just before the return-statement**, where the last computation is performed
 - ▶ **Exception handling:** Similarly to `return`-statements, `raise/throw`-statements also identify tail-positions!
- ▶ Because computation may proceed non-linearly, there many be more than one tail-position
- ▶ To find the tail-positions, find all points in the code where a `return`-statement or the `ret` instruction could be placed

The tail-position (*continued*)

Example:

```
(lambda (x)
  (f (g (g x))))
```



tail-call

The tail-position (*continued*)

Example:

```
(lambda (x)
  (f (lambda (y)
        (g x y))))
```



- 👉 Each lambda-expression has its own return-statement, and therefore, each lambda-expression has its own tail-position!

The tail-position (*continued*)

Example:

```
(lambda (x y z w)
  (if (foo? x)
      (goo y)
      (boo (doo z))))
```

► o tail-calls

- ☞ If an if-expression is in tail-position, then the then-expression & else-expression are **also** in tail-position

The tail-position (*continued*)

Example:

```
(lambda (x y z)
  (f (if (g? x)
           (h y)
           (w z))))
```

► tail-call

- ☞ If an if-expression **is not** in tail-position, then neither are its then-expression & else-expression

The tail-position (*continued*)

Example:

```
(lambda (a b)
  (f a)
  (g a b)
  (display "done!\n"))
  ^-- tail-call
```

- ☞ If a sequence, whether explicit or implicit, is in tail-position, the last expression in the sequence is **also** in tail-position

The tail-position (*continued*)

Example:

```
(lambda ()  
  (and (f x) (g y) (h z)))
```

- ▶ tail-call 
- ▶ If an and-expression is in tail-position then its last expression is also in tail-position
- ▶ While it is possible to return after computing previous expressions [within an and-expression], it is only from the **last expression** that return is possible immediately, without first testing the value of the expression

The tail-position (*continued*)

Example:

```
(lambda ()  
  (or (f (g x)) y))
```

- ▶ The above example contains no application in tail-position
- 👉 Similarly to and-expression, if an or-expression is in tail-position then its last expression is in tail-position

The tail-position (*continued*)

Example:

```
(lambda ()  
  (set! x (f y)))
```

👉 The body of a set!-expression is **never** in tail-position!

The tail-position (*continued*)

Example:

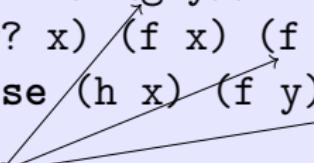
```
(lambda ()  
  (set! x (f (lambda (y)  
                (g x y)))))
```

- ▶ tail-call 
- 👉 Even though the body of the set!-expression is **not** in tail-position, “every lambda-expression has its own *return*”
 - ▶ The marked application is in tail-position relative to its enclosing lambda-expression

The tail-position (*continued*)

Example:

```
(lambda (x y z)
  (cond ((f? x) (g y))
        ((g? x) (f x) (f y))
        (else (h x) (f y) (g (f x)))))
```

- ▶ tail-calls 
- 👉 If a cond-expression is in tail-position then the **last expression** in the implicit sequence of each cond-rib is also in tail-position

The tail-position (*continued*)

Example:

```
(let ((x (f y))
      (y (g x)))
  (goo (boo x) y))
```



tail-call

- ☞ The body of a let-expression is the body of a lambda-expression.
 - ▶ Therefore, the last expression in the implicit sequence of the body is in tail-position

The tail-position (*continued*)

Example:

```
(lambda (s)
  (apply f s))
```

This is an interesting example:

- ▶ On the one hand, there is only one tail-call that appears **statically** in the source code.
- ▶ On the other hand, there is another tail-call that shall take place **at run-time**: The application of *f* must re-use the top activation frame!
- 👉 The implementation of *apply* duplicates part of the code for the tail-call-optimization, so that the frame for the call to *f* overwrites the frame for the call to *apply*!

The tail-position (*continued*)

Example:

```
let disj (nt1 : α parser) (nt2 : α parser) =  
  ((fun str index ->  
    try (nt1 str index)  
    with X_no_match -> (nt2 str index)) : α  
  parser);;
```

- ▶ This application is **not** in tail-position
 - ⌚ Upon returning successfully, the code needs to **pop** an exception-handler!
- ▶ This application is in tail-position

Chapter 5

Agenda

- ✓ Intuition about the tail-calls, tail-position, & the tail-call-optimization
- ✓ The tail-position, tail-call
- ▶ The TCO
- ▶ Loops & tail-recursion
- ▶ Annotating the tail-call
- ▶ What TCO code looks like
- ▶ Implementing the TCO

The tail-call optimization

The tail-call optimization is a recycling of activation frames on the stack:

- ▶ Function/method calls ordinarily open new activation frames on the stack
- ▶ Function/method calls in tail-position need not open new activation frames, but may re-use the current/top frame
- ▶ Only two items in the current activation frame need to be preserved during a tail-call:
 - ▶ The **old frame-pointer**
 - ▶ The **return-address**
- ▶ Everything else on the stack —
 - ▶ The lexical environment
 - ▶ The values of arguments
 - ▶ Any local values

are all overwritten & replaced by the contents of the new frame

The tail-call optimization (*continued*)

- ▶ The tail-call optimization optimizes stack-frames
 - ▶ In some situations this may also optimize time
 - ▶ We shall not encounter many other optimizations of **space**
 - ▶ The tail-call optimization is very important!

Chapter 5

Agenda

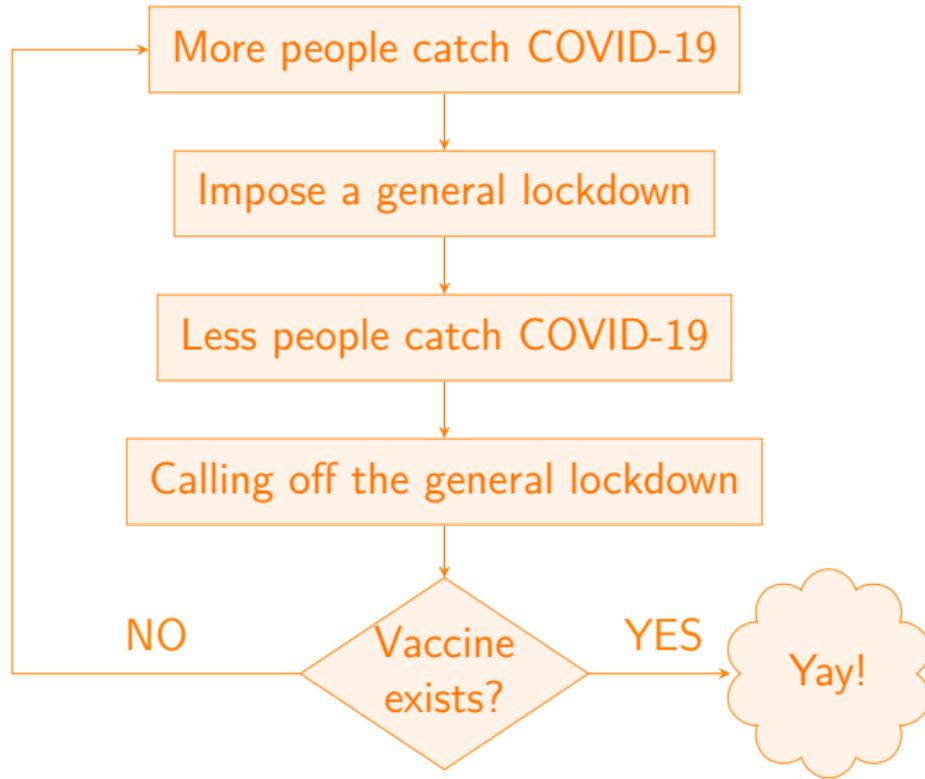
- ✓ Intuition about the tail-calls, tail-position, & the tail-call-optimization
- ✓ The tail-position, tail-call
- ✓ The TCO
- ▶ Loops & tail-recursion
- ▶ Annotating the tail-call
- ▶ What TCO code looks like
- ▶ Implementing the TCO

The tail-call optimization (*continued*)

The significance of the TCO

- ▶ All loops are special cases of recursive functions where the recursive call is in tail-position
 - ▶ The TCO is our license to use recursive procedures to implement iteration
 - ▶ Absent the TCO, iteration using recursion would be prohibitively expensive:
 - ⚠ The amount of stack consumed in the execution of a loop would be proportional to the number of iterations
 - ⚠ Large loops would exhaust the stack

Loops



Loops (*cont*)

- ▶ By now, we know how to identify the tail-position
- ▶ We made the claim that the TCO is important because it gives us license to implement loops using tail-recursive functions
- 👉 We now need to pay this debt, and show that loops are tail-recursive functions!

Loops (*continued*)

Example: while-loops

```
(define while
  (lambda (test body)
    (if (test)
        (begin
          (body)
          (while test body)))))
```

► o tail-call

Loops (*continued*)

Example: while-loops (*cont*)

```
> (let ((i 0))
  (while
    (lambda () (< i 10))
    (lambda ()
      (set! i (+ 1 i))
      (display
        (format "~a~%" i)))))
```

1
2
...
9
10

Loops (*continued*)

Example: for-loops

```
(define for
  (lambda (from to body)
    (if (< from to)
        (begin
          (body from)
          (for (+ 1 from) to body)))))
```

- ▶ Notice that the body is parameterized by the **index-variable** of the loop
- ▶ tail-call

Loops (*continued*)

Example: for-loops (*cont*)

```
> (for 1 6
      (lambda (i)
        (for 1 6
            (lambda (j)
              (display
                (format "\t~a"
                       (* i j))))))
      (newline)))
```

1	2	3	4	5
2	4	6	8	10
3	6	9	12	15
4	8	12	16	20
5	10	15	20	25

The tail-call optimization (*continued*)

- ▶ The tail-call optimization is not just important for loops
- ▶ Consider *Ackermann's Function*:

$$\begin{aligned}\text{Ack}(0, q) &= q + 1 \\ \text{Ack}(p + 1, 0) &= \text{Ack}(p, 1) \\ \text{Ack}(p + 1, q + 1) &= \text{Ack}(p, \text{Ack}(p + 1, q))\end{aligned}$$
$$\begin{aligned}\text{Ack}(2, 2) &= 7 \\ \text{Ack}(3, 3) &= 61\end{aligned}$$

- ▶ tail-calls ○
- ▶ non-tail-call ○
- ▶ The TCO cuts reduces the number of frames needed by 50%, which lets us compute *Ackermann's Function* for larger inputs

The tail-call optimization (*continued*)

One disadvantage of the TCO:

- ▶ When a frame is overwritten, **debug information** is lost
 - ▶ Smalltalk & Java have great debuggers
 - ▶ Scheme does not!
 - ▶ Language implementations that do not implement the TCO should, at the very least, offer efficient & convenient looping mechanisms
- 👉 **Compromise:** Turn the TCO on/off while debugging
- ▶ No Scheme compiler does this
 - ▶ It's easy (if tedious) to do by hand:
`(define id (lambda (x) x))`
 - Just wrap each tail-call with `id` 😊

Chapter 5

Agenda

- ✓ Intuition about the tail-calls, tail-position, & the tail-call-optimization
- ✓ The tail-position, tail-call
- ✓ The TCO
- ✓ Loops & tail-recursion
- Annotating the tail-call
- What TCO code looks like
- Implementing the TCO

Annotating tail-calls in our compiler

- ▶ Annotating tail-calls is done in a single pass over the AST of `expr`
- ▶ You are given the type `expr'` which includes the type constructor `ApplicTP'` of `expr' * (expr' list)` for encoding tail-calls
- ▶ The simplest way to annotate tail-calls is to carry along an auxiliary parameter `in_tp` (read: *in tail-position*) to indicate whether the current expression is in tail-position
- ▶ The initial value of `in_tp` is false
- ▶ When an `Applic` is encountered and the value of `in_tp` is true, an `ApplicTP'` is used to package the result of the recursive calls over the procedure and the list of arguments
- ▶ Upon entering lambda-expressions (of any kind), the value of `in_tp` is reset back to true

Putting it all together...

```
(lambda (a) (a (a (a (lambda (b) (b (b (a c))))))))  
  
LambdaSimple' (["a"] ,  
    ApplicTP' (Var' (VarParam' ("a" , 0)) ,  
        [Applic' (Var' (VarParam' ("a" , 0)) ,  
            [LambdaSimple' (["b"] ,  
                ApplicTP' (Var' (VarParam' ("b" , 0)) ,  
                    [Applic' (Var' (VarParam' ("b" , 0)) ,  
                        [Applic' (Var' (VarBound' ("a" , 0 , 0)) ,  
                            [Var' (VarFree' "c")]])]))]))]))
```

Chapter 5

Agenda

- ✓ Intuition about the tail-calls, tail-position, & the tail-call-optimization
- ✓ The tail-position, tail-call
- ✓ The TCO
- ✓ Loops & tail-recursion
- ✓ Annotating the tail-call
- What TCO code looks like
- Implementing the TCO

TCO Ackermann in C

The TCO is simplest to demonstrate on a function with immediate tail-recursion:

- ▶ The new frame is identical to the old frame, both in size and types

Consider *Ackermann's Function*:

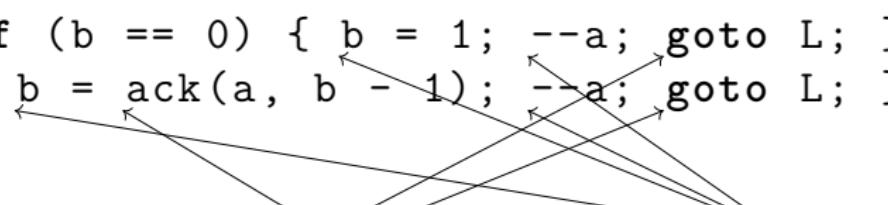
```
int ack(int a, int b) {  
    if (a == 0) { return b + 1; }  
    else if (b == 0) { return ack(a - 1, 1); }  
    else { return ack(a - 1, ack(a, b - 1)); }  
}
```

- 👉 A tail-call is identified by the pattern `return f(...)`
- 👉 There are two tail-recursive function calls

TCO Ackermann in C (*continued*)

Optimizing the tail-calls in *Ackermann's Function* results in:

```
int ack(int a, int b) {  
    L:  
    if (a == 0) { return b + 1; }  
    else if (b == 0) { b = 1; --a; goto L; }  
    else { b = ack(a, b - 1); --a; goto L; }  
}
```



- ▶ Notice that the current frame is changed in place.
- ▶ Notice that `goto` replaces the function call
- ▶ Notice that one non-tail-call remains

TCO Ackermann in x86/64

Here is *Ackermann's Function*, with the TCO, in x86/64:

ack:

```
    cmp rdi, 0
    jz .A
    cmp rsi, 0
    jz .B
    push rdi
    dec rsi
    call ack
    mov rsi, rax
    pop rdi
    dec rdi
    jmp ack
```

.A:

```
    lea rax, [rsi + 1]
    ret
.B:
    dec rdi
    mov rsi, 1
    jmp ack
```

Tail-calls and a non-tail-call

The tail-call optimization (*continued*)

- ▶ When the tail-call is not immediately-recursive
 - ▶ non-recursive, or
 - ▶ mutually-recursive

we cannot demonstrate the TCO in high-level C

- ▶ This would require that we goto from within the body of one procedure into a label that is **local** to another procedure
- ▶ The new frame can be very different in size, argument count, and argument type from the old frame, so we cannot overwrite it in a high-level language

The tail-position (*continued*)

- ▶ Tail-recursion, whether immediate or mutual, is a **special case** of the tail-call optimization, where the call is recursive or mutually recursive
- ▶ The general tail-recursion optimization is all that is required to support the implementation of loops using recursion
 - ▶ This is required by the standard for Scheme
- ▶ The tail-call optimization is more general than the tail-recursion optimization, and optimizes all tail-calls
- ▶ The tail-call optimization crosses the boundaries of functions/procedures/methods, and is implemented in assembly/machine language
- 👉 You will implement the **tail-call-optimization** in your compilers

Chapter 5

Agenda

- ✓ Intuition about the tail-calls, tail-position, & the tail-call-optimization
- ✓ The tail-position, tail-call
- ✓ The TCO
- ✓ Loops & tail-recursion
- ✓ Annotating the tail-call
- ✓ What TCO code looks like
- Implementing the TCO

The tail-call optimization (*continued*)

Upon tail-call

- ① Evaluate the arguments, and push their values onto the stack in reverse order (from last to first)
- ② Push the argument count
- ③ Evaluate the procedure expression
 - ▶ Verify that we have a closure!
- ④ Push the lexical environment of the closure
- ⑤ Push the return-address of the current frame
- ⑥ Restore the old frame-pointer register (on x86/64: rbp)
- ⑦ Overwrite the existing frame with the new frame
 - ▶ Loop, `memcpy`, whatever...
- ⑧ `jmp` to the code-pointer of the closure

The tail-call optimization (*continued*)

Upon return from a procedure-call

- ① Add a wordsize (8 bytes) to the stack-pointer (on x86/64: `rsp`) to move past the lexical environment
- ② Pop off the argument count
 - ▶ This can be different from the number of arguments you pushed:
 - ▶ If you call a procedure with n arguments and it tail-calls a procedure with k arguments, and it calls a procedure with r arguments, which returns, then you get back r arguments on the stack!
 - ▶ Variadic lambda-expressions and lambda-expressions with optional arguments will alter the stack to number of their parameters: If you call the procedure `(lambda (a b c . r) ...)` with 29 arguments, and it returns, you can expect 4 arguments on the stack, the last one of which being a list of the remaining 26 arguments...

The tail-call optimization (*continued*)

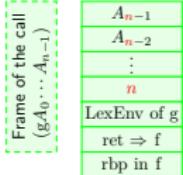
Upon return from a procedure-call

- ② Pop off the argument count
- ③ Remove as many arguments off the stack as indicated by the argument count: Add word-size * argument-count to the stack-pointer register

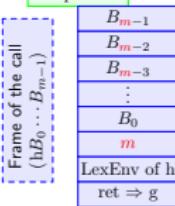
Comparing stacks for non-TCs and TCs

Setting up the stack for a **non-tail-call** to h

A **non-tail-call** to
($gA_0 \dots A_{n-1}$) from
within the body of f

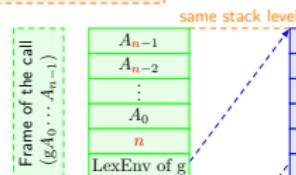


Setting up the stack
for a **non-tail-call** to
($hB_0 \dots B_{m-1}$) from
within the body of g
Once called, h shall
push the rbp

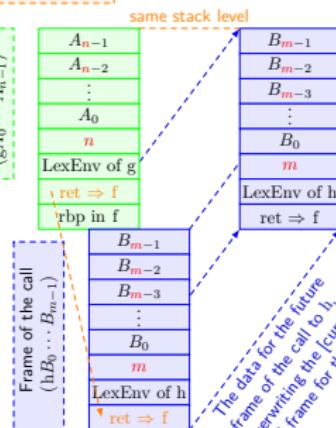


Setting up the stack for a **tail-call** to h

A **non-tail-call** to
($gA_0 \dots A_{n-1}$) from
within the body of f



Setting up the stack
for a **tail-call** to
($hB_0 \dots B_{m-1}$) from
within the body of g
Once called, h shall
push the rbp



The stack set up for
the **tail-call**
($hB_0 \dots B_{m-1}$) after
overwriting the frame
from the call to g,
and before jumping
to f

The data for the future
frame of the call to h
overwriting the [current
top] frame for the call to g

- ▶ The stack for a **non-tail-call** to h
 - ▶ A non-tail-call from f to g
 - ▶ A **non-tail-call** from g to h
- ▶ The stack for a **tail-call** to h
 - ▶ A non-tail-call from f to g
 - ▶ A **tail-call** from g to h

Chapter 5

Agenda

- ✓ Intuition about the tail-calls, tail-position, & the tail-call-optimization
- ✓ The tail-position, tail-call
- ✓ The TCO
- ✓ Loops & tail-recursion
- ✓ Annotating the tail-call
- ✓ What TCO code looks like
- ✓ Implementing the TCO

Further reading



Goals

- ▶ Understanding the significance of loop-optimizations
- ▶ Understanding the basic loop-optimizations
- ▶ Ability to implement simple versions of various loop-optimizations
- ▶ Understanding how modern programming languages get in the way of loop-optimizations

Roadmap

- ▶ Understanding the significance of loop-optimizations
- ▶ Understanding some of the main loop-optimizations
 - ▶ Factoring out expressions that are invariant in a loop
 - ▶ loop-fusion
 - ▶ Loop-unrolling
 - ▶ Static branch prediction
 - ▶ Dynamic branch prediction
 - ▶ Software pipelining
- ▶ The decline of loop-optimizations

Roadmap

- 👉 Understanding the significance of loop-optimizations
- ▶ Understanding some of the main loop-optimizations
 - ▶ Factoring out expressions that are invariant in a loop
 - ▶ loop-fusion
 - ▶ Loop-unrolling
 - ▶ Static branch prediction
 - ▶ Dynamic branch prediction
 - ▶ Software pipelining
- ▶ The decline of loop-optimizations

SELF-STUDY Loops are interesting

 During most of the run-time of a program, the CPU is performing some kind of loop:

- ▶ It takes very little time to pass through all of RAM linearly
- ▶ Most computer programs take longer to execute than it takes to pass through all of RAM
- 👉 Hence the CPU is performing a loop!

- ▶ A loop consists of **administration** and **body**
 - ▶ Administration
 - ▶ Initializing variables
 - ▶ Testing
 - ▶ Updating
 - ▶ Jumping
 - ▶ Body
 - ▶ The code we wish to execute over and over

- ▶ The time it takes to compute a loop is proportional to the number of times the body of the loop is executed, and is **roughly** equal to

$$\text{initialization} + (\text{testing} + \text{update} + \text{jumping} + \text{body}) \times \text{count}$$

where *count* is the number of times the loop is performed, i.e., the number of times the body of the loop is executed

- ▶ Any significant improvements in the time a loop executes must come from the iteration (**marked**):

$$\text{initialization} + (\text{testing} + \text{update} + \text{jumping} + \text{body}) \times \text{count}$$

- ▶ Of course, if the loop is **nested within another loop**, then the *initialization* component may become significant

SELF-STUDY Loops are interesting (*cont*)

- 👉 Optimizing loops requires we change:
 - ▶ *count*, i.e., the number of times a loop runs
 - ▶ *testing*, i.e., what tests are performed
 - ▶ *update*, i.e., how various counters are managed
 - ▶ *jumping*, i.e., how we return to the head of the loop
 - ▶ *body*, i.e., the work done inside the loop
- 👉 Each loop-optimization affects **some** of these components in order to reduce the time spent inside the loop

SELF-STUDY Components of a loop

The Components of Loops in Pascal

```
for i := 1 to 100 do
begin
  writeln('iteration ', i);
  A[i] := 100 - i;
end;
```

initialization

count

testing is implicit

update is implicit

jumping is implicit

body

The Components of Loops in C

```
for (i = 0, j = 100; i < j; ++i, --j) {  
    printf("iteration %d\n", i);  
    A[i] = j;  
}
```

- ▶ *initialization*
- ▶ *count is implicit*
- ▶ *testing*
- ▶ *update*
- ▶ *jumping is implicit*
- ▶ *body*

 SELF-STUDY Components of a loop (*cont*)

- ▶ In high-level languages, some components of a loop are **implicit**, that is, they are not mentioned explicitly, but appear in the resulting machine-language
- ▶ In low-level languages, such as assembly/machine-language, or unstructured BASIC, the components of a loop are generally more explicit
- ▶ Even low-level languages support FOR-loops (BASIC), or loop-instructions (Intel x86) that hide some of the components

SELF-STUDY Components of a loop (*cont*)

```
10 INPUT "N = "; N
20 R = 1
30 IF N = 0 THEN 70
40 R = R * N
50 N = N - 1
60 GOTO 30
70 PRINT "FACTORIAL = "; R
80 END
```

- *initialization*
- *count*
- *testing*
- *update*
- *jumping*
- *body* can be thought of as empty...

- ▶ Loops are special cases of applying **recursive functions** [in tail-position]
- ▶ Recursive functions are defined over the inductive structure of **precisely one of its parameters**

(1/2)

For example, the inductive structure of the set of natural numbers \mathbb{N} is generally given using the inductive definition of **Peano Numbers**:

- ▶ $0 \in \mathbb{N}$
- ▶ $S(n) \in \mathbb{N}$, when $n \in \mathbb{N}$

- ▶ Loops are special cases of applying recursive functions [in tail-position]
- ▶ Recursive functions are defined over the inductive structure of precisely one of its parameters

(2/2)

This is why recursive functions over \mathbb{N} are generally defined over the above two cases

- ⚠ The set of integer \mathbb{Z} is not ordinarily defined inductively
- 👉 There is no natural principle of induction over the integers
- 👉 Recursive functions over the integers, as often found in real programming [languages], represents a problematic compromise both in theory and in practice!

SELF-STUDY Components of a loop (*cont*)

-  The term **recursive function** can be used to mean at least two very different things:
 - ① A function that calls itself
 - ② A function that is defined according to the **axiom schema for General Recursion**
-  Throughout this course, when we mention recursive functions, we generally mean the sense of [①]
-  Specifically, when discussing the current topic, of **optimisations of loops**, we mean the sense of [②]

SELF-STUDY Components of a loop (*cont*)

- ▶ The **induction-variable of a loop** is precisely the inductive parameter in the corresponding recursive function
- ▶ The induction-variable in a loop might not always be apparent:
 - ▶ In some cases, the compiler can transform a loop, so as to make apparent the induction-variable
 - ▶ This would correspond to transforming the definition of a recursive function in the sense of [①] into an equivalent definition in the sense of [②]
 - ▶ In some cases, the compiler might not be able to identify an induction-variable
 - ▶ Although a clever mathematician might be able to do so!

- ▶ The induction-variable in a loop might not always be apparent:
 - ▶ In some cases, the compiler can **transform the loop** into an equivalent one, with an explicit induction-variable
- ▶ In some cases, the induction-variable does not exist, either explicitly or implicitly

Example of a loop without an induction variable

```
for (;;) {  
    printf("Find my induction variable! :-)\n");  
}
```

- ▶ This is an infinite loop, uncontrolled by any variable
 - ▶ In principle, some infinite loops could be optimized, but not this one
-  Think why!

Example of a loop without an induction variable

```
extern int f(int i);  
...  
for (i = 0; i < MAX; ) {  
    i = f(i);  
}
```



This loop **is** controlled by the variable *i*, but is it an induction variable?

- ▶ To be an induction variable, *i* must **decrease** in some sense
- ▶ Since *f* is unknown to the compiler, it cannot know how *i* changes
- ▶ Since the compiler cannot prove that *i* decreases [in any sense], *i* is not an induction variable

- ▶ Many of the loop-optimizations assume that an induction-variable exist
- ▶ If the induction-variable cannot be identified, the compiler will simply fail to apply the given optimization
- ▶ An optimization may affect the efficiency of code, but not its semantics (meaning, behaviour, etc)

SELF-STUDY Loops are interesting (*cont*)

- ▶ In some sense, loop-optimizations are the most interesting optimizations:
 - ▶ Even the smallest change in **what happens inside the loop** is then multiplied by the **number of times the loop is performed**
initialization + (testing + update + jumping + body) × count
 - ▶ If we can **also** reduce the number of times the loop itself is performed, i.e., reduce *count*, then the savings will be even greater
- ▶ Optimizing loops improves performance more than any other kind of optimization

SELF-STUDY Loops are interesting (*cont*)

- ▶ Our goal at this point is to go through some of the main loop-optimizations:
 - ▶ Understand when we can apply them
 - ▶ Understand which components of the loop do they affect
 - ▶ See how they transform a loop

Roadmap

- ✓ Understanding the significance of loop-optimizations
- Understanding some of the main loop-optimizations
 - ☞ Factoring out expressions that are invariant in a loop
 - ▶ loop-fusion
 - ▶ Loop-unrolling
 - ▶ Static branch prediction
 - ▶ Dynamic branch prediction
 - ▶ Software pipelining
 - The decline of loop-optimizations



SELF-STUDY Factoring expressions out of a loop

Consider the following pseudo-code in some C-like language:

```
for (i = 0; i < MAX; ++i) {  
    A[i] = i * sin(0.39);  
}
```

This is not very good code, as the expression $\sin(0.39)$ is computed over and over inside the loop. How might we improve on it?



SELF-STUDY Factoring expressions out of a loop (*cont*)

You might consider pulling the expression `sin(0.39)` out of the loop, storing its value in a variable, and using this variable within the loop:

```
double sin_0_39 = sin(0.39);
```

```
for (i = 0; i < MAX; ++i) {  
    A[i] = i * sin_0_39;  
}
```

Where `sin_0_39` is a **fresh variable** that does not conflict with any parameter or local variable.

- ▶ Now `sin(0.39)` is computed only once.
- But is this a **valid** transformation?
 - ▶ Can this transformation change the **meaning** of the code?

We are making several assumptions about the code:

- ▶ That `sin` computes the mathematical *sine*-function over the real numbers
 - ▶ That `sin` has no side-effects that are visible outside its body
-  But aren't these **reasonable** assumptions?
-  In fact, these assumptions are language-specific!



SELF-STUDY Factoring expressions out of a loop (*cont*)

- ▶ In Fortran, Pascal, Algol, etc
 - ▶ The **name** `sin` is reserved for the mathematical function
 - ▶ The mathematical library is standardized, and is defined as part of the language
 - ▶ The user **cannot override** the names `sin`, `cos`, `exp`, `log`, etc
 - 👉 The compiler **knows at compile-time** everything there is to know about the function `sin`, and may use this information to generate better code!



SELF-STUDY Factoring expressions out of a loop (*cont*)

- ▶ In C/C++, etc
 - ▶ The **name** `sin` is defined in the library `libm`
 - ▶ The **type** of `sin` is introduced via `#include <math.h>`, but neither its definition nor its semantics
 - ▶ You link object-files with this library via the `-lm` option to the **linker**
 - 👉 If you pass `-lm` to the compiler, but it just passes it to the linker, if the linker is invoked, or ignores it if it is not
 - ▶ During **compilation** (not **linking!**) the compiler **knows nothing** about the libraries with which you might link
 - ▶ The name `sin` and library `libm` **are not special**:
 - ❓ A user is perfectly entitled to link with the library `-lcatholic`, *to tally up the value of his sins* before attending confession...
 - 👉 The compiler is **unable** to use this information to optimize code



SELF-STUDY Factoring expressions out of a loop (*cont*)

So this rather straightforward optimization:

```
for (i = 0; i < MAX; ++i) {  
    A[i] = i * sin(0.39);  
}
```



```
double sin_0_39 = sin(0.39);  
for (i = 0; i < MAX; ++i) {  
    A[i] = i * sin_0_39;  
}
```

is possible in some languages, and not possible in others!



SELF-STUDY Factoring expressions out of a loop (*cont*)

If the compiler cannot show that a sub-expression in the loop is invariant in it, then the transformation is **unsafe**, and may be **invalid**:

- ▶ If the return value changes from one call to the next (e.g., `randomInteger(1, 100)`), then factoring it out will ensure the value does not change
- ▶ If the expression has side-effects (e.g., `(i *= j)`, `printf("I told you!")`, `malloc(SIZE)`), then factoring out the expression will ensure the side-effects occur only once



SELF-STUDY Factoring expressions out of a loop (*cont*)

How is the optimization performed

To perform this optimization we need to

- ▶ Find in the loop sub-expressions that are invariant
- ▶ Generate **fresh** names for as many variables as expressions that we wish to factor
- ▶ Factor out the expressions just before the loop
- ▶ Rewrite the body of the loop to use the corresponding variable names rather than the expressions

Conclusion

- ▶ This optimization reduces the time spent on computing the *body* of the loop, by factoring out expressions the values of which are invariant under the loop
- ▶ The transformation itself is straightforward, and the code is just what we would expect a competent programmer to write
- ▶ The optimization is language-specific
- ▶ Some languages are defined in ways that deny the compiler information that would be necessary to enable this optimization
 - ▶ In such languages, you may optimize the code by hand
- ▶ In languages where the compiler can establish invariance, the optimization can lead to significant improvements in run-time

Roadmap

- ✓ Understanding the significance of loop-optimizations
- ▶ Understanding some of the main loop-optimizations
 - ✓ Factoring out expressions that are invariant in a loop
 - 👉 loop-fusion
 - ▶ Loop-unrolling
 - ▶ Static branch prediction
 - ▶ Dynamic branch prediction
 - ▶ Software pipelining
- ▶ The decline of loop-optimizations

SELF-STUDY Loop-fusion

Loop-fusion is an optimization that combines several loops into one:

- ▶ This optimization improves the code by combining the administration of loops, thereby saving on
 - ▶ *testing*
 - ▶ *update*
 - ▶ *jumping*
- ▶ The savings in the administration of the loop are then multiplied by the number of times the loop is performed
- ▶ Sometimes, it is possible to save on *initialize* as well, but this is insignificant, unless we consider nested loops

SELF-STUDY Loop-fusion (*cont*)

Consider the following pseudo-code in some C-like language:

```
for (i = 0; i < MAX; ++i) {  
    A[i] = i;  
}  
for (i = 0; i < MAX; ++i) {  
    B[i] = 1;  
}
```

Parts of the code are duplicated. How might we improve on it?

SELF-STUDY Loop-fusion (*cont*)

You might consider joining the bodies of these two loops within a single loop:

```
for (i = 0; i < MAX; ++i) {  
    A[i] = i;  
    B[i] = 1;  
}
```

- What have gained by combining the loops?
- When is such a transformation valid?

SELF-STUDY Loop-fusion (*cont*)

What have gained by combining the loops?

component changed by loop-fusion	before	after
<i>initialization</i>	2	1
<i>testing</i>	$2 + 2 \cdot \text{MAX}$	$1 + \text{MAX}$
<i>update</i>	$2 \cdot \text{MAX}$	MAX
<i>jumping</i>	$2 \cdot \text{MAX}$	MAX



Remember: The loop executes completely n times, and fails the $n + 1$ -test!

When is such a transformation valid?

We are making several assumptions about the loops we are merging:

- ▶ The compiler can find an induction-variable (possibly transforming the code to do so)
- ▶ The compiler can show that all these loops run the same number of times
- ▶ The bodies of each loop can be shown not to **conflict**:
 - ▶ Let $BodyA_i$ be the i -th iteration of the body of the first loop
 - ▶ Let $BodyB_j$ be the j -th iteration of the body of the second loop
 - ▶ Notice that $\text{time}_{BodyA_i} < \text{time}_{BodyB_j}, \forall i, j < n$
 - 👉 Therefore, a **conflict** is a read/write-dependency conflict between $BodyA_i, BodyB_j$, for all $\forall i, j < n$

SELF-STUDY Loop-fusion (*cont*)

Example of a conflict between two loop-bodies

Consider the following code:

```
for (i = 0; i < MAX; ++i) {  
    printf("a");  
}  
  
for (i = 0; i < MAX; ++i) {  
    printf("b");  
}
```

 Clearly any $BodyB_j$ must happen **after all** $BodyA_i$, for any $i, j < n$

 We cannot perform the loop-folding optimization on these loops!

SELF-STUDY Loop-fusion (*cont*)

Example of a conflict between two loop-bodies

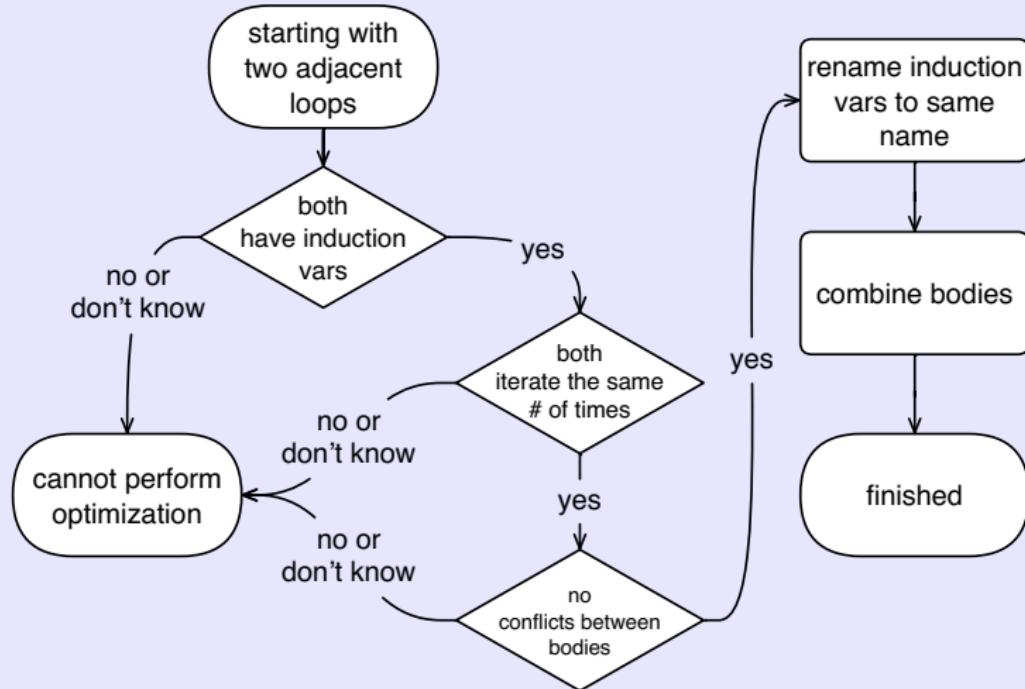
Consider the following code:

```
for (i = 0; i < MAX; ++i) {  
    A[i] = i;  
}  
  
for (i = 0; i < MAX; ++i) {  
    B[i] = A[MAX - 1 - i];  
}
```

- Clearly any $BodyB_j$ must happen **after** $BodyA_{MAX-1-i}$, for any $j < n$
- We cannot perform the loop-folding optimization on these loops!

SELF-STUDY Loop-fusion (*cont*)

How is the optimization performed



 SELF-STUDY Loop-splitting

- ☞ Notice that **Loop-splitting** does not appear as a separate item in the Roadmap slides
 - ▶ That is because we are still discussing **loop-fusion**
 - ▶ And also because loop-splitting is **not** an optimization
- ▶ Loop-splitting is a transformation that extends the usefulness of loop-fusion
 - ▶ Loop-fusion is appropriate only in special cases
 - ▶ Two loops that happen to be of equal size and appear one after the other
 - ▶ Loop-splitting is a transformation that splits the longer of the two loops
 - ▶ We then have a total of 3 loops, of which 2 can be fused

SELF-STUDY Visualizing loop-splitting

First loop iterates less than second (1/2)

Before the split & fusion, the code iterates n times over the body A , followed by $n + m$ times over the body B

Two adjacent loops



Before the split

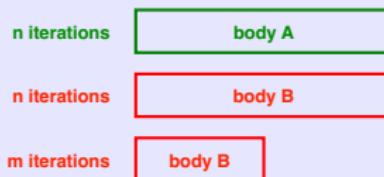


SELF-STUDY Visualizing loop-splitting

First loop iterates less than second (2/2)

After the split & fusion, the code iterates n times over the fused body A, B , followed by m times over the body B

After the split



After fusion



SELF-STUDY Visualizing loop-splitting

First loop iterates more than second (1/2)

Before splitting & fusion, the code iterates $m + n$ times over the body *A*, followed by n times over the body of *B*

Two adjacent loops



Before the split



SELF-STUDY Visualizing loop-splitting

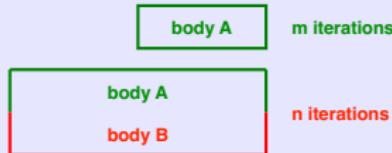
First loop iterates more than second (2/2)

After the split & fusion, the code iterates m times over the body A , followed by n times over the fused body A, B

After the split



After fusion



Example

We start with two loops of unequal size, the bodies of which are independent of each other:

```
for (i = 0; i < 150; ++i) {
    A[i] = 0;
}
for (i = 0; i < 200; ++i) {
    B[i] = i;
}
```

Example

We split the larger of the two into two loops, giving a total of three loops:

```
for (i = 0; i < 150; ++i) {  
    A[i] = 0;  
}  
  
for (i = 0; i < 150; ++i) {  
    B[i] = i;  
}  
  
for (; i < 200; ++i) {  
    B[i] = i;  
}
```

Example

We merge the bodies of the two loops of equal size:

```
for (i = 0; i < 150; ++i) {
    A[i] = 0;
    B[i] = i;
}
for (; i < 200; ++i) {
    B[i] = i;
}
```

Limitations

- ▶ Loop-fusion is language-dependent:
 - ▶ In some languages (C/C++, Fortran, etc), it is possible for arrays to **overlap**:
 - ▶ For example:

```
int A[105];
int *B = &A[5];
/* == A + 5 by pointer arithmetic! */
```
- ▶ Loop-fusion is language-dependent (*cont*):
 - ▶ Arrays can overlap dynamically, i.e., at run-time, in ways about which the compiler cannot know

Limitations (*cont*)

- ▶ In languages that allow for arrays to overlap, before the compiler can perform an optimization such as loop-fusion, it will need to prove that either no overlapping occurs in this particular case, or else that this overlap does not change the result
- 👉 If the compiler is unable to prove that an optimization is safe/valid, it cannot perform it!

Limitations (*cont*)

- 👉 If the optimization is safe, and the compiler is unable to prove this, and the optimization is critical for a performance bottleneck in the code, then the programmer can:
 - ▶ Change the code so that the compiler can prove the optimization is safe/valid
 - ▶ Perform the optimization manually (not recommended in general)

Roadmap

- ✓ Understanding the significance of loop-optimizations
- 👉 Understanding some of the main loop-optimizations
 - ✓ Factoring out expressions that are invariant in a loop
 - ✓ Loop-fusion
 - 👉 Loop-unrolling
 - ▶ Static branch prediction
 - ▶ Dynamic branch prediction
 - ▶ Software pipelining
 - ▶ The decline of loop-optimizations

SELF-STUDY Loop-unrolling

Loop-unrolling duplicates the body of a loop so as

- ▶ To eliminate the loop altogether (for small loops)
- ▶ To reduce the number of times a loop executes (for large loops)
 - 👉 This enlarges the body of the loop
- ▶ There are two reasons to perform loop-unrolling:
 - ① Eliminate or reduce the cost of the administration of the loop
 - ② Add opportunities to parallelize the body of the loop

Recurring administrative costs

Consider a loop with a small body:

- ▶ The cost of administration represents a higher ratio of the cost of the loop than in a loop with a larger body
- ▶ Negligible as it may seem, the cost of administration can be multiplied by the number of times this loop is performed
- ▶ This is especially significant for nested loops in which the inner loop has a small body and runs a small number of times

Example

Consider the following code:

```
for (i = 0; i < 1000000; ++i) {  
    for (j = 0; j < 2; ++j) {  
        A[i][j] = 0;  
    }  
}
```

- ▶ The inner loop runs twice
- ▶ The body of the inner loop is small
- 👉 It might be better just to duplicate the body and eliminate the inner loop

Adding parallelism

Consider the loop

```
for (i = 0; i < 1000000; ++i) {  
    A[i] = 0;  
}
```

- ▶ The body of the loop consists of a **single assignment statement**
- ▶ A naive translation of this loop into assembly language will not utilize the capabilities for Instruction-level Parallelism (ILP) in a modern microprocessor

SELF-STUDY Loop-unrolling (*cont*)

Adding parallelism (*cont*)

Starting with the original code:

```
for (i = 0; i < 1000000; ++i) {  
    A[i] = 0;  
}
```

Suppose we unroll the loop once:

```
for (i = 0; i < 1000000; i += 2) {  
    A[i] = 0;  
    A[i + 1] = 0;  
}
```

The body of the new loop consists of **two instructions** that can both be done in parallel

SELF-STUDY Loop-unrolling (*cont*)

Adding Parallelism (*cont*)

Of course, nothing prevents us from unrolling the loop 3 times, giving us a body of a four instructions that can be performed in parallel:

```
for (i = 0; i < 1000000; i += 4) {  
    A[i] = 0;  
    A[i + 1] = 0;  
    A[i + 2] = 0;  
    A[i + 3] = 0;  
}
```

The only hard limit we need to consider is the **width** of the pipeline

 There is no point in unrolling a loop beyond what the hardware can perform in parallel

SELF-STUDY Loop-unrolling (*cont*)

The general structure of the loop is as follows:

```
for ( <loop administration> ) {  
    <Body> <induction variable>  
}
```

- 👉 Notice that the **body of the loop** is parameterized by the **induction-variable** of the loop
 - ▶ This means that statements within the body will make use of this variable, and depend on its value at the n -th iteration
- 👉 Unrolling the loop will create a body that will combine expressions parameterized by the induction variable **at different iterations**
 - ▶ This may require more variables

SELF-STUDY Loop-unrolling (*cont*)

Suppose we unroll the loop three times:

```
for (i = 0; i < 1000000; ++i) {  
    ⟨Body⟩i  
}
```

We shall get

```
for (i = 0; i < 1000000; i += 4) {  
    ⟨Body⟩i  
    ⟨Body⟩i+1  
    ⟨Body⟩i+2  
    ⟨Body⟩i+3  
}
```

SELF-STUDY Loop-unrolling (*cont*)

```
for (i = 0; i < 1000000; i += 4) {  
    ⟨Body⟩i  
    ⟨Body⟩i+1  
    ⟨Body⟩i+2  
    ⟨Body⟩i+3  
}
```

The thing is that $i + 1, i + 2, i + 3$ are not variables in the language, but expressions!

- ▶ If the body doesn't **change** the value of i , then we can get away with **substituting** $(i + k)$ for i in $\langle\text{Body}\rangle_{i+k}$, for each k
 - ◀ This may involve re-computing $(i + k)$ several times!
- ▶ If $\langle\text{Body}\rangle_i$ changes i , then we need the **location** of i in addition to its **value**

SELF-STUDY Loop-unrolling (*cont*)

```
for (i = 0; i < 1000000; i += 4) {  
    ⟨Body⟩i  
    ⟨Body⟩i+1  
    ⟨Body⟩i+2  
    ⟨Body⟩i+3  
}
```

But $i + 1, i + 2, i + 3$ are not variables in the language, but expressions!

- ▶ Alternately, we can define variables i_k to represent the k -th unrolled version of the induction-variable i

SELF-STUDY Loop-unrolling (*cont*)

The loop becomes

```
for (i0 = 0, i1 = 1, i2 = 2, i3 = 3;  
     i0< 1000000;  
     i0 += 4, i1 += 4, i2 += 4, i3 += 4) {  
    <Body>i0  
    <Body>i1  
    <Body>i2  
    <Body>i3  
}
```

 The induction-variables in loops are ordinarily mapped to **registers**, to give better performance. How will loop-unrolling affect register-allocation in register-impooverished architectures, such as the Intel x86?

Conclusion

Loop-unrolling —

- ▶ ...can be full or partial
 - ▶ Full unrolling eliminates the loop altogether
 - ▶ Partial unrolling ($k - 1$)-times results in a loop that executes $1/k$ times as the original
- ▶ ...grows the body of the loop in proportion to how many times it was unrolled
 - ▶ This adds parallelism
 - ▶ This can consume registers quickly

Roadmap

- ✓ Understanding the significance of loop-optimizations
- 👉 Understanding some of the main loop-optimizations
 - ✓ Factoring out expressions that are invariant in a loop
 - ✓ loop-fusion
 - ✓ Loop-unrolling
 -  Static branch prediction (planned)
 -  Dynamic branch prediction (planned)
 -  Software pipelining (planned)
-  The decline of loop-optimizations (planned)

SELF-STUDY References

- 🔗 The Axioms for Peano Arithmetic — Check out the definition for the natural numbers!
- 🔗 The Axiom Schema for General Recursion (for general recursive functions)
- 🔗 Loop-Optimizations (Wikipedia)
- 🔗 Loop-fusion (Wikipedia)
- 🔗 Loop-Unrolling (Wikipedia)
- 🔗 SIMD
- 🔗 Software-Pipelining

Chapter 6

Roadmap

Code Generation:

- ▶ Constants
- ▶ Symbols & Free Variables
- ▶ The Code Generator

Constants

Perhaps surprisingly, handling constants is rather subtle:

- ▶ Constants are **static**, allocated during **compile-time**, and loaded into memory by the loader
- ▶ Constants **must** be allocated before the user-code executes: It would be an actual error to allocate them at afterwards
 - ▶ It might not be obvious that there is a real and serious issue at hand
 - ▶ The following example should help convince you of this

The anomaly of quote

- ▶ The **anomaly of quote** is the name given to a phenomenon that has to do with the **creation-time** of constants
- ▶ The anomaly was discovered & defined in the world of LISP/Scheme, and hence the name
- ▶ However, the anomaly is not unique to LISP/Scheme, and occurs in many other languages, including C/C++, Python, etc

The anomaly of quote (*continued*)

The procedure `last-pair` takes a list and returns the **last pair** of that list:

```
(define last-pair
  (letrec ((loop
            (lambda (s r)
              (if (pair? r)
                  (loop r (cdr r))
                  s))))
    (lambda (s)
      (loop s (cdr s))))))
```

The anomaly of quote (*continued*)

Here is how to run last-pair:

```
> (last-pair '(1 . 2))  
(1 . 2)  
> (last-pair '(a))  
(a)  
> (last-pair '(a b c))  
(c)
```

The anomaly of quote (*continued*)

Consider the two procedures foo and goo:

```
(define foo
  (lambda ()
    (let ((s (list 'he 'said:)))
      (set-cdr! (last-pair s)
                 (list 'ha 'ha))
      s)))
(define goo
  (lambda ()
    (let ((s '(he said:)))
      (set-cdr! (last-pair s)
                 (list 'ha 'ha))
      s)))
```

The anomaly of quote (*continued*)

Notice the following behaviour:

```
> (foo)  
(he said: ha ha)  
> (eq? (foo) (foo))  
#f
```

The anomaly of quote (*continued*)

Notice the following behaviour:

```
> (goo)
(ha said: ha ha)
> (goo)
(ha said: ha ha ha ha)
> (goo)
(ha said: ha ha ha ha ha ha)
> (eq? (goo) (goo))
#t
```

The anomaly of quote (*continued*)

Why the different behaviour of foo and goo?

- ▶ The key idea is to **trace** the inner loop in last-pair:

```
(define last-pair
  (letrec ((loop
            (trace-lambda last-pair>loop (s r)
              (if (pair? r)
                  (loop r (cdr r))
                  s))))
    (lambda (s)
      (loop s (cdr s))))))
```

The anomaly of quote (*continued*)

```
> (foo)
|(last-pair>loop (he said:) (said:))
|(last-pair>loop (said:) ())
|(said:)
(he said: ha ha)
> (foo)
|(last-pair>loop (he said:) (said:))
|(last-pair>loop (said:) ())
|(said:)
(he said: ha ha)
> (foo)
|(last-pair>loop (he said:) (said:))
|(last-pair>loop (said:) ())
|(said:)
(he said: ha ha)
```

The anomaly of quote (*continued*)

```
> (goo)
|(last-pair>loop (he said:) (said:))
|(last-pair>loop (said:) ())
|(said:)
(he said: ha ha)
> (goo)
|(last-pair>loop (he said: ha ha) (said: ha ha))
|(last-pair>loop (said: ha ha) (ha ha))
|(last-pair>loop (ha ha) (ha))
|(last-pair>loop (ha) ())
|(ha)
(he said: ha ha ha ha)
```

The anomaly of quote (*continued*)

```
> (goo)
|(last-pair>loop (he said: ha ha ha ha) (said: ha ha ha h
|(last-pair>loop (said: ha ha ha ha) (ha ha ha ha))
|(last-pair>loop (ha ha ha ha) (ha ha ha))
|(last-pair>loop (ha ha ha) (ha ha))
|(last-pair>loop (ha ha) (ha))
|(last-pair>loop (ha) ())
|(ha)
.he said: ha ha ha ha ha)
```

The anomaly of quote (*continued*)

Observations

We see immediately that

- ▶ Within `foo` the list remains the same length (2)
- ▶ Within `goo` the list keeps growing by 2 with each application of `goo`, so `last-pair` works more from one call of `goo` to the next...
- ▶ Since `(eq? (foo) (foo))` returns `#f`, we realize that a new list is **created** each time `(foo)` is evaluated, even though the list looks the same from one call to `foo` to the next...
- ▶ Since `(eq? (goo) (goo))` returns `#t`, we realize that the same list is **created returned** each time `(goo)` is evaluated, even though the list looks different from one call to `goo` to the next...

The anomaly of quote (*continued*)

Taking another look at foo & goo:

```
(define foo
  (lambda ()
    (let ((s (list 'he 'said:)))
      (set-cdr! (last-pair s)
                 (list 'ha 'ha))
      s)))
(define goo
  (lambda ()
    (let ((s '(he said:)))
      (set-cdr! (last-pair s)
                 (list 'ha 'ha))
      s)))
```

The anomaly of quote (*continued*)

Conclusion

- ▶ In foo we have (let ((s (list 'he 'said:))) ...)
- ▶ In goo we have (let ((s '(he said:))) ...)
- ▶ Each time we call foo a new list is allocated afresh, and is then extended, so the length remains constant
 - ▶ With each call to foo, the variable s gets assigned a new value, a new address of a newly-allocated list
- ▶ Each time we call goo the same list gets extended further
 - ▶ If the original list exists at address L , the expression (let ((s '(he said:))) ...) keeps re-assigning the [same] value of L to the variable s
- ▶ You might think that the **anomaly of quote** is caused by side-effects. **This is incorrect**, as the next example shall demonstrate...

The anomaly of quote (*continued*)

Consider the following code:

```
(define foo
  (let ((f1 (lambda () '(a b)))
        (f2 (lambda () (list 'a 'b))))
    (lambda ()
      (list (g f1)
            (g f2)
            (g f1)
            (g f1)
            (g f2)))))

(define g
  (lambda (f)
    (if (eq? (f) (f))
        'statically-allocated-lists
        'dynamically-allocated-lists)))
```

The anomaly of quote (*continued*)

We now run foo:

```
> (foo)
(statically-allocated-lists
 dynamically-allocated-lists
 statically-allocated-lists
 statically-allocated-lists
 dynamically-allocated-lists)
```

- ▶ There are no side-effects in either foo or g
- 👉 Notice that g has no difficulties in distinguishing statically-allocated data from dynamically-allocated data!

The anomaly of quote (*continued*)

This “anomaly” affects C/C++ code as well:

- ▶ When you define a string as `char *s = "hello";` you've performed an **implicit casting** (!)
 - ▶ The type of "hello" is `const char *`
 - ▶ The type of s is `char *`
 - ▶ You just told the compiler not to worry about it...
- ▶ Just like LISP/Scheme, C/C++ will let you **intermix** static and dynamic data **freely**
- ▶ But the static data has been marked `.section .rodata` by `gcc`
 - ▶ `.rodata` means **read only**
 - ▶ `.rdata` means **read or write**
- ▶ If you try to **change** your mixed data,
 - ▶ changes to the dynamic data will work just fine
 - ▶ changes to the static data will generate a **segmentation fault**

The anomaly of quote (*continued*)

This “anomaly” affects C/C++ code as well:

- ▶ If you try to compile your code with `-g` (for debugging) and single-step through it in the debugger, your program will likely perform correctly to the end... 😊
- ⚠ Debuggers are written to trace, inspect, & debug programs
 - ▶ The debugger will **ignore** the page-protection bits and load your data into pages that have permission bits set to read, write, & execute, to give you the maximum flexibility...
 - ▶ So your data is **read only** in the shell, but **read, write, execute** within the debugger...
 - ▶ This is one situation in which the debugger will not reflect the normal execution environment of your program (!)

The anomaly of quote (*continued*)

This “anomaly” affects C/C++ code as well:

- ▶ This bug is “the monster that **really** does live under the bed!”
 - ▶ It’s **never** there when you inspect (using a debugger)
 - ▶ But you just know it’s there!
- ▶ The **only** way to fix this bug is to look for **places in your code where static & dynamic data intermix!**



The anomaly of quote in Python (*continued*)

From the Python Pocket Ref (1/2)

Function defaults and attributes

Mutable default argument values are evaluated once at `def` statement time, not on each call, and so can retain state between calls. However, some consider this behavior to be a caveat, and classes and enclosing scope references are often better state-retention tools; use `None` defaults for mutable and explicit tests to avoid unwanted changes, as shown in the following's comments:

The anomaly of quote (*continued*)

From the Python Pocket Ref (2/2)

```
>>> def grow(a, b=[]):      # def grow(a, b=None):
...     b.append(a)          #     if b == None: b = []
...     print(b)              #     ...
...
>>> grow(1); grow(2)
[1]
[1, 2]
```

The anomaly of quote (*continued*)

Conclusion

- ▶ Allocation-time is crucial
- ▶ Constants must be created/allocated before program-execution
- ▶ Side-effects on constants are defined to be undefined in most programming languages
- ▶ While testing for **identity** (e.g., using `eq?`) **is not a side-effect**, it can, just like side-effects, expose issues related to allocation-time and data-sharing, and result in code with undefined semantics!
- ▶ The anomaly of quote occurs in **any** programming language that permits a mix of statically and dynamically allocated memory
 - 👉 This is not “merely” a problem in Scheme or C!

The anomaly of quote (*continued*)

Further reading

-  The Anatomy of LISP, by *John Allen*

The constants-table

- ▶ The constants-table is a **compile-time data structure**:
 - ▶ It exists until your compiler is done generating code
 - ▶ It does not exist when the [generated] code is running
- ▶ The constants-table serves several purposes:
 - ▶ Lays out constants where constants shall reside in memory when your program executes
 - ▶ Helps to pre-compute the **locations** of the constants in your program
 - ▶ The locations are needed to lay out other constants in memory (constants that contain other constants)
 - ▶ The locations are needed during code-generation
 - ▶ Constants are compiled into a **single** `mov` instruction
 - ▶ The size/depth/complexity of the constant are of no significance
 - ▶ The run-time behaviour for constants is always the same, always efficient

The constants-table (*continued*)

Issue: Constants can be nested

- ▶ A sub-constant is also a constant
 - ▶ It must be allocated at compile-time
 - ▶ Its address needs to be known at compile-time
- ▶ Relevant data types:
 - ▶ Pairs
 - ▶ Vectors
 - ▶ Symbols
 - ▶ Symbols are special; we shall discuss symbols later

Example in C: The linked list (4 9 6 3 5 1)

```
typedef struct LL {  
    int value;  
    struct LL *next;  
} LL;  
const LL c1 = {1, (struct LL *)0};  
const LL c51 = {5, &c1};  
const LL c351 = {3, &c51};  
const LL c6351 = {6, &c351};  
const LL c96351 = {9, &c6351};  
const LL c496351 = {4, &c96351};
```

- 👉 The constants c1, c51, c351, c6351, c96351 are all sub-constants of c496351
 - ▶ They need to be defined, laid out in memory, and their address known **before** we can define c496351

The constants-table (*continued*)

Issue: Sharing sub-constants

- ▶ Since constants are, by definition, immutable, we can save space by **factoring out & sharing** common sub-constants:
- ▶ That side-effects on constants are undefined means that we cannot assume any specific behaviour when performing side-effects on them
- ▶ This gives us license to share sub-constants
- ▶ Most Scheme implementations **do not** factor-out & share sub-constants
- 👉 Our implementation **shall** factor-out & share sub-constants

The constants-table (*continued*)

Interactivity

- ▶ Most Scheme systems are **interactive**
- ▶ Interactivity is not the same as being interpreted
 - ▶ Chez Scheme is interactive
 - ▶ Chez Scheme has no interpreter
 - ▶ Expressions are compiled & executed on-the-fly

The constants-table (*continued*)

Interactivity

- ▶ Interactive systems are conversational
 - ▶ The “conversation” takes place at the REPL
 - ▶ REPL stands for Read-Eval-Print-Loop
 - ▶ **Read:** Read an expression from an input channel (scan, read, tag-parse)
 - ▶ **Eval:** Compute the value of the expression, either by interpreting it, or by compiling & executing it on-the-fly
 - ▶ **Print:** Print the value of the expression (unless it's #<void>)
 - ▶ **Loop:** Return to the start of the REPL
 - ▶ The REPL executes until the end-of-file is reached or the (exit) is evaluated

The constants-table (*continued*)

Interactivity (*continued*)

- ▶ Interactive systems need to create constants on-the-fly, as expressions are entered at the REPL
- ▶ Creating constants on-the-fly is not conducive to sharing sub-constants:
 - ▶ There would be a great performance penalty to “looking up constants” at run-time
 - ▶ Some constants would/should be garbage-collected, which would make this process imperfect
- ▶ So interactive systems do not factor & share constants

The constants-table (*continued*)

Interactivity (*continued*)

- 👉 But we are not writing an interactive compiler!
 - ▶ Writing interactive compilers requires generating machine-code on-the-fly, which is harder than generating and writing assembly-instructions (which are just text) to a text file
 - ▶ Interactive compilers are harder to debug, since we cannot invoke a system debugger (such as *gdb*) on an executable
 - ▶ While interactive compilers are fun, the exercise would be time-consuming, and would not offer a great added benefit to the course

The constants-table (*continued*)

- ▶ We are writing an offline/batch compiler
 - ▶ It's not conversational
 - ▶ We see all the source code at compile-time
 - ▶ We won't be implementing the load procedure
 - ▶ So code cannot be loaded during run-time
 - ▶ This would require the compiler to be available during run-time too, which would be about as difficult as writing an interactive compiler
 - ▶ In particular, we get to see **all** the constants in our code, ahead of time
 - ▶ So it makes sense that we factor/share sub-constants

The constants-table (*continued*)

Constructing the constants-table

- ① Scan the AST (one recursive pass) & collect the sexprs in all ScmConst records
 - ▶ The result is a list of sexprs
- ② Convert the list to a set (removing duplicates)
- ③ Expand the list to include all sub-constants
 - ▶ The list should be sorted **topologically**
 - ▶ Each sub-constant should appear in the list **before** the const of which it is a part
 - ▶ For example, (2 3) should appear before (1 2 3)
- ④ Convert the resulting list into a set (remove all duplicates, again)

The constants-table (*continued*)

Constructing the constants-table

- ⑤ Go over the list, from first to last, and create the constants-table:
 - ① For each sexpr in the list, create a 3-tuple:
 - ▶ The address of the constant sexpr
 - ▶ The constant sexpr itself
 - ▶ The **representation** of the constant sexpr as a list of bytes
 - ② The first constant should start at address zero (0)
 - ▶ The TAs will instruct you how to make use of this address in your code
 - ③ The constant sexpr is used as a **key** for **looking up** constants in the constants-table
 - ④ The representation of a constant is a list of numbers:
 - ▶ Each number is a byte, that is, a non-negative integer that is less than 256

The constants-table (*continued*)

Constructing the constants-table

- ⑤ Go over the list, from first to last, and create the constants-table:
- ⑤ As you construct the constants-table, you shall need to consult it, in its intermediate state, to look-up the addresses of sub-constants
 - ▶ The list of 3-tuples contains all the information needed to lookup & extend the constants-table

The constants-table (*continued*)

How the constants-table is used

- ① The representations of the constants initialize the memory of your program
 - ▶ They are laid out in memory by the code-generator
 - ▶ They are allocated in assembly-language by the compiler
 - ▶ They are assembled into data stored in a **data section**
 - ▶ They are loaded by the **system loader** when you run your program
 - ▶ They are available in memory before the program starts to run
- ② The addresses of the constants are used to determine the representation of other constants that contain them
- ③ The addresses of the constants are used by the code-generator to create and issue the `mov` instructions that evaluate the constants at run-time

The constants-table (*continued*)

How/where sharing of sub-constants takes place

- ▶ When constructing the constants-table, we twice converted lists to sets, i.e., **removed duplicates**
- ▶ This means that for any constant sexpr \mathfrak{S} will appear **only once** in the constants-table
 - ▶ All sexprs that contain \mathfrak{S} will use the same address of the one and only occurrence of the constant sexpr \mathfrak{S}
 - ▶ So \mathfrak{S} has been “factored out” of all constant sexprs in which it appears, and is shared by all of them

The constants-table (*continued*)

You still need some information...

The code to generate the constants-table is straightforward to write, but please don't start on it just yet. The TAs will give you some additional information:

- ▶ The TAs will give you the layout, i.e., the schema for representing the various constants in memory
- ▶ In particular, you need to know how to encode the RTTI for the various data types
- ▶ For Strings, Pairs, Vectors, you need to know how to handle sub-constants
- ▶ Symbols are complicated (will be covered later on)

Chapter 6

Roadmap

Code Generation:

- ✓ Constants
- ▶ Symbols & Free Variables
- ▶ The Code-Generator

Symbols & Free Vars

- ▶ Just as the implementation of constants is different in **interactive systems** vs **batch** systems, the implementation of **symbols & free variables** is different too
- ▶ The implementation of symbols & free variables in Scheme (and similar languages) developed over decades, and is by now a fundamental aspect of these languages, so understanding the implementation is essential
 - ▶ We first consider how symbols & free variables are implemented in a standard, interactive system
 - ▶ Then we consider how batch systems are different
 - ▶ Finally, we detail what you should implement in your system

Symbols & Free Vars (*continued*)

Interactive Systems

- ▶ Symbols are **hashed strings**
 - ▶ The **hash table** is also known as the **symbol table**
 - ▶ Each symbol has a **representative string** that serves as a **key** and is known as its **print name**
 - ▶ To see the print-names, use the procedure **symbol->string**:
`> (symbol->string 'moshe)
"moshe"`

Symbols & Free Vars (*continued*)

Interactive Systems (*continued*)

- ▶ Symbols are hashed strings
 - ▶ Dr Racket returns a **duplicate** of the representative string
 - ▶ Chez Scheme returns the **exact, identical string**
 - ▶ This is one area where Chez's behaviour is problematic:
 - ▶ If the original representative string is returned, it can be modified using `string-set!`
 - ▶ This shall render the symbol **inaccessible**:
 -  The hash function maps the symbol to the modified string
 -  This is recognized as a **collision**
 -  Handling this collision will resolve to a different bucket!
 -  This [mis-]behaviour was intentional in Chez, and was used as a hackish way to “hide” data. Today it’s an unnecessary anachronism...

Symbols & Free Vars (*continued*)

Interactive Systems (*continued*)

- ▶ In interactive Scheme, new symbols are added all the time as new expressions get typed at the REPL or loaded from files
 - ▶ The scanner is in charge of
 - ▶ Recognizing the symbol token
 - ▶ Hashing the symbol string to obtain a **bucket** (whether original or pre-existing)
 - ▶ Creating the symbol object: A symbol is a tagged object containing the **address of the corresponding bucket**
 - ▶ The bucket contains 2 cells:
 - ▶ The **print cell**, pointing to the **representative string**
 - ▶ The **value cell**, holding the **value** of the **free variable** by the same name

Symbols & Free Vars (*continued*)

Interactive Systems (*continued*)

- ▶ The symbol-table serves two purposes:
 - ▶ Managing the symbol data structure as a collection of hashed strings
 - ▶ Managing the global-variable bindings via the top-level
- ▶ These two purposes may appear unrelated, but, in fact, they are closely related in interactive systems:
 - ▶ Every free variable was once a symbol...
 - ▶ Every symbol is hashed
 - ▶ Free variables and symbols can be loaded during run-time

The value-cell

- ▶ The view of the symbol-table across the dimension of the value cells is known as **the top-level**
 - ▶ The top-level holds the **global bindings** in Scheme
 - ▶ For example, the procedures car, cdr, cons, and other builtins are defined at the top-level
 - ▶ Modern versions of Scheme (R⁶RS) & modern dialects of LISP come with **namespaces**, **packages**, **modules**, as ways of aggregating groups of functions and variables
 - ▶ The top-level, in such systems, is just a system namespace that is exported by default

- ▶ Scheme buckets come with a name-cell and a value-cell
- ▶ Some dialects of LISP come with more cells
 - ▶ A value-cell & and a function-cell
 - ▶ Such systems are known as **2-LISP** systems, because beyond the name-cell, they contain 2 additional cells
 - ▶ In this sense, Scheme is a **1-LISP**

n-LISP (*continued*)

What does it mean to have a value-cell & function-cell?

(x x)

- ▶ The same variable name can refer both to a procedure and a value
 - ▶ This does not mean you cannot store a procedure in a value cell
 - ▶ To apply a procedure in a value cell, you need to use the procedure `funcall`
 - ▶ To obtain the closure in the function-cell (to be passed as data), you need to use the special form `function` (which has the reader-macro form `#'`)

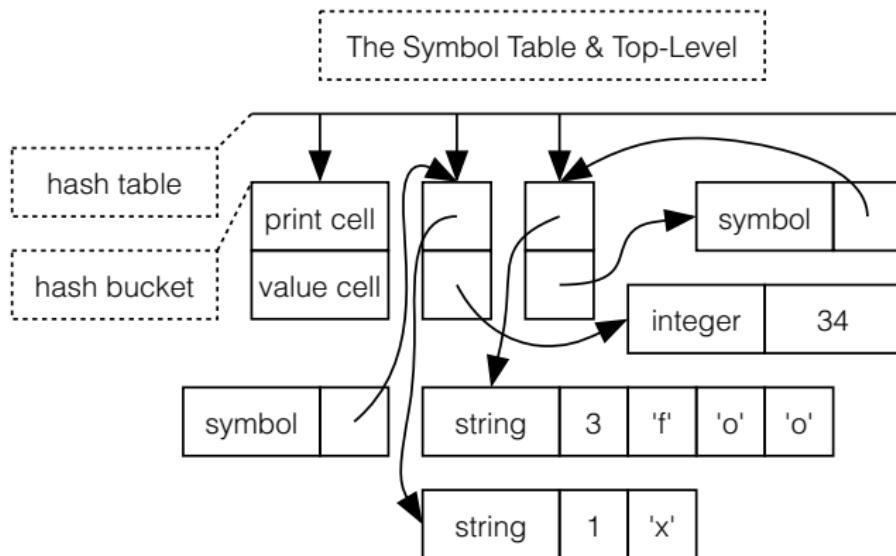
n-LISP (*continued*)

- ▶ What are the advantages of 2-LISP languages?
 - 👉 There are **NONE!**
 - ▶ A long time ago, some people thought the ability to overload a name adds power to the language
- ▶ So why bother with 2-LISP languages??
 - ▶ Well, there's this hardly-known, esoteric, programming language by the name of **Perl**, which is a 5-LISP language... 😊
 - ▶ Every name in Perl can be used for
 - ▶ A function
 - ▶ A scalar
 - ▶ An array
 - ▶ A hash table
 - ▶ A file handle
 - ▶ So knowing about this nonsense might reduce your learning curve if you ever need to learn Perl!

Symbols & Free Vars (*continued*)

Part of the symbol-table & top-level for the code:

```
> (define foo 'foo)  
> (define x 34)
```



Symbols & Free Vars (*continued*)

- ▶ There is a strict grammar for literal symbols, but **any** string can become the print-name for a symbol:

```
> (string->symbol "a234")
a234
> (string->symbol "A234")
A234
> (string->symbol "A 234")
A\x20;234
> (string->symbol "this is a bad symbol!")
this\x20;is\x20;a\x20;bad\x20;symbol!
```

Symbols & Free Vars (*continued*)

- ▶ Because a symbol can be created from any string, symbols that do not resemble literal strings are printed in **peculiar ways**, using hexadecimal characters, **so as to avoid confusion**
- ▶ The `string->symbol` procedure may be thought of as the API to the hash function for the symbol-table
- ▶ As of R⁶RS, Scheme supports hash tables as first-class objects, so programmers may use them as freely as **dictionaries** in other languages

Symbols & Free Vars (*continued*)

- ▶ When a new symbol is hashed, a bucket for it is created, and the initial value is #<undefined> (a special object that signifies that the global variable hasn't been defined & holds no value)
- ▶ Global variables are defined by means of the `define-expression`
 - ▶ Attempts to assign an undefined variable is defined to be an **error**, although Chez Scheme is tolerant of this, and tacitly defines the variable before setting it
 - ▶ Re-defining a variable changes its value & is somewhat similar to `set!`

Symbols & Free Vars (*continued*)

- ▶ When expressions are read, either at the REPL, or from a file, either in textual or compiled form, each `expr` is **first** read as an `sexpr`:
 - ▶ At this stage, symbols are hashed & the corresponding symbol objects are created
 - ▶ If, upon parsing, such a symbol turns out to denote a free variable, the variable cell is accessible for definition/set/get via the hash bucket of the symbol
 - ▶ Thus free variable access is but a pointer dereference away from the symbol & the hash bucket

Symbols & Free Vars (*continued*)

- ▶ Because symbols are hashed by the scanner, it makes no sense to ask whether a given symbol is in the symbol-table: The answer is always affirmative.
- ▶ The list of print-names from the symbol-table is available via the procedure `oblist`:

```
> (oblist)
(entry-row-set! entry-col-set! entry-screen-cols-set!
 ...
entry-top-line asmop-add entry-bot-line
record-constructor
entry-mark Effect)
> (length (oblist))
9420
```

Symbols & Free Vars (*continued*)

- ▶ Symbols for which there are buckets in the hash-table are said to be **interned symbols** (in the sense that they are internal to the hash table)
- ▶ Another kind of symbols are the **uninterned symbols**, which are a special kind of symbols that are not hashed
 - ▶ The vast majority of symbols in the system are interned and hashed
 - ▶ Uninterned symbols are a hack that was added intentionally to break the definition of symbols:
 - ▶ Uninterned symbols are used in situations where we require a **fresh symbol** that does not appear anywhere in the system, and is not equal to any other symbol (in the sense of `eq?`)
 - ▶ Such symbols are used when we need **unique names**, such as for variable names in **hygienic macro-expanders**
 - ▶ Uninterned symbols are created by means of the procedure `gensym`, and are also known as *gensyms*

Symbols & Free Vars (*continued*)

Uninterned symbols are supported via the following API:

- ▶ `gensym` generates uninterned symbols, usually with numbered names, such as `g1`, `g2`, `g3`, etc
- ▶ The `symbol?` predicate returns `#t` for an uninterned symbol
- ▶ The `eq?` predicate returns `#f` whenever an uninterned symbol is compared to anything but itself:
 - ▶ Either an interned symbol, including a symbol by the same name:

```
> (gensym)  
g0  
> (eq? 'g1 (gensym))  
#f
```

- ▶ Or another *gensym*:
- ```
> (eq? (gensym) (gensym))
#f
```

# Symbols & Free Vars (*continued*)

Uninterned symbols are supported via the following API:

- ▶ The `symbol->string` procedure will generate a string of the form "g1", "g2", etc., that may look like the one generated for an uninterned symbol:

```
> (list (symbol->string 'g1)
 (symbol->string (gensym)))
("g1" "g1")
```
- ▶ Uninterned symbols can be identified via the `gensym?` procedure

# Symbols & Free Vars (*continued*)

## Our implementation of symbols

- ▶ The implementation of symbols is simplified by the fact that ours is a **static compiler**, and therefore symbols **shall not be loaded** during run-time
- ▶ To simplify matters further, you **should not implement** the procedure `string->symbol`, so that new symbol objects cannot be created, at run-time, from strings
- ▶ This means that **all** symbols in our system are **static, literal constants**, and this simplifies matters considerably
- ▶ All symbols shall have the respective representative string as a **sub-constant**
  - 👉 This affects the way you construct the constants-table

# Symbols & Free Vars (*continued*)

## Our implementation of symbols

- ▶ The symbol data-structure is a tagged data structure that points to the representative string, which itself is a tagged, constant, string-object
  - ▶ The eq? procedure should compare the **address fields** of the two symbol objects
  - ▶ The symbol->string procedure shall create and return a **copy** of the representative string of the symbol

# Symbols & Free Vars (*continued*)

## Our implementation of free variables

- ▶ Just as before, our implementation of free variables is simplified by the fact that ours is a **static compiler**, so free variables are **not loaded** during run-time
- ▶ Global variables in our system are not much more than names that serve as shorthand for assembly-language **labels** that point to global storage in the **data section**
- ▶ Your goal is to create a **free-variables-table** to serve during the code-generation phase of the compiler pipeline

# Symbols & Free Vars (*continued*)

## Our implementation of free variables (*continued*)

- ▶ Just as you collected a list of constants, by traversing the AST of the user-code, so you collect a list of strings that are the names of all the free variables that occur in the AST of the user code
- ▶ Create a set from the above list of strings, by removing duplicate strings
- ▶ Create a list of **pairs**, based on the above set, by associating with each name-string of a free variable a unique, indexed string for a **label** in the x86/64 assembly language: "v1", "v2", "v3", etc

# Symbols & Free Vars (*continued*)

## Our implementation of free variables (*continued*)

- ▶ The list of pairs is the **free-variables-table**:
  - ▶ This table **must** be available to the code-generator
  - ▶ Here is how the code-generator uses it:
    - ▶ For a **get** to a free variable, the code-generator issues a `mov` instruction **from** the respective label/variable  $v_n$
    - ▶ For a **set** to a free variable, the code-generator issues a `mov` instruction **to** the respective label/variable  $v_n$

# Chapter 6

## Roadmap

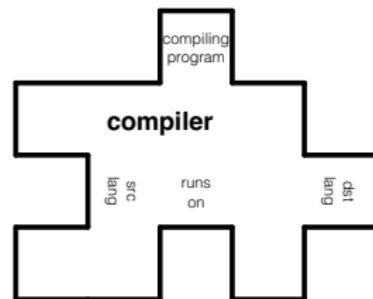
### Code Generation:

- ✓ Constants
- ✓ Symbols & Free Variables
- The Code Generator

# Code Generation



Compilör\*



\*Some assembly required

# Code Generation

- ▶ The code-generator is a function  $\text{expr}' \rightarrow \text{string}$ 
  - ▶ We look at  $\text{expr}'$  **after** the semantic analysis phase is complete
  - ▶ After the constants-table and free-vars-table have been set up
- ▶ The string returned is x86/64 assembly language code, line by line...

# Code Generation (*continued*)

## Assumptions about the code-generator

We make several assumptions concerned our code-generator, that we shall have to satisfy:

- ▶ **Notation:** The notation  $\llbracket \cdot \rrbracket$  stands for the code-generator
- ▶ **The induction hypothesis of the code-generator:** For any expression  $\mathcal{E}$ ,  $\llbracket \mathcal{E} \rrbracket$  is a string of instructions in x86/64 assembly language, that evaluate  $\mathcal{E}$ , and place its value in register `rax`
  - ▶ We need this assumption to convince ourselves that for each node in the AST of  $\text{expr}'$ , we generate code that has the correct behaviour
  - ▶ The relative correctness of each part of the code-generator is then combined to form a proof of correctness for the entire code-generator, and consequently, for the compiler

# Code Generation (*continued*)

## Assumptions about the code-generator (*continued*)

- ▶ The calling conventions on the x86/64 architecture specify that the first 6 non-floating-point arguments are passed through 6 general-purpose registers, 8 floating point arguments are passed through 8 SSE registers, and any additional arguments are passed on the system-stack.
  - ▶ This calling convention is very nice for C, because most procedures take far less than 6 arguments
  - ▶ This calling convention is not very nice for Scheme, because of the extensive use of `apply`, variadic procedures & procedures with optional arguments, and the relatively little use of floating-point numbers

# Code Generation (*continued*)

## Assumptions about the code-generator (*continued*)

- ▶ The calling conventions on the x86/64 architecture are known as the **Application Binary-Interface**, or ABI
- ▶ The 64-bit ABI specifies that the first 6 non-floating-point arguments are passed through 6 general-purpose registers, 8 floating point arguments are passed through 8 SSE registers, and any additional arguments are passed on the system-stack.
- 👉 We shall use the ABI-conventions when calling system-functions, as well as functions written in C (e.g., `printf`)

# Code Generation (*continued*)

## Assumptions about the code-generator (*continued*)

- 👉 The code we generate shall not adhere to the Linux 64-bit ABI **for calling our own code**, but shall use the system-stack, organized into activation frames, as presented in the diagrams and pseudo-code in these slides, to pass all the arguments, regardless of their number, type, & size
  - ▶ Using the stack makes more sense for languages such as Scheme, that support procedures that can be called with arbitrarily-many arguments
- 💣 The x86/64 calling conventions are OS-dependent, and are **slightly** different between Linux, Windows, and OS X.
- 👉 For this project you **must** use the Linux conventions!

# Code Generation (*continued*)

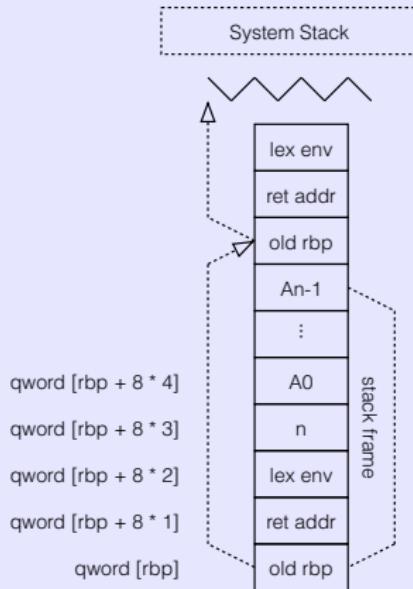
## Assumptions about the code-generator (*continued*)

- ▶ We shall assume the availability of four singleton, literal constants, that are always present in the run-time system, even if they are not present statically in the user-code (they are required for the support libraries):
  - ▶ The void object #<void>
    - ▶ located at label `sob_void`
  - ▶ The empty list ()
    - ▶ located at label `sob_nil`
  - ▶ The Boolean value *false* #f
    - ▶ located at label `sob_false`
  - ▶ The Boolean value *true* #t
    - ▶ located at label `sob_true`

# Code Generation (*continued*)

## Assumptions about the code-generator (*continued*)

- We assume the following structure for all activation frames:



# Code Generation (*continued*)

## Assumptions about the code-generator (*continued*)

- ▶ We shall assume there is **always** at least one activation frame
  - ▶ This means that the code-generator assumes that we are within the body of some lambda-expression that has been applied. For example, within the body of a null let-expression:  
`(let () ... )`
  - ▶ We will need to support/maintain this assumption by setting up an initial dummy-frame at the start of the program

# Code Generation (*continued*)

We shall now go over each of the nodes in the `expr'` AST, and describe in pseudo-code, what the code-generator returns for each and every node.

- 👉 Please note that the code-generator shall intermix **static** and **dynamic** strings
  - ▶ Static strings shall be noted in **black**
  - ▶ Dynamic strings shall be noted in **red**

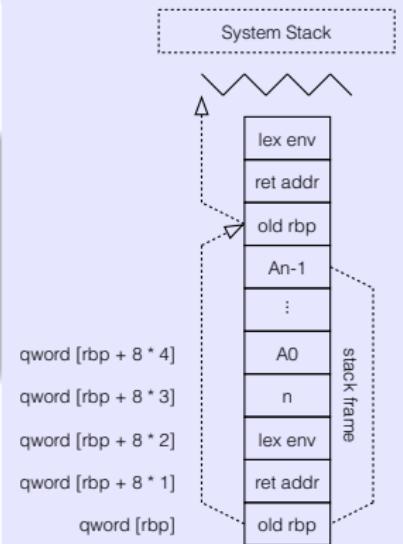
# Code Generation (*continued*)

## Constants

$\llbracket \text{Const}'(\text{c}) \rrbracket$

$= \text{mov rax, AddressInConstTable}(\text{c})$

## The frame

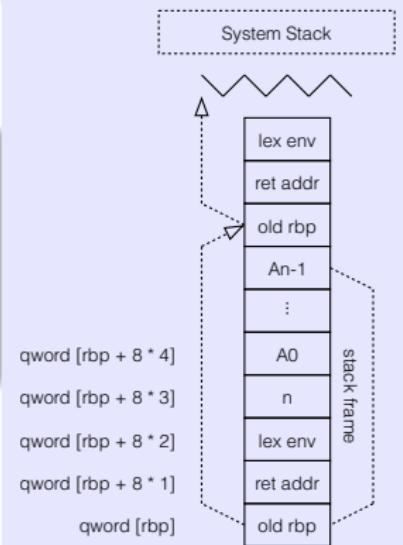


# Code Generation (*continued*)

## Parameters / get

```
[[Var' (VarParam' (_ ,minor))]]
= mov rax, qword [rbp + 8 * (4 + minor)]
```

## The frame



# Code Generation (*continued*)

## Parameters / set

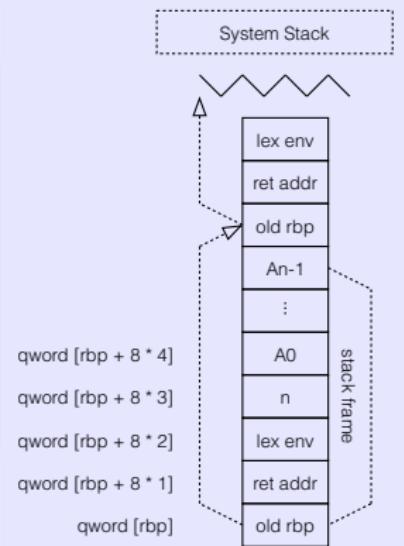
`[[Set(Var' (VarParam' (_ ,minor)) , E)]]`

= `[[E]]`

`mov qword [rbp + 8 * (4 + minor)], rax`

`mov rax, sob_void`

## The frame

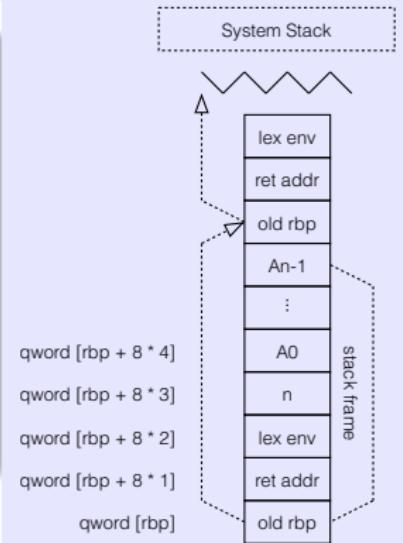


# Code Generation (*continued*)

## Bound vars / get

```
[[Var' (VarBound' (_ ,major ,minor))]]
= mov rax, qword [rbp + 8 * 2]
 mov rax, qword [rax + 8 * major]
 mov rax, qword [rax + 8 * minor]
```

## The frame



# Code Generation (*continued*)

## Bound vars / set

```
[[Set' (Var' (VarBound' (_,
 major,
 minor)),

 ε)]] =

[[ε]]
```

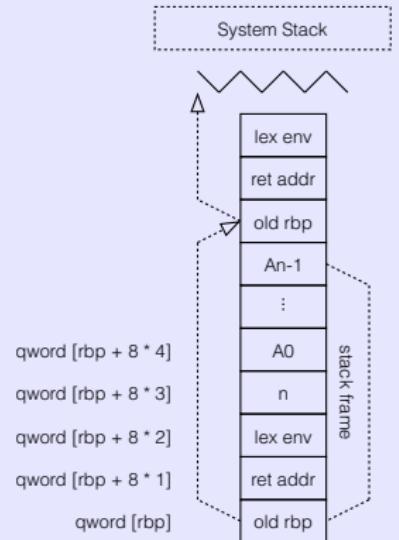
```
mov rbx, qword [rbp + 8 * 2]

mov rbx, qword [rbx + 8 * major]

mov qword [rbx + 8 * minor], rax

mov rax, sob_void
```

## The frame



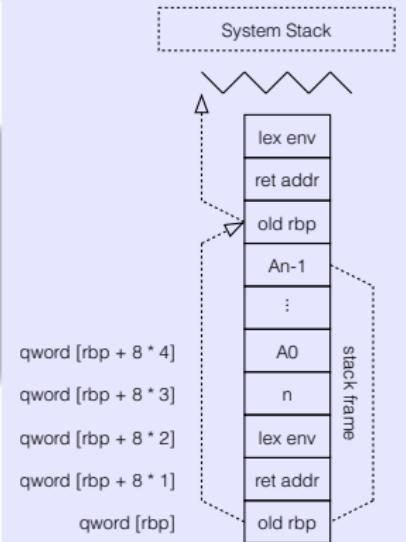
# Code Generation (*continued*)

Free vars / get

[[Var' (VarFree' (v))]]

= mov rax, qword [LabelInFVarTable(v)]

## The frame



# Code Generation (*continued*)

## Free vars / set

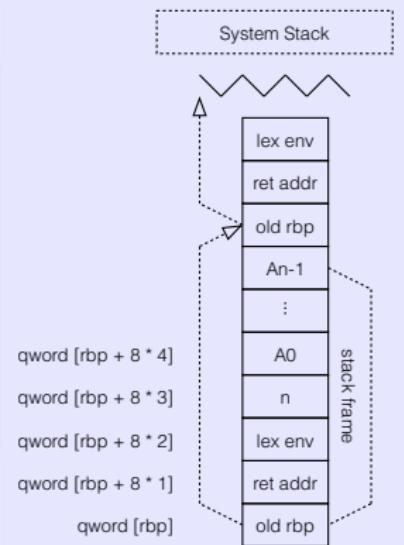
$\llbracket \text{Set}(\text{Var}'(\text{VarFree}'(v)), \mathcal{E}) \rrbracket$

$= \llbracket \mathcal{E} \rrbracket$

`mov qword [LabelInFVarTable(v)], rax`

`mov rax, sob_void`

## The frame

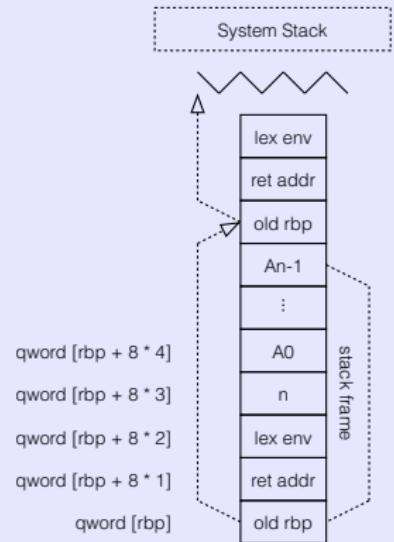


# Code Generation (*continued*)

## Sequences

$$\begin{aligned} & \llbracket \text{Seq}([\mathcal{E}_0; \mathcal{E}_1; \dots; \mathcal{E}_{n-1}]) \rrbracket \\ &= \llbracket \mathcal{E}_0 \rrbracket \\ &\quad \llbracket \mathcal{E}_1 \rrbracket \\ &\quad \dots \\ &\quad \llbracket \mathcal{E}_{n-1} \rrbracket \end{aligned}$$

## The frame



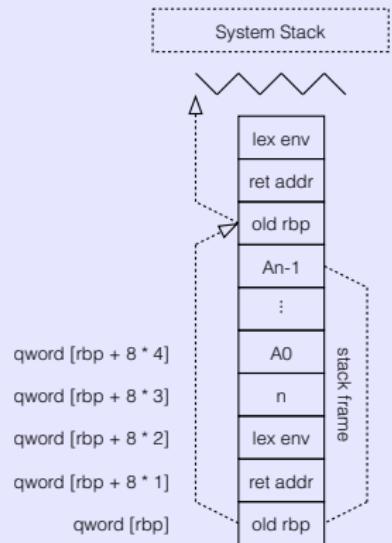
# Code Generation (*continued*)

Or

```
[[Or'([E0; E1; ...; En-1])]]
= [[E0]]
 cmp rax, sob_false
 jne Lexit
 [[E1]]
 cmp rax, sob_false
 jne Lexit
 ...
 [[En-1]]

Lexit:
```

The frame



# Code Generation (*continued*)

Or

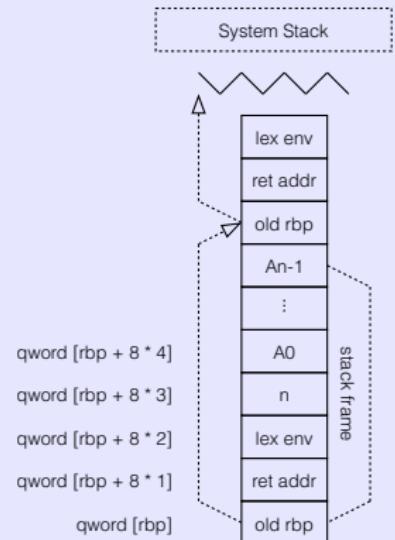
- ▶ In our pseudo-code, we use the label Lexit
- ▶ In our project, we must **index** our labels, to make them unique:
  - ▶ Lexit<sup>1</sup>, Lexit<sup>2</sup>, Lexit<sup>3</sup>, etc

# Code Generation (*continued*)

If

```
[[If'($\mathcal{Q}, \mathcal{T}, \mathcal{E}$)]]
= [[\mathcal{Q}]]
 cmp rax, sob_false
 je Lelse
 [[\mathcal{T}]]
 jmp Lexit
Lelse:
 [[\mathcal{E}]]
Lexit:
```

## The frame



# Code Generation (*continued*)

## If

- ▶ In our pseudo-code, we use two labels Lelse, Lexit
- ▶ In our project, we must **index** each pair of labels, to make them unique, and so they match a specific instance of an if-expression:
  - ▶ Lelse1, Lexit1
  - ▶ Lelse2, Lexit2
  - ▶ Lelse3, Lexit3
  - ▶ etc

# Code Generation (*continued*)

## Boxes

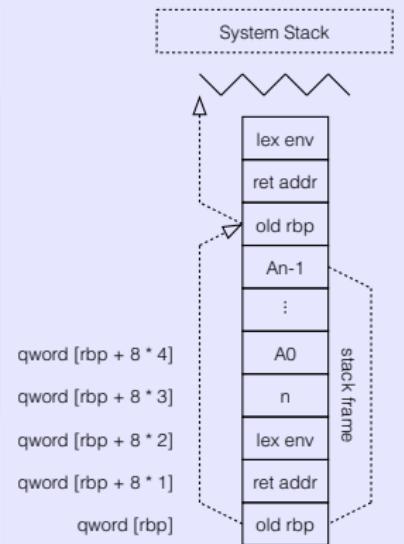
- ▶ Boxes provide one extra level of indirection to the value
- ▶ Boxes can be implemented as untagged arrays of size 1
  - ▶ That they are untagged means that boxes do not contain RTTI
  - ▶ This is probably the simplest implementation, but nevertheless, only one of many possible implementations

# Code Generation (*continued*)

Box / get

```
[[BoxGet' (Var' (v))]]
= [[Var' (v)]]
 mov rax, qword [rax]
```

## The frame

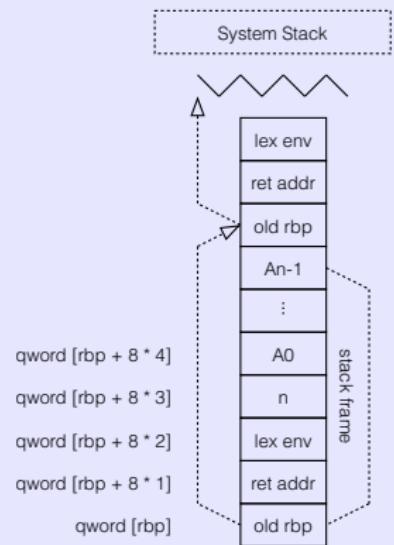


# Code Generation (*continued*)

## Box / set

```
[[BoxSet' (Var' (v) , E)]] =
 [[E]]
 push rax
 [[Var' (v)]]
 pop qword [rax]
 mov rax, sob_void
```

## The frame



# Code Generation (*continued*)

## LambdaSimple

$\llbracket \text{LambdaSimple}'([p_0; \dots; p_{m-1}], \text{body}) \rrbracket$  in pseudo-code:

- ▶ Allocate the ExtEnv (the size of which is known statically, and is  $1 + |\text{Env}|$ )
- ▶ Copy pointers of minor vectors from Env (on the stack) to ExtEnv (with offset of 1):

```
for (i = 0, j = 1; i < |Env| ;
 ++i, ++j) {
 ExtEnv[j] = Env[i];
}
```

## Outline

Closure-Creation Code

Create ExtEnv

Allocate closure object

Closure → Env == ExtEnv  
Closure → Code == Lcode  
jmp Lcont

Lcode:  
push rbp  
mov rbp, rsp  
[ body ]  
leave  
ret

Lcont:

closure body

# Code Generation (*continued*)

## LambdaSimple (*continued*)

$\llbracket \text{LambdaSimple}'([\text{p}_0; \dots; \text{p}_{m-1}], \text{body}) \rrbracket$  in pseudo-code:

- ▶ Allocate ExtEnv[0] to point to a vector where to store the parameters
- ▶ Copy the parameters off of the stack:

```
for (i = 0; i < n; ++i)
 ExtEnv[0][i] = Parami;
```

- ▶ Allocate the closure object; Address in `rax`
- ▶ Set `rax → env = ExtEnv`
- ▶ Set `rax → code = Lcode`
- ▶ `jmp Lcont`

## Outline

Closure-Creation Code

Create ExtEnv

Allocate closure object

Closure → Env == ExtEnv  
Closure → Code == Lcode  
`jmp Lcont`

Lcode:  
push rbp  
mov rbp, rsp  
[ body ]  
leave  
ret

Lcont:

closure body

# Code Generation (*continued*)

## LambdaSimple (*continued*)

$\llbracket \text{LambdaSimple}'([p_0; \dots; p_{m-1}], \text{body}) \rrbracket$  in  
pseudo-code:

► Lcode :

```
push rbp
mov rbp, rsp
 $\llbracket \text{body} \rrbracket$
leave
ret
```

Lcont :

## Outline

Closure-Creation Code

Create ExtEnv

Allocate closure object

Closure → Env == ExtEnv  
Closure → Code == Lcode  
jmp Lcont

Lcode:  
push rbp  
mov rbp, rsp  
 $\llbracket \text{body} \rrbracket$   
leave  
ret

closure body

Lcont:

# Code Generation (*continued*)

## LambdaSimple (*continued*)

- ▶ During the **creation** of the closure, we perform only the code in blue
- ▶ During the **application** of the closure, only the code in orange executes
- ▶ The code in orange is embedded within the code in blue
  - ▶ This makes our code-generator **compositional**
    - ▶ We can combine the output of the code-generator
    - ▶ The downside is that we need to pay with the `jmp Lcont` instruction

## Outline

Closure-Creation Code

Create ExtEnv

Allocate closure object

`Closure → Env == ExtEnv`  
`Closure → Code == Lcode`  
`jmp Lcont`

`Lcode:`  
`push rbp`  
`mov rbp, rsp`  
`[ body ]`  
`leave`  
`ret`

`Lcont:`

closure body

# Code Generation (*continued*)

## LambdaSimple (*continued*)

- ▶ We could have saved the `jmp Lcont` instruction at the expense of making our code-generator non-compositional
  - ▶ We could not have combined the output of the code-generator
  - ▶ We would have needed some place, “out of the way” where to place the code generated for the **body** of a lambda-expression, so that the normal program-flow would not reach it by mistake, and it would not execute unless the closure was applied

## Outline

Closure-Creation Code

Create ExtEnv

Allocate closure object

Closure → Env == ExtEnv  
Closure → Code == Lcode  
`jmp Lcont`

Code:  
`push rbp`  
`mov rbp, rsp`  
[ body ]  
`leave`  
`ret`

closure body

Lcont:

# Code Generation (*continued*)

## LambdaSimple (*continued*)

- ▶ In our pseudo-code, we use two labels Lcont, Lcode
- ▶ In our project, we shall use several additional labels to manage all the tests and loops we specified in the pseudo-code. We must **index** each of those labels, to make them unique, and so they match a specific instance of a lambda-expression:
  - ▶ Lcont`1`, Lcode`1`, etc
  - ▶ Lcont`2`, Lcode`2`, etc
  - ▶ Lcont`3`, Lcode`3`, etc
  - ▶ etc

# Code Generation (*continued*)

## Application

$\llbracket \text{Applic}'(\text{proc}, [\text{Arg}_0; \dots; \text{Arg}_{n-1}]) \rrbracket$  in pseudo-code:

$\llbracket \text{Arg}_{n-1} \rrbracket$

push rax

:

$\llbracket \text{Arg}_0 \rrbracket$

push rax

push  $n$

$\llbracket \text{proc} \rrbracket$

Verify that rax has type closure

push rax  $\rightarrow$  env

call rax  $\rightarrow$  code

# Code Generation (*continued*)

## Application (*continued*)

$\llbracket \text{Applic}'(\text{proc}, [\text{Arg}_0; \dots; \text{Arg}_{n-1}]) \rrbracket$  in pseudo-code:

```
add rsp, 8*1 ; pop env
pop rbx ; pop arg count
lea rsp, [rsp + 8*rbx] ; or alternately:
; shl rbx, 3 ; rbx = rbx * 8
; add rsp, rbx ; pop args
```

- ▶ Notice that upon return, we consult the argument count on the stack before popping off the arguments
  - ▶ This takes into account the fact that the number we need to pop might be different from the number originally pushed:
    - ▶ lambda-expressions with optional arguments
    - ▶ The tail-call optimization

# Code Generation (*continued*)

## Outline

### Lambda with optional args

$\llbracket \text{LambdaOpt}'([\text{p}_0; \dots; \text{p}_{m-1}], \text{opt}, \text{body}) \rrbracket$  in  
pseudo-code:

- ▶ The code is essentially the same as for `LambdaSimple'`
- ▶ The difference occurs in the body of the procedure, i.e., at `Lcode`

ClosureOpt-Creation Code

Create ExtEnv

Allocate closure object

Closure  $\rightarrow$  Env := ExtEnv  
Closure  $\rightarrow$  Code := Lcode  
Imp Lcont

Lcode:

Adjust stack for  
opt args

push rbp  
mov rbp, rsp  
[ body ]  
leave  
ret

closure body

Lcont:

# Code Generation (*continued*)

## Lambda with optional args

`[[LambdaOpt'([p0; ⋯ ; pm-1], opt, body)]]` in  
pseudo-code:

Lcode :

Adjust the stack for the  
optional arguments

```
push rbp
mov rbp, rsp
[[body]]
leave
ret
```

Lcont :

## Outline

ClosureOpt-Creation Code

Create ExtEnv

Allocate closure object

Closure → Env := ExtEnv  
Closure → Code := Lcode  
Imp Lcont

:Lcode:

Adjust stack for  
opt args

```
push rbp
mov rbp, rsp
[body]
leave
ret
```

closure body

:Lcont:

# Code Generation (*continued*)

## Lambda with optional args

- ▶ Suppose  
`(lambda (a b c . d) ... )` is applied to the arguments 1, 1, 2, 3, 5, 8
- ▶ Six arguments are passed
- ▶ The body of the procedure expects 4 arguments
  - ▶ The last argument, d, is supposed to point to the list (3 5 8)
- ▶ The stack needs to be adjusted
  - ▶ Notice that the number of arguments must change too!

## Outline

The stack as it is

The stack as expected



# Code Generation (*continued*)

## Lambda with optional args

- ▶ Suppose  
`(lambda (a b c . d) ...)` is applied to the arguments 1, 1, 2
- ▶ Three arguments are passed
- ▶ The body of the procedure expects 4 arguments
  - ▶ The last argument, d, is supposed to point to the empty list ()
- ▶ The stack needs to be adjusted
  - ▶ Notice that the number of arguments must change too!

## Outline

The stack as it is:

The stack as expected

|     |     |
|-----|-----|
| 2   | 0   |
| 1   | 2   |
| 1   | 1   |
| 3   | 1   |
| env | 4   |
| ret | env |
|     | ret |

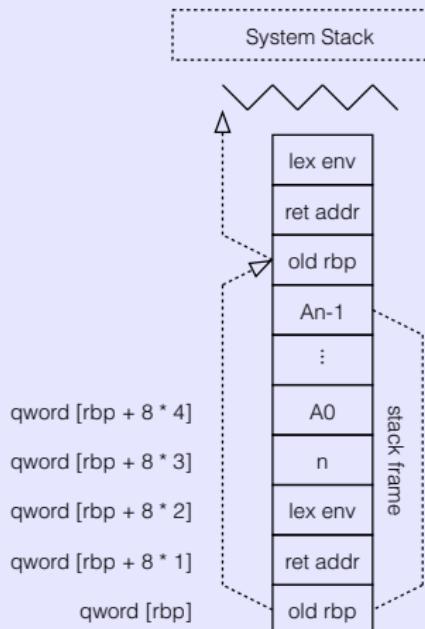
# Code Generation (*continued*)

## Lambda with optional args

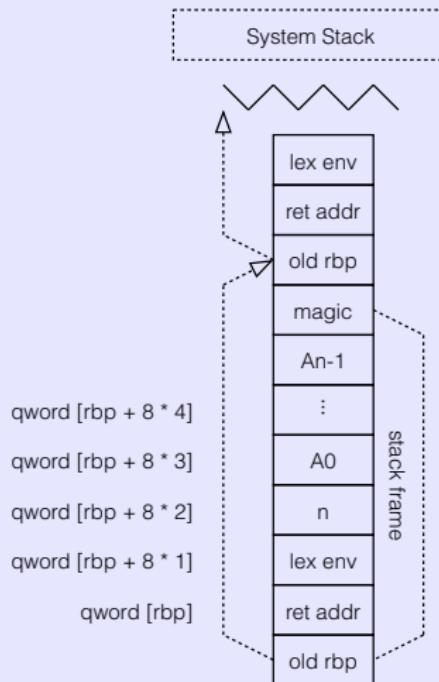
- ▶ As you can see
  - ▶ Sometimes we need to **shrink** the top frame
  - ▶ Sometimes we need to **enlarge** the top frame by one
    - ▶ When the number of arguments matches precisely the number of **required parameters**, there is no room in the frame to place the empty list
    - ▶ We shift the contents of the frame down by one [8-byte] word to make room for opt
- ▶ We can test during run-time and decide what to do
  - ▶ This is the **basic approach**
- ▶ We can also use **magic** to save us from having to test and shift down... 
  - ▶ To use **magic**, we need to change the structure of our activation frame:

# Code Generation (*continued*)

## Without Magic



## With Magic



# Code Generation (*continued*)

## Lambda with optional args with magic

- ▶ Suppose  
`(lambda (a b c . d) ...)` is applied to the arguments 1, 1, 2, 3, 5, 8
- ▶ Six arguments are passed
- ▶ The body of the procedure expects 4 arguments
  - ▶ The last argument, d, is supposed to point to the list (3 5 8)
- ▶ The stack needs to be adjusted
  - ▶ Notice that the number of

## Outline

The stack as it is:

The stack as expected



# Code Generation (*continued*)

## Lambda with optional args **with** magic

- ▶ Suppose  
`(lambda (a b c . d) ...)` is applied to the arguments 1, 1, 2
- ▶ Three arguments are passed
- ▶ The body of the procedure expects 4 arguments
  - ▶ The last argument, d, is supposed to point to the empty list ()
- ▶ The stack needs to be adjusted
  - ▶ Notice that the number of arguments must change too!

## Outline

The stack as it is:

The stack as expected

|       |     |
|-------|-----|
| magic | ()  |
| 2     | 2   |
| 1     | 1   |
| 1     | 1   |
| 3     | 3   |
| env   | env |
| ret   | ret |

# Code Generation (*continued*)

## Lambda with optional args (*continued*)

To summarize:

- ▶ Using **magic** means reserving a word at the start of **each** frame
  - ▶ **All** frames grow by one word, regardless of whether or not they belong to procedures with optional arguments!
  - ▶ We do not include **magic** in the **argument count** on the stack!
- ▶ If you choose to use **magic** you need to remember to remove it from the frame after returning from an application
- 👉 You are free to use either the **basic approach** or **magic**, depending on your taste/style...

# Code Generation (*continued*)

## Tail-call applications

$\llbracket \text{ApplicTP}'(\text{proc}, [\text{Arg}_0; \dots; \text{Arg}_{n-1}]) \rrbracket$  in pseudo-code:

$\llbracket \text{Arg}_{n-1} \rrbracket$

push rax

:

$\llbracket \text{Arg}_0 \rrbracket$

push rax

push  $n$

$\llbracket \text{proc} \rrbracket$

Verify that rax has type **closure**

push rax  $\rightarrow$  env

push qword [rbp + 8 \* 1] ; old ret addr

push qword [rbp] ; the same old rbp

Fix the stack (see next slide)

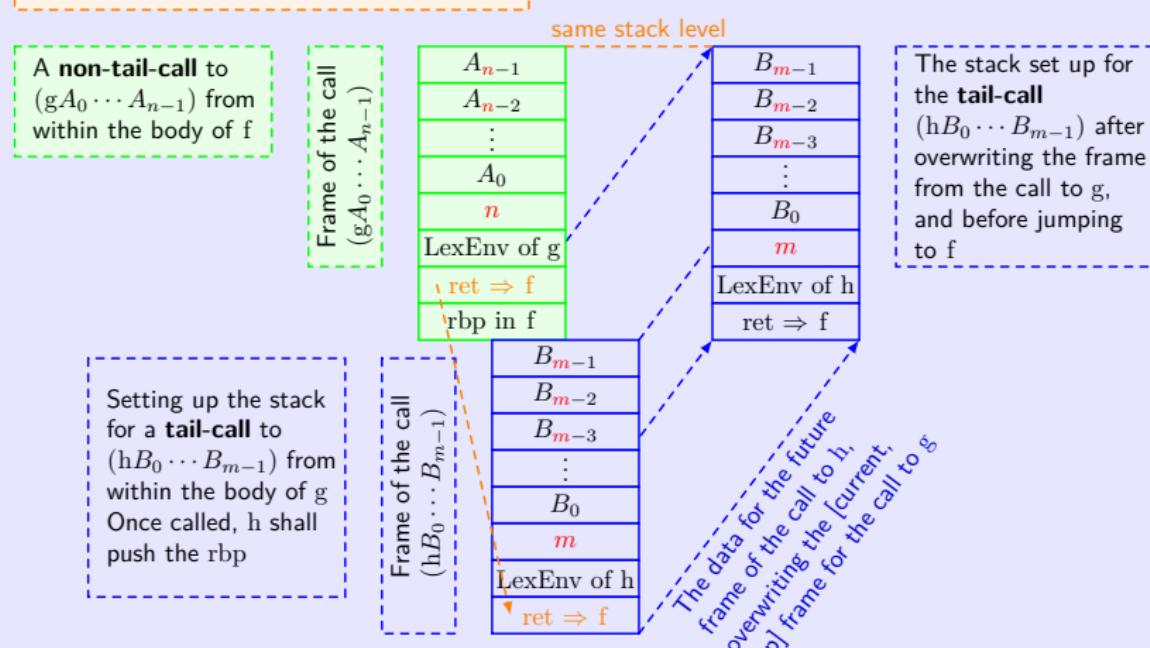
pop rbp ; restore the old rbp

jmp rax  $\rightarrow$  code

# Code Generation (*continued*)

## Tail-call applications — fixing the stack

Setting up the stack for a **tail-call** to h



# Code Generation: A summary

- ▶ The code-generator is a recursive function  $\text{expr}' \rightarrow \text{string}$
- ▶ The return string is an assembly-language code-fragment
- ▶ To convert this fragment into a standalone program, we need to “sandwich it” between two pieces of assembly-language code known as the **prologue** and the **epilogue**:
  - ▶ The **prologue**
    - ▶ defines the various sections
    - ▶ lays out **constants** in the data section
    - ▶ lays out the **free variables** in the data (bss) section
    - ▶ sets up the **initial dummy-frame**
    - ▶ calls the user-code
  - ▶ The **epilogue** contains the code for the **primitive procedures** (`car`, `cdr`, `cons`, `+`, etc)

# Builtins

- ▶ The builtin procedures are the procedures that come with the system
- ▶ There are two kinds of builtin procedures:
  - ▶ Low-level builtins, or **primitives**, which are low-level system-procedures that must be hand-coded in assembly language
    - ▶ Such procedures include `car`, `pair?`, `apply` and others
    - 👉 We shall supply you a complete list of primitives you shall need to support in your compiler
  - ▶ Higher-level builtins, which are procedures that could either be hand-coded, or written in Scheme and compiled
    - ▶ Such procedures include `map`, `length`, `list`, etc
    - ▶ We shall supply you with a Scheme source file containing definitions of procedures we would like you to compile using your compiler, and make available to your users

# Builtins (*continued*)

## The apply builtin

- ▶ The general format of the apply procedure is  
 $(apply\ proc\ x_0 \dots x_{n-1}\ s)$ 
  - ▶  $s$  is a proper list
  - ▶  $x_0 \dots x_{n-1}$  is a possibly-empty sequence of Scheme expressions
  - ▶  $proc$  is an expression the value  $c$  of which is a closure
- ▶ Let  $w \equiv '(,x_0 \dots ,x_{n-1} ,@s)$ . The closure  $c$  should be such that it can be applied to the arguments in  $w$ , both in terms of their number and types
- ▶ The closure  $c$  is applied to the arguments in  $w$  in **tail-position**
- ▶ An implementation of apply **must** duplicate the frame-recycling copy that takes place during code-generation for ApplicTP'

# How to work on the the final project

- ▶ Start with template code for the code-generator. It should do nothing but raise a *not-yet-implemented*-exception for any input
- ▶ Compose the procedures in the assignments so far:
  - ▶ You are given code for opening and reading a textfile
  - ▶ Open the Scheme source file, read in the text
  - ▶ Apply the reader to the list of characters
    - ▶ Your grammar should be for  $\langle \text{sexpr} \rangle^*$
    - ▶ You should have a usable `read_sexprs` procedure for doing just that
  - ▶ Map over the list of sexprs a procedure that
    - ▶ tag-parses the sexpr
    - ▶ perofrms semantic analysis on the parsed-expression resulting in a list of `expr`'
- ▶ Build the constants-table & free-variable-table

## How to work on the final project (*cont*)

- ▶ Apply the code-generator to each `expr'` in the list
  - ▶ Append to each of the resulting strings a call to an x86/64 subroutine that
    - ▶ examines the contents of `rax`, and
    - ▶ print the Scheme object if not `void`
- ▶ Catenate all the strings together; This is your `code-fragment`
- ▶ Sandwich the code-fragment between a prologue and epilogue (defined above)
- ▶ Write the resulting string into a text-file using the code we shall supply you
- ▶ Then run `nasm` to assemble the assembly file
- ▶ Then run `gcc` to link the assembly file
  - ▶ If you link with linux standard C libraries, you will find it very difficult to link with `ld`, because of the many libraries used
  - ▶ You are using `gcc` as a “smart linker” ☺

## How to work on the final project (*cont*)

- ▶ Run the executable and examine the output to *stdout*
  - 👉 You would do well to automate the creation of the executable by means of a *makefile*
- ▶ By the end of all these steps, you will have **completed the cycle**
  - ▶ You can now go from Scheme source code to a linux executable
  - ▶ The only problem is that your code generator supports **nothing** yet!

# How to work on the final project (*cont*)

You are now ready to work on the code generator:

- ▶ Add support for constants, and test!
- ▶ Add support for Seq', and test!
- ▶ Add support for If', and test!
  - ▶ Test support for and (which macro-expands to nested-if-expressions)
- ▶ Add support for Or', and test!
- ▶ Add support for defining/setting/getting free variables, and test!
- 👉 **Temporarily**, remove the annotation of tail-calls from the semantic-analysis phase. This will guarantee that you generate no ApplicTP' records

## How to work on the final project (*cont*)

- ▶ Add support for LambdaSimple'
- ▶ Add support for Applic', and test thoroughly!
- ▶ Implement some primitives and test thoroughly!
  - ▶ Start with type-predicates such as pair?, null?, number?, zero?, etc
  - ▶ Throw in car, cdr, cons, etc
- 👉 Re-introduce the annotation of tail-calls into the semantic-analysis phase. This will cause tail-calls to be tagged with the ApplicTP' records
- ▶ Add support for ApplicTP', and test thoroughly!
- ▶ Add support for LambdaOpt', and test!
- ▶ Add support for the rest of the primitives, and test thoroughly!

# How to work on the final project (*cont*)

- ▶ Include the Scheme code we shall provide you, and test thoroughly!
  - ▶ When you read a Scheme source file, be sure to append it to the string containing the Scheme code we provide
  - ▶ This will make it appear that the Scheme code we provide was part of your test file
  - ▶ Your compiler will compile all the code together
- ▶ By now you have a working compiler! Congratulations!
  - ▶ Run your compiler at the linux labs
  - ▶ Make sure everything builds properly
- ▶ Submit your compiler according to the instructions we provide
- 👉 Start studying for the final exam! 😊

# How to work on the final project (*cont*)

## What you **may not** do

- ▶ Share/show code to others
- ▶ Include code from students from previous years, from your friends, from the internet
- ▶ Slack off, and leave most/all of the work to your partner
- ▶ Make your code public (e.g., put on it github)

# How to work on the final project (*cont*)

## What you may do

- ▶ Share tests with your classmates
- ▶ Share scripts to automate testing with your classmates
- ▶ Test after the tiniest changes/additions to your code
- ▶ Gloat, boast, be proud, & happy when your compiler finally works! 🎉



# Further reading

## Roadmap

### Parameter-Passing Mechanisms

- ▶ Introduction
- ▶ The mechanisms we consider
  - ▶ Call-by-Value
  - ▶ Call-by-Reference
  - ▶ Call-by-Sharing / Call-by-Object
  - ▶ Call-by-Name
  - ▶ Call-by-Need

# Parameter-Passing Mechanisms

## Introduction

- ▶ Parameter-Passing Mechanisms (PPMs) are mechanisms for communicating & sharing arguments during a call:
  - ▶ **Call:** A function, procedure, or method call
  - ▶ **Caller:** A point in the code where a call is made
  - ▶ **Callee:** The target of the call — The function, procedure, or method being called
- ▶ In short, PPMs are mechanisms that control how arguments are passed
- ▶ If you are familiar with 1-2 programming languages, you may not have seen many PPMs, and might not be aware of the variety of different mechanisms

# Parameter-Passing Mechanisms

## Introduction (*continued*)

- ▶ Some mechanisms are important —
  - ▶ ...for historical reasons
  - ▶ ...for theoretical reasons
  - ▶ ...because they are unique, have interesting properties, are subtle, come with a steep learning-curve, etc
  - ▶ ...appear in important programming languages
  - ▶ ...are often misunderstood by language developers, and so are implemented incorrectly, or idiosyncratically (!)
- 👉 When studying a programming language, it is **crucial** that you understand the PPMs available in it

# Parameter-Passing Mechanisms

## Introduction (*continued*)

- ▶ In this course, we shall focus on the most commonly-known PPMs:
  - ▶ Call-by-Value (CBV)
  - ▶ Call-by-Reference (CBR)
  - ▶ Call-by-Sharing / Call-by-Object (CBS/CBO)
  - ▶ Call-by-Name (CBName)
  - ▶ Call-by-Need (CBNeed)
- ▶ Many more PPMs exist; The above are just the most basic, commonly-used, and important
- ▶ Each PPM was invented as a way to address a special need, issue, problem:
  - 👉 When you study PPMs, it is important to be clear on what need, issue, problem are being addressed, and how

## Roadmap

### Parameter-Passing Mechanisms

- ✓ Introduction
- The mechanisms we consider
  - ▶ Call-by-Value
  - ▶ Call-by-Reference
  - ▶ Call-by-Sharing / Call-by-Object
  - ▶ Call-by-Name
  - ▶ Call-by-Need

# Parameter-Passing Mechanisms

## Call-by-Value (CBV)

You wish to purchase a cellular phone. The seller agrees to your offer of US\$100, and the one remaining question is how to pass him the money. You choose to use **CBV**:

- ▶ Some people write on their banknotes or tear them
- ▶ You wish to avoid damaging your banknotes
- ▶ Therefore, you photocopy a US\$100 banknote and send the copy to the seller
- ▶ Two weeks later, you get in the mail a photo of your new cellular phone...

# Parameter-Passing Mechanisms

## Call-by-Value (*continued*)

- ▶ CBV is a PPM that attempts to restrict the ways by which caller & callee can communicate, by **making it impossible for a change** (pronounced “side effect”) **by the callee to the arguments to be seen by the caller**
- ▶ Such a restriction makes communication better-resemble functions in mathematics:
  - ▶ The caller communicates with the callee via the **arguments**
  - ▶ The callee communicates with the caller via the **return value**
- ▶ Under CBV, it is still possible for caller & callee to communicate via, e.g., global variables

## Call-by-Value (*continued*)

- ▶ CBV is **implemented** using a **deep copy** of the arguments before the callee can see them
  - ▶ Changes by the callee to the arguments affect its copies of the arguments, and are not visible outside the callee
  - ▶ The copies are actually **located on the run-time stack**
    - ▶ They are popped off the stack upon return
    - ▶ No **dynamic memory management** (**garbage-collection** or **reference-counting**) is required

## Call-by-Value — Pros

- ▶ **Safety:** A callee cannot change the state of the caller by modifying its arguments
- ▶ **Closer to mathematics:** Easier to prove properties about the code

# Parameter-Passing Mechanisms

## Call-by-Value — Cons

- ▶ Costly for large arguments: Copying large arguments
  - ▶ ...is wastefully time-consuming
  - ▶ ...consumes stack-space rapidly
- ▶ Very limiting
  - ▶ Example: Sorting a large array
    - ▶ The array cannot be sorted *in-situ*
    - ▶ Passing a large array can be impractical
    - ▶ Returning a large array as the return value from a function is both impractical and not possible in many languages
- ▶ Makes little sense in some situations
  - ▶ Example: Passing by value a graph with pointers between nodes: A deep copy cannot be done in pointers are an exposed data type

## Call-by-Value — Summary

- ▶ Under CBV, callee arguments are evaluated & copied onto the stack
- ▶ Copying is deep... Arbitrarily deep
  - ▶ Passing a multi-dimensional array of structure will take a long time and take up much stack space
- ▶ Impractical in many situations
- ▶ Usually reserved for simple, small, primitive, non-aggregate data types

## Roadmap

### Parameter-Passing Mechanisms

- ✓ Introduction
- ▶ The mechanisms we consider
  - ✓ Call-by-Value
  - ▶ Call-by-Reference
  - ▶ Call-by-Sharing / Call-by-Object
  - ▶ Call-by-Name
  - ▶ Call-by-Need

# Parameter-Passing Mechanisms

## Call-by-Reference (CBR)

You wish to purchase a cellular phone. The seller agrees to your offer of US\$100, and the one remaining question is how to pass him the money. You choose to use **CBR**:

- ▶ You give the person a letter of authorization to your bank, that effectively makes him your partner.
  - ▶ In scenario 1, all goes well, the seller withdraws exactly US\$100 from the account and delivers your phone
  - ▶ In scenario 2, the seller withdraws US\$10,000 from the account and delivers nothing
  - ▶ In scenario 3, you no longer hear from the seller. When you go to the branch office of your bank, you learn that the seller removed all funds and closed the account
- ▶ Welcome to CBR: You are at the mercy of the seller!

# Parameter-Passing Mechanisms

## Call-by-Reference (*continued*)

- ▶ CBR was developed around the same time as CBV, and attempts to avoid some of its limitations [while introducing another ☺]
- ▶ CBR is a way of sharing variables: **Only variables** can be passed by reference
- ▶ When a variable in the caller is passed by reference, the name of the corresponding parameter in the callee becomes an **alias/synonym** for the variable name in the caller
- ▶ This means that **anything** that can be done to/with such a variable in the caller can also be done in the callee

# Parameter-Passing Mechanisms

## Call-by-Reference — Example

The following behavior should be very familiar to you even if you had not previously worked with CBR:

### Caller

```
A = {4, 9, 6, 3, 5, 1};
foo(A);
print_array(A);
{42, 9, 6, 3, 5, 1}
```

### Callee

```
procedure foo(CBR B) {
 B[0] = 42;
}
```

# Parameter-Passing Mechanisms

## Call-by-Reference — Example

The following behavior might seem odd to you:

### Caller

```
A = {4, 9, 6, 3, 5, 1};
foo(A);
print_array(A);
```

{1, 1, 2, 3, 5, 8}

### Callee

```
procedure foo(CBR B) {
 B = {1, 1, 2, 3, 5,
 8};
}
```

## Call-by-Reference (*continued*)

- ▶ When the variable in the caller & parameter in the callee are **aliases**, the callee can change the variable in the caller to point to a different data-structure (assuming the programming language permits assignments of a structure to a variable)
- ▶ This is the behavior of CBR in Algol68, PL/I, Pascal, & many other languages
- ▶ CBR is implemented by transparently passing the address of the variable, so the parameter is **2 pointers away** from the object

## Call-by-Reference — Pros

- ▶ CBR can be implemented efficiently
- ▶ CBR, through aliasing variables, permits arrays and other structures to be shared, changed by the callee, etc. For example, an *in-situ* sort-routine

# Parameter-Passing Mechanisms

## Call-by-Reference — Cons

- ▶ Restrictive: Only variables can be passed by reference
- ▶ Dangerous: While the first example, of changing an element of an array seems perfectly reasonable & familiar, the second example grants the callee a lot of power:
  - ▶ Merely wanting to pass an array to be sorted should not be taken as consent to switch arrays!
  - ▶ The second example illustrates side effects that may be particularly difficult to trace; This can result in bugs that are difficult to understand
- ▶ Bad Style: The second example, taken to an extreme, treats aliased variables as if they were **global**: This can lead to poor style, overly-complex program-logic in the spirit of language where global variables are used excessively: COBOL, Basic, etc.

# Parameter-Passing Mechanisms

## Call-by-Reference — Summary

- ▶ Only variables can be passed by reference, and they become aliases
- ▶ CBR is efficient
- ▶ CBR gives the callee more control over the state of the caller than is ordinarily required or desired
  - ▶ When abused, this extra control will result in complex program-logic that is harder to trace & debug

## Roadmap

### Parameter-Passing Mechanisms

- ✓ Introduction
- ▶ The mechanisms we consider
  - ✓ Call-by-Value
  - ✓ Call-by-Reference
  - ▶ Call-by-Sharing / Call-by-Object
  - ▶ Call-by-Name
  - ▶ Call-by-Need

# Parameter-Passing Mechanisms

## Call-by-Sharing (CBS) / Call-by-Object (CBO)

You wish to purchase a cellular phone. The seller agrees to your offer of US\$100, and the one remaining question is how to pass him the money. You choose to use CBS/CBO:

- ▶ You're happy to hand the seller money, but you might need the money at some future point...
- ▶ You take a US\$100 banknote, and punch 2 holes in it: You tie a piece of cord to each hole. You hand over one cord to the seller, and continue to hold on to the other
- ▶ The idea is that when you need those US\$100, you can always pull on the cord and retrieve the banknote, regardless of who is in its possession
- ▶ The seller hands you a piece of cord tied to his cellular phone...

## Call-by-Sharing / Call-by-Object (*continued*)

- ▶ As long as you continue to hold on to the cord, you may use the phone
- ▶ If you ever drop the phone, you lose access to it
  - ▶ The owner may still continue to hold on to his end of the cord...

# Parameter-Passing Mechanisms

## Call-by-Sharing / Call-by-Object (*continued*)

- ▶ CBS & CBO mean exactly the same thing
  - ▶ The distinction comes from the programming paradigms in the context of which CBS/CBO was invented and re-invented...
- ▶ CBS permits for sharing data structures efficiently (more efficiently than CBR)
- ▶ CBS does not enable the callee to set variables in the caller, as is the case with CBR

# Parameter-Passing Mechanisms

## Call-by-Sharing / Call-by-Object (*continued*)

- ▶ When an expression is passed by sharing/object, it is first evaluated in the caller, and a pointer to the object is then copied onto the stack
- ▶ Access in the callee, through the corresponding parameter automatically & transparently de-references the pointer
- ▶ When an array is passed by sharing/object, its elements can be accessed for set/get, and the effect is visible in the caller
- ▶ When an object is passed by sharing/object, its methods can be called, and any effects are visible in the caller
- ▶ When some datum is passed (e.g., a **pair** in Scheme), its **mutators** (`set-car!`, `set-cdr!`) can change it, and the change is visible in the caller
- ▶ Access for set/get is **1 pointer away** from the datum.

# Parameter-Passing Mechanisms

## Call-by-Sharing / Call-by-Object — Example

The following behavior should be amply familiar to you, e.g., from Java, Scheme, and Python:

### Caller

```
A = {4, 9, 6, 3, 5, 1};
foo(A);
print_array(A);
{42, 9, 6, 3, 5, 1}
```

### Callee

```
procedure foo(CBS B) {
 B[0] = 42;
}
```

# Parameter-Passing Mechanisms

## Call-by-Sharing / Call-by-Object — Example

This is really what you had come to expect in Java, Scheme, and Python:

### Caller

```
A = {4, 9, 6, 3, 5, 1};
foo(A);
print_array(A);
{4, 9, 6, 3, 5, 1}
```

### Callee

```
procedure foo(CBS B) {
 B = {1, 1, 2, 3, 5,
 8};
}
```

# Parameter-Passing Mechanisms

## Call-by-Sharing / Call-by-Object — Pros

- ▶ Efficient
- ▶ Simple to implement
- ▶ Fewer restrictions than with CBR
- ▶ A very sensible way of communicating data between caller & callee
  - ▶ What you would expect; No surprises as with CBR
  - ▶ Simple to understand & reason about
- ▶ Available in all modern programming languages, from functional to object-oriented programming languages

# Parameter-Passing Mechanisms

## Call-by-Sharing / Call-by-Object — Cons

- ☞ It requires **dynamic memory-management** (either **reference-counting** or **garbage-collection**)
  - ▶ This is **probably** why the designer of C++ preferred [a version of] CBR to CBS/CBO
  - ▶ All modern programming languages that come with dynamic memory-management support CBS/CBO

## Call-by-Sharing / Call-by-Object — Summary

- ▶ Changes that **de-reference the parameters** in the callee are visible in the caller
- ▶ Changes that **set the parameters** in the callee **are not visible** in the caller
- ▶ Straightforward, predictable, sensible, and efficient
- ▶ Used commonly, often in conjunction with CBV for primitive types

## Roadmap

### Parameter-Passing Mechanisms

- ✓ Introduction
- ▶ The mechanisms we consider
  - ✓ Call-by-Value
  - ✓ Call-by-Reference
  - ✓ Call-by-Sharing / Call-by-Object
  - ▶ Call-by-Name
  - ▶ Call-by-Need

# Parameter-Passing Mechanisms

## Call-by-Name (CBName)

You wish to purchase a cellular phone. The seller agrees to your offer of US\$100, and the one remaining question is how to pass him the money. You choose to use **CBName**:

- ▶ You take the phone, and hand the seller an IOU ("I-owe-you")
- ▶ When the seller needs some cash, you go off to earn the money:
  - ▶ You help elderly people carry their groceries to their cars
  - ▶ You deliver the daily paper in your neighborhood
  - ▶ You open a lemonade stand...
- ▶ When you have the cash, you pay the seller

## Call-by-Name (*continued*)

- ▶ The seller gets to keep the IOU slip
- ▶ The next time the seller needs some cash, he retrieves the IOU slip and asks you to honor it afresh...
- ▶ If you're not careful, this may end up being the most expensive cellular phone you ever bought!
- ▶ On the other hand, if the owner never asks for the cash, and this will happen from time to time, you don't need to pay for the phone or do any work to earn the money!

# Parameter-Passing Mechanisms

## Call-by-Name (*continued*)

- ▶ CBName is one of the oldest PPMs: It first appeared in Algol60, and has subsequently been removed in the next language standard — Algol68, to be replaced by CBR
- ▶ Since Algol60, no programming language has attempted to include CBName as a PPM
  - ▶ Recently, Scala introduced a broken and incorrect implementation of CBName... [😢]
- ▶ Back in the days of Algol60, CBName was considered:
  - ▶ Ingenious
  - ▶ Difficult to learn & understand
  - ▶ Tricky to implement and use correctly & efficiently
- ▶ It was then replaced cheerfully with CBR, which simpler to use & to implement

# Parameter-Passing Mechanisms

## Call-by-Name — Then why bother??

- ▶ From the standpoint of programming languages theory, CBN is “the correct thing”:
- 💡 Recall the **substitution model** from your PPL course:
  - ▶ The substitution model is how expressions are evaluated in mathematics & logic
  - ▶ The substitution model is “broken” by **side effects & recursion**; This motivated the transition to the environment model
  - ▶ In moving to the environment model, you tacitly changed the evaluation strategy in your interpreters, from **normal-order of evaluation** to **applicative-order of evaluation**
  - ▶ Changing the order of evaluation affected the behavior of your programs with respect to termination & side-effects

# Parameter-Passing Mechanisms

## Call-by-Name — Then why bother?? (*continued*)

- ▶ CBName is the way to switch from the substitution model to the environment model while changing the behavior of programs **as little as possible**
  - ▶ In the absence of any side-effects, your code should behave as if it were being evaluated under normal-order of evaluation
  - ▶ For code with side-effects & recursion, CBName “does the right thing”

# Parameter-Passing Mechanisms

## Call-by-Name — Then why bother?? (*continued*)

- ▶ Even though CBName is not implemented in any current PL (with the exception of an incorrect implementation in Scala), in a functional setting, CBName is the most efficient implementation of the substitution model:
  - ▶ All results in programming languages theory are proven, first under CBName, and only then under CBV — The proofs are simpler & cleaner
  - ▶ CBName is thus fundamental to programming languages theory, even if not implemented in any practical, up-to-date programming language

# Parameter-Passing Mechanisms

## Call-by-Name: Why so difficult to understand & implement?

- ▶ CBName is very different from other PPMs that were used more commonly
- ▶ Theoreticians often fail to understand CBName, because they mainly consider it in purely functional setting, in the complete absence of side-effects
- ▶ As we shall soon see, when CBName & side-effects mix, strange & wonderful interactions take place, and [only] then, CBName becomes tricky to understand
- ▶ In fact, it's simple to implement efficiently. Certain intuitions were simply lacking in back in the 1960's...

# Parameter-Passing Mechanisms

## Call-by-Name: Why isn't all this stuff way too advanced??

- ▶ When it comes to CBName, you (yes, you!) have a distinct advantage over programmers & computer scientists in the 1960's:
  - ▶ CBName is actually simple to understand & implement if explained in terms of object-oriented programming (!)
  - ▶ Object-orientation first appeared in 1971, long after Algol60 has been out-of-favor
  - ▶ So you (yes, you!) have a major, conceptual advantage in learning and in implementing CBName, since you are familiar with object-orientation

# Parameter-Passing Mechanisms

## Call-by-Name (*continued*)

- ▶ When an expression is passed by name, what in fact passes **is a container object with setter & getter methods**, in which the expression
  - ▶ ...is the body of the getter-method
  - ▶ ...appears on the LHS of an assignment-statement in the setter-method
- ▶ Every get-reference to the parameter in the callee is implemented as a call to the getter-method
- ▶ Every assignment to the parameter in the callee is implemented as a call to the setter-method
- ▶ If the specific expression cannot appear on the LHS of an assignment an error is issued; Sometimes at run-time (!)
- ▶ And that's all there is to implementing CBName

## Call-by-Name (*continued*)

CBName has surprising implications:

- ▶ Each time the value of a CBName parameter is queried, the original expression that was passed by name is [re-]evaluated
- ▶ To avoid re-evaluation, simply assign the parameter to a **local variable** within the callee, and query the local variable instead of the parameter

# Parameter-Passing Mechanisms

## Call-by-Name: An example

- ▶ Each time you query dice1 & dice2 within the procedure backgammon, the corresponding expressions are re-evaluated, and new random numbers are generated.
- ▶ These truly are “random variables”

### Caller

```
backgammon(
 1 + random(6),
 1 + random(6)
) ;
```

### Callee

```
procedure backgammon(
 CBName dice1,
 CBName dice2) {
 ...
}
```

# Parameter-Passing Mechanisms

## Call-by-Name: An example

- ▶ A get to B
- ▶ A de-reference of the value of B

### Caller

```
A = {4, 9, 6, 3, 5, 1};
foo(A);
print_array(A);
```

⇒

{42, 9, 6, 3, 5, 1}

### Callee

```
procedure foo(CBName B)
{
 B[0] = 42;
}
```

# Parameter-Passing Mechanisms

## Call-by-Name: An example

- ▶ A set to B

### Caller

```
A = {4, 9, 6, 3, 5, 1};
foo(A);
print_array(A);
{1, 1, 2, 3, 5, 8}
```

### Callee

```
procedure foo(CBName B)
{
 B = {1, 1, 2, 3, 5,
 8};
}
```

# Parameter-Passing Mechanisms

## Call-by-Name (*continued*)

- ▶ As you can see, in the last example, the function `foo` behaves similarly when its parameter is passed by reference or by name
- ▶ Concerning CBR, we pointed out that this level of exposure of a variable in the caller to the code of the callee is dangerous
  - ▶ CBName is **far more** powerful than CBR, but for a number of reasons, the programming languages community is more forgiving when it comes to CBName: *Quod licet Jovi, non licet bovi...*

# Parameter-Passing Mechanisms

## Call-by-Name: Example — The Jensen Device

After the call to voodoo, what is the value of v in the caller?

### Caller

```
var A = {4, 9, 6, 3, 5,
 1};
var i;
var v = voodoo(A[i], i,
 6);
```

### Callee

```
function voodoo(
 CBName x, CBName y,
 CBV z) {
 var s = 0;
 for (y = 0;
 y < z;
 ++y) s += x;
 return s;
}
```

# Parameter-Passing Mechanisms

## Call-by-Name: Example — The Jensen Device

Answer: 28

### Caller

```
var A = {4, 9, 6, 3, 5,
 1};
var i;
var v = voodoo(A[i], i,
 6);
```

### Callee

```
function voodoo(
 CBName Ai, CBName i,
 CBV limit) {
 var sum = 0;
 for (i = 0;
 i < limit;
 ++i) sum += Ai;
 return sum;
}
```

# Parameter-Passing Mechanisms

## Call-by-Name: Example — The Jensen Device (*cont*)

- ▶ The parameter  $A_i$  holds an object,
  - ▶ ...the getter-method of which is `int getValue() { return A[i]; }`
  - ▶ ...the setter-method of which is `void setValue(int v) { A[i] = v; }`
- ▶ The parameter  $i$  holds an object,
  - ▶ ...the getter-method of which is `int getValue() { return i; }`
  - ▶ ...the setter-method of which is `void setValue(int v) { i = v; }`
- ▶ So just as their names in the **second version** suggest, there is a deep semantic relationship between  $A_i$  and  $i$  in the callee — they both depend on the value of  $i$  in the caller

# Parameter-Passing Mechanisms

## Call-by-Name: Example — The Jensen Device (cont)

- ▶ The caller calls voodoo with  $A[i]$  and  $i$ , but the value of  $i$  has not been initialized!?
  - ▶ Precisely!
    - ▶ When you pass an argument by name, the argument is not evaluated
    - ▶ When you pass arguments by name, you are defining an object and two methods per each argument passed by name, and the expressions themselves are evaluated only when the setter-methods and getter-methods are invoked
    - ▶ So in fact, we were not passing **the value** of  $i$  or  $A[i]$  but an objects with methods that set and get these expressions
  - ▶ And this is the magic of CBName: Mixed with side-effects, CBName creates deep connections between various parts of the code

# Parameter-Passing Mechanisms

## Call-by-Name: Example — The Jensen Device (*cont*)

- ▶ This technique is known as the **Jensen Device**:
  - ▶ There is nothing in the procedure voodoo, especially not in the way it was written originally, that could indicate that it **sums up** the elements of the array and returns their sum
  - ▶ Rather, the behavior of “summing up” emerges from a combination of how voodoo is defined, along with how it is called: The caller & callee must “co-operate” to cause the “summing up” to happen.
    - ▶ Put otherwise, you could call voodoo with a different set of arguments, and it would not return a sum of elements of an array

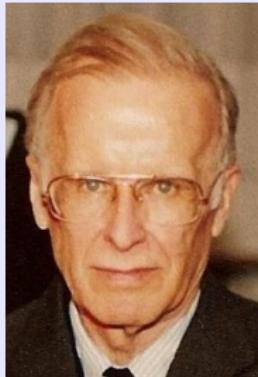
# Parameter-Passing Mechanisms

## Call-by-Name: Who's Jensen? [pronounced "Yensen"]

- ▶ Remember BNF? BNF stands for the Backus-Naur Form:

John Backus

*John Backus from IBM NY*



Peter Naur

*Peter Naur from the University of Copenhagen*



Jørn Jensen

*Jørn Jensen was a programmer who collaborated with Peter Naur*



# Parameter-Passing Mechanisms

## Call-by-Name: Example

- ▶ Consider the following code:

```
procedure while(
 CBName test ,
 CBName body) {
 if (test) {
 body;
 while(test , body);
 }
}
```

- ▶ Because the test, body are passed by name, they are not evaluated until consulted
- ▶ This **delays evaluation**, something we exploit to implement **while** as a procedure, rather than a special form

# Parameter-Passing Mechanisms

## Call-by-Name: Example

```
boolean function and(
 CBV a,
 CBName b) {
 if (a)
 return b;
 return false;
}
```

- ▶ The code eagerly evaluates a, but **delays the evaluation** of b
- ▶ This is sufficient to implement and as a function, with Scheme-like semantics, rather than a special form

# Parameter-Passing Mechanisms

## Call-by-Name: Example

```
boolean function or(
 CBV a,
 CBName b) {
 if (a)
 return a;
 return b;
}
```

- ▶ The code eagerly evaluates a, but **delays the evaluation** of b
- ▶ This is sufficient to implement or, with Scheme-like semantics, as a function rather than a special form

# Parameter-Passing Mechanisms

## Call-by-Name — Pros

- ▶ A natural extension of the substitution model & the normal-order of evaluation to side-effects & recursion
- ▶ Provides a natural way to delay computation: This can replace hygienic macros in many cases, allowing us to implement as functions & procedures code that would otherwise be implemented as macros. This simplifies debugging considerably!
- ▶ May be implemented with reasonable efficiency

# Parameter-Passing Mechanisms

## Call-by-Name — Cons

- ▶ Creates complex interrelationships between callers and callees
- ▶ Unusual & unfamiliar, it is a source of confusion for programmers and implementers
  - ▶ For example, from most programming languages, we are used to **callees** encapsulating intention & functionality
  - ▶ With CBName, where it is necessary to use such exotic tricks as the **Jensen Device**, the intention & functionality is **split between caller and callee**
    - ▶ For example, We used voodoo to sum up numbers from an array
    - ▶ We could not name voodoo `sum_up_numbers`, because it would only sum up numbers **if called in a particular way**
    - 👉 Our inability to give voodoo a name that represents what it does demonstrates that CBName diverges greatly from how intention & functionality are implemented using more accepted PPMs

# Parameter-Passing Mechanisms

## Call-by-Name — Summary

- ▶ CBName is an interesting and unique PPM
- ▶ Passing expressions by name involves means that we are passing **container objects** that have the expressions in their setter- and getter-methods
- ▶ Implementation is quite efficient
- ▶ Programming with CBName offers a natural way to delay evaluation, and control the order of evaluation explicitly, allowing us to write first-class procedures that in other languages are encoded as macros

## Call-by-Name — Summary (*cont*)

- ▶ We already mentioned that CBR is dangerous, in that it permits promiscuous access to caller internals: In this sense, CBName is far more dangerous than CBR
- ▶ CBName has been implemented partially in industrial languages beyond Algol60 (Scala)
- ▶ CBName continues to be important in programming languages theory

## Roadmap

### Parameter-Passing Mechanisms

- ✓ Introduction
- ▶ The mechanisms we consider
  - ✓ Call-by-Value
  - ✓ Call-by-Reference
  - ✓ Call-by-Sharing / Call-by-Object
  - ✓ Call-by-Name
  - ▶ Call-by-Need

# Parameter-Passing Mechanisms

## Call-by-Need (CBNeed)

You wish to purchase a cellular phone. The seller agrees to your offer of US\$100, and the one remaining question is how to pass him the money. You choose to use **CBNeed**:

- ▶ Basically, you have a similar arrangement as in the CBName transaction
  - ▶ The one difference is that once you hand over the money for the phone, the seller signs a receipt to the effect that you paid in full
  - ▶ If the seller ever asks for payment again, you just show him the receipt, and you need not pay any more

# Parameter-Passing Mechanisms

## Call-by-Need (*continued*)

- ▶ CBNeed aims at **avoiding the re-computation** that is inherent and implicit in CBName
- ▶ CBNeed = CBName + Caching
  - ▶ Just as with CBName, when passing an expression by need, a container object is created and passed for an argument
  - ▶ The container object has two methods: a setter and a getter
  - ▶ The difference between CBName and CBNeed is that with CBNeed, the getter method evaluates the expression **at most once**.

# Parameter-Passing Mechanisms

## Call-by-Need (CBNeed)

- ▶ Having once evaluated an expression, the value is kept within the container object, and returned immediately upon further get calls
- ▶ When Passing `factorial(1000000)` as an argument using CBNeed, with parameter name `n`, we may appreciate the advantage of not having to recompute the value of `n` each and every time `n` is consulted
- ▶ However, if we were to pass the expression `(1 + random(6))` by need, as a kind of digital dice, this would make for a very inflexible and uninteresting game...

## Call-by-Need — Pros

- ▶ For many problems, the caching mechanism of CBNeed is very convenient
- ▶ Caching a computed value, when this makes sense semantically, can significantly reduce computation-time

## Call-by-Need — Cons

- ▶ For many problems, this form of caching is unnecessary
- ▶ If people find CBName difficult to reason about, they're not going to find CBName + caching to be any easier!

# Parameter-Passing Mechanisms

## Call-by-Need — Summary

- ▶ CBNeed = CBName + caching
- ▶ Sometimes this caching behavior is very useful; Other times it is not
- ▶ CBNeed is available in the well-known language Haskell and the lesser-known languages Galina & Daisy

# Further reading

## Roadmap

- ▶ The expansion of letrec, revisited
- ▶ Recursion & Circularity
- ▶ Fixed-Point Theory
- ▶ Defining circular structures with recursion
- ▶ Defining circular structures with self-application

## Revisiting the expansion of letrec

This is the macro-expansion we presented for letrec:

$$\begin{aligned} \llbracket (\text{letrec } ((f_1 \langle Expr_1 \rangle) &= \llbracket (\text{let } ((\textcolor{red}{f}_1 \text{ 'whatever}) \\ &\quad (f_2 \langle Expr_2 \rangle) && (\textcolor{red}{f}_2 \text{ 'whatever}) \\ &\quad \dots && \dots \\ &\quad (f_n \langle Expr_n \rangle)) && (\textcolor{red}{f}_n \text{ 'whatever})) \\ \langle expr_1 \rangle \dots \langle expr_m \rangle) \rrbracket && (\text{set! } \textcolor{red}{f}_1 \langle Expr_1 \rangle) \\ && (\text{set! } \textcolor{red}{f}_2 \langle Expr_2 \rangle) \\ && \dots \\ && (\text{set! } \textcolor{red}{f}_n \langle Expr_n \rangle) \\ && (\text{let } () \\ && \quad \langle expr_1 \rangle \dots \langle expr_m \rangle) \rrbracket \end{aligned}$$

## Revisiting the expansion of letrec (*continued*)

- ▶ When we introduced this expansion, we commented that it was disappointing that we needed to resort to side-effects in order to implement an idea as fundamental to functional programming as recursion.
- ▶ At that time, we added that there is a purely functional way, without resorting to side-effects, to define recursive functions
- ▶ This is the topic we are now entering

# Chapter 8

## Roadmap

- ✓ The expansion of letrec, revisited
- ▶ Recursion & Circularity
- ▶ Fixed-Point Theory
- ▶ Defining circular structures with recursion
- ▶ Defining circular structures with self-application

- ▶ Recursive functions are an example of a **statically-defined, circular** data-structure
  - ▶ For **local**, recursive functions, the **value** of the recursive function is a **closure**
    - ▶ Closures have 2 fields: A lexical environment, and a code pointer
    - ▶ That the function is recursive means that the code pointer  $L$  of which points to some code (e.g., in *machine language*) that references a **bound variable** the lexical address of which points to a location in the lexical environment of the closure, a location the value at which is  $L$ ...
  - ▶ For **global**, recursive functions, the value of the function is too a closure, the code pointer  $L$  of which points to a some code (e.g., in *machine language*) that references a **free variable** defined in the top-level environment, a free variable the value of which is  $L$ ...

- ▶ The linguistic ability of a programming language, to define recursive functions, is the linguistic ability to define a particular, **circular data-structure** statically, so that the compiler may know, at compile-time, that a cycle exists:
  - ▶ The cycle is defined by using the **name** of the procedure as a label, to which to point from within the body of the procedure
  - ▶ Some programming languages provide linguistic facilities with which to define circular data structures of other kinds too: Scheme, Prolog, C, assembly, etc
  - ▶ Other programming languages have no facility for defining any kind of circular data-structures statically: In such languages, circular data-structures are created at run-time

- ▶ It is significant that the circularity is defined and be recognizable by the compiler, statically:
  - ▶ Circular data-structures require special handling: Input, output, & many other operations can go into infinite loops if circularity is not handled properly
  - ▶ When the circular structure is a **function type**, a compiler can sometimes detect problems with termination conditions, infinite loops, etc
  - ▶ When a functional type is **not** a circular structure, the compiler is free **not** to use a stack, either to pass arguments or to save the return address

## Static, circular data in C

```
struct LL {
 int value;
 struct LL *next;
};
```

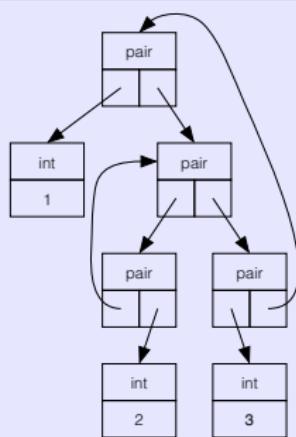
```
extern struct LL db4;
struct LL db1 = {1, &db4};
struct LL db5 = {5, &db1};
struct LL db3 = {3, &db5};
struct LL db6 = {6, &db3};
struct LL db9 = {9, &db6};
struct LL db4 = {4, &db9};
```

## Static, circular data in Scheme

### Data

```
> '#0=(1 . #1=((#1# . 2) 3 . #0#))
#0=(1 . #1=((#1# . 2) 3 . #0#))
```

### Shape



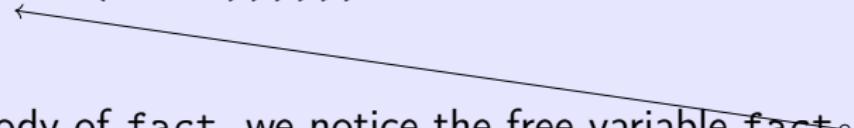
## Static, circular data in other languages

- ▶ By far, the easiest language in which to define static, circular data is Prolog
- ▶ In Java, the only static, circular data-types that can be defined are the **method**, **class**, and **interface**
- ▶ In Python, the only static, circular, data-types that can be defined are the **function**, **method**, and **class**
- ▶ In some languages, the only recursive data-type is the function
  - ▶ In PL/I, this must be declared with the keyword **RECURSIVE**
  - ▶ In FORTH, single recursion is supported by a keyword, and mutual recursion can only be defined at **run-time**
- ▶ Some languages don't even permit recursive function: COBOL (up to the 1985 standard), Fortran (up to the 1977 standard)

## Writing fact without recursion

- We start with the recursive definition of fact:

```
(define fact
 (lambda (n)
 (if (zero? n)
 1
 (* n (fact (- n 1))))))
```



- Examining the body of fact, we notice the free variable ~~fact~~, that serves as an edge that closes a circular graph.

## Writing fact without recursion

- ▶ Notice that fact is a **free variable**
- ▶ In the functional world, free variables are dead-weight:
  - ▶ In the functional world, we change variables using functional composition:

$$\begin{aligned}f(x) &= \cos \boxed{x} \\g(x) &= x^2 + 1 \\(f \circ g)(x) &= \cos \boxed{(x^2 + 1)}\end{aligned}$$

- ▶ Before the change ○
- ▶ After the change ○

- ▶ Free variables can only be changed using **assignment**, which is outside the functional paradigm

## Writing fact without recursion

- ▶ Therefore, the first step we take is to “close over” the free variable fact, by enclosing the entire expression without (lambda (fact) ... ).
- ▶ We call this expression Ffact:

```
(define Ffact
 (lambda (fact)
 (lambda (n)
 (if (zero? n)
 1
 (* n (fact (- n 1)))))))
```

## Writing fact without recursion

- ▶ The relationship between fact & Ffact is a rich one:
  - ▶ On the one hand, Ffact is related to fact **syntactically**, meaning there is a simple derivation of the syntax of Ffact from the syntax of fact — As we mentioned earlier, we just surround the body of fact with `(lambda (fact) ... )` to obtain Ffact
  - ▶ On the other hand, this relationship goes much deeper than mere syntax

## Writing fact without recursion

- ▶ We claim that **any implementation** of the factorial function, regardless of how it computes factorial, whether following the recursive definition (`fact`), or iteratively, or using some mathematical wizardry such as the **Gamma function**, etc — Regardless of how it is implemented, any implementation of the factorial function satisfies two properties:
  - ▶ It is a fixed-point of `Ffact`
  - ▶ Any other fixed-point of `Ffact` **can also** compute the factorial function (**leastness** of fixed-point)

# Chapter 8

## Roadmap

- ✓ The expansion of letrec, revisited
- ✓ Recursion & Circularity
- Fixed-Point Theory
- Defining circular structures with recursion
- Defining circular structures with self-application

# Fixed Points

- ▶ You meet a friend, ask for her phone number, and failing to have handy paper & pencil with which to write it down, you punch her number on your handy pocket calculator...
- ▶ Later on, sitting in some [non-Compiler-Construction] lecture, your mind wanders, and you begin to play with your calculator, hitting the  $\sqrt{x}$  key over and over...
- ▶ You notice how with each key-press, the numbers change on your screen:

5552626  
2356.40106943  
48.5427756667  
6.96726457562

...

1

1

## Fixed Points (*continued*)

When you swap out of your day-dreaming, you realize some important facts:

- ▶ The  $\sqrt{x}$  key no longer has any effect on the number displayed on your calculator screen
- ▶ The number you've reached, 1, is what is known as a **fixed-point** of the square-root function
- ▶ You've lost your friend's phone number...

# Fixed Points (*continued*)

- ▶ For a function  $f: \mathcal{D} \rightarrow \mathcal{D}$ , a point  $x_0 \in \mathcal{D}$  is called a **fixed-point** of  $f$ , if  $f(x_0) = x_0$ 
  - ▶  $x_0$  is a point that **is not changed** by the function
- ▶ The fixed-points of a function tell us a lot about the function
- ▶ Techniques for working with fixed-points can often greatly simplify proofs and computations, so such techniques should be in your “bag of tricks”...

# Fixed Points (*continued*)

Exmaple: Computing a limit

$$\frac{1}{3} + \frac{1}{3^2} + \frac{1}{3^3} + \dots$$

After thinking about it for a while, we realize the sum must be  $x$  ☺  
Then

$$\begin{aligned} 3x &= 1 + \frac{1}{3} + \frac{1}{3^2} + \frac{1}{3^3} + \dots \\ &= 1 + x \end{aligned}$$

$$3x - 1 = x$$

So we are searching for a solution to the equation  $f(x) = x$ , where  $f(x) = 3x - 1$ , and so  $x_0 = \frac{1}{2}$

# Fixed Points (*continued*)

Example: Computing a limit

$$2 + \sqrt{2 + \sqrt{2 + \dots}}$$

After thinking about it for a while, we realize the sum must be  $x$  😊  
Then

$$\begin{aligned}(x - 2)^2 &= 2 + \sqrt{2 + \sqrt{2 + \dots}} \\&= x \\x^2 - 5x + 4 &= 0\end{aligned}$$

So we are searching for a solution to the equation  $f(x) = x$ , where  $f(x) = (x - 2)^2$ , and so  $x_1 = 1, x_2 = 4$ . The solution is, of course,  $x_2 = 4$

## Existence of a fixed-point

- ▶ Regarding functions in mathematics, it is clear that many functions have no fixed-points.
- ▶ For example,  $f(x) = x^2 + 1$ , over  $\mathbb{R}$ , or  $g(x) = x - 1$ , over either  $\mathbb{R}$  or  $\mathbb{C}$
- ▶ Assuming the existence of a fixed-point, when such does not exist, will lead to a contradiction

# Fixed Points (*continued*)

## Existence of a fixed-point

- ▶ In pure functional calculus, such as the  $\lambda$ -calculus, expressions always have fixed-points:
  - ▶ If we think of such expressions as **computer programs**, the fixed-point is just a kind of iteration over the program, which too is a program
  - ▶ The fixed-point may be a non-terminating program, but for our purposes, this is irrelevant, as we shall see when we explore **least fixed-points**
  - ▶ Without distracting from the beauty of the theory, that a fixed-point exists always indicates that the theory is fundamentally trivial...

# Fixed Points (*continued*)

## Functions as fixed-points

- ▶ From your studies in mathematics, you are already familiar with the idea of the fixed-point of a function being a number
- ▶ You might be surprised that a function can itself be the fixed-point of a [higher-order] function
- ▶ **Example 1:** Consider the *identity* function (define id (lambda (x) x)). Obviously anything is a fixed-point of the *identity* function, including itself!

```
> (id log)
#<procedure log>
> (id acos)
#<procedure acos>
> (id id)
#<procedure id>
```

# Fixed Points (*continued*)

## Functions as fixed-points (*continued*)

- ▶ **Example 2:** Consider the function K, defined as follows:

```
(define K
 (lambda (x) (lambda (y) x)))
```

- ▶ For any term x, x is a fixed-point of (K x):

```
> ((K "moshe") "moshe")
"moshe"
> ((K '(1 2 3)) '(1 2 3))
(1 2 3)
> ((K K) K)
#<procedure k>
> ((K exp) exp)
#<procedure exp>
```

# Fixed Points (*continued*)

## Relationship of Ffact & fact

- ▶ **Claim:** fact is a fixed-point of Ffact
- ▶ **Proof:** The claim we need to show is that  $(Ffact\ fact) = fact$ . We prove this in two cases.
- ▶ **Case 1:**  $((Ffact\ fact)\ 0) = 1$ , by inspection. This is trivial because when  $n$  is 0, Ffact ignores its first argument, so:

```
> ((Ffact 'potato) 0)
1
```

## Ffact

```
(define Ffact
 (lambda (fact)
 (lambda (n)
 (if (zero? n)
 1
 (* n (fact (- n 1)))))))
```

# Fixed Points (*continued*)

## Relationship of Ffact & fact

- ▶ **Case 2:** Assume  $n > 0$ . So  $((\text{Ffact} \text{ fact}) n) = (* n (\text{fact} (- n 1)))$   
 $= n * (n - 1)!$ , because  $(\text{fact} (- n 1))$  is just  $(n - 1)!$
- ▶ This is **not** a proof by induction, and we neither mentioned nor used an induction hypothesis

## Ffact

```
(define Ffact
 (lambda (fact)
 (lambda (n)
 (if (zero? n)
 1
 (* n (fact (- n 1))))))))
```

## Fixed Points (*continued*)

- ▶ In fact, merely being a fixed-point of  $F_{fact}$  isn't going to cut it:  
Expressions can have more than one fixed-point:
  - ▶ For example, **anything** is a fixed-point of the identity function  
`(lambda (x) x)`. So what is so interesting about  $fact$  being possibly one of possibly several fixed-points of  $F_{fact}$ ?
- ▶ What we need to show is that there is some deeper relationship between  $fact$  &  $F_{fact}$  — That  $fact$  isn't just any old fixed-point, but that in some sense, it's the most basic fixed-point of  $F_{fact}$ .

# Fixed Points (*continued*)

## Relationship of Ffact & fact

- ▶ **Claim:** fact is the least fixed-point of Ffact, in the sense that any other fixed-point of Ffact can also compute the factorial function.
- ▶ **Proof:** Let g be a fixed-point of Ffact, so that  $(Ffact\ g) = g$ . We want to show that for all  $n \in \mathbb{N}$ , we have  $(g\ n) = n!$ . We prove this by induction on n:
  - ▶ **Base Case:**  $(g\ 0) = ((Ffact\ g)\ 0) = 1$ , by inspection

## Ffact

```
(define Ffact
 (lambda (fact)
 (lambda (n)
 (if (zero? n)
 1
 (* n (fact (- n 1)))))))
```

# Fixed Points (*continued*)

## Relationship of Ffact & fact

- ▶ **Induction Hypothesis:**  $\forall k < n, (g\ k) = k!$
- ▶ **Induction Step:** For  $n > 0, (g\ n) = ((Ffact\ g)\ n)$  [by definition]  $= (*\ n\ (g\ (-\ n\ 1)))$  [since  $n > 0$ ]  $= n * (n - 1)! = n!$

## Ffact

```
(define Ffact
 (lambda (fact)
 (lambda (n)
 (if (zero? n)
 1
 (* n (fact (- n 1))))))))
```

# Fixed Points (*continued*)

## Relationship of Ffact & fact (*continued*)

- ▶ But doesn't this mean that fact is the one and only fixed-point of Ffact?
- ▶ NO!
  - ▶ We showed that forall  $g$ , the fact that  $(\text{Ffact } g) = g$  implies that forall  $n \in \mathbb{N}$ ,  $(g \ n) = n!$
  - ▶ This means that  $g$  can compute the fact
  - ▶ It does not mean that fact can compute  $g$
- ▶ Huh??
  - ▶ Recall your course in *Logic & Set Theory*: A function is a set of ordered-pairs
  - ▶ By showing that  $g$  can compute fact, we showed that all the pairs in fact are also pairs in  $g$ , or that taken as sets,  $\text{fact} \subseteq g$
  - ▶ In order to prove that  $\text{fact} = g$ , we will need to show that  $g \subseteq \text{fact}$ , and this we haven't done

# Fixed Points (*continued*)

## Relationship of Ffact & fact (*continued*)

- ▶ But... What does it mean for  $g \not\subseteq \text{fact}$ ??
  - ▶ It would mean that  $g$  might contain additional pairs that are not in  $\text{fact}$ : That the domain of  $g$  is **larger** than the domain of  $\text{fact}$
  - ▶ We really know nothing about  $g$  except that it is one of possibly-many fixed-points of  $\text{Ffact}$

# Fixed Points (*continued*)

## Relationship of Ffact & fact (*continued*)

We now demonstrate that the *factorial* function is a fixed-point of Ffact, regardless of how it is implemented

- ▶ Let fact be the usual, **recursive** implementation of the *factorial* function
- ▶ Let iterative-fact be the iterative implementation of the *factorial* function

Both (Ffact fact) and (Ffact iterative-fact) implement the *factorial* function!

# Fixed Points (*continued*)

## Recursive factorial

```
(define fact
 (lambda (n)
 (if (zero? n)
 1
 (* n (fact (- n 1))))))
```

## Iterative factorial

```
(define iterative-fact
 (letrec ((loop
 (lambda (n r)
 (if (zero? n)
 r
 (loop (- n 1)
 (* n r)))))))
 (lambda (n)
 (loop n 1))))
```

## Fixed Points (*continued*)

We see that both fact, iterative-fact, (Ffact fact), and (Ffact iterative-fact) all compute the *factorial* function:

```
> (fact 5)
120
> (iterative-fact 5)
120
> (fact 10)
3628800
> (iterative-fact 10)
3628800
> ((Ffact fact) 5)
120
> ((Ffact iterative-fact) 5)
120
> ((Ffact fact) 10)
3628800
> ((Ffact iterative-fact) 10)
3628800
```

# Fixed Points (*continued*)

## Summary so far

- ▶ fact is not just any old fixed-point of Ffact, but it is also the least fixed-point, in the sense that any fixed-point of Ffact can also compute fact...
- ▶ These two facts give us license to search for any fixed-point of Ffact:
  - 👉 We are confident in the knowledge that any fixed-point of Ffact will be able to compute the factorial function

# Fixed-Points (*continued*)

## How are fixed-points found

- ▶ There are many tricks and methods in analysis for finding fixed-points
- ▶ There is one super-simple technique that in analysis converges only in specific, well-defined conditions, and when it does converge, the rate of convergence can vary greatly from function to function
- ▶ This technique is known as the **fixed-point iteration**:
  - ▶ In the bottom-up version, we start with an initial “guess”, apply the function repeatedly, until we converge

# Fixed-Points (*continued*)

## How are fixed-points found

- ▶ In your courses on *Numerical Analysis & Numeric Methods*, you learned that if the derivative  $f'$  is bound by a number strictly less than 1, namely if  $\forall x \in I$  we have  $f(I) \subseteq I$ , and  $|f'(x)| \leq M < 1$ , then  $\exists x_0 \in I$ , such that  $f(x_0) = x_0$ , and the fixed-point iteration shall converge.
- ▶ As we mentioned, there is something fundamentally trivial about fixed-point theory in programming languages, and now we shall see what it is:
  - ▶ Our  $f$  is a higher-order function, mapping functions to functions (e.g., `Ffact`)
  - ▶ The fixed-point iteration we shall be using, does not start in a **bottom-up** fashion, from some guess, but starts **top-down**
  - ▶ The fixed-point iteration is too a computer program. Does it converge/terminate?

# Fixed-Points (*continued*)

## How are fixed-points found

- ▶ Since the least fixed-point of  $\text{Ffact}$  non-empty, that is, since it contains ordered pairs of the form  $\langle n, n! \rangle$ , and since **any** fixed-point of  $\text{Ffact}$  can compute factorial, we needn't look any further than the fixed-point obtained through the top-down fixed-point iteration: It must contain all the pairs of the form  $\langle n, n! \rangle$
- 👉 And in general, when we define a higher-order functional the least fixed-point of which is some computable, non-empty function, the fixed-point iteration **must** terminate over it

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration

- ▶ The simplest way to map  $f$  to  $(f\ (f\ (f\ \dots\ )))$  is to use recursion
- ▶ The argument is somewhat circular (pun intended! 😊) but that'll do for now, just as a demonstration:

```
(define fix
 (lambda (f)
 (f (fix f))))
```

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration

- ▶ Can the above definition of fix really work?
  - ▶ Yes, under call-by-name
    - ▶ You can see that indeed we are generating the infinite application  $(f\ (f\ (f\ (\dots)\ )))$
    - ▶ Under call-by-name, we would not be evaluating the arguments upon application, but only when the respective parameter is read within the body of  $f$
  - ▶ Under Scheme's applicative-order, call-by-value/call-by-sharing, the system would attempt to evaluate the argument before calling the code pointer in the closure of the procedure.

Attempting to evaluate the argument of  
 $\underbrace{(f\ (f\ (f\ (f\ (\dots)\ ))))}$ , since it itself is an application, would result in first attempting to evaluate the argument of that application, etc., which would result in an unbounded computation

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

- ▶ But amazingly, with a bit of analysis, we can get fix to work, or put otherwise, we can *fix* fix so that it works under applicative-order:
  - ▶ We wish to fix Ffact, or rather compute  $(Ffact\ (\ Ffact\ (\ Ffact\ \dots)))$ . We know that the argument to  $Ffact^\circ$  is supposed to compute the factorial function
  - ▶ The factorial function is a procedure of 1 argument
  - ▶ We can wrap the recursive call in fix with  $(\lambda(n)\ (\dots n))$  to delay the computation until the parameter (fact) is read, which is precisely what happens under call-by-name
  - ▶ This transformation, this wrapping with  $(\lambda(n)\ (\dots n))$ , is known as  $\eta$ -expansion ( $H/\eta$  are the uppercase/lowercase Greek letter “eta”)

# $\eta$ -Reduction / $\eta$ -Expansion

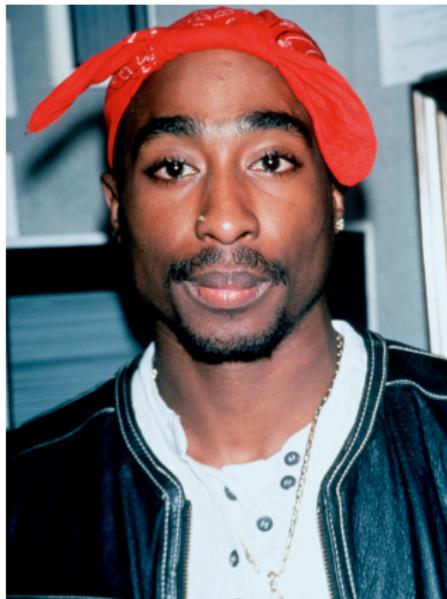
- ▶ You are a TA for a *Numerical Methods*
- ▶ You assign your class the homework assignment of implementing the *cosine* function
- ▶  $N - 1$  students submitted just what you expected:
  - ▶ They wrote a program to sum up the first terms of the Taylor-Maclaurin series  $1 - \frac{1}{2!}x^2 + \frac{1}{4!}x^4 - \frac{1}{6!}x^6 + \dots$ , until the error was within the accepted tolerance
  - ▶ One student submitted the following code:

```
(define my-cosine (lambda (x) (cos x)))
```

essentially calling the builtin library function `cos`
  - ▶ You might give him a grade of 0:
    - ▶ He basically did nothing
    - ▶ His code is merely a **wrapper** for a procedure he did not write

# $\eta$ -Reduction / $\eta$ -Expansion (*continued*)

- ▶ What does it mean to be a wrapper:
  - ▶ Not a rapper:



But a wrapper...

# $\eta$ -Reduction / $\eta$ -Expansion (continued)

- ▶ What does it mean to be a wrapper:
  - ▶  **$\eta$ -Reduction:** Given  $(\lambda (x) (M x))$ , where  $x$  is some variable, and  $x \notin \text{FreeVars}(M)$ , and  $M$  is a one-place function, then
$$(\lambda (x) (M x)) \rightarrow_{\eta} M$$
  - ▶  **$\eta$ -Expansion:** Given a one-place function  $M$ , and a variable  $x \notin \text{FreeVars}(M)$ ,  $M \rightarrow_{\eta^{-1}} (\lambda (x) (M x))$ .  $\eta$ -Expansion thus creates a wrapper around  $M$  without altering its behavior
  - ▶  $M$  must be a function: "Moshe" is not equivalent to  
`(lambda (x) ("Moshe" x))`
  - ▶ The above holds for Curried, single-valued functions
    - ▶ Clearly `cons` is not equivalent to `(lambda (x) (cons x))`

# $\eta$ -Reduction / $\eta$ -Expansion (continued)

- ▶ What about functions of different arity?
  - ▶ If M takes 0 arguments, it can be  $\eta$ -expanded to  
`(lambda () (M))`
  - ▶ If M takes 1 argument, it can be  $\eta$ -expanded to  
`(lambda (x) (M x))`
  - ▶ If M takes 2 arguments, it can be  $\eta$ -expanded to  
`(lambda (x y) (M x y))`
  - ▶ If M is variadic, or we don't know how many arguments it takes, we can  $\eta$ -expand it using a variadic lambda-expression, to  
`(lambda s (apply M s))`

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

- ▶ The recursive call  $\circ$  is precisely what we need to delay
- ▶ Applying the variadic  $\eta$ -expansion, we get the version that is suitable for CBV/CBS  $\circ$

### CBName fix

```
(define fix
 (lambda (f)
 (f (fix f))))
```

### CBV/CBS

```
(define fix
 (lambda (f)
 (f (lambda s
 (apply (fix f) s))))))
```

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

We can now put fix to some good use:

```
> (define fact
 (fix (lambda (!)
 (lambda (n)
 (if (zero? n)
 1
 (* n (! (- n 1))))))))

> (fact 5)

120

> (fact 20)

2432902008176640000
```

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

And here is another example of using fix:

```
> (define ack
 (fix (lambda (ackermann)
 (lambda (p q)
 (cond ((zero? p) (+ 1 q))
 ((zero? q) (ackermann (- p 1) 1))
 (else (ackermann (- p 1)
 (ackermann p (- q 1)))))))))

> (ack 2 2)

7

> (ack 3 3)

61
```

Good that we used variadic  $\eta$ -expansion! 

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

- ▶ For any function  $f$ , let  $S \subseteq \text{Domain}(f)$  be a subset of the domain of  $f$ . The function  $g = \{\langle x, f(x) \rangle : x \in S\}$  defined over  $S$  is called a **partial function** of  $f$
- ▶ For example, the empty set represents the empty partial function of **any other function** — It computes & returns nothing
  - ▶ In programming languages theory, the empty set thus represents the **infinite loop**, because it terminates on nothing
- ▶ For example, the set  $\{\langle 0, 1 \rangle, \langle 1, 1 \rangle, \langle 2, 2 \rangle, \langle 3, 6 \rangle, \langle 4, 24 \rangle\}$  is the partial factorial function  $\text{fact}_{<5}$  that computes factorial over the partial domain  $\{0, 1, 2, 3, 4\}$
- ▶ Using a proof similar to what we did in our **leastness** claim, it is straightforward to show by induction that:  
 $(\text{Ffact } \text{fact}_{<n}) = \text{fact}_{<n+1}$

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

Here's a demonstration of computing with the partial factorial functions:

```
> (define fact<0
 (lambda (n)
 (error 'fact<0
 (format "Cannot compute (fact<0 ~a)" n))))
> (fact<0 0)
```

Exception in fact<0: Cannot compute (fact<0 0)  
Type (debug) to enter the debugger.

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

```
> (define fact<1 (Ffact fact<0))
> (fact<1 0)
1
> (fact<1 1)
```

Exception in fact<0: Cannot compute (fact<0 0)  
Type (debug) to enter the debugger.

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

```
> (define fact<2 (Ffact fact<1))
> (fact<2 0)
1
> (fact<2 1)
1
> (fact<2 2)
```

Exception in fact<0: Cannot compute (fact<0 0)  
Type (debug) to enter the debugger.

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

```
> (define fact<3 (Ffact fact<2))
> (fact<3 0)
1
> (fact<3 1)
1
> (fact<3 2)
2
> (fact<3 3)
```

Exception in fact<0: Cannot compute (fact<0 0)  
Type (debug) to enter the debugger.

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

```
> (define fact<4 (Ffact fact<3))
> (fact<4 0)
1
> (fact<4 1)
1
> (fact<4 2)
2
> (fact<4 3)
6
> (fact<4 4)
```

Exception in fact<0: Cannot compute (fact<0 0)  
Type (debug) to enter the debugger.

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

- ▶ We demonstrated that we can implement the factorial function as a fixed-point of `Ffact`, however our procedure `fix` itself was written recursively
- ▶ Our challenge now is to find a non-recursive implementation of `fix`
- ▶ We shall still be computing fixed-points, only without recursion

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

- ▶ Recall our definition of Ffact:

```
(define Ffact
 (lambda (fact)
 (lambda (n)
 (if (zero? n)
 1
 (* n (fact (- n 1)))))))
```

- ▶ Based on Ffact, we define Gfact:

```
(define Gfact
 (lambda (fact n)
 (if (zero? n)
 1
 (* n (fact fact (- n 1))))))
```

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

- ▶ Notice the differences between Ffact & Gfact:
  - ▶ Ffact is Curried; Gfact isn't
  - ▶ Ffact has fact be a function that takes 1 argument; Gfact has fact take 2
- ▶ Gfact is derived from Ffact textually:
  - ▶ We make no claims at this point about Gfact, what it computes, etc.
- ▶ Notice just this:  $(Gfact\ Gfact\ 0) = 1$ , by inspection

# Fixed-Points (*continued*)

:PROPERTIES:

## Computing with the fixed-point iteration (*continued*)

- ▶ **Claim:**  $(Gfact\ Gfact\ n) = n!$
- ▶ **Proof:** By induction on  $n$ :
  - ▶ **Base Case:** We already mentioned that  $(Gfact\ Gfact\ 0) = 1$ , by inspection
  - ▶ **Induction Hypothesis:** For all  $k < n$ , we have  $(Gfact\ Gfact\ k) = k!$
  - ▶ **Induction Step:**  $n > 0$ , we have  $(Gfact\ Gfact\ n) = (*\ n\ (Gfact\ Gfact\ (-\ n\ 1)))$  [because by our IH,  $(Gfact\ Gfact\ (-\ n\ 1)) = (n - 1)!$ ]  $= n * (n - 1)! = n!$

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

- ▶ In fact, we can encode Gfact directly in C:

```
int Gfact(void *fact, int n) {
 if (n == 0) return 1;
 else return
 n * ((int (*)(void *, int))fact)(fact, n - 1);
}
```

- ▶ Note that Gfact is **not** recursive!
- ▶ The code compiles & runs without a warning, in perfectly legal, portable, standard C
- ▶ We can call this code as follows:

```
int result = Gfact(&Gfact, 5); /* result == 120 */
```

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

- ▶ Why/how does this code work?
  - ▶ We replaced recursion, namely a static, circular data-structure, with self-application: We're passing the **address** of a function onto itself at runtime, and thus closing at runtime the loop of our circular structure
- ▶ What's with the types & casting?
  - ▶ The type `(void *)` is a way of telling the C compiler not to worry about the type of fact; That things shall work out...
  - ▶ In fact, fact is a function, so before we apply it, we must inform the C compiler that it is indeed a function, by casting it to a function-type: `(int (*)(void *, int))`
  - ▶ The point is that we are going to pass `fact` to itself, so we tell the C compiler that `fact` is of type `(void *)`.

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

- ▶ If we were to use the more complete type for fact, `(int (*)(void *, int))` in place of `(void *)`, this would start an infinite regression of evermore complete types:
  - ▶ `(int (*)(void *, int))`
  - ▶ `(int (*)(int (*)(void *, int), int))`
  - ▶ `(int (*)(int (*)(int (*)(void *, int), int), int))`
  - ▶ ...
- ▶ In other words, our use of the `(void *)` type in this example is not innocent at all: It's a way of circumventing a limitation in the type-system of the C programming language, its inability to handle **recursive types**

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

- ▶ Getting back to Gfact:

```
(define Gfact
 (lambda (fact n)
 (if (zero? n)
 1
 (* n (fact fact (- n 1))))))
```

- ▶ The next step is to Curry Gfact & correspondingly to associate to the left the call (fact fact (- n 1)), giving Hfact:

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

- ▶ 

```
(define Hfact
 (lambda (fact)
 (lambda (n)
 (if (zero? n)
 1
 (* n ((fact fact) (- n 1)))))))
```
- ▶ Just as with Gfact, it is straightforward to show that  $((Hfact\ Hfact)n) = n!$
- ▶ **Proof:** By induction on  $n$ 
  - ▶ **Base Case:**  $((Hfact\ Hfact)\ 0) = 1$ , by inspection

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

► **Proof:** By induction on  $n$

- **Induction Hypothesis:**  $\forall k < n, ((\text{Hfact Hfact})\ k) = k!$
- **Induction Step:** For  $n > 0, ((\text{Hfact Hfact})\ n) =$   
 $(* n ((\text{Hfact Hfact}) (- n 1)))$  [because  $n \neq 0$ ]  
 $= n * (n - 1)! [by IH] = n!$

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

- ▶ Notice how close is Hfact to Ffact
- ▶ We can obtain Hfact from Ffact by composition

### Ffact

```
(define Ffact
 (lambda (fact)
 (lambda (n)
 (if (zero? n)
 1
 (* n ((fact
 (- n 1))))))))
```

### Hfact

```
(define Hfact
 (lambda (fact)
 (lambda (n)
 (if (zero? n)
 1
 (* n ((fact fact)
 (- n 1))))))))
```

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

- We can obtain Hfact from Ffact through composition:

$$\begin{aligned} \text{Hfact} &= \text{Ffact} \circ (\lambda (x) (x\ x)) \\ &= (\lambda (x) (\text{Ffact} (x\ x))) \end{aligned}$$

- One subtlety is that, under applicative order, just as with fix before, the application  $(x\ x)$ , which is supposed to be the factorial function, is being evaluated too soon, **before it is used**, so the above definition will only work under call-by-name
- But the same remedy that worked before, the use of variadic  $\eta$ -expansion, will work again

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

- We replace  $(x\ x)$  with its variadic  $\eta$ -expansion  
`(lambda s (apply (x x) s))`, giving

```
Hfact ≡ (lambda (x)
 (Ffact (lambda s
 (apply (x x) s))))
```

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

- In fact, we are not interested in Hfact in and of itself, but rather in fact, which is (Hfact Hfact):

```
fact ≡ (Hfact Hfact)
 ≡ ((lambda (x)
 (Ffact (lambda s
 (apply (x x) s)))))

 (lambda (x)
 (Ffact (lambda s
 (apply (x x) s)))))
```

- The above expression is parameterized by Ffact. The thing to do is to **abstract** over Ffact to take it as an argument.

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

```
(define Y
 (lambda (f)
 ((lambda (x)
 (f (lambda s
 (apply (x x) s))))
 (lambda (x)
 (f (lambda s
 (apply (x x) s)))))))
```

- The term  $Y$ , also known as the  $Y$ -combinator, is a version of fix written using self application and without recursion.

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

- There are 3 instances of self-application in Y:

```
(define Y
 (lambda (f)
 ((lambda (x)
 (f (lambda s
 (apply (x x) s))))))
 (lambda (x)
 (f (lambda s
 (apply (x x) s)))))))
```

- ◦ The first instance of self-application
- ◦ The second instance of self-application
- ◦ The third instance of self-application

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

We can now define fact using the Y-combinator:

```
(define fact
 (Y (lambda (!)
 (lambda (n)
 (if (zero? n)
 1
 (* n (! (- n 1))))))))
```

And test the code:

```
> (fact 5)
120
```

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

We define *Ackermann's Function* using the Y-combinator:

```
(define ack
 (Y (lambda (ackermann)
 (lambda (p q)
 (cond ((zero? p) (+ 1 q))
 ((zero? q) (ackermann (- p 1) 1))
 (else (ackermann (- p 1)
 (ackermann p (- q 1)))))))))
```

And test the code

```
> (ack 3 3)
```

61

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

And even though we haven't extended the theory of fixed-points to multiple fixed-points, here's a simple workaround, inspired by the standard construction in set-theory. We use it to define mutually-recursive is-even?, is-odd? functions:

```
(define is-even?-and-is-odd?
 (Y (lambda (even?odd?)
 (lambda (u)
 (u (lambda (n)
 (or (zero? n)
 ((even?odd? (lambda (e? o?) o?))
 (- n 1))))
 (lambda (n)
 (and (positive? n)
 ((even?odd? (lambda (e? o?) e?))
 (- n 1))))))))
```

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

```
(define is-even?
 (is-even?-and-is-odd?
 (lambda (ev? od?) ev?)))
```

```
(define is-odd?
 (is-even?-and-is-odd?
 (lambda (ev? od?) od?)))
```

# Fixed-Points (*continued*)

## Computing with the fixed-point iteration (*continued*)

And test the code:

```
> is-even?-and-is-odd?
#<procedure at foo.scm:359>
> (is-even? 10)
#t
> (is-even? 11)
#f
> (is-odd? 10)
#f
> (is-odd? 11)
#t
```

# Chapter 8

## Roadmap

- ✓ The expansion of letrec, revisited
- ✓ Recursion & Circularity
- ✓ Fixed-Point Theory
- ▶ Defining circular structures with recursion
- ▶ Defining circular structures with self-application

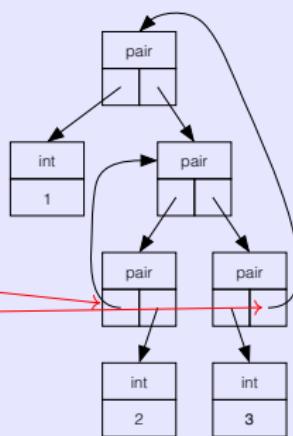
# Back to circular data

## Definition

Defined dynamically, with side-effects:

```
> '#0=(1 . #1=((#1# . 2) 3 .
 #0#))
#0=(1 . #1=((#1# . 2) 3 . #0#))
> (let ((x (cons 1
 (cons (cons 'a 2)
 (cons 3 'b))))))
 (set-car! (cadr x) (cdr x))
 (set-cdr! (cddr x) x)
 x)
#0=(1 . #1=((#1# . 2) 3 . #0#))
```

## Shape



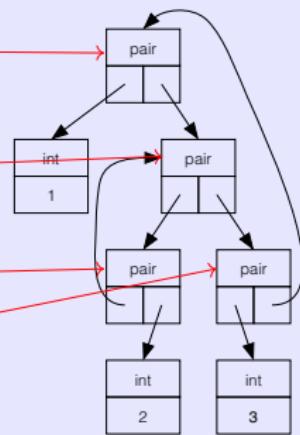
# Back to circular data

## Definition

Modeled recursively:

```
> (define x
 (letrec ((make-x○
 (lambda ()
 (cons (lambda () 1)
 make-y)))
 (make-y○
 (lambda ()
 (cons make-z make-w)))
 (make-z○
 (lambda ()
 (cons make-y
 (lambda () 2))))
 (make-w○
 (lambda ()
 (cons (lambda () 3)
 make-x))))
 (make-x)))
```

## Shape



# Back to circular data

## Definition

Testing the recursive model:

```
> x
(#<procedure at foo.scm:58>
 .
 #<procedure make-y at foo.scm:101>)
> ((car x))
1
> ((cdr ((car ((cdr x))))))
2
> ((cdr ((car ((car ((car ((cdr x))))))))))
2
> ((car ((cdr ((cdr x))))))
3
> ((car ((cdr ((cdr ((cdr x))))))))
1
```

# Chapter 8

## Roadmap

- ✓ The expansion of letrec, revisited
- ✓ Recursion & Circularity
- ✓ Fixed-Point Theory
- ✓ Defining circular structures with recursion
- Defining circular structures with self-application

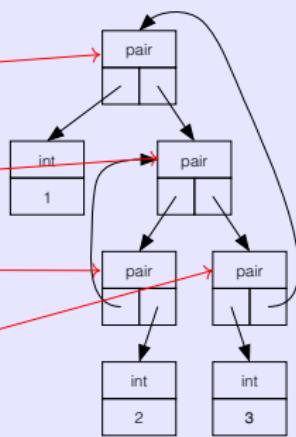
# Back to circular data

## Definition

Modeled using a fixed-point:

```
> (define x
 ((Y (lambda (make-x)
 (lambda () o
 (cons
 (lambda () 1)
 (lambda () o
 (cons
 (lambda () o
 (cons
 (lambda ()
 ((cdr (make-x))))
 (lambda () 2)))
 (lambda () o
 (cons
 (lambda () 3)
 make-x)))))))))))
```

## Shape



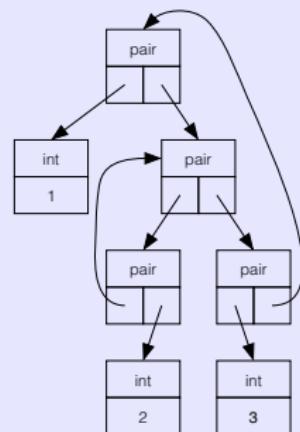
# Back to circular data

## Definition

Testing the fixed-point model:

```
> x
(#<procedure at foo.scm:226> .
 #<procedure at foo.scm:247>)
> ((car x))
1
> ((cdr ((car ((cdr x))))))
2
> ((cdr ((car ((car ((car ((cdr x))))))))))
2
> ((car ((cdr ((cdr x))))))
3
> ((car ((cdr ((cdr ((cdr x))))))))
1
```

## Shape



# Chapter 8

## Roadmap

- ✓ The expansion of letrec, revisited
- ✓ Recursion & Circularity
- ✓ Fixed-Point Theory
- ✓ Defining circular structures with recursion
- ✓ Defining circular structures with self-application

# Fixed-Points (*continued*)

## What have we seen

- ▶ We do not need side-effects to implement recursion & circular structures, either locally or globally
- ▶ The following are equivalent and inter-definable:
  - ▶ Recursion
  - ▶ Self-application
  - ▶ Circular data-structures
  - ▶ Side-effects
  - ▶ Recursive types

# Fixed-Points (*continued*)

## What have not we seen

- ▶ We have not seen how to extend the theory to systems of multiple fixed-point equations
- ▶ We have not seen how to write a general expander for letrec that can be used to define arbitrarily-many mutually-recursive procedures
- ▶ See you next semester, in my course *Introduction to Functional Programming*... ☺

# Further reading

## Roadmap

- 👉 Control in Programming Languages
- ▶ Continuations
- ▶ Continuation-Passing Style
- ▶ The Stack-Machine

# Control in Programming Languages

- ▶ Programming is an activity that manages **data & control**
- ▶ **Data** is the information created, moved, worked on by the computer
- ▶ **Control** is the mechanism by which the computer decides/manages **what** to do
  - ▶ On very simple computers, this could mean nothing more than the management of the Program-Counter/Instruction-Pointer (PC/IP)
  - ▶ On more advanced machines, more than one activity can take place simultaneously, and cannot be described by a single PC/IP
    - ▶ On such systems, a set of PC/IPs determines control
  - ▶ For the sake of simplicity, we shall refer to a single PC/IP

# Control in Programming Languages (*continued*)

- ▶ Control is the state of the PC/IP
- ▶ To manage control is to affect the PC/IP in a **non-linear** way
  - ▶ This is a low-level characterization
  - ▶ It applies both to low-level & to high level views of programming
- ▶ All machine-language instructions alter the PC/IP in a trivial, **linear** way
  - ▶ Even a sequence of `nop` instructions!

# Control at the level of the machine

- ▶ Some instructions change the PC/IP in a non-trivial, **non-linear** way:
  - ▶ Conditional & unconditional jumps & branches
  - ▶ Loops
  - ▶ Call & return
  - ▶ Interrupts & return from interrupts
- etc
- ▶ For more advanced architectures we include thread-switching, process-management, etc.

# Control in high-level languages

- ▶ Most HL programming languages offer a rich collection of HL control structures:
  - ▶ Loops, bounded & unbounded
  - ▶ Conditionals, guards, switches
  - ▶ Function/procedure/method call & return
  - ▶ Operations for asynchronous computing
  - ▶ Exception handling

etc

# Control in high-level languages (*continued*)

- ▶ Object-oriented programming is about **objects**, namely **data**. OOPLs generally come with richer facilities for working with data than for working with control
  - ▶ OOPLs model control structures as data, and then use the language's facility for working with data in order to make control more expressive & flexible
    - ▶ Examples: **Design Patterns** such as **Command**, **Strategy**, **Visitor**, **Observer**, **Iterator**, etc

# Control in high-level languages (*continued*)

- ▶ Functional Programming, by contrast, is based on the  $\lambda$ -calculus, which is an abstract theory of higher-order functions, and is data-impoveryed
  - ▶ The  $\lambda$ -calculus comes with no data other than  $\lambda$ -expressions, and all other data, including numbers, Booleans, etc., must be modeled using functions

# First-Class Objects

- ▶ Think “first-class citizens” vs “second-class citizens”
- ▶ To say that something is first-class means that it enjoys all the capabilities of any native data:
  - ▶ To be stored
  - ▶ Manipulated
  - ▶ Passed as an argument to a function or method
  - ▶ Returned from a function-call or method-call

etc

# First-Class Objects (*continued*)

- ▶ Not to be first-class means that some category of operations that applies to “ordinary data” is inapplicable to the object of interest
  - ▶ Functions in C are second-class — They cannot be stored, passed around, or changed (though their addresses can)
  - ▶ Macros in C are second-class
  - ▶ Classes in Java are second-class, though in Smalltalk they are first-class

etc

# Context

- ▶ The concept of **context** has to do with the code that surrounds an expression
- ▶ The context starts at the prompt, that is, at the REPL
- ▶ The context exists during run-time & cannot, in general, be known at compile-time
- ▶ **Example:** What is the context of the marked sub-expression in:

```
> (+ (* 2 3) (* 4 5))
```

▶ **Answer:** Removed the marked expression, leaving a “hole”:

```
> (+ (* 2 3))
```

- ▶ **Example:** What is the context of the marked sub-expression in:

```
> x
```

▶ **Answer:** Removed the marked expression, leaving a “hole”:

```
>
```

## Roadmap

- ✓ Control in Programming Languages
- 👉 Continuations
- ▶ Continuation-Passing Style
- ▶ The Stack-Machine

# Continuations

- ▶ Continuations are part of a general, high-level, functional theory of control
- ▶ Continuations are a way of representing the **rest of the computation**, whatever else remains to be evaluated/done, as a **first-class data-structure**:
  - ▶ “The rest of the computation” can be stored, manipulated, passed as an argument, returned from a call, etc.
  - ▶ Any general category of operations that is applicable to first-class data is applicable to the rest of the computation

# Continuations (*continued*)

- ▶ **Example:** We can have several continuations simultaneously, representing several “happy endings” to the story of our computation. We can then **wait to the end** to decide which ending is most suitable to us, given what we know **by the end...**
- ▶ **Example:** Using continuations, it is straightforward to implement **backtracking** of the kind found in Prolog
- ▶ **Example:** Using continuations, it is possible to implement a multi-threading kernel without any support from either the operating system or the hardware

# Continuations (*continued*)

- ▶ There are fundamentally two ways to work with continuations:
  - ▶ To write code in a special style, known as **Continuation-Passing Style** (CPS)
  - ▶ To use special procedures & forms that are unique to Scheme, to create and access a continuation object
- ▶ Writing in CPS is by far, the most flexible way to work with continuations
  - ▶ You can work with different kinds of continuations, and you can work in **any** programming language
  - ▶ The downside is that **all** the code that works with continuations needs to be written in CPS

# Continuations (*continued*)

- ▶ Using Scheme's builtin procedures & forms for grabbing & using continuations imposes less constraints on your programming style, and the API of your program. So it is easy to add code that works with continuations to an existing project, without having to rewrite existing code
  - ▶ On the downside, these procedures & forms are available in very few languages, they are harder to work with, and they are not as general as CPS

## Roadmap

- ✓ Control in Programming Languages
- ✓ Continuations
- 👉 Continuation-Passing Style
- ▶ The Stack-Machine

CPS is many things, and is used in many different ways:

- ▶ CPS is a high-level language with which to express control
  - ▶ Any kind of control can be implemented using CPS
  - ▶ If your languages doesn't support procedures, recursion, backtracking, asynchronous computing, exceptions, etc — these can all be modeled using CPS
  - ▶ CPS can even be used to model a virtual machine with a unique control architecture: E.g., to model backtracking or asynchronous computing at the VM level

# CPS (*continued*)

- ▶ CPS can serve as an intermediate language in a compiler
  - ▶ Many kinds of program transformations & compiler optimizations are simpler to describe, understand, and implement in CPS
- ▶ CPS is a programming technique that can often stretch the boundaries of your programming language. As such, it is a useful technique for your bag of tricks

# CPS — Intuition

- ▶ With CPS, your code is divided into two parts: What needs to happen **now**, and what needs to happen **afterwards**
- ▶ CPS requires you to order linearly expressions and sub-expressions in your code, sometimes, arbitrarily, so that these are evaluated according to this fixed order
- ▶ **Example:** Consider the expression  $(f \ (g \ (h \ x)))$ . There is a linear ordering to its evaluation:
  - ▶ Call  $h$
  - ▶ Call  $g$
  - ▶ Call  $f$

# CPS — Intuition (*continued*)

- ▶ Example: Consider the expression  $(f \ (g \ x) \ (h \ y))$ . Exactly two orderings are possible:

## First ordering

- ▶ Call g
- ▶ Call h
- ▶ Call f

## Second ordering

- ▶ Call h
- ▶ Call g
- ▶ Call f

- ▶ The choice of ordering would determine the result/output when there are side-effects in g, h
- ▶ Such examples are defined to be undefined in Scheme; Still the choice can be significant for other reasons

## CPS — Intuition (*continued*)

- ▶ **Example:** Consider the code

```
(f (g x)
 (h (g x)))
```

- ▶ Assuming there are **no side-effects** in g, the best ordering would be to compute the first argument (g x) first, and then **re-use the value** in computing the second argument (h (g x))

# CPS – Example

```
(w (f (if (g? x)
 (h (r y))
 (p (q z))))))
```

- ▶ Call g?
- ▶ Depending on the value of (g? x), do either:

## Then-clause

- ▶ Call r
- ▶ Call h
- ▶ Call f
- ▶ Call w

## Else-clause

- ▶ Call q
- ▶ Call p
- ▶ Call f
- ▶ Call w

- ▶ Despite the **fork**, only one ordering is possible here: The **fork** means there is a **choice** based on the return value of g?
- 👉 Notice that the calls to f, w are **duplicated**

## CPS (*continued*)

Programming in CPS changes the interface to all procedures that are not **builtin**: All user-defined procedures in CPS take an additional argument, a **continuation**:

- ▶ Initially, we shall represent the continuation as a **procedure**
- ▶ Later on, we shall see how to implement the continuation using **other data-structures**
- ▶ The continuation represents the **rest of the computation**, the **context**, as a function
- ▶ Because the continuation represents **all** the rest of the computation, it is always applied in tail-position

## CPS (*continued*)

- 👉 If you find yourself applying the continuation **not** in tail-position, this means you're doing something wrong!
- 👉 It is not possible to apply the continuation in non-tail position & get away with it: This is **always** a bug!
- ▶ The application of the continuation is governed by the **Induction Hypothesis of CPS**: When computing a value  $x$  in CPS, with continuation  $k$ ,  $k$  shall always be applied, in tail-position to  $x$

# Converting to CPS

Example: `(lambda (x) (f (g (h x))))`

- We add a **fresh** variable `k` to the lambda-expression:

`(lambda (x k) ...)`

- Then, we compute `(h x)`:

`(lambda (x k) (h$ x (lambda (roh) ...)))`

- `h$` is the CPS version of `h`
- The `(lambda (roh) ...)` is the continuation
- The name `roh` stands for *result of h*

# Converting to CPS (*continued*)

- ▶ The IH/CPS tells us that the continuation is applied to the result of `h`
  - 👉 You won't see that application of the continuation to the result; It's buried within `h$`!
  - 👉 We're not discussing how to convert `h` to `h$`; Only how to convert `(lambda (x) (f (g (h x))))` to CPS!
  - 👉 You must always trust the IH/CPS! 😊

# Converting to CPS (*continued*)

- ▶ Then, we compute  $(g \text{ roh})$ :

```
(lambda (x k)
 (h$ x
 (lambda (roh)
 (g$ roh (lambda (rog) ...))))
```

- ▶  $g\$$  is the CPS version of  $g$
- ▶ The function  $(\lambda (rog) \dots)$  is the continuation
- ▶ The variable name  $rog$  stands for *result of g*

# Converting to CPS (*continued*)

- ▶ The IH/CPS tells us that the continuation is applied to the result of h
  - ⌚ You won't see that application of the continuation to the result; It's buried within g\$
  - ⌚ You must always trust the IH! ☺
- ▶ We then call f with the result of g:

```
(lambda (x k)
 (h$ x
 (lambda (roh)
 (g$ roh
 (lambda (rog)
 (f$ rog
 (lambda (rof) ...)))))))
```

# Converting to CPS (*continued*)

- ▶  $f\$$  is the CPS version of  $f$
- ▶ The function `(lambda (rof) ...)` is the continuation
- ▶ The name `rof` stands for *result of f*
- ▶ The IH/CPS tells us that the continuation is applied to the result of  $f$ 
  - 💡 You won't see the application of the continuation to the result; It's buried within  $f\$$
  - 💡 You must always trust the IH! 😊

# Converting to CPS (*continued*)

- ▶ We are left to think about the continuation  
`(lambda (rof) ...)`
  - ▶ There are no more procedures to call in the body of the original procedure
  - ▶ We have one continuation `k` that is our “connection” to whomever might need to call our code
    - ⚠ If you did not use `k` in your continuations, you had better have a **very** good reason; This will usually be **wrong**!
  - ▶ We **must** use `k` & its argument is `rof`
  - ▶ The continuation `k` needs to be applied to the result & so the continuation becomes `(lambda (rof) (k rof))`

# Converting to CPS (*continued*)

- ▶ We are left to think about the continuation  
`(lambda (rof) ...)`
  - ▶ Notice that `rof`  $\notin$   $FreeVars(k)$ , so we can  $\eta$ -reduce it:

$$\begin{aligned} (\lambda (rof) &\rightarrow_{\eta} k \\ &(k \ rof)) \end{aligned}$$

- ▶ Putting it all together, we get:

```
(lambda (x k)
 (h$ x
 (lambda (roh)
 (g$ roh
 (lambda (rog)
 (f$ rog k)))))))
```

# Converting to CPS (*continued*)

Complete example:

## Direct Style

```
(lambda (x)
 (f (g (h x))))
```

## CPS

```
(lambda (x k)
 (h$ x
 (lambda (roh)
 (g$ roh
 (lambda (rog)
 (f$ rog k)))))))
```

# Converting to CPS (*continued*)

- ▶ Let us re-examine what we did with the innermost continuation:

$$\begin{aligned} (\lambda \text{ (rof)} &\rightarrow_{\eta} \text{ k} \\ &(\text{k rof})) \end{aligned}$$

- ▶ Imagine we worked in a Scheme-like system, where the TCO was **not implemented**:
- ▶ How many frames we built during the evaluation of the LHS:  
Two: The frame in which we apply  $(\lambda \text{ (rof)} \dots)$ , and the frame in which we apply  $\text{k}$ .
- ▶ How many frames we built during the evaluation of the RHS:  
One. Just the frame built when applying  $\text{k}$



$\eta$ -reducing the innermost continuation **is** an implementation of the TCO!



CPS turns the code “inside out”; This makes the TCO is easier to detect and implement!

# Converting to CPS (*continued*)

Here is a more complex example:

```
(lambda (x y)
 (f (g x)
 (h y)))
```

- ▶ As we mentioned earlier, CPS forces a linear ordering. In the previous example, only one ordering was possible. In this example, there are two:
  - ▶ First ordering:
    - ▶ Call g with x, result  $\Rightarrow$  rog
    - ▶ Call h with y, result  $\Rightarrow$  roh
    - ▶ Call f with rog & roh, result  $\Rightarrow$  rof
  - ▶ Second ordering:
    - ▶ Call h with y, result  $\Rightarrow$  roh
    - ▶ Call g with x, result  $\Rightarrow$  rog
    - ▶ Call f with rog & roh, result  $\Rightarrow$  rof

## Converting to CPS (*continued*)

- ▶ The choice of ordering can be significant when there are side-effect in the arguments to  $f$ , or when sub-expressions can be reused profitably
- ▶ Let us **arbitrarily** select the **second ordering**
- ▶ We add a **fresh** variable  $k$  to the lambda-expression:

(lambda (x y k) ...)

- ▶ Then, we evaluate  $(h\ y)$ :

(lambda (x y k)  
  (h\$ y (lambda (roh) ...)))

- ▶  $h$$  is the CPS version of  $h$
- ▶ The function  $(lambda (roh) ...)$  is the continuation
- ▶ The variable name  $roh$  stands for *result of h*

# Converting to CPS (*continued*)

- ▶ Then, we compute  $(g\ x)$ :

```
(lambda (x y k)
 (h$ y
 (lambda (roh)
 (g$ x (lambda (rog) ...))))
```

- ▶  $g\$$  is the CPS version of  $g$
- ▶ The function  $(lambda\ (rog)\ ...)$  is the continuation
- ▶ The variable  $rog$  stands for *result of g*

# Converting to CPS (*continued*)

- ▶ Then, we compute  $(f \text{ } \text{rog} \text{ } \text{roh})$ :

```
(lambda (x y k)
 (h$ y
 (lambda (roh)
 (g$ x
 (lambda (rog)
 (f$ rog roh (lambda (rof) ...)))))))
```

- ▶  $f\$$  is the CPS version of  $f$
- ▶ The function  $(\lambda (rof) \dots)$  is the continuation
- ▶ The variable  $rof$  stands for *result of f*

# Converting to CPS (*continued*)

- ▶ We are left to think about the innermost continuation  
`(lambda (rof) ...)`
  - ▶ There is nothing more to do in the original procedure
  - ▶ We need to return `rof` by applying to it the continuation `k`
  - ▶ The body of the continuation is therefore `(k rof)`
  - ▶ The continuation `(lambda (rof) (k rof))`  $\eta$ -reduces to `k`
- ▶ The entire code looks like:

```
(lambda (x y k)
 (h$ y
 (lambda (roh)
 (g$ x
 (lambda (rog)
 (f$ rog roh k))))))
```

# Converting to CPS (*continued*)

- ▶ Going with the first ordering gives:

```
(lambda (x y k)
 (g$ x
 (lambda (rog)
 (h$ y
 (lambda (roh)
 (f$ rog roh k))))))
```

- ▶ For “well-behaved” choices of g, h, these two orderings will return **the same result**
- ▶ When g, h are **not** “well-behaved”, there are **side-effects** the results of which depend upon the **order of evaluation**

# Converting to CPS (*continued*)

- ▶ Not so “well-behaved” code will have side effects in g, h that can result in
  - ▶ Different output
  - ▶ Different return-values
  - ▶ Infinite loops
  - ▶ Run-time error-messages
- ▶ Even in well-behaved code, certain orderings may be more efficient than others

# Converting to CPS (*continued*)

## Mixing builtin functions & user-defined functions

- ▶ The point of converting code to CPS is so we can use the continuation in ways that affect the rest of the computation
- ▶ At the moment, we are learning to convert code to CPS & we're just practicing the process of chaining together pieces of code, so we're not yet doing anything very interesting with the continuations. This will come later on
- ▶ In the meantime, we can realize that since the builtin code is not written in CPS, only user-code could ever do anything with the continuation

# Converting to CPS (*continued*)

## Mixing builtin functions & user-defined functions

- 👉 Builtin code is therefore not converted to CPS, but rather kept as is
- 👉 You will find examples on the internet where builtins are wrapped in a way that makes them have a CPS API, even though they cannot really be converted to CPS; Please ignore such misguided examples!

# Converting to CPS (*continued*)

## Mixing builtin functions & user-defined functions

- ▶ Example of what not to do: To convert `(lambda (x) (car (cdr x)))` to CPS, some books and online sources will write

```
(lambda (x k)
 (cdr$ x
 (lambda (rocdr)
 (car$ rocdr k))))
```

and assume the following definitions:

```
(define car$ (lambda (x k) (k (car x))))
(define cdr$ (lambda (x k) (k (cdr x))))
```

- ▶ while calls to `car$`, `cdr$` are actually **not** in tail-position!
- 💡 On the one hand, you get to use `car$` and `cdr$` as if they were in CPS, but on the other hand, their definitions clearly indicate that they're not!

# Converting to CPS (*continued*)

## Mixing builtin functions & user-defined functions

- ▶ **Example of what to do:** The code `(lambda (x) (+ x x))` is converted as `(lambda (x k) (k (+ x x)))`
- ▶ **Example of what to do:** The code  
`(lambda (x y) (+ (* x x) (* y y)))` is converted as  
`(lambda (x y k) (k (+ (* x x) (* y y)))))`

# Converting to CPS (*continued*)

## Mixing builtin functions & user-defined functions

- ▶ Example: The code

(lambda (x y)  
 (car (f (g (car x)) (h (q (reverse (cdr y)))))))  
is converted as  
~~(lambda (x y k)  
 (q\$ o (reverse (cdr y))  
 (lambda (roq)  
 (h\$ o roq  
 (lambda (roh)  
 (g\$ o (car x))  
 (lambda (rog)  
 (f\$ o rog roh  
 (lambda (rof) (k (car rof)))))))))))~~

# Converting to CPS (*continued*)

## Mixing builtin functions & user-defined functions

- ▶ Example: The code

```
(lambda (x y)
 (car (f (g (car x)) (h (q (reverse (cdr y)))))))
 1st arg to f 2nd arg to f
```

- ▶ Other possibilities would have the orderings:

- ▶  $q \Rightarrow g \Rightarrow h \Rightarrow f$
- ▶  $g \Rightarrow q \Rightarrow h \Rightarrow f$

# Converting to CPS (*continued*)

## Converting conditionals to CPS

- Example: The code

```
(lambda (x)
 (if (zero? x)
 y
 (f x)))
```

- First, we add a **fresh** variable  $k_o$  to the lambda-expression:

~~(lambda (x k) ...)~~

-  The *test*-clause of an *if*-expression is **never** in tail-position
-  The *then*-clause and *else*-clause of an *if*-expression are **both** in tail-position if the *if*-expression itself is in tail-position
-  We **don't** convert builtin procedures to CPS

# Converting to CPS (*continued*)

```
(lambda (x k)
 (if (zero? x)
 ...
 ...))
```

- We need to return y by passing it to the continuation k:

```
(lambda (x k)
 (if (zero? x)
 (k y)
 ...))
```

# Converting to CPS (*continued*)

- ▶ Next, we compute  $(f\ x)$ :

```
(lambda (x k)
 (if (zero? x)
 (k y)
 (f$ x
 (lambda (rof) ...))))
```

- ▶  $f\$$  is the CPS version of  $f$
- ▶ The function  $(lambda\ (rof)\ ...)$  is the continuation
- ▶ The continuation is applied in tail-position to the result of the call to  $f\$$ , so within the continuation it will be the value of the variable  $rof$ , which stands for *result of f*

# Converting to CPS (*continued*)

- We need to return `rof` to the continuation `k`, so the body of `(lambda (rof) ...)` becomes:

```
(lambda (rof)
 (k rof))
```

- This continuation  $\eta$ -reduces to `k`
- The complete code is thus:

```
(lambda (x k)
 (if (zero? x)
 (k y)
 (f$ x k)))
```

# Converting to CPS (*continued*)

- ▶ Example: The code

```
(lambda (x y)
 (if (foo? x)
 (goo x)
 (boo x (goo y))))
```

- ▶ First we add a **fresh** variable **k** for the continuation:

```
(lambda (x y k) ...)
```

- ▶ Next, we need to consider the *test*-clause of the if-expression:

- ▶ Unlike the previous example, the *test*-clause (foo? x) is **not** builtin, but rather user-defined, so we need to convert it to CPS:

```
(lambda (x y k)
 (foo?$ x
 (lambda (rof?)
 ...)))
```

- ▶ foo?\$ is the CPS version of foo?

# Converting to CPS (*continued*)

- ▶ Next, we have the if-expression:

```
(lambda (x y k)
 (foo? $ x
 (lambda (rof?)
 ...)))
```

## Converting to CPS (*continued*)

- ▶ Next, we call goo:

```
(lambda (x y k)
 (foo?$ x
 (lambda (rof?)
 (if rof?
 (goo$ x
 (lambda (rog)
 ...)))
 ...))))
```

- ▶ goo\$ is the CPS version of goo
- ▶ The function (lambda (rog) ...)) is the continuation
- ▶ rog is the variable that shall contain the *result of goo*

# Converting to CPS (*continued*)

- ▶ Next, we return `rog` to the continuation `k`:

```
(lambda (x y k)
 (foo?$ x
 (lambda (rof?)
 (if rof?
 (goo$ x
 (lambda (rog)
 (k rog)))
 ...))))
```

# Converting to CPS (*continued*)

- ▶ The continuation `(lambda (rog) (k rog))`  $\eta$ -reduces to `k`, giving:

```
(lambda (x y k)
 (foo? $ x
 (lambda (rof?)
 (if rof?
 (goo$ x k)
 ...))))
```

# Converting to CPS (*continued*)

- ▶ Next, we need to convert `(boo x (goo y))` to CPS. What needs to happen next is to call `goo` on `y`:

```
(lambda (x y k)
 (foo?$ x
 (lambda (rof?)
 (if rof?
 (goo$ x k)
 (goo$ y
 (lambda (rog) ...)))))))
```

# Converting to CPS (*continued*)

- ▶ Next, we compute boo:

```
(lambda (x y k)
 (foo?$ x
 (lambda (rof?)
 (if rof?
 (goo$ x k)
 (goo$ y
 (lambda (rog)
 (boo$ x rog
 (lambda (rob) ...))))))))
```

# Converting to CPS (*continued*)

- We return `rob` by applying to it the continuation `k`:

```
(lambda (x y k)
 (foo? $ x
 (lambda (rof?)
 (if rof?
 (goo$ x k)
 (goo$ y
 (lambda (rog)
 (boo$ x rog
 (lambda (rob)
 (k rob))))))))))
```

## Converting to CPS (*continued*)

- ▶ Notice that  $(\lambda (rob) (k \ rob))$   $\eta$ -reduces to  $k$ , so the code becomes:

```
(lambda (x y k)
 (foo? $ x
 (lambda (rof?)
 (if rof?
 (goo$ x k)
 (goo$ y
 (lambda (rog)
 (boo$ x rog k)))))))
```

- ▶ In converting an if-expression to CPS, the continuation is **duplicated** both in the *then-clause* and in the *else-clause*.

# Converting to CPS (*continued*)

- ▶ When the if-expression is in tail-position this duplication means that the continuation parameter appears **twice**
- ▶ When the if-expression is **not** in tail-position, duplicating the continuation may require duplicating a lot of code. For nested if-expressions, this duplication becomes exponential in the size of the nesting

# Converting to CPS (*continued*)

- ▶ Example: The code

```
(lambda (x y z)
 (f (g (if (h? x)
 (p y)
 (q (r z))))))
```

- ▶ First, we add a **fresh** continuation variable  $k$  to the list of parameters:

```
(lambda (x y z k) ...)
```

- ▶ Next, we evaluate the *test*-clause by calling  $h?$ :

```
(lambda (x y z k)
 (h?$ x
 (lambda (roh?)
 ...)))
```

# Converting to CPS (*continued*)

- ▶ Next, we test the `roh?`, the value of the *test-clause*:

```
(lambda (x y z k)
 (h? $ x
 (lambda (roh?)
 (if roh?
 ...
 ...))))
```

# Converting to CPS (*continued*)

- ▶ Next, we evaluate the *then*-clause by calling p:

```
(lambda (x y z k)
 (h? $ x
 (lambda (roh?)
 (if roh?
 (p$ y
 (lambda (rop)
 ...)))
 ...))))
```

- ▶ We have now exited one branch of the if-expression, only to fall back on the code that surrounds it: (f (g ...)).
  - ▶ We need to convert this code to CPS

# Converting to CPS (*continued*)

- We evaluate (g rop):

```
(lambda (x y z k)
 (h?$ x
 (lambda (roh)
 (if roh?
 (p$ y
 (lambda (rop)
 (g$ rop
 (lambda (rog)
 ...))))
 ...
))))
```

# Converting to CPS (*continued*)

- ▶ The return value of k is then passed on to f:

```
(lambda (x y z k)
 (h? $ x
 (lambda (roh?)
 (if roh?
 (p$ y
 (lambda (rop)
 (g$ rop
 (lambda (rog)
 (f$ rog
 (lambda (rof)
 ...
)))))))))
 ...
))))
```

# Converting to CPS (*continued*)

- We return `rof` by applying to it the continuation `k`:

```
(lambda (x y z k)
 (h? $ x
 (lambda (roh?)
 (if roh?
 (p$ y
 (lambda (rop)
 (g$ rop
 (lambda (rog)
 (f$ rog
 (lambda (rof)
 (k rof))))))))
 ...))))
```

# Converting to CPS (*continued*)

- The continuation `(lambda (rof) (k rof))`  $\eta$ -reduces to `k`, giving:

```
(lambda (x y z k)
 (h? $ x
 (lambda (roh?)
 (if roh?
 (p$ y
 (lambda (rop)
 (g$ rop
 (lambda (rog)
 (f$ rog k)))))))
 ...))))
```

## Converting to CPS (*continued*)

- Moving on with the *else*-clause of the *if*-expression, we need to CPS (q (r z)), so we evaluate (r z):

```
(lambda (x y z k)
 (h? $ x
 (lambda (roh?)
 (if roh?
 (p$ y
 (lambda (rop)
 (g$ rop
 (lambda (rog)
 (f$ rog k))))))
 (r$ z
 (lambda (ror)
 ...))))))
```

## Converting to CPS (*continued*)

- The result is passed on to q, so we call q with ror:

```
(lambda (x y z k)
 (h$ x
 (lambda (roh?)
 (if roh?
 (p$ y
 (lambda (rop)
 (g$ rop
 (lambda (rog)
 (f$ rog k))))))
 (r$ z
 (lambda (ror)
 (q$ ror
 (lambda (roq)
 ...)))))))
```

# Converting to CPS (*continued*)

- We have now exited the second branch of the if-expression, only to fall back on the code that surrounds it: (f (g ...))
- You should have a strong sense of déjà vu; We've been down that path before:

```
(lambda (x y z k)
 (h? $ x
 (lambda (roh?)
 (if roh?
 (p$ y
 (lambda (rop)
 (g$ rop
 (lambda (rog)
 (f$ rog k))))))
 (r$ z
 (lambda (ror)
 (q$ ror
 (lambda (roq)
 (g$ roq
 (lambda (rog)
 (f$ rog k)))))))))))
```

Compare these two continuations:

## continuation in *then-clause*

```
(lambda (rop)
 (g$ rop
 (lambda (rog)
 (f$ rog k))))
```

## continuation in *else-clause*

```
(lambda (roq)
 (g$ roq
 (lambda (rog)
 (f$ rog k))))
```

The only difference between them is the use of different parameter-names here  $\circ$  and here  $\circ$ . So we have a perfect duplication of the continuation

# CPS

We use a let-expression to factor-out the common code:

```
(lambda (x y z k)
 (let ((k2 (lambda (x)
 (g$ x
 (lambda (rog)
 (f$ rog k))))))
 (h?$ x
 (lambda (rob?)
 (if roh?
 (p$ y k2)
 (r$ z
 (lambda (ror)
 (q$ ror k2))))))))
```

Rather than duplicating whole lines of code, we duplicate a single variable name.

# Factorial in CPS

We want to write the recursive factorial directly in CPS (rather than converting it from the direct-style version)

The original function takes a single parameter  $n$ , so we add a **fresh** variable  $k$  :

```
(define fact$
 (lambda (n k)
 ...))
```

## Factorial in CPS (*continued*)

We need to test for zero, and `zero?` is builtin, so we don't convert it to CPS:

```
(define fact$
 (lambda (n k)
 (if (zero? n)
 ...
 ...)))
```

When  $n = 0$ , we return 1, so we pass it on to k:

```
(define fact$
 (lambda (n k)
 (if (zero? n)
 (k 1)
 ...)))
```

# Factorial in CPS (*continued*)

If  $n > 0$ , we recurse on  $n - 1$ :

```
(define fact$
 (lambda (n k)
 (if (zero? n)
 (k 1)
 (fact$ (- n 1)
 (lambda (n-1!)
 ...))))))
```

## Factorial in CPS (*continued*)

What do we do with  $(n - 1)!$ ? — Multiply it by  $n$ , and return, as we pass  $(* n n-1!)$  to  $k$ :

```
(define fact$
 (lambda (n k)
 (if (zero? n)
 (k 1)
 (fact$ (- n 1)
 (lambda (n-1!)
 (k (* n n-1!)))))))
```

# Factorial in CPS (*continued*)

How might we run fact\$?

```
> (fact$ 5 (lambda (5!) 5!))
120
> (fact$ 5 list)
(120)
> (fact$ 5
 (lambda (5!)
 ` (the answer is ,5!)))
(the answer is 120)
```

# The initial continuation

- ▶ Asking what should be the initial continuation is pretty much the same as asking what to do with the value of the function-call in direct style
  - ☞ The answer is: Whatever you want!
- ▶ When you have a function-call in direct style, and the call appears in some context:

```
> (... (foo x y) ...)
```

The way to call the CPS version of the function is to pass, as initial continuation, the CPS version of the context:

```
> (foo$ x y (lambda (result) (... result ...)))
```

# Fibonacci in CPS

We want to write the recursive function Fibonacci directly in CPS.  
The function in direct style takes a single argument  $n$ , so we add a  
fresh variable  $k$ :

```
(define fib$
 (lambda (n k)
 ...))
```

The function starts with the test ( $< n 2$ ), which is a call to the  
builtin predicate  $<$ , which is therefore not converted to CPS:

```
(define fib$
 (lambda (n k)
 (if (< n 2)
 ...
 ...)))
```

# Fibonacci in CPS (*continued*)

If the test holds, we return  $n$ , since  $(\text{fib } 0) \Rightarrow 0$ , and  $(\text{fib } 1) \Rightarrow 1$ . Returning  $n$  means, in CPS, passing it to the continuation  $k$ :

```
(define fib$
 (lambda (n k)
 (if (< n 2)
 (k n)
 ...)))
```

## Fibonacci in CPS (*continued*)

If the test does not hold, we need to add the *Fibonacci* of  $n - 1$  to the *Fibonacci* of  $n - 2$ . We can start with either recursive call, and we arbitrarily pick *Fibonacci* of  $n - 1$ :

```
(define fib$
 (lambda (n k)
 (if (< n 2)
 (k n)
 (fib$ (- n 1)
 (lambda (fib-n-1)
 ...)))))
```

# Fibonacci in CPS (*continued*)

Once back from the call, the IH/CPS tells us that the continuation is applied to *Fibonacci* of  $n - 1$ , so the value of the parameter `fib-n-1` corresponds to its name!

We then evaluate *Fibonacci* of  $n - 2$ :

```
(define fib$
 (lambda (n k)
 (if (< n 2)
 (k n)
 (fib$ (- n 1)
 (lambda (fib-n-1)
 (fib$ (- n 2)
 (lambda (fib-n-2)
 ...)))))))
```

# Fibonacci in CPS (*continued*)

Just as before, the value of `fib-n-2` is indeed the *Fibonacci* of  $n - 2$ . Mindful that  $\text{Fibonacci}(n) = \text{Fibonacci}(n - 1) + \text{Fibonacci}(n - 2)$ , we wish to return the sum of `fib-n-1` & `fib-n-2`, which means, in CPS, that we pass on the sum to the continuation `k`:

```
(define fib$
 (lambda (n k)
 (if (< n 2)
 (k n)
 (fib$ (- n 1)
 (lambda (fib-n-1)
 (fib$ (- n 2)
 (lambda (fib-n-2)
 (k (+ fib-n-1 fib-n-2))))))))))
```

# Fibonacci in CPS (*continued*)

How might we run fib\$?

```
> (fib$ 10 (lambda (fib10) fib10))
55
> (fib$ 10
 (lambda (fib10)
 ` (the result is ,fib10)))
(the result is 55)
```

# Fibonacci, revisited

- ▶ Up to now, we only experienced CPS as a representation into which we converted code written originally in direct style
- ▶ CPS was just a particularly obnoxious way to write code
- ▶ It's high time we have some fun with CPS, and use it in a way that may not correspond immediately to a program in direct style, and which showcases the special power of CPS
- ▶ At the moment, we have two ways of implementing the *Fibonacci* function:
  - ▶ We can use the recursive definition: This is terribly inefficient, exhibiting exponential complexity of  $O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right)$
  - ▶ We can use the iterative definition, that exhibits linear complexity

# Fibonacci, revisited

- We can do better!
- Consider the vector  $\begin{pmatrix} Fib_{n+1} \\ Fib_n \end{pmatrix}$ . The matrix  $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$  maps:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} Fib_{n+1} \\ Fib_n \end{pmatrix} = \begin{pmatrix} Fib_{n+2} \\ Fib_{n+1} \end{pmatrix}$$

- It follows by induction that

$$\begin{aligned} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n+1} \cdot \begin{pmatrix} Fib_1 \\ Fib_0 \end{pmatrix} &= \begin{pmatrix} Fib_{n+2} & Fib_{n+1} \\ Fib_{n+1} & Fib_n \end{pmatrix} \cdot \begin{pmatrix} Fib_1 \\ Fib_0 \end{pmatrix} \\ &= \begin{pmatrix} Fib_{n+2} \\ Fib_{n+1} \end{pmatrix} \end{aligned}$$

# Fibonacci, revisited

- ▶ Raising a matrix to the  $n$ -th power is straightforward:

$$\begin{aligned}M^{2n} &= (M^n)^2 \\ M^{2n+1} &= M \cdot M^{2n}\end{aligned}$$

- ▶ Putting this all together, it seems we can compute *Fibonacci* numbers with complexity that is **logarithmic** to  $n$
- ▶ How to represent the matrix?
  - ▶ We could use lists of lists, or vectors of vectors, but these would require allocation & garbage-collection

# Fibonacci, revisited

- ▶ How to represent the matrix?
  - ▶ We can use the stack!
    - ▶ We need to return 4 numbers — the components of a  $2 \times 2$  matrix
    - ▶ In assembly language, this would have been simple, since we can invent our own calling conventions
    - ▶ Most programming languages do not support the control paradigm of **multiple return-values**
    - ▶ We can easily model this paradigm using CPS:
    - ▶ We can use continuations that take 4 arguments, modeling a simultaneous return of 4 values from a function-call.

# Fibonacci, revisited

- ▶ How to represent the matrix?
  - ▶ We write an internal procedure `run`, that takes 6 arguments:
    - ▶ `a, b, c, d`: 4 elements of a  $2 \times 2$  matrix
    - ▶ `n`: the power of the exponentiation
    - ▶ `ret-abcd`: a 4-place continuation used to return the components of a  $2 \times 2$  matrix

# Fibonacci, revisited

- ▶ How to write an efficient *Fibonacci*:
  - ▶ We first test for  $n = 0$ : If satisfied, we return the identity matrix
  - ▶ We then test whether the power is even: If satisfied, we call `run` recursively on  $n/2$ , square the resulting matrix, and return:

$$\begin{aligned}M^2 &= \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} a & b \\ c & d \end{pmatrix} \\ &= \begin{pmatrix} a^2 + bc & b(a + d) \\ c(a + d) & bc + d^2 \end{pmatrix}\end{aligned}$$

- ▶ Otherwise, we call `run` recursively on  $\frac{n-1}{2}$ , square the result, multiply by the characteristic matrix, and return:

$$\begin{aligned}\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot M &= \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} a & b \\ c & d \end{pmatrix} \\ &= \begin{pmatrix} a+c & b+d \\ a & b \end{pmatrix}\end{aligned}$$

# Fibonacci, revisited

The code is as follows:

```
(define fib
 (letrec ((run
 (lambda (a b c d n ret-abcd)
 (if (zero? n)
 (ret-abcd 1 0 0 1)
 (run a b c d (quotient n 2)
 (lambda (a b c d)
 (let ((b*c (* b c))
 (a+d (+ a d)))
 (let ((a (+ (* a a) b*c))
 (b (* b a+d))
 (c (* c a+d))
 (d (+ b*c (* d d)))))
 (if (even? n)
 (ret-abcd a b c d)
 (ret-abcd (+ a c) (+ b d)
 a b)))))))))))
 (lambda (n)
 (run 1 1 1 0 n
 (lambda (fib-n+1 fib-n _fib-n fib-n-1)
 fib-n))))))
```

# *Fibonacci*, revisited

So how good is our strange, CPS-version of *Fibonacci*?

- ▶ The recursive implementation is out of the race quickly. Its exponential behavior means that it will take impractically-long to compute *Fibonacci* numbers for  $n > 60$  or so.

# Fibonacci, revisited

- The iterative implementation can compute *Fibonacci* numbers for quite large values of  $n$ . Here are some stats generated on my computer:

```
> (time (begin (fib-iterative 1000000) 'moshe))
(time (begin (fib-iterative 1000000) ...))
 1749 collections
 18.235483726s elapsed cpu time, including 0.135496834s
 collecting
 18.235416674s elapsed real time, including 0.140029295s
 collecting
 43407361504 bytes allocated, including 43366500208 bytes
 reclaimed
moshe
> (time (begin (fib 1000000) 'moshe))
(time (begin (fib 1000000) ...))
 no collections
 1.438607759s elapsed cpu time
 1.438611206s elapsed real time
 1307184 bytes allocated
moshe
```

# Fibonacci, revisited

- ▶ And for a larger value of  $n$ :

```
> (time (begin (fib-iterative 5000000) 'moshe))
(time (begin (fib-iterative 5000000) ...))
 9750 collections
 462.900038369s elapsed cpu time, including 3.110764949s
 collecting
 462.951576890s elapsed real time, including 3.139074913s
 collecting
 1084839153888 bytes allocated, including 1084638858736 bytes
 reclaimed
moshe
> (time (begin (fib 5000000) 'moshe))
(time (begin (fib 5000000) ...))
 no collections
 35.741472107s elapsed cpu time
 35.741057564s elapsed real time
 6530496 bytes allocated
moshe
```

# CPS & Higher-Order Functions



Recall Ffact, the factorial functional:

```
(define Ffact
 (lambda (fact)
 (lambda (n)
 (if (zero? n)
 1
 (* n (fact (- n 1)))))))
```

# CPS & Higher-Order Functions (*continued*)



Recall Ycurry, Curry's Y-combinator, which we applied to Ffact to obtain the factorial function:

```
(define Ycurry
 (lambda (f)
 ((lambda (x) (f (lambda (a) ((x x) a))))
 (lambda (x) (f (lambda (a) ((x x) a)))))))
```

Previously, we considered a **variadic** version of Ycurry, but we're not getting into converting variadic functions to CPS, so we shall just consider a **unary** version...

# CPS & Higher-Order Functions (*continued*)

- ▶ Let us convert **both** to CPS, so that we might define the **recursive** CPS version of factorial
- ▶ Starting with Ffact, we shall need to add a continuation k:

```
(define Ffact$
 (lambda (fact$ k)
 ...))
```

Notice that we take fact\$ rather than fact, just to remind us that we're now working with procedures in CPS

- ▶ We need to return the function (lambda (n) ....). Returning functions is just like returning numbers or strings or Booleans: We pass them to the continuation

# CPS & Higher-Order Functions (*continued*)

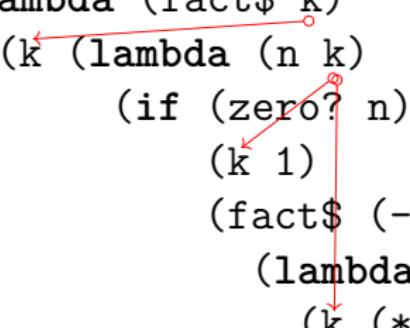
- ▶ Note too that we have a choice here: We could pass a non-CPS procedure (lambda (n) ...), or we could convert **that** procedure to CPS, which would require us to add a continuation variable to its parameter list, and convert its body to CPS. We are choosing the latter option:

```
(define Ffact$
 (lambda (fact$ k)
 (k (lambda (n k)
 ...))))
```

# CPS & Higher-Order Functions (*continued*)

- ▶ From now 'til the end, there are no surprises, and we continue just as we would have converted the recursive factorial to CPS:

```
(define Ffact$
 (lambda (fact$ k)
 (k (lambda (n k)
 (if (zero? n)
 (k 1)
 (fact$ (- n 1)
 (lambda (n-1!)
 (k (* n n-1!))))))))))
```



# CPS & Higher-Order Functions (*continued*)

- ▶ To work with `Ffact$`, we need to convert `Ycurry` as well. We throw in the added continuation variable `k`:

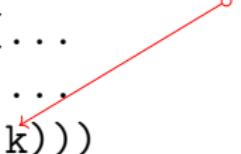
```
(define Ycurry$
 (lambda (f$ k)
 ...))
```

- ▶ Notice that we not only added `k`, but also noted that `f` is to be given in CPS as well. This is just a notational convenience, but it's an important one, as it shall keep us from messing up.

# CPS & Higher-Order Functions (*continued*)

- ▶ Notice that the body of `Ycurry` contains a tail-call, so `k` shall be passed on to it automatically:

```
(define Ycurry$
 (lambda (f$ k)
 (...
 ...
 k))))
```



- ▶ Also notice that the procedure and the first argument are **identical**

# CPS & Higher-Order Functions (*continued*)

- ▶ Next, we convert `(lambda (x) (f (lambda (a) ((x x) a)))` to CPS:
- ▶ We start by adding to `(lambda (x) ...)` a continuation `k` to its list of parameters:

```
(define Ycurry$
 (lambda (f$ k)
 ((lambda (x k) ...)
 (lambda (x k) ...)
 k)))
```

# CPS & Higher-Order Functions (*continued*)

- ▶ The `(lambda (x k) ...)` procedure has a tail-call in its body, so we pass it `k`:

```
(define Ycurry$
 (lambda (f$ k)
 ((lambda (x k)
 (f$...
 k))
 (lambda (x k)
 (f$...
 k))
 k)))
```

# CPS & Higher-Order Functions (*continued*)

- We add a continuation  $k$  to the `(lambda (a) ...)`, and continue as usual:

```
(define Ycurry$
 (lambda (f$ k)
 ((lambda (x k)
 (f$ (lambda (a k)
 (x x (lambda (rox) (rox a k))))
 k))
 (lambda (x k)
 (f$ (lambda (a k)
 (x x (lambda (rox) (rox a k))))
 k))
 k)))
```

The diagram illustrates the CPS transformation applied to the Ycurry\$ function. Red arrows highlight the flow of continuations. One arrow points from the outermost  $k$  to the innermost  $k$  in the first pair of nested lambdas. Another arrow points from the innermost  $k$  to the  $rox$  parameter in the nested lambda. A third arrow points from the outermost  $k$  to the innermost  $k$  in the second pair of nested lambdas, and so on.

# CPS & Higher-Order Functions (*continued*)

- ▶ Testing our code:

```
> ((Ycurry Ffact) 5)
120
> (Ycurry$ Ffact$
 (lambda (fact$)
 (fact$ 5
 (lambda (5!)
 ` (the result is ,5!))))))
(the result is 120)
```

# Multiple Continuations

- ▶ It is often useful to manage more than one continuation
- ▶ **Example:** You have already encountered the problem of identifying a parameter-list of a lambda-expression. We wish to distinguish between:
  - ▶ proper list, which denotes a simple lambda-expression
  - ▶ improper list, which denotes a lambda-expression that may take optional arguments
  - ▶ symbol, which denotes a variadic lambda-expression
- ▶ When we identify an improper list, we want to separate the proper list to the left of the *dot* from the [non-nil] S-expression to the right of the *dot*
  - ▶ We do all this in a single pass, using 3 continuations

# Multiple Continuations (*continued*)

- ▶ Our procedure `param-list$` takes 4 arguments:
  - ▶ `e`: the S-expression we are analyzing
  - ▶ `ret-simple`: a continuation invoked when the parameter-list identifies a simple lambda-expression
  - ▶ `ret-opt`: a continuation invoked when the parameter-list identifies a lambda-expression with optional arguments
  - ▶ `ret-var`: a continuation invoked when the parameter-list identifies a variadic lambda-expression
- ▶ We begin with the template of the procedure:

```
(define param-list$
 (lambda (e ret-simple ret-opt ret-var)
 ...))
```

# Multiple Continuations (*continued*)

- ▶ We distinguish 3 cases:
  - ▶ e is the empty list: This identifies a simple lambda-expression
  - ▶ e is a symbol: This identifies a variadic lambda-expression **at this point**
  - ▶ e is a pair: We need to examine further We develop the template:

```
(define param-list$
 (lambda (e ret-simple ret-opt ret-var)
 (cond ((null? e) (ret-simple))
 ((symbol? e) (ret-var))
 ((pair? e) ...)
 (else (error 'param-list$
 (format "Not a parameter
list"))))))
```

# Multiple Continuations (*continued*)

- ▶ The `ret-simple`, `ret-var` continuations take no arguments:  
We found information that needs to be returned...
- ▶ In case of a *pair* we call `param-list$` recursively on `(cdr e)`:

```
(define param-list$
 (lambda (e ret-simple ret-opt ret-var)
 (cond ((null? e) (ret-simple))
 ((symbol? e) (ret-var))
 ((pair? e)
 (param-list$ (cdr e)
 ...
 ...
 ...))
 (else (error 'param-list$
 (format "Not a parameter list"))))))
```

# Multiple Continuations (*continued*)

- ▶ Nothing changes if the param-list\$ of (cdr e) is simple, so the ret-simple continuation does not change:

# Multiple Continuations (*continued*)

- ▶ In case  $(cdr e)$  is an improper list, then so is  $e$ . The only thing that changes are the *left* and *right* parts of the improper list
- ▶ If  $s$  and  $a$  are the *left* and *right* parts of  $(cdr e)$ , then  $`(, (car e) , @s)$  and  $a$  are the *left* and *right* parts of  $e$ , so this is what we return:

```
(define param-list$
 (lambda (e ret-simple ret-opt ret-var)
 (cond ((null? e) (ret-simple))
 ((symbol? e) (ret-var))
 ((pair? e)
 (param-list$ (cdr e)
 ret-simple
 (lambda (s a)
 (ret-opt `(, (car e) , @s) a))
 ...))
 (else (error 'param-list$
 (format "Not a parameter list"))))))
```

# Multiple Continuations (*continued*)

- ▶ The last case is the subtlest: If  $(cdr e)$  is a symbol (which is why  $\text{ret-var}$  was called), this means that  $e$  is an improper list
- ▶ The *left* and *right* parts of the improper list are  $`(, (car e))$  and  $(cdr e)$ , so we return these using  $\text{ret-opt}$ :

```
(define param-list$
 (lambda (e ret-simple ret-opt ret-var)
 (cond ((null? e) (ret-simple))
 ((symbol? e) (ret-var))
 ((pair? e)
 (param-list$ (cdr e)
 ret-simple
 (lambda (s a)
 (ret-opt `',(car e) ,@s) a))
 (lambda ()
 (ret-opt `',(car e)) (cdr e))))))
 (else (error 'param-list$
 (format "Not a parameter list"))))))))
```

# Multiple Continuations (*continued*)

- ▶ To test param-list\$ conveniently, we write an API function:

```
(define analyze-param-list
 (lambda (e)
 (param-list$ e
 (lambda () `(~(,e denotes a simple lambda)))
 (lambda (s a)
 `(~(,e denotes a lambda with
 required args ,s and
 optional ,a)))
 (lambda () `(~(,e denotes a variadic lambda))))))
```

# Multiple Continuations (*continued*)

- We can run it as follows:

```
> (analyze-param-list '(a b c))
((a b c) denotes a simple lambda)
> (analyze-param-list '(a b . c))
((a b . c) denotes a lambda with required args (a b)
 optional c)
> (analyze-param-list 'moshe)
(moshe denotes a variadic lambda)
```

# Further reading

-  *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Grady Booch
-  *Head First Design Patterns*, by Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra
-  *Refactoring to Patterns*, by Joshua Kerievsky
  -  Control
  -  Continuation
    -  CPS
  -  OOP
  -  Functional Programming

# Further reading (*continued*)

## ► Loop mechanisms

-  Iteration
-  *do-while-loop*
-  *for-loop*
-  *foreach-loop*
-  *while-loop*

## Design Patterns

### Behavioural Design Patterns

-  *Command*
-  *Iterator*
-  *MVC*
-  *Observer*
-  *Strategy*
-  *Visitor*

# The stack machine

## Roadmap

- ✓ Control in Programming Languages
- ✓ Continuations
- ✓ Continuation-Passing Style
- 👉 The Stack-Machine
  - ▶ CPS
  - ▶ Representation-independence of the continuation
  - ▶ Continuations as records
  - ▶ Continuations as flat records
  - ▶ CISC assembly language

# From direct style to CPS

- ▶ A pipeline of 5 transformations carries us from a program in direct style to CISC-like assembly language
- ▶ The first & most significant of these is the CPS transformation, which linearizes control
- ▶ After CPS, a sequence of relatively straightforward transformations changes the representation of the continuation, and converts the code to CISC-like assembly language
- ▶ By the end of this presentation, you should have a solid understanding
  - ▶ How a CPS translator is substantially a compiler
  - ▶ How to implement **any** control paradigm in any Turing-complete programming language
  - ▶ What continuations are all about...

# State I — CPS

- ▶ Starting with a program in direct style, we convert it to CPS:

## Direct Style

```
(define fact
 (lambda (n)
 (if (zero? n)
 1
 (* n (fact (- n 1))))))
```

## CPS

```
(define fact$
 (lambda (n k)
 (if (zero? n)
 (k 1)
 (fact$ (- n 1)
 (lambda (x)
 (k (* n x)))))))
```

## Usage:

```
> (fact$ 5 (lambda (x) x))
```

120

## Roadmap

- ✓ Control in Programming Languages
- ✓ Continuations
- ✓ Continuation-Passing Style
- The Stack-Machine
  - ✓ CPS
  - Representation-independence of the continuation
  - Continuations as records
  - Continuations as flat records
  - CISC assembly language

# Stage II

Starting with the program in CPS, we transform it so that it is Representation-Independent with respect to the continuation.

## Representation-Independence

- ▶ This may come as a surprise, since you have been using closures to represent continuations ever since we first saw continuations, but closures are just one of many possible data-types with which to represent continuations
- ▶ Since closures are a relatively high-level structure, and we are interested in compiling code to assembly language, we would ultimately like to **replace** closures as our representation for continuations with another, simpler data-structure that is closer to the underlying machine

## Stage II (*continued*)

### Representation-Independence (*continued*)

- ▶ The representation of the continuation is currently exposed in our CPS version of factorial. Our immediate goal is to move the details of the representation behind an API that provides access, but hides implementation.
- ▶ Such a transformation would make the factorial function **representation-independent relative to the continuation**
  - ▶ This means we could change the underlying representation without having to change the code for the factorial function

## Stage II (*continued*)

- ▶ Looking at our code in CPS, we notice several places that disclose that the continuation is implemented as a closure:

```
(define fact$
 (lambda (n k)
 (if (zero? n)
 (k 1)
 (fact$ (- n 1)
 (lambda (x)
 (k (* n x)))))))
```

- ▶ ◦ Application of the continuation-closure
- ▶ ◦ Creation of the continuation-closure

## Stage II (*continued*)

- ▶ Continuations are either **created** or **applied**: A natural API for working with continuations should provide for these two operations:

```
(define fact$ri
 (lambda (n k)
 (if (zero? n)
 (apply-k k 1)
 (fact$ri (- n 1)
 (^k-fact n k))))))
```

## Stage II (*continued*)

```
(define apply-k
 (lambda (k x)
 (k x)))
(define ^k-init
 (lambda ()
 (lambda (x) x)))
(define ^k-fact
 (lambda (n k)
 (lambda (x)
 (apply-k k (* n x))))))
```

**Notation:** My own notational convention is to read the  $\wedge$  character as “make”, so that  $\wedge k\text{-init}$  is read as *make-k-init*...

# Stage II (*continued*)

## ► Questions

- ▶ How do we know to pass n, k to  $\text{^k-fact}$ ?
- ▶ Why does  $\text{^k-fact}$  take 2 arguments?

## ► Answer

- ▶ The original code for the continuation is  
`(lambda (x) (k (* n x)))`
  - ▶ \* is a **free var**, and comes from the top-level env
  - ▶ x is a **parameter**, and comes off the stack
  - ▶ n, k are **bound vars**, and come from the lexical environment of the closure that represents the continuation
- ▶ We only need to pass the bound vars!
- 👉 Always pass the continuation k as the **last argument** (explained later)

## Stage II (*continued*)

### ► Question

- ▶ How did we know to pass nothing to  $\text{^k-init}$ ?
- ▶ Why does  $\text{^k-init}$  take no arguments?

### ► Answer

- ▶ The original code for the continuation is `(lambda (x) x)`
  - ▶ There are no bound variables in this code!

Here is how you run the code:

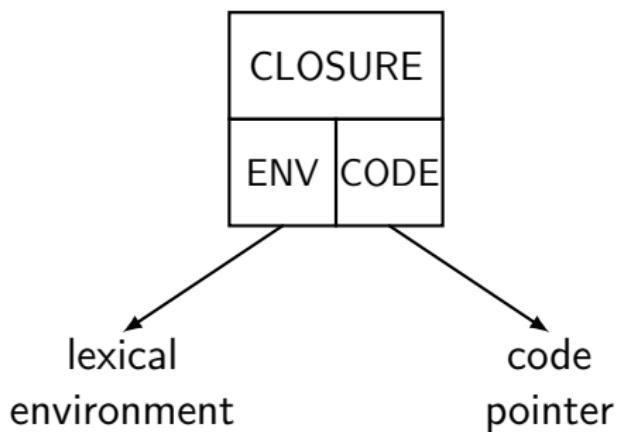
```
> (fact$ri 5 (^k-init))
120
```

## Roadmap

- ✓ Control in Programming Languages
- ✓ Continuations
- ✓ Continuation-Passing Style
- The Stack-Machine
  - ✓ CPS
  - ✓ Representation-independence of the continuation
  - Continuations as records
  - Continuations as flat records
  - CISC assembly language

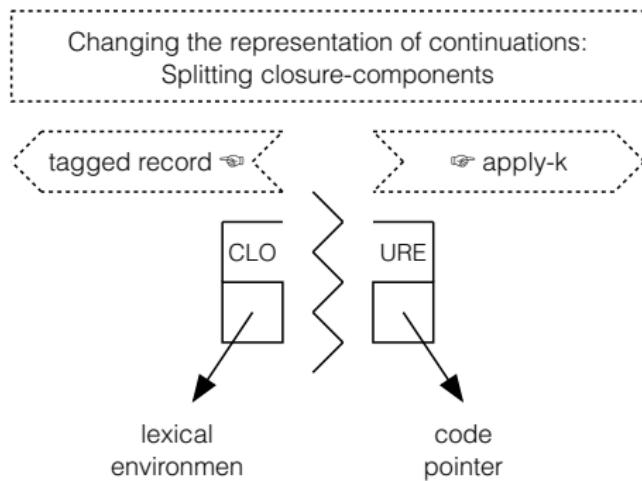
# Stage III

- ▶ Now that the code is independent of the way we represent continuations, we are able to change easily the representation of continuations, from closures, to other data structures, just by changing the definitions of  $\hat{k}\text{-init}$ ,  $\hat{k}\text{-fact}$ , and  $\text{apply-}k$
- 💡 A closure is a data structure that combines a **lexical environment** and a **code pointer**, that is **data** and **functionality**:



## Stage III (*continued*)

- Rather than using closures, we are going to represent continuations using **tagged records**, that contain a **name** for the continuation, and any **data** in the lexical environments, that is the **values** of the bound variables
- The **functionality** of the closure shall move into apply-k



## Stage III (*continued*)

We change the API functions of our continuations:

```
(define ^k-init
 (lambda ()
 `(k-init)))
(define ^k-fact
 (lambda (n k)
 `(k-fact ,n ,k)))
```

- 💣 Of course, this change breaks our existing code. The problem shall come up in `apply-k`, where we attempt to **apply** these lists, in place of closures:

```
(define apply-k
 (lambda (k x) (k x)))
```

# Scheme break...

- ▶ This is nasty:

```
(let ((first (car s))
 (second (cadr s))
 (third (caddr s))
 (fourth (cadaddr s))
 ; cXr is defined up to 4 {A|D}s
 (fifth (cadaddr (cdr s))))
 (sixth (cadaddr (cddr s))))
 <something interesting>)
```

# Scheme break...

This is neater:

```
(apply (lambda (first second third fourth fifth sixth)
 <something interesting>)
 s)
```

But since the procedure can span several lines, it's neater-yet to have it last, so we define:

```
(define with (lambda (s f) (apply f s)))
```

and then we can write:

```
(with s
 (lambda (first second third fourth fifth sixth)
 <something interesting>))
```

## Stage III (*continued*)

We need to move to apply-k the functionality (i.e., the activities at the code pointer) of the continuation-closures:

```
(define apply-k
 (lambda (k x)
 (cond ((eq? (car k) 'k-init) x)
 ((eq? (car k) 'k-fact)
 (with k
 (lambda (_ n k)
 (apply-k k (* n x))))))
 (else (error 'apply-k
 "I don't recognize this
continuation")))))
```

## Stage III (*continued*)

And now `fact$ri` works as usual:

```
> (fact$ri 5 (^k-init))
120
```

Because the code is representation-independent with respect to the continuations, the implementation of `fact$ri` is **invariant** under a change of representation!

## Stage III (*continued*)

- Our continuations are **nested**, and have the structure:

```
(k-fact 1
 (k-fact 2
 (k-fact 3
 (k-fact 4
 (k-fact 5
 (k-init))))))
```

- The reason we insisted on the continuation being the **last** argument to our continuation-constructor procedures ( $\text{^k-init}$ ,  $\text{^k-fact}$ ) is so that it also be the last field in the **continuation-record**
- This gives us license to **flatten** the continuation structure, and avoid the nesting

## Roadmap

- ✓ Control in Programming Languages
- ✓ Continuations
- ✓ Continuation-Passing Style
- The Stack-Machine
  - ✓ CPS
  - ✓ Representation-independence of the continuation
  - ✓ Continuations as records
  - Continuations as flat records
  - CISC assembly language

# Stage IV

As before, we can swap the representation for continuations by changing only the continuation-API procedures:

## Nested Continuations

```
(define ^k-init
 (lambda ()
 `(k-init)))
(define ^k-fact
 (lambda (n k)
 `(k-fact ,n ,k)))
```

## Flat Continuations

```
(define ^k-init
 (lambda ()
 `(k-init)))
(define ^k-fact
 (lambda (n k)
 `(k-fact ,n . ,k)))
```

# Stage IV

## ► Nested Continuations

```
(define apply-k
 (lambda (k x)
 (cond ((eq? (car k) 'k-init) x)
 ((eq? (car k) 'k-fact)
 (with k
 (lambda (_ n k)
 (apply-k k (* n x))))))
 (else (error 'apply-k
 "I don't recognize this continuation")))))
```

## ► Flat Continuations

```
(define apply-k
 (lambda (k x)
 (cond ((eq? (car k) 'k-init) x)
 ((eq? (car k) 'k-fact)
 (with k
 (lambda (_ n . k)
 (apply-k k (* n x))))))
 (else (error 'apply-k
 "I don't recognize this continuation")))))
```

## Stage IV (*continued*)

- ▶ Just as before, the change of representation is transparent to the fact\$ri procedure

```
> (fact$ri 5 (^k-init))
120
```

Our continuations are now flat:

```
(k-fact 1 k-fact 2 k-fact 3 k-fact 4 k-fact 5 k-init)
```

- ▶ A continuation-tag now denotes a continuation “from here to the end”
  - ▶ Had k not been the last field this would not have been possible!

## Roadmap

- ✓ Control in Programming Languages
- ✓ Continuations
- ✓ Continuation-Passing Style
- The Stack-Machine
  - ✓ CPS
  - ✓ Representation-independence of the continuation
  - ✓ Continuations as records
  - ✓ Continuations as flat records
  - CISC assembly language

# Stage V

- ▶ Now that we have a flat LIFO structure, we write it top-to-bottom, and we get ourselves a standard, micro-architectural stack!
- ▶ We are ready for the final transformation to the **Stack Machine**

Zhè shì yīgè duīzhàn

k\_init  
5  
k-fact  
4  
k-fact  
3  
k-fact  
2  
k-fact  
1  
k-fact  
)

这  
是  
一  
个  
堆  
栈

## Stage V (*continued*)

- ▶ The Stack Machine is an abstraction of a CISC-like micro-architecture:
  - ▶ We assume the machine has as many registers as needed
  - ▶ We can perform **atomic** operations
    - ▶ Any primitive Scheme procedure! 😊
  - ▶ We have labels
  - ▶ We have branches, unconditional & conditional
  - ▶ We have conditionals
  - ▶ We have assignments
  - ▶ We have arbitrary memory
  - ▶ We have a stack

## Stage V (*continued*)

- ▶ The Stack Machine is an abstraction of a CISC-like micro-architecture:
  - ▶ All functions are thunks (take 0 arguments)
  - ▶ All function-calls are in tail-position
  - ▶ This means that functions are merely **labels**, and function-calls are merely **jumps to labels**
  - ▶ All function-bodies are flat (except for if and cond nestings)

## Stage V (*continued*)

This is not a realistic architecture, but it's pretty close:

- ▶ Builtin Scheme procedures can be managed through call to a library function
- ▶ if, cond nestings can be managed through cmp and conditional branches
- ▶ Registers can be managed through **register-allocation** algorithms, register-stacks, etc
- ▶ Basically, any instruction in our “assembly language” should be possible to compile to Intel x86, ARM, or some other real assembly using 1-2-3 instructions (modulo exotic stuff like floating-point numbers)

# Stage V (*continued*)

## CPS/RI/Records

```
(define fact$ri
 (lambda (n k)
 (if (zero? n) (apply-k k 1)
 (fact$ri (- n 1) (k-fact n k))))))
```

## Stack Machine

```
(define fact$sm
 (lambda ()
 (if (zero? n)
 (begin
 (set! x 1)
 (set! k (pop))
 (apply-k$sm))
 (begin
 (push n)
 (set! n (- n 1))
 (push 'k-fact)
 (fact$sm))))))
 → (if (zero? n)
 (begin
 (set! x 1)
 (set! k (pop))
 (apply-k$sm))
 (begin
 (push n)
 (set! n (- n 1))
 (push 'k-fact)
 (fact$sm))))
```

# Stage V (*continued*)

## CPS/RI/Records

```
(define apply-k
 (lambda (k x)
 (cond ((eq? (car k)
 'k-init)
 x)
 ((eq? (car k)
 'k-fact)
 (with k
 (lambda (_ n○k)
 (apply-k○k○
 (* n x○)))))
 (else ...))))
```

## Stack Machine

```
(define apply-k$sm
 (lambda ()
 (cond ((eq? k 'k-init)
 →x)
 ((eq? k 'k-fact)
 →(set! n (pop)))
 →(set! x (* n x)))
 →(set! k (pop))
 →(apply-k$sm))
 (else ...))))
```

## Stage V (*continued*)

This is our “hardware”, if you can believe it:

```
(define n 'moshe) ; register
(define k 'moshe) ; register
(define x 'moshe) ; register
(define stack 'moshe) ; we had better initialize!
(define push
 (lambda (e)
 (set! stack
 (cons e stack))))
(define pop
 (lambda ()
 (let ((e (car stack)))
 (set! stack (cdr stack))
 e)))
```

## Stage V (*continued*)

We write a simple API function so it's nice to call fact\$sm:

```
(define fact
 (lambda (_n)
 (set! n _n)
 (set! stack '())
 (push 'k-init)
 (fact$sm)))
```

And we call fact as usual:

```
> (fact 5)
120
```

# Stage V (*continued*)

## Is this real assembly code?

- ▶ What's the deal with `apply-k$sm`: “I wrote many x86 assembly-language programs in my architecture course, and never wrote this kind of code!”
- ▶ What's the deal with **symbols** as continuation-tags:
  - ▶ There are no symbols in x86
  - ▶ We can replace symbols with an *enum* or some other set of integer constants, but this doesn't look like normal assembly code!

# Stage V (*continued*)

## Is this real assembly code?

- ▶ In fact, `apply-k$sm` doesn't exist: It's an optical illusion!
- ▶ There are no continuation-tags: They're optical illusions too!
- ▶ `apply-k$sm` is an abstract dispatcher for **return addresses**
  - ▶ It's a way of talking about what happens at an address, when the address is given abstractly, by **label**, rather than concretely, by some **integer**
  - ▶ `apply-k$sm` doesn't really exist in assembly language:
    - ▶ The cond-ribs in `apply-k$sm` are just labels in assembly code
    - ▶ A jump to `apply-k$sm` (tail-position!) is always preceded by `(set! k (pop))`
    - ▶ The combination  
`(set! k (pop))`  
`(apply-k$sm); tail-call!`  
is called in x86 `ret` ☺

# Stage V (*continued*)

## Is this real assembly code?

- ▶ Pushing a continuation-tag is **always** done before a tail-call to a procedure

- ▶ The combination

...

(push 'k-something)

(P) ; this is a tail-call!

corresponds to the assembly language code

...

call P

k\_something:

where k\_something is a label the value of which is the **return address** of the call

- ▶ We are **very close** to a realistic assembly language!

## Stage V (*continued*)

Just once, just for the sake of a demonstration, let's go all the way:

- ▶ We convert fact to x86/64 assembly language
- ▶ We display **in parallel** the original Scheme code for the **stack machine**
  -  Notice what is involved in the change
  -  Decide for yourself just how straightforward is the translation
- ▶ To keep things as similar as possible to the **stack machine** code, we define the following aliases:

```
%define N rbx
%define X rax
```

- ▶ Now you don't need to remember which register in x86/64 is used for which register in the **stack machine**

# Stage V (*continued*)

fact:

```
 cmp N, 0 ; (if (zero? n)
 jne .not_zero ; (begin
 mov X, 1 ; (set! x 1)
 ; (set! k (pop))
 ret ; (apply-k))
```

.not\_zero:

```
; (begin
 push N ; (push n)
 dec N ; (set! n (- n 1))
 ; (push 'k-fact)
 call fact ; ((fact)()))
; Inside APPLY-K:
; ((eq? k 'k-fact)
 pop N ; (set! n (pop))
 cqo ;
 mul N ; (set! x (* n x))
 ; (set! k (pop))
 ret ; (apply-k))
```

# Stage V (*continued*)

## Conclusion

- ▶ The code in x86/64 assembly is actually shorter than the pseudo-assembly in Scheme
- ▶ The code should be amply readable to anyone who can program in x86/64 assembly
- ▶ The only “complications” come from the way x86/64 provides for multiplication & division
- 👉 Check out the `cps-examples.{scm, pdf}` files, on the course website
- 👉 Check out the complete examples of converting the **stack machine** to assembly, on the course website

# Further reading\*

-  Essentials of Programming Languages, 2nd edition, by *Daniel P Friedman, Mitchell Wand, Christopher T Haynes*. Yes, I specifically intend the **second edition**, and not a later one!
-  Compiling with Continuations, by *Andrew Appel*

## Roadmap

- ✓ Control in Programming Languages
- ✓ Continuations
- ✓ Continuation-Passing Style
- ✓ The Stack-Machine
  - ✓ CPS
  - ✓ Representation-independence of the continuation
  - ✓ Continuations as records
  - ✓ Continuations as flat records
  - ✓ CISC assembly language

## Goals

- 👉 Asynchronous Computing
- ▶ Coroutines, Threads & Processes
- ▶ Context-switching & tail-position
- ▶ The two-thread architecture

# Asynchronous Computing

- ▶  $\alpha$ - — the prefix “not”, “un-”
- ▶  $\sigma\nu$ - — the prefix “with”, “together”
- ▶  $\chi\rho\nu\omega\varsigma$  — the Greek God of time; time
- ▶  $\sigma\nu\chi\rho\nu\omega\varsigma$  — in time, in order, in phase, in step
- ▶ **Synchronous** computing means **sequential** computing
- ▶ **Asynchronous** computing means non-sequential computing, or computing **in parallel**



# Asynchronous Computing (*continued*)

- ▶ True asynchronous computing requires truly independent computing mechanisms that can operate concurrently: More than one ALU, core, processor, etc.
- ▶ Asynchronous computing is often simulated or augmented through **interleaving computation**

# Asynchronous Computing (*continued*)

## How to interleave computation

- We begin with several, independent computations:

COMPUTATION A

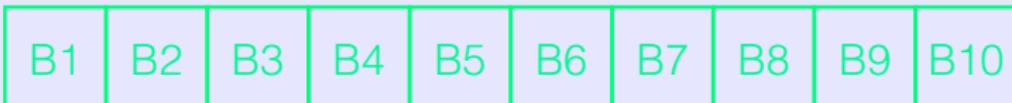
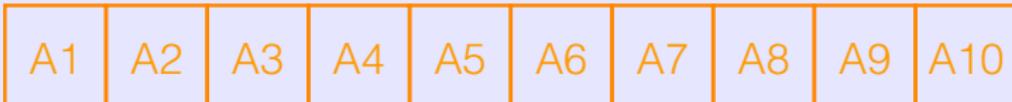
COMPUTATION B

COMPUTATION C

# Asynchronous Computing (*continued*)

## How to interleave computation (*cont*)

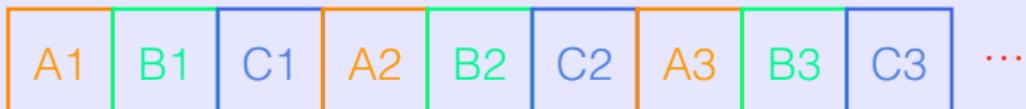
- ▶ Each computation is split into small, sequential chunks:



# Asynchronous Computing (*continued*)

## How to interleave computation (*cont*)

- ▶ The chunks of the different computations are interleaved, and performed in sequence:



creating an illusion of true, parallel computation

- 👉 Often interleaved and true asynchronous computing are combined, to create a user-experience of greater asynchronous capabilities than the hardware can offer
- 👉 The transition between chunks of different computations is known as **task-switching**, or **context-switching**

# Task Switching

There are different ways to perform task-switching:

## [①] Coöperative multitasking

- ▶ Under coöperative multitasking, each task must relinquish control voluntarily and explicitly, by means of specific operators
  - ▶ If it fails to do so, either by design or because of a bug, the task will continue indefinitely, often hanging the system, or until it terminates
  - ▶ Coöperative multitasking was used in Mac OS, the operating system used on Macintosh computers before the advent of OSX

# Task Switching

There are different ways to perform task-switching:

## [②] Pre-emptive multitasking

- ▶ Special hardware (e.g., a timer-interrupt facility) is used to wrest control from the current task and pass it onto another
- ▶ No coöperation is required of the current task
- ▶ Generally, this kind of task-switching cannot be prevented
- ▶ This is how task-switching is implemented on modern operating systems running on modern hardware
- ▶ The opposite of pre-emptive multitasking is **non-pre-emptive multitasking**, also known as **cooperative multitasking**...

# Task Switching

There are different ways to perform task-switching:

## [③] Multitasking through instrumentation

- ▶ **Instrumentation** means that the code is processed/transformed/manipulated by the compiler, prior to execution, so as to relinquish control explicitly
- ▶ As with preemption, instrumented code cannot affect or prevent task-switching
- 👉 The technique we shall present in this chapter is a form of instrumentation:
  - ▶ We show how to do it **manually**
  - ▶ Since the transformation is algorithmic, it can be done **automatically**

# Chapter 10

## Goals

- ✓ Asynchronous Computing
- 👉 Coroutines, Threads & Processes
- ▶ Context-switching & tail-position
- ▶ The two-thread architecture

# Asynchronous Tasks

Asynchronous tasks are distinguished by the **kinds of information** shared among different tasks:

- ▶ **Coroutines:** All coroutines share the same stack and the same heap
  - ▶ **Threads:** Each thread has its own stack, but all threads share the same heap
  - ▶ **Processes:** Each process has its own stack & heap
- 👉 In this presentation, we focus on **threads**

# Chapter 10

## Goals

- ✓ Asynchronous Computing
- ✓ Coroutines, Threads & Processes
- 👉 Context-switching & tail-position
- ▶ The two-thread architecture

# Context-Switching in Threads & Tail-Position

- ▶ Each thread may consist of procedures calling each other, and stacking up activation frames
- ▶ Since each thread comes with its own stack, the activation frames of one thread never intermix with those of other threads
- ▶ Rather than implement several stacks, we require that all non-builtin calls (i.e., calls to user-defined procedures) be in **tail-position**, so they either do not require the stack, or they use **continuations** to model the stack:
  - ▶ We use the **CPS-transformation** to convert all user-defined code into tail-position
  - ▶ By applying the CPS-transformation, the thread-specific stack is implemented using a **continuation**
  - ▶ Because of lexical scope, **one thread cannot access the continuations (i.e., stack) of another thread**, which is precisely how we know that we've implemented **threads rather than coroutines**

# Chapter 10

## Goals

- ✓ Asynchronous Computing
- ✓ Coroutines, Threads & Processes
- ✓ Context-switching & tail-position
- 👉 The two-thread architecture
  - ▶ Instrumenting code for the 2-thread architecture
  - ▶ Racing & termination
  - ▶ Detecting circularity
  - ▶ Prioritization
  - ▶ Threads & Types in Ocaml

# The two-thread architecture

- ▶ We present a very simple, 2-thread architecture, that we are tempted to call **thread-passing style** (TPS)
  - ▶ The name “2-thread architecture” refers to the API, and is not meant to imply that it is limited to two threads: In fact, any number of threads can be interleaved
- ▶ A thread is implemented as a procedure that takes a single thread  $t$ :
  - ▶ The thread performs some simple, atomic operation, after which the thread  $t$  is applied (in **tail-position**) to another thread that continues the original computation
  - ▶ The thread variable  $t$  is always used as a **parameter**
- ▶ Each user-defined procedure & continuation takes a thread as an additional argument, and invokes it after some simple, atomic operation

# Chapter 10

## Goals

- ✓ Asynchronous Computing
- ✓ Coroutines, Threads & Processes
- ✓ Context-switching & tail-position
- 👉 The two-thread architecture
  - 👉 Instrumenting code for the two-thread architecture
    - ▶ Racing & termination
    - ▶ Detecting circularity
    - ▶ Prioritization
    - ▶ Threads & Types in Ocaml

# Instrumenting code

We now present some simple examples of code that is instrumented to run as a thread:

- ▶ The code is just the instrumented code
- ▶ It does not come with any code to invoke it
- ▶ Later, after we are proficient in converting Scheme source code to instrumented code, we shall consider complete examples, i.e., schedule threads to run concurrently

# Instrumenting code (*continued*)

Example 1: (lambda (x) (f (g x)))

We start by converting the code to CPS:

```
(lambda (x k)
 (g$ x
 (lambda (rog)
 (f$ rog k))))
```

# Instrumenting code (*continued*)

Example 1: `(lambda (x) (f (g x)))`

We now instrument the code to run as a thread:

```
(lambda (x k t)
 (t (lambda (t)
 (g$ x
 (lambda (rog t)
 (t (lambda (t)
 (f$ rog k t))))))
 t))))
```

# Instrumenting code (*continued*)

Example 2: (lambda (x y) (f (g x) (h y)))

- ▶ As before, we convert the code to CPS
- ▶ We choose, arbitrarily, to start with the application (g x)

```
(lambda (x y k)
 (g$ x
 (lambda (rog)
 (h$ y
 (lambda (roh)
 (f$ rog roh k))))))
```

# Instrumenting code (*continued*)

Example 2: (lambda (x y) (f (g x) (h y)))

We instrument the code to take and pass control onto another thread:

```
(lambda (x y k t)
 (t (lambda (t)
 (g$ x
 (lambda (rog t)
 (t (lambda (t)
 (h$ y
 (lambda (roh t)
 (t (lambda (t)
 (f$ rog roh k t)))))))
 t))))))
 t))))
```

# Instrumenting code (*continued*)

Example 3:

```
(lambda (x) (if (w? x) (f x) (g (h x))))
```

We start by converting the code to CPS:

```
(lambda (x k)
 (w?$ x
 (lambda (row?)
 (if row?
 (f$ x k)
 (h$ x
 (lambda (roh)
 (g$ roh k)))))))
```

# Instrumenting code (*continued*)

Example 3:

```
(lambda (x) (if (w? x) (f x) (g (h x))))
```

We now instrument the code to take and pass control onto another thread:

```
(lambda (x k t)
 (t (lambda (t)
 (w?$ x
 (lambda (row? t)
 (if row?
 (t (lambda (t)
 (f$ x k t)))
 (t (lambda (t)
 (h$ x
 (lambda (roh t)
 (t (lambda (t)
 (g$ roh k t))))))))
 t)))))
```

# Complete examples of instrumented, two-thread code

We now define threaded versions of procedures that perform meaningful, well-known computation:

- ▶ Starting with the program in direct style, we convert it to CPS
- ▶ We then instrument the CPS version to run as a thread
- ▶ After all the procedures have been converted to threads, we demonstrate how to run them the computation in an interleaved fashion

## Example 4: Factorial

### Example 4: Direct Style

```
(define fact
 (lambda (n)
 (if (zero? n)
 1
 (* n (fact (- n 1))))))
```

## Example 4: Factorial

### Example 4: CPS

```
(define fact$
 (lambda (n k)
 (if (zero? n)
 (k 1)
 (fact$ (- n 1)
 (lambda (rof)
 (k (* n rof)))))))
```

## Example 4: Factorial

### Example 4: Instrumented for multi-threading

```
(define t$fact$
 (lambda (n k t)
 (t (lambda (t)
 (if (zero? n)
 (t (lambda (t)
 (k 1 t)))
 (t (lambda (t)
 (t$fact$ (- n 1)
 (lambda (rof t)
 (t (lambda (t)
 (k (* n rof) t))))
 t))))))))
```

## Example 5: Fibonacci

### Example 5: Direct Style

```
(define fib
 (lambda (n)
 (if (< n 2)
 n
 (+ (fib (- n 1))
 (fib (- n 2))))))
```

# Example 5: Fibonacci

## Example 5: CPS

```
(define fib$
 (lambda (n k)
 (if (< n 2)
 (k n)
 (fib$ (- n 1)
 (lambda (fib-1)
 (fib$ (- n 2)
 (lambda (fib-2)
 (k (+ fib-1 fib-2))))))))))
```

# Example 5: Fibonacci

## Example 5: Instrumented for multi-threading

```
(define tfib
 (lambda (n k t)
 (t (lambda (t)
 (if (< n 2)
 (t (lambda (t)
 (k n t)))
 (t (lambda (t)
 (tfib (- n 1)
 (lambda (fib-1 t)
 (t (lambda (t)
 (tfib (- n 2)
 (lambda (fib-2 t)
 (t (lambda (t)
 (k (+ fib-1 fib-2) t))))
 t))))
 t)))))))
```

# Example 6: Ackermann

## Example 6: Direct Style

```
(define ack
 (lambda (a b)
 (cond ((zero? a) (+ 1 b))
 ((zero? b) (ack (- a 1) 1))
 (else (ack (- a 1)
 (ack a (- b 1)))))))
```

## Example 6: Ackermann

### Example 6: CPS

```
(define ack$
 (lambda (a b k)
 (cond ((zero? a) (k (+ 1 b)))
 ((zero? b) (ack$ (- a 1) 1 k))
 (else (ack$ a (- b 1)
 (lambda (roa)
 (ack$ (- a 1) roa k)))))))
```

## Example 6: Ackermann

### Example 6: Instrumented for multi-threading

```
(define tack
 (lambda (a b k t)
 (t (lambda (t)
 (cond ((zero? a)
 (t (lambda (t)
 (k (+ 1 b) t))))
 ((zero? b)
 (t (lambda (t)
 (tack (- a 1) 1 k t))))
 (else
 (t (lambda (t)
 (tack a (- b 1)
 (lambda (roa t)
 (t (lambda (t)
 (tack (- a 1) roa k t))))
 t))))))))))
```

# Example 7: The length of a list

## Example 7: Direct Style

```
(define len
 (lambda (s)
 (if (null? s)
 0
 (+ 1 (len (cdr s))))))
```

# Example 7: The length of a list

## Example 7: CPS

```
(define len$
 (lambda (s k)
 (if (null? s)
 (k 0)
 (len$ (cdr s)
 (lambda (rol)
 (k (+ 1 rol)))))))
```

## Example 7: The length of a list

### Example 7: Instrumented for multi-threading

```
(define tlen
 (lambda (s k t)
 (t (lambda (t)
 (if (null? s)
 (t (lambda (t)
 (k 0 t)))
 (t (lambda (t)
 (tlen (cdr s)
 (lambda (rol t)
 (t (lambda (t)
 (k (+ 1 rol) t))))
 t))))))))
```

# Example 8: Mapping a function over a list

## Example 8: Direct Style

```
(define map
 (lambda (f s)
 (if (null? s)
 '()
 (cons (f (car s))
 (map f (cdr s))))))
```

## Example 8: Mapping a function over a list

Example 8: CPS, version A: f is converted to CPS too

```
(define map$
 (lambda (f$ s k)
 (if (null? s)
 (k '())
 (f$ (car s)
 (lambda (rof)
 (map$ f$ (cdr s)
 (lambda (rom)
 (k (cons rof rom))))))))
```

## Example 8: Mapping a function over a list

Example 8: CPS, version B: f is **not** converted to CPS

```
(define map$
 (lambda (f s k)
 (if (null? s)
 (k '())
 (map$ f (cdr s)
 (lambda (rom)
 (k (cons (f (car s))
 rom)))))))
```

# Example 8: Mapping a function over a list

## Example 8: Version A instrumented for multi-threading

```
(define tmap
 (lambda (f$ s k t)
 (t (lambda (t)
 (if (null? s)
 (t (lambda (t)
 (k '() t)))
 (t (lambda (t)
 (f$ (car s)
 (lambda (rof t)
 (t (lambda (t)
 (tmap f$ (cdr s)
 (lambda (rom t)
 (t (lambda (t)
 (k (cons rof rom) t))))
 t)))))))
 t)))))))
```

# Example 8: Mapping a function over a list

## Example 8: Version B instrumented for multi-threading

```
(define t$map
 (lambda (f s k t)
 (if (null? s)
 (t (lambda (t)
 (k '() t)))
 (t (lambda (t)
 (t$map f (cdr s)
 (lambda (rom t)
 (t (lambda (t)
 (k (cons (f (car s))
 rom)
 t))))))))
 t)))))
```

## Example 8: Mapping a function over a list

### Why two versions of `map`?

- ▶ If we wish to map procedures in CPS over a list, we need Version A, the one that assumes the function being mapped is itself in CPS
  - ▶ Good for general programming in CPS
- ▶ If we wish to create a **safe** version of `map`, one that generates an error message if the list is contains a cycle, but otherwise we wish to use ordinary procedures in direct style, we need to use Version B

# Chapter 10

## Goals

- ✓ Asynchronous Computing
- ✓ Coroutines, Threads & Processes
- ✓ Context-switching & tail-position
- 👉 The two-thread architecture
  - ✓ Instrumenting code for the two-thread architecture
  - 👉 Racing & termination
    - ▶ Detecting circularity
    - ▶ Prioritization
    - ▶ Threads & Types in Ocaml

# Racing & termination

- ▶ We know that our naïve, recursive definition of the Fibonacci function has exponential complexity
- ▶ We also know that the computational complexity of Ackermann's function is far far larger than an exponential function
- ▶ We wish to write a function that takes 3 natural numbers  $a$ ,  $b$ ,  $c$ , and returns either  $\text{Ackermann}(a, b)$  or  $\text{Fibonacci}(c)$ , depending on **whichever finishes computing first**
- ▶ One obvious stipulation is that our function should fail only when **both** the Ackermann and Fibonacci functions fail on their respective inputs, given our resources (RAM, CPU time, etc)
- ▶ This problem is a race to the finish. We are not concerned by off-by-one results, and we can arbitrarily choose to let Ackermann's function run first. But the computation must be **interlaced**

# Racing & termination (*continued*)

```
(define ack-vs-fib-race
 (lambda (a b c)
 (let ((thread-ack
 (lambda (t)
 (tack a b
 (lambda (x t) `((ack ,a ,b) ==> ,x))
 t)))
 (thread-fib
 (lambda (t)
 (tfib c
 (lambda (x t) `((fib ,c) ==> ,x))
 t))))
 (thread-ack thread-fib))))
```

# Racing & termination (*continued*)

```
> (ack-vs-fib-race 2 2 10)
((ack 2 2) ==> 7)
> (ack-vs-fib-race 2 2 5)
((fib 5) ==> 5)
> (ack-vs-fib-race 3 3 20)
((ack 3 3) ==> 61)
> (ack-vs-fib-race 200 200 20)
((fib 20) ==> 6765)
```

# Racing & termination (*continued*)

How might we schedule 3 threads?

```
(define sched-3
 (lambda (t-1 t-2 t-3)
 (t-1 (lambda (t-a)
 (t-2 (lambda (t-b)
 (t-3 (lambda (t-c)
 (sched-3 t-a t-b t-c))))))))))
```



- Notice that the scheduler is **recursive**: Why is it?
- Clearly, this approach to scheduling extends to any number of tasks

# Racing & termination (*continued*)

## How might we schedule 3 threads? (*cont*)

We can now schedule a race between three threads:

```
(define ack-vs-fib-vs-fact-race
 (lambda (a b c d)
 (let ((thread-ack
 (lambda (t)
 (tack a b
 (lambda (x t) `((ack ,a ,b) ==> ,x))
 t)))
 (thread-fib
 (lambda (t)
 (tfib c
 (lambda (x t) `((fib ,c) ==> ,x))
 t)))
 (thread-fact
 (lambda (t)
 (t$fact$ d
 (lambda (x t) `((fact ,d) ==> ,x))
 t)))))

 (sched-3 thread-ack thread-fib thread-fact))))
```

# Racing & termination (*continued*)

## How might we schedule 3 threads? (*cont*)

```
> (ack-vs-fib-vs-fact-race 2 2 10 10)
((fact 10) ==> 3628800)
> (ack-vs-fib-vs-fact-race 2 2 10 20)
((fact 20) ==> 2432902008176640000)
> (ack-vs-fib-vs-fact-race 2 2 10 40)
((ack 2 2) ==> 7)
> (ack-vs-fib-vs-fact-race 2 20 10 40)
((fact 40)
 ==>
 815915283247897734345611269596115894272000000000)
> (ack-vs-fib-vs-fact-race 2 20 10 1000)
((fib 10) ==> 55)
> (ack-vs-fib-vs-fact-race 2 20 20 1000)
((ack 2 20) ==> 43)
```

# Racing & termination (*continued*)

## Discussion

Let us look at the initial continuations for Ackermann & Fibonacci:

- ▶ For Ackermann: `(lambda (x t) `((ack ,a ,b) ==> ,x))`
- ▶ For Fibonacci: `(lambda (x t) `((fib ,c) ==> ,x))`
- ▶ Notice how each initial continuation ignores the thread it receives:
  - ▶ The thread received by the initial continuation for Ackermann has to do with running Fibonacci
  - ▶ The thread received by the initial continuation for Fibonacci has to do with running Ackermann
- 👉 Computation is terminated by not passing control onto the other thread

# Chapter 10

## Goals

- ✓ Asynchronous Computing
- ✓ Coroutines, Threads & Processes
- ✓ Context-switching & tail-position
- 👉 The two-thread architecture
  - ✓ Instrumenting code for the two-thread architecture
  - ✓ Racing & termination
  - 👉 Detecting circularity
  - ▶ Prioritization
  - ▶ Threads & Types in Ocaml

# Detecting circularity

Consider the naïve implementation of the procedure length:

```
(define length
 (if (null? s)
 0
 (+ 1 (length (cdr s)))))
```

- ▶ For circular structures, e.g., #0=(moshe . #0#), length enters an infinite loop
- ▶ In Chez Scheme, length detects circularity in such situations, and generates an error message rather than looping indefinitely:  
> (length '#0=(moshe . #0#))

Exception in length: (moshe moshe moshe moshe moshe moshe m  
Type (debug) to enter the debugger.

- ▶ This means that the implementation of length is not the naïve one

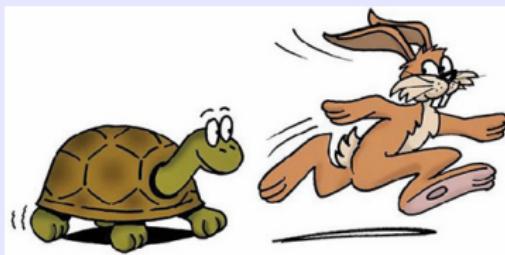
## Detecting circularity (*continued*)

- ▶ In principle, we can have all procedures in Scheme that compute some function over finite, proper lists, first detect circularity, and only if the argument is not circular, proceed to compute the given function
  - 💡 Using our approach to interleaving computation, we can can interleave the test for circularity together with the computation itself

# Detecting circularity (*continued*)

## The tortoise & the hare

- ▶ The classical algorithm for detecting circularity is known as **the tortoise & the hare**. The idea is to use 2 pointers:
  - ▶ A **slow pointer** (*tortoise*) that proceeds **one** cdr down the list at each iteration
  - ▶ A **fast pointer** (*hare*) that proceeds **two** cdr-s down the list at each iteration
- ▶ If the pointers are ever equal **again** then the list is circular



# Detecting circularity (*continued*)

Here is the circularity-detection algorithm in direct style:

```
(define circular?
 (letrec ((run
 (lambda (tortoise hare)
 (or (eq? tortoise hare)
 (and (pair? hare)
 (pair? (cdr hare))
 (run (cdr tortoise)
 (cddr hare)))))))
 (lambda (s)
 (and (pair? s)
 (run s (cdr s)))))))
```

## Detecting circularity (*continued*)

And now we test it:

```
> (circular? '(a b c))
#f
> (circular? '(a b c . #0=(d e f . #0#)))
#t
```

## Detecting circularity (*continued*)

In fact, for our purposes here, we are not interested in **knowing** whether a list is circular, but in **issuing an error message and terminating** if it is. We can rewrite our code to suit this task better:

```
(define die-if-circular
 (letrec ((run
 (lambda (tortoise hare)
 (cond ((eq? tortoise hare)
 (error 'die-if-circular
 "Circularity detected!"))
 ((and (pair? hare) (pair? (cdr hare)))
 (run (cdr tortoise)
 (cddr hare))))
 (else (void))))))
 (lambda (s)
 (if (pair? s)
 (run s (cdr s)))))))
```

## Detecting circularity (*continued*)

- ▶ We are not interested in the **return value**
- ▶ We could have skipped the `else-clause` in `run` altogether, but that is considered “bad style”, so we prefer to be explicit about our intentions
- ▶ The behavior of `die-if-circular` is a bit different than of `circular?`:

```
> (die-if-circular '(a b c))
> (die-if-circular '(a b c . #0=(d e f . #0#)))
```

Exception in `die-if-circular`: Circularity detected!  
Type `(debug)` to enter the debugger.

## Detecting circularity (*continued*)

A curious issue comes up when we try to convert die-if-circular to a threaded version: What should die-if-circular do when given a non-circular list?

- ▶ Currently, we return the `void` object
- ▶ In a multi-threaded context, we would just want to continue with the other thread
- ▶ Our architecture is committed to running two threads
- ▶ Welcome to the `do-nothing` thread:

```
(define t$do-nothing
 (lambda (t)
 (t t$do-nothing)))
```

- ▶ Running `t$do-nothing` concurrently with another thread, just runs that other thread (via `double-dispatch`)
- ⚠️ Running two `t$do-nothing` threads concurrently is an infinite-loop

# Detecting circularity (*continued*)

-  Converting die-if-circular to our 2-thread architecture does not require the use of CPS, since the recursive call to run is in tail-position
-  This means that if you did convert die-if-circular to CPS you would only be passing the continuation; Not creating new ones
-  This approach departs from our general approach of first converting to CPS and only then continuing to the 2-threaded architecture
  - ▶ If this change makes you uncomfortable or insecure, then just convert to CPS as usual

# Detecting circularity (*continued*)

Below is the code for the t\$die-if-circular code:

```
(define t$die-if-circular
 (letrec ((run
 (lambda (tortoise hare t)
 (cond ((eq? tortoise hare)
 (t (lambda (t)
 (error 'die-if-circular
 "Circularity detected!"))))
 ((and (pair? hare) (pair? (cdr hare)))
 (t (lambda (t)
 (run (cdr tortoise)
 (cddr hare)
 t))))
 (else (t t$do-nothing))))))
 (lambda (s t)
 (if (pair? s)
 (t (lambda (t)
 (run s (cdr s) t)))
 (t t$do-nothing))))))
```

## Detecting circularity (*continued*)

We can now combine the `t$len$` thread and the `t$die-if-circular` thread to get a definition of `length` that interleaves the computation of the length with the detection of circularity:

```
(define safe-length
 (lambda (s)
 (let ((thread-length
 (lambda (t)
 (tlen s (lambda (len t) len) t)))
 (thread-circularity
 (lambda (t)
 (t$die-if-circular s t))))
 (thread-length thread-circularity))))
```

## Detecting circularity (*continued*)

```
> (safe-length '())
0
> (safe-length '(a b c))
3
> (safe-length '(a b c . #0=(d e f . #0#)))
```

Exception in die-if-circular: Circularity detected!  
Type (debug) to enter the debugger.

## Detecting circularity (*continued*)

- 👉 We could still use `t$len$` to write implement the naïve version of `length` by interleaving it with `t$do-nothing`
- ▶ It is straightforward to combine `t$die-if-circular` with other threads, to create “safe” versions of all the list-processing procedures in Scheme: `reverse`, `append`, `map`, `fold-left`, `fold-right`, etc.

## Detecting circularity (*continued*)

The procedure `safe-map` combines Version B of `map`, namely `$t$map` with `die-if-circular`, to test for circularity as we map along:

```
> (safe-map list '(a b c))
((a) (b) (c))
> (safe-map list '(a b c . #0=(d e . #0#)))
```

Exception in `die-if-circular`: Circularity detected!  
Type `(debug)` to enter the debugger.

# Chapter 10

## Goals

- ✓ Asynchronous Computing
- ✓ Coroutines, Threads & Processes
- ✓ Context-switching & tail-position
- 👉 The two-thread architecture
  - ✓ Instrumenting code for the two-thread architecture
  - ✓ Racing & termination
  - ✓ Detecting circularity
  - 👉 Prioritization
  - ▶ Threads & Types in Ocaml

# Prioritization

- ▶ If we think of access to the CPU as a **resource**, then prioritization is a way of allocating this resource among the various tasks that are running on a computer
- ▶ Important or urgent tasks are assigned higher priority, and this results in a greater share of the CPU-time relative to other tasks that are currently running
- ▶ Ideally, the priority of a task is something users should be able to control
  - ▶ with high resolution
  - ▶ during run-time
- ▶ Our 2-thread architecture is not as flexible
  - ▶ We can **lower** the priority of a task by adding more indirections before it is performed

# Prioritization (*continued*)

## Example: Immediate

```
(lambda (t)
 (foo ... t))
```

## Example: Lower

```
(lambda (t)
 (t (lambda (t)
 (foo ... t))))
```

# Prioritization (*continued*)

Example: Lower yet

```
(lambda (t)
 (t (lambda (t)
 (t (lambda (t)
 (foo ... t))))))
```

# Prioritization (*continued*)

## Possible use for prioritization

- ▶ The double-dispatch involved in lowering the priority of a task is generally cheap relative to the computation it performs
- ▶ De-facto, most lists in Scheme are **non-circular**
  - ▶ It might be more performant to detect circularity with an extremely low-priority thread

# Prioritization (*continued*)

```
(define t$low-priority-die-if-circular
 (letrec ((run
 (lambda (tortoise hare t)
 (t (lambda (t)
 (t (lambda (t)
 (t (lambda (t)
 (t (lambda (t)
 (cond ((eq? tortoise hare)
 (error 'die-if-circular
 "Circularity detected!"))
 ((and (pair? hare)
 (pair? (cdr hare)))
 (run (cdr tortoise)
 (cddr hare)
 t)))
 (else (t
 t$do-nothing)))))))))))))))
```

```
(lambda (s t)
 (if (pair? s)
 (t (lambda (t)
 (run s (cdr s) t))))
 (t t$do-nothing)))))
```

# Chapter 10

## Goals

- ✓ Asynchronous Computing
- ✓ Coroutines, Threads & Processes
- ✓ Context-switching & tail-position
- 👉 The two-thread architecture
  - ✓ Instrumenting code for the two-thread architecture
  - ✓ Racing & termination
  - ✓ Detecting circularity
  - ✓ Prioritization
- 👉 Threads & Types in Ocaml

# Threads & Types in Ocaml

- ▶ It is straightforward to encode `t$fact$`, `t$fib$`, and other threaded procedures in ocaml
- ⚠ Problems begin when we try to schedule two concurrent threads
  - ▶ Threads are applied **to each other**
  - ▶ This means that the type of a thread must be a subtype of of the **self-applicator** or `(lambda (x) (x x))`
  - ▶ The **self-applicator** cannot be typed directly in ocaml:
  - ▶ `(lambda (x) (x x) : τ → σ`, therefore
  - ▶  $x : \tau$ , and
  - ▶  $(x\ x) : \sigma$ , from which we get that  $x$  has type
  - ▶  $x : \tau \rightarrow \sigma$ ,
  - ▶ Attempting to unify  $x : \tau$  and  $x : \tau \rightarrow \sigma$  gives us an infinite type for  $x$ :  $((\dots) \rightarrow \sigma) \rightarrow \sigma$
  - ▶ We can encode such a type using either **recursive types** or **fixed-point operators over types**

# Threads & Types in Ocaml (*continued*)

- ▶ The Hindley-Milner type system used in ocaml does not support recursive types or fixed-point operators over types
- ▶ Ergo,  $(\text{fun } x \rightarrow x\ x)$  does not have a type in ocaml:

```
fun x -> x x;;
Characters 11-12:
 fun x -> x x;;
```

Error: This expression has type 'a  $\rightarrow$  'b  
but an expression was expected of type 'a  
The type variable 'a occurs inside 'a  $\rightarrow$  'b

- 👉 So before we can write threaded code in ocaml, we must find a way to overcome the limitation of the Hindley-Milner type system...

# Threads & Types in Ocaml (*continued*)

There are three ways to solve the problem of recursive types in ocaml:

## Explicitly enabling recursive types

The first way to enable recursive types within ocaml is to re-start ocaml with the `-rectypes` option:

```
gmayer@curry> ocaml -rectypes
OCaml version 4.05.0
```

```
fun x -> x x;;
- : ('a -> 'b as 'a) -> 'b = <fun>
```

And you can see that ocaml manages to type the applicator.

# Threads & Types in Ocaml (*continued*)

The second way to enable recursive types within ocaml is to use the directive `#rectypes;;` from within an existing ocaml session:

## Explicitly enabling recursive types (*cont*)

```
#rectypes;;
fun x -> x x;;
- : ('a -> 'b as 'a) -> 'b = <fun>
```

# Threads & Types in Ocaml (*continued*)

## Using **equirecursive** types

If we define a new type that is parameterized by type  $\alpha$  and contains a function type from the parameterized type to  $\alpha$ , ocaml will accept this:

```
type 'a circular = Circ of ('a circular -> 'a);;
```

- If we were to name '`'a circular`' as '`'b`', we would get the type equation '`'b = Circ of ('b -> 'a)`', which is a special kind of recursive function type defined using the type constructor `Circ`. The recursive type '`'b = 'b -> 'a`' and the **projection** or **unfolding** of the type '`'b = Circ of ('b -> 'a)`' forms an **isomorphism**, and the types are said to be **isorecursive**. If our type system considers these two types **equal**, then the types are said to be **equirecursive**

# Threads & Types in Ocaml (*continued*)

## Using **e**quirecursive types (*cont*)

- ▶ Even though the ocaml type system does not permit general recursive types, any untyped  $\lambda$ -expression can be modeled in ocaml via **i**sorecursive types
- ☞ The expression `let omega = fun (Circ x) -> x (Circ x)` **in** `omega (Circ omega);;` goes into an infinite loop, just as would, were we to enable recursive function types, the expression `let omega = fun x -> x x` **in** `omega omega;;`
- 💡 This is the same as if we were to type in Scheme:  
`(let ((omega (lambda (x) (x x)))) (omega omega))`

# Threads & Types in Ocaml (*continued*)

## Encoding threads using **equirecursive** types

Armed with this little-bit of facility with recursive types, we have a third way to write threaded code: Using an equirecursive type:

```
type 'a thread = Thr of ('a thread -> 'a);;
```

We can now define t\_fib and t\_ack

# Threads & Types in Ocaml (*continued*)

```
let rec t_fib n k (Thr t) =
 t (Thr (fun (Thr t) ->
 if n < 2 then
 t (Thr (fun (Thr t) ->
 k n (Thr t)))
 else
 t (Thr (fun (Thr t) ->
 t_fib (n - 1)
 (fun r1 (Thr t) ->
 t (Thr (fun (Thr t) ->
 t_fib (n - 2)
 (fun r2 (Thr t) ->
 t (Thr (fun (Thr t) ->
 k (r1 + r2) (Thr t))
)))
 (Thr t))))))
 (Thr t))));;
```

# Threads & Types in Ocaml (*continued*)

```
let rec t_ack a b k (Thr t) =
 t (Thr (fun (Thr t) ->
 if a = 0 then
 t (Thr (fun (Thr t) ->
 k (b + 1) (Thr t)))
 else
 t (Thr (fun (Thr t) ->
 if b = 0 then
 t (Thr (fun (Thr t) ->
 t_ack (a - 1) 1 k (Thr t)))
 else
 t (Thr (fun (Thr t) ->
 t_ack a (b - 1)
 (fun r (Thr t) ->
 t (Thr (fun (Thr t) ->
 t_ack (a - 1) r k (Thr t)
))))
 (Thr t))))));;
```

# Threads & Types in Ocaml (*continued*)

```
let ack_vs_fib a b c =
 let thread_ack = Thr(fun (Thr t) ->
 t_ack a b
 (fun x (Thr t) ->
 Printf.sprintf "ack %d %d = %d" a b x)
 (Thr t)) in
 let thread_fib = Thr(fun (Thr t) ->
 t_fib c
 (fun x (Thr t) ->
 Printf.sprintf "fib %d = %d" b x)
 (Thr t)) in
 (fun (Thr t) -> t thread_fib)(thread_ack);;
```

# Threads & Types in Ocaml (*continued*)

We can now run our code:

```
ack_vs_fib 2 2 10;;
- : string = "ack 2 2 = 7"
ack_vs_fib 20 20 10;;
- : string = "fib 10 = 55"
```

# Threads & Types in Ocaml (*continued*)

## Interleaving Computation — Conclusion

- ▶ CPS can be used to **slice** computation into small, sequential chunks: Each chunk is a single application in tail-position, or a test of an if-expression, etc
- ▶ The code can be instrumented for task switching, by taking a thread as an additional argument, and interlacing threads before each chunk is evaluated
- ▶ Uninstrumented code forms a **critical section**: An area of code that is evaluated without task-switching
- ▶ Double-dispatching or gratuitous task-switching is used to hard-code a lower priority for a task

# Further reading

-  *Compiling with Continuations*, by Andrew W Appel
-  *Essentials of Programming Languages*, by Daniel P Friedman & Mitchell Wand, Chapters 5–7
-  Robert W Floyd's **Tortoise and Hare** algorithm for detecting circularity
-  The Wikipedia entry on recursive types: Check out isorecursive/equirecursive types
-  Class notes on types in ocaml: Check out the isorecursive/equirecursive types

# Chapter 10

## Goals

- ✓ Asynchronous Computing
- ✓ Coroutines, Threads & Processes
- ✓ Context-switching & tail-position
- ✓ The two-thread architecture
  - ✓ Instrumenting code for the two-thread architecture
  - ✓ Racing & termination
  - ✓ Detecting circularity
  - ✓ Prioritization
  - ✓ Threads & Types in Ocaml

# Chapter 11

# Further reading