Chapter 1: Introduction

Why so many languages?

- 4 factors:
  - **Evolution**: learned how to do things better → e.g. structured programming (such as if-statements) over gotos
  - **Purpose-built**: Javascript for web programs, Rust for systems programming
  - For **program specific hardware**: CUDA for GPUs
  - **Socio-economic factors**: learn Swift to program iPhone apps, Java to program android apps
    - Proprietary interests & network effects → works in apple ecosystem, not in other ecosystems

SMoL

- **Minimal (+ but all core features)** programming language → designed for learning
- **Has all the core functions of most programming languages**
  - Eager evaluation of expressions
  - Variables and assignment
  - Lexical scoping rules
  - First-class functions and closures
  - First class mutable structures (arrays, vectors)
  - Automated memory management (e.g. garbage collection)
- What is **NOT** present in SMoL
  - Not always present in every programming language: **static types**
  - Much variation between programming languages: **objects**

Example programs

- **Interpreter**: consumes a program in a language and produces the result of that program. Interpreter::Program$_L$ → Value
- **Type-checker**: consume program → determines true/false based on whether type annotations are correct or not
- **Pretty-printer**: consumes program and prints a pretty version of that program
- **Transformer**: consumes programs in a language and produce related but different programs **in the same language**
  - Different program → text is different
  - Example: macros, for/while loop?
- **Compiler:** consumes programs in one language and produce related programs in a different language compiler:: Program$_L$ → Program$_T$

"The interpreter of a programming language is just another program"

PLAI and PLAIT

- plai: large overlaps with Racket but has two extra features: allows pattern matching, allows the definition of unit tests
- plait: typed version of plai
- Difference between `let` and `define`
  - `let ([]) (body)` → bindings only exist within the body
  - `define (name) (expr)` → does not have to be evaluated within a body

Chapter 2: SMoL & Q&A

- (verify?) A is a **metalanguage** for B: language B is implemented in another language A
- (verify?) language B is implemented in another another language A: there exists an interpreter written in language A for the language B

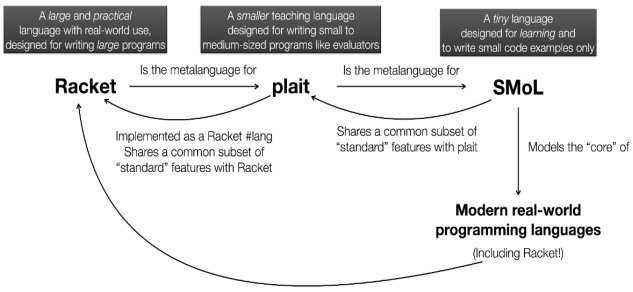


Figure 1: 319x146

Modern programming languages core features

- **Lexical scoping**: search for a variable binding in the current scope first, if it is not present, search for the binding

```
(def (f y 3)
  (def (g x a)
    (+ x y)))
```

  in the outer scopes recursively

- Eager evaluation of expressions: compute results before you need them
- Updatable variables (= mutations/assignments)
- Functions can be
  - higher-order (lambdas): 1) passed as arguments, 2) bound to variables 3) serve as return values
  - close over lexical bindings (closures)
- Scope can be nested
- Automatic memory management (e.g. garbage collection)
- Mutable first-class structures: vectors/arrays and objects
  - **First-class**

Function definitions vs Function calls

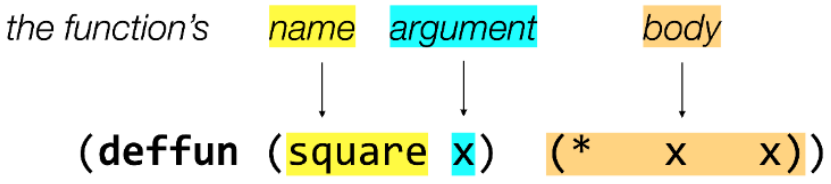


Figure 2: Function definition|424x91

- Argument to
  - Function definition: **formal parameter**
  - Function application/calls: **actual parameter**

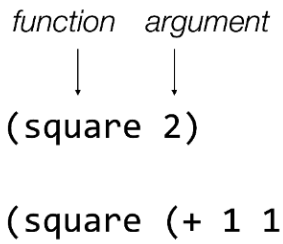


Figure 3: Function application|186x128

S-Expressions

- Able to express: expressions or pure data

```
s-exp ::= atom
        | (s-exp s-exp* )

atom  ::= number
        | symbol
```

Figure 4: atom = pure data

Stacker: key concepts

- **Stack**: list of function calls waiting for a value → most recent function call: at the bottom
- **Environment**: list/set of bindings: variable name → value
  - environments extend other environments: variable binding not found locally → search in extended environment
  - acts like a **cache** for substituted values
- **Current task**: expression being evaluated + environment its being evaluated in
- **Heap**: set of heap-allocated values (vectors or functions)
- **Address**: unique references to environments and heap values
- **Block**: *sequence* of definitions and expressions
  - Blocks form a *tree-like* structure: blocks may contain other blocks
  - Example blocks: top-level block, function body
  - Definitions and expressions are evaluated in *reading order* (top-bottom, left-right)
- **Primordial block**: *invisible* block that *contains* the top-level block
  - Defines the variables and functions provided by the language itself
- **Scope** of a variable: the region of code where a variable can be referred to

  - A variable reference is **in the scope of a declaration** ⇔ the former refers to the latter (the `x` in `(+ x 1)` in in the scope of the declaration of `x`: `defvar x 23`))

- **Shadowing:** When a variable is redefined in a sub-block, this variable in the sub-block **shadows** the variable in the superblock.
  - A binding in the child environment *overrides* a binding in the parent environment

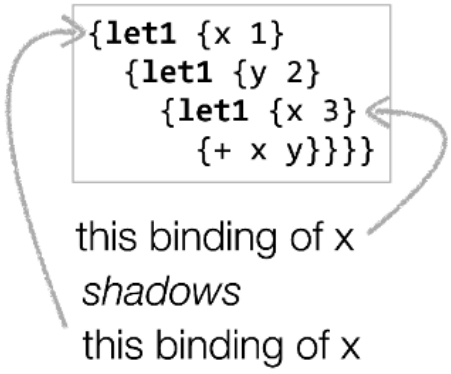
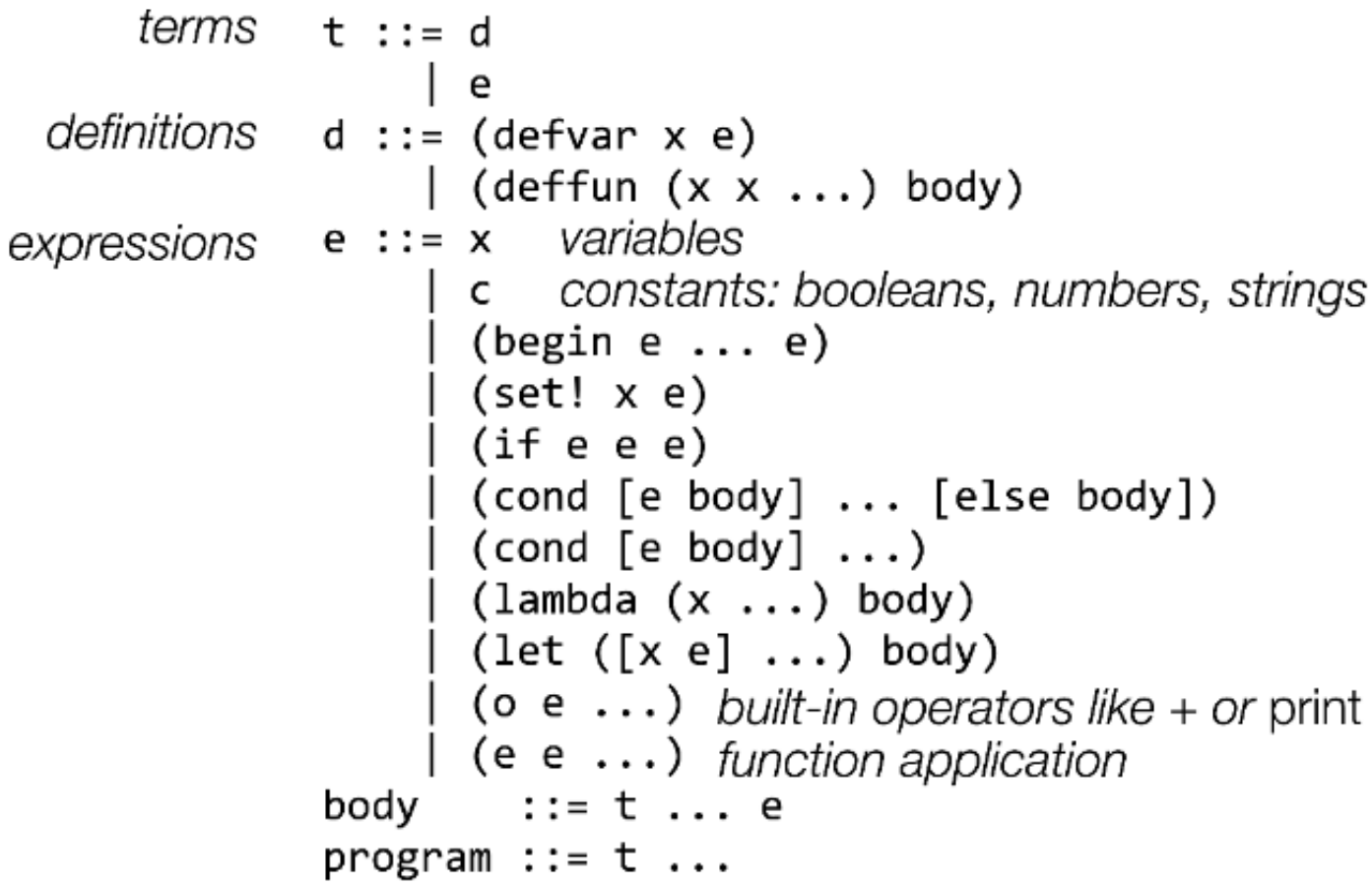


Figure 5: shadowing example|237x193

- **Aliasing**: multiple references sharing the same object
  - Vectors can be **aliased**
  - Vectors are not copied in the following situations (aliases are made):
    * Binding another variable to a vector
    * Passing a vector as an argument to a function call
    * Creating a new vector that refers to existing vectors
- Variable assignments mutate existing bindings, they do not create new bindings
  - `set! x 0`: mutates existing binding
  - `let ([x 0])`: creates a new binding in an extended environment

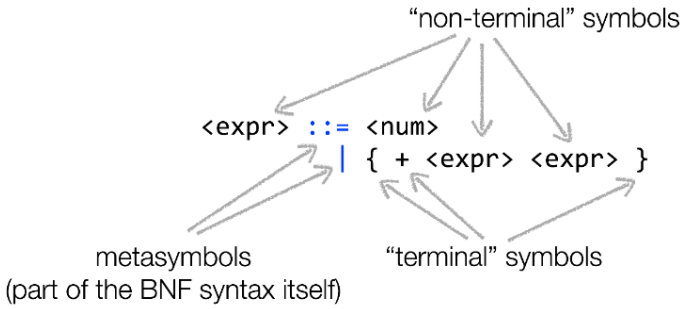
SMoL syntax



Chapter 3: Evaluation

- Two types of **evaluators**:
  - **Interpreters**: easier to write, debug and update (in case we need extra features in the language)
  - **Compilers**: aim to generate efficient code in a target language
- **Meta-programs**: programs that operate on other programs
- There is no such thing as an interpreted language or compiled language → a language does not specify how it should be evaluated
- **Just-in-time compilation**: the evaluator for the programming language is first the interpreter, but switches to compiler for pieces of code that are executed often.
- **Abstract syntax**: representation of a program in the computer
  - Commonly represented as a **tree**: **Abstract Syntax Tree (AST)**
    - Represents programs in programs
    - ASTs enforce unambiguity in the program representation
      - Program evaluation can be done unambiguously
- **Parsing**: turns a linear sequence of tokens into a tree-shaped representation
  - Parsing always produces a consistent unambiguous representation, even if the input is ambiguous
- Functions refers to the latest values of the variables defined outside of their definitions
  - A function **does not remember** the values of those variables at the time the function was defined → ***BUT* it remembers the bindings/environment!**

BNF format



- Initial language
- 3 things to support conditionals:
  1) extend datatype representing expressions
  2) extend the evaluator to handle these new expressions
  3) extend the parser to produce these new representations
- The branches of a conditional statement must have the same type in statically typed languages

```
(define (calc e)
  (type-case Exp e
    [(num n) n]
    [(plus l r) (+ (calc l) (calc r))]
    [(cnd c t e) (if (zero? (calc c))
                     (calc t)
                     (calc e))]))
```

Figure 6: new abstract syntax representation

- We need to distinguish between type-constructors that produce **abstract syntax** and type-constructors that produce **values**

```
(calc : (Exp -> Number))
```

```
(calc : (Exp -> Value))
```

(?) If you don't have variables and functions → you don't have a program

- **Binding**: assign a value to variable
- **Local**: limited to some region of the program, not outside of it
  - `(let (x 1))`: creates a new environment that **extends** the current environment. Binds `x` to `1` in this **new** environment
  - `(set! x 1)` → **updates** the binding in the **current** environment

Functions in the language

- Functions can be
  - Top-level

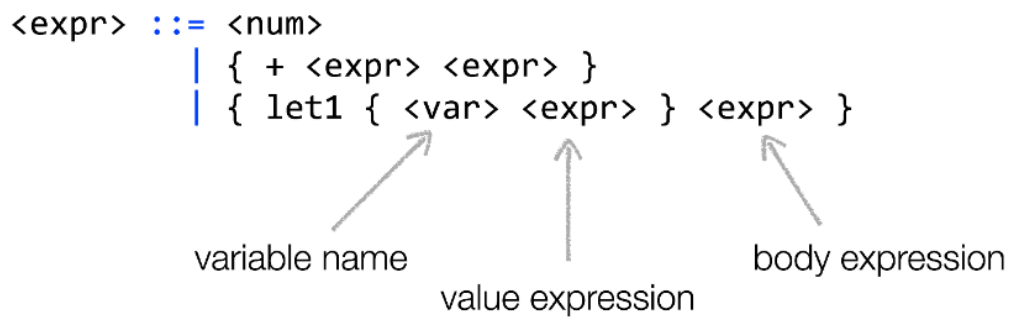


Figure 7: 510x160

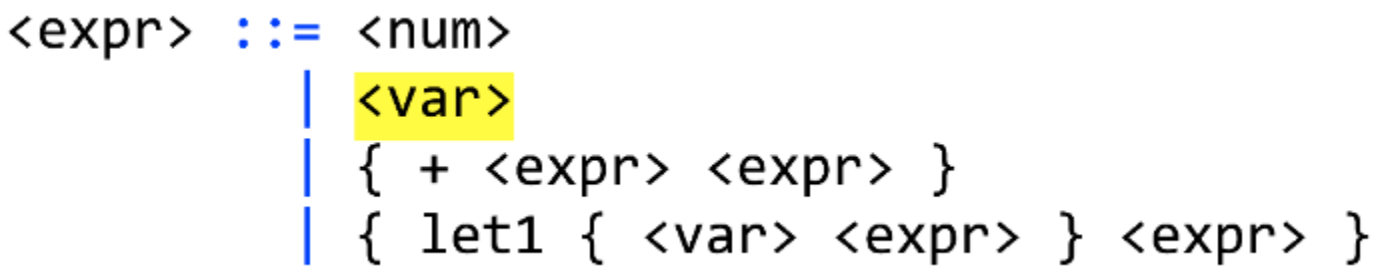


Figure 8: 356x78|<var> to use a bound variable

  - Nested inside other functions
- Functions are **first-class** values (see definition of first class)
  - **First-class**: it can do everything a normal value can (just to emphasize)
    * Passed as arguments
    * Bound to variables
    * Serve as return values
    * Vectors can refer to first-class values
- **Anonymous functions**: functions without a name
  - **Lambda expression**: *expression* that creates an (anonymous) function
- Two new types of expressions: lambda expressions & function call expressions
- Lambda expressions should evaluate to a new function type that can be represented in AST
- Argument evaluation order:
  - 1) get the function value corresponding to function variable `interp fun env`
  - 2) get the values corresponding to the arguments
  - 3) bind the values to the variables → get a new environment
  - 4) use the new environment to evaluate the body

```
(define (interp e env)
  (type-case Exp e
    …
    [(appE fun arg)
     (type-case Value (interp fun env)
       [(funV var body)
        (let ([new-env (extend env var (interp arg env))])
          (interp body new-env))]
       [else …])]))
```

- **Eager evaluation**: evaluate argument expression before passing them as arguments to the function

- **Lazy evaluation**: defer evaluation of argument expressions when referenced in the body of the function

```
(interp : (Exp Env -> Value))
(define (interp e env)
  (type-case Exp e
    [(numE n) (numV n)]
    [(varE s) (lookup env s)]
    [(plusE l r) (add (interp l env) (interp r env))]
    [(let1E var val body)
     (let ([new-env (extend env var (interp val env))])
       (interp body new-env))]
    [(lamE var body) (funV var body)]
    [(appE fun arg)
     (type-case Value (interp fun env)
       [(funV var body)
        (let ([new-env (extend env var (interp arg env))])
          (interp body new-env))]
       [else (error 'appE "expects function value")])]))
```

- **Static scoping**: binding of a variable is determined by the variable's position in the program, not the program's execution
- **Dynamic scoping**: binding of a variable is determined by its most recently executed binding
- binding != value of a variable. Value can change for the same binding.

- Dynamic scoping: evaluation result of `coin-flip` cannot be determined before execution → cannot determine binding of x before execution!

```
(let ([x 1])
  (let ([add (lambda (y) (+ x y))])
    …
    (if (coin-flip)
        (let ([x 2])
          (add 10))
        (add 10))
```

- A person that reads programs can't be sure about the binding structure of our programs
- **Closure**: function value that contains environment the function was defined in
  - *Running* a closure is valid terminology (like *running a function*)
  - From function expression to closure: `[(e-lam param body) (v-fun param body env)]`
- **Closed expression**: expression with no unbound variables.
  - `(+ x 1)` where x is not bound → unclosed expression

Chapter 4: Syntactic Sugar

- **Syntactic sugar**: extra constructs that make your code easier to program and understand
- **Desugaring**: translating from a bigger surface language into a smaller core language

CPL Summary 2026-01-19 18.17.07.excalidraw

Figure 9: CPL Summary 2026-01-19 18.17.07.excalidraw

- **Macro systems**: systems in which a (slightly abstracted) program source is rewritten into program source before parsing takes place
  - ??? Macrossystem? `(+ 1 2)` → `(+ 2 1)`
- Few languages expose macros to the programmer

- Some do with varying degrees flexibility
- In languages with eager evaluation you cannot define new control structures as plain functions
- **Hygiene**: property of keeping the variable bindings in the macro definition separate from the variable bindings at the point of macro expansion
- In languages with eager evaluation, you cannot define new control structures as plain functions
  - Requirement: functions that operate on the syntax of their arguments
  - **Macro**: function that take syntax as input an transform it to produce new syntax
- Local binding (let): syntactic sugar for function application
  # Chapter 5: Objects
- We define objects as syntactic sugar

This pattern:

```
(define (<class-name> <constructor-params>)
  …
  (lambda (m)
    (case m
      [(<field-name>) …]
      [(<method-name>) (lambda (x) …)]
      …)))
```

Is equivalent to this pattern in Java:

```
class <class-name> {
  public <class-name>(<constructor-params>) {
    …
  }
  public int <field-name> = …;
  public int <method-name>(int x) {…}
  …
}
```

Figure 10: class pattern (without extras)|391x93

This pattern:

```
(define <class-name>
  (lambda (<constructor-params>)
    …
    (let ([<var> …])
      (lambda (m)
        (case m
          [(<field-name>) …]
          [(<method-name>) (lambda (x) …)]
          …)))))
```

Is equivalent to this pattern in Java:

```
class <class-name> {
  private int <var> = …;
  public <class-name>(<constructor-params>) {
    …
  }
  public int <field-name> = …;
  public int <method-name>(int x) {…}
  …
}
```

Figure 11: class pattern (with private state)|472x114

Important concepts

- **Objects**: functions that dispatch based on a message name (= **object pattern**)
  - Bundling of data and operations over the data
  - Generalisation of closures
- **Classes**: functions that make objects (**= the object pattern**)
- **Static members**: defined in environments extended by all objects of the same class
  - bindings are **shared** across objects of this class

Figure 12: class pattern (with private state and static members)|0x0

This pattern:

```
(define <class-name>
  (let ([<static-var> …])
    (lambda (<constructor-params>)
      …
      (let ([self 'dummy] [<var> …])
        (begin
          (set! self
            (lambda (m)
              (case m
                 [(<field-name>) …]
                 [(<method-name>)
                   (lambda (x) … self …)]
                 …)))
          self)))))
```

Is equivalent to this pattern in Java:

```
class <class-name> {
  private static <static-var> = …;

  private int <var> = …;
  public <class-name>(<constructor-params>) {
    …
  }
  public int <field-name> = …;
  public int <method-name>(int x) {…this…}
  …
}
```

Figure 13: class pattern (private, static and self-reference)|572x204

- Function can be represented as an object with a single entry point

- Object can be represented as a function with multiple entry points
- **Dynamic dispatch**: the method that should be called is determined **at runtime** based on the structure of the object
- **Members**: fields and methods of a class or object
  - Two interesting questions about classes/objects:
    - Are the **set of member names** statically fixed or changeable dynamically?
    - Is the **member being accessed** statically fixed or can it be computed dynamically?
      - (the member name you use)

	Member name is static	Member name is computed
Fixed set of members	Example: a basic Java class	Example: in Java, can use the Java Reflection API to lookup a class member at runtime using a String value
Variable set of members	Non-sensical (how would one statically address a new member that is added at runtime?)	Most "scripting languages" (notably JavaScript, Python, Ruby, Perl). E.g. JavaScript's `obj[name]` syntax.

Member name design space examples

Inheritance

- **Inheritance**: allows the child object to reuse the members of the parent object without having to modify it
  - The lookup for a message in the child object continues in the parent object

Difference a-child vs make-child

Figure 14: Difference a-child vs make-child

FYI

```
(define (make-parent)
  (lambda (m)
    (case m
      [(add1) (lambda (x) (+ x 1))]
      [(sub1) (lambda (x) (- x 1))])))

(define (make-child)
  (lambda (m)
    (case m
      [(add2) (lambda (x) (+ x 2))]
      [(sub2) (lambda (x) (- x 2))]
      [else ((make-parent) m)])))

(let ([a-child (make-child)])
  ((a-child 'add2) 5)  ; => 7
  ((a-child 'add1) 5)) ; => 6 (inheritance!)
```

Figure 15: creates a new parent instance for every msg lookup in parent|266x200

```
(define (make-parent)
  (lambda (m)
    (case m
      [(add1) (lambda (x) (+ x 1))]
      [(sub1) (lambda (x) (- x 1))])))

(define (make-child)
  (let ([a-parent (make-parent)])
    (lambda (m)
      (case m
        [(add2) (lambda (x) (+ x 2))]
        [(sub2) (lambda (x) (- x 2))]
        [else (a-parent m)]))))

(let ([a-child (make-child)])
  ((a-child 'add2) 5)  ; => 7
  ((a-child 'add1) 5)) ; => 6 (inheritance!)
```

Figure 16: creates only one instance of the parent|275x221

Open recursion

- **Open recursion**: parent object allows the child object to modify the behaviour of existing methods by overriding members on self. ## Other forms of inheritance
- **Class-based inheritance**: each child has its own personal copy of the parent object
  - Specify in advance which class you are going to extend
  - In object pattern: implemented via 'chaining' message lookups
- **Prototypal inheritance**: When two or more child objects inherit from a **common parent object**
  - Changes in common parent object are visible in all child objects
  - prototype-based inheritance
- **Mixin-based inheritance (= mixins)**: classes that can be mixed into any class hierarchy as needed
  - **Mixin**: class that can be mixed into any class hierachy as needed
  - Mixin's superclass becomes a **parameter** of the mixin → mixin can be viewed as function parameterised with parent class
  - Enables modular class extensions
- **Multiple inheritance**: a class can inherit multiple classes
  - Benefits: flexibility in modelling complex hierarchies without code duplication
  - Drawbacks: problems arise when the superclasses define members with the same name → which one to choose?

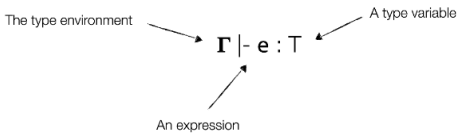
```scala
trait Flyable {
  def fly(): Unit = println("I'm flying!")
}

class Bird

val eagle = new Bird with Flyable   // mixin applied here
eagle.fly()  // prints "I'm flying!"
```

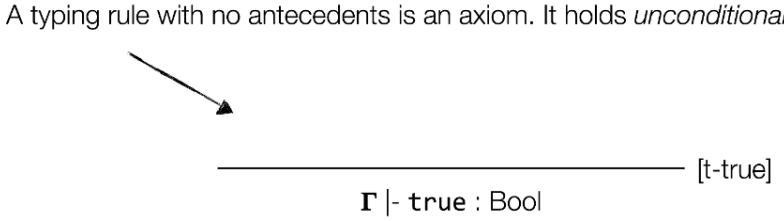
Figure 17: Mixin's can be applied as needed. Flyable is not implemented in advance.|361x116

Chapter 6: Types



The environment Γ proves that the expression e has type T

- **Judgement**: tree that results from applying the typing rules until every part of the tree results in an **axiom**
- **Type-error**: failure to construct a judgement (Γ|-"hi" : Num is an invalid statement → tree cannot be constructed)
- **Axiom**: typing rule with no antedecents



  - Holds unconditionally

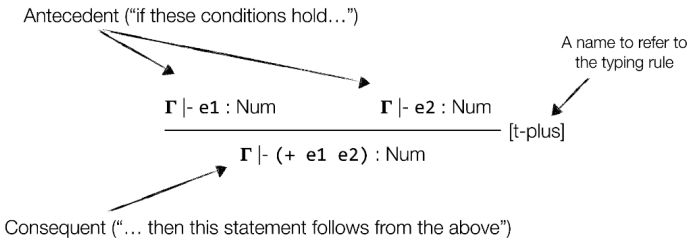


Figure 18: 356x126

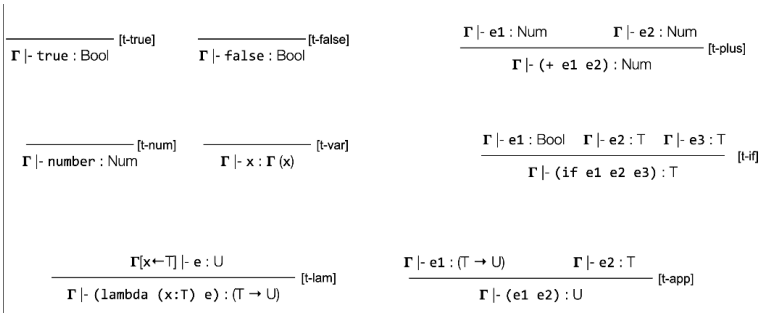


Figure 19: 392x154

- **Antedecent**: the extended environment where x is bound to type T proves that e is of type U
- **Consequent**: the environment proves that the lambda that takes an argument x of type T and has body e is of type $T \to U$.

$$\frac{\Gamma[x \leftarrow T] \vdash e : U}{\Gamma \vdash (\texttt{lambda (x:T) e}) : (T \to U)}$$

  - This is because in the body of the lambda, the environment Γ is extended to $\Gamma[x \leftarrow T]$.

- Type checking: start from the bottom!
- $\Gamma(x)$: type that is bound to x in environment Γ
- Γ: Type environment. Binds variables to types (e.g. $\Gamma(x) = Num$)
- A type system is **sound** $\Leftrightarrow$ if $e : t$, and if $e \to v$ then $v : t$
  - Even if the type-checker is sound: there may exist expressions for which the type-checker predicts a type $e : t$ but will never reduce to a value of that type because the program never terminates.

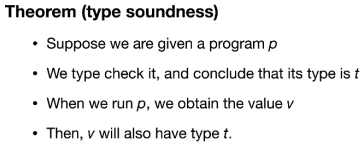


Figure 20: 191x80

Similarities and difference interpretation and type-checking

- Similarities:
  - Both use **environments**
  - They are **both evaluators**: turn programs into answers

Property	Type checker	Interpreter
Result	true/false	computed value
Input	Sees only program text	Executes actual program data
Environment	Binds variables to types	Binds variables to *values*
Termination	Terminates	Might not terminate
Nr of passes in the body	**once**	**zero to infinite**

- Type-safety benefits:
  - **Savings in space** (more compact memory representations of values)
    - No need to keep **type-tag around**
  - **Savings in time** (prevent run-time type checking)
    - no need to runtime check for **substitutability** (= whether you can pass it as argument or not)
  - **Progress**: given a program p, if it passes the type checker then
    - or p is a value
    - or p can take a step
  - **Preservation**: if p takes a step, then the type of p remains the same

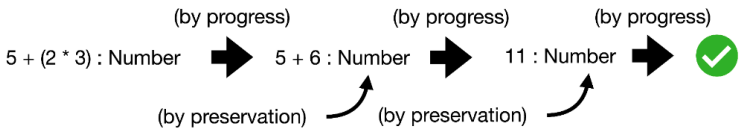


Figure 21: 398x73

- Every soundness theorem is a **contract** between the language and the user: The set of published, permitted exceptions or error conditions that *may* still occur
  - E.g. array out of bounds exception is not within this contract
- Are languages without static typing doomed?
  - No:
    - 1) lack of static typ
    - 2) dynamically typed languages check **at runtime**
    - 3) statically typed languages also do checks at runtime for statically untractable checks:
      - dereferencing a pointer

	SAFE	**UNSAFE**
TYPED	ML, Haskell, Java	C, C++
NOT TYPED	Python, Javascript, Racket	Machine code?

Figure 22: 292x220

- **Typed but unsafe**: the type computed by the type-checker is not the same as the type of the value that is computed (**not sound**)
  - E.g. in C++ you can cast `int*` to `char*` and then dereference: `char a = *charPtr`.
    - If the value was `1000` then char a wouldn't make sense.
- Strongly typed and weakly typed languages are poorly defined concepts
  - Its about the **guarantees of the runtime** (type soundness), not how strong the type checker is.

- You can have a type checker that always predicts `Number`, but that doesn't make any guarantees about the types of the computed values of the program

Lecture 7: Continuations

- **Control operation**: operation that causes an operation to proceed because it controls the program counter of the machine.
  - **Function call**:
  - **Exception handling:**

Local control operation: **Non-local control operation**:

- **Local I/O**: performed by tradition programs (e.g. command-line program)

- **Remote I/O**: performed by web-based programs
- **Evaluation context**: function calls stored on the stack
  - exists **implicitly** on the stack

  - We need to make this evaluation context **explicit**

Figure 23: evaluation context (left) stored in a lambda function to make it explicit|0x0

When a program needs external input → control is passed to the OS.

- Making transfer of control explicit:
  - 1) Explicitly suspend the program at each point in execution where it performs I/O
  - 2) Enable the OS to resume the program with external input from the point at which the program was suspended
    - OS has control: so OS should take action to resume program

SMoL

```
(get-number/k "first"
  (lambda (v1)
    (get-number/k "second"
      (lambda (v2)
        (print (+ v1 v2))))))
```

Figure 24: get-number/k takes a closure representing the rest of the computation (evaluation context)|361x124

- **Continuations**: closures that represent the rest of the computation (= closures that represent evaluation context)
  - denoted with the letter **k**
  - **Continuation-passing-style (CPS)**: style where programs use continuations

Inversion of control

- **Inversion of control**: instead of the innermost operation being done first and the outermost being done last, the innermost operation is done last and the outermost operation is done first.

Non-CPS: `print (+ (get-number) (get-number))`

In **both** cases: `get-number` needs to be called first. In CPS style, `get-number/k` gets the number, and then calls the continuation (evaluation context). This is implicit in non-CPS style.

Call/CC and Let/CC

- `(let/cc k body ...) = (call/cc (lambda (k) body ...)`
  - Merely syntactic sugar → `let/cc` is used in the summary

```
SMoL
(get-number/k "first"
  (lambda (v1)
    (get-number/k "second"
      (lambda (v2)
        (print (+ v1 v2))))))
```

Figure 25: 442x152

- `let/cc` allows you to customise what the continuation is after evaluation of the body → the evaluation context of the body is forgotten!
  - If `k` is called in the body → result of `let/cc k body` is result of `k`
  - If `k` is not called in the body → result of `let/cc k body` is result of `body`
  - Unlike with a function call → calling a continuation **never returns**: the evaluation context in which continuation is called is discarded!

example of how early return is implemented using continuations

Figure 26: example of how early return is implemented using continuations

Advanced control flow

- **Cooperative multi-threading**: each thread must explicitly give up 'control' to the thread scheduler to resume another thread
- **Pre-emptive multi-threading**: any thread can yield implicitly after executing a certain period of time or # of instructions

---

- CPS transformation is a **global transformation**: all functions require an extra parameter.
- CPS transformation is hard to understand due to inversion of control

Chapter 8: javascript

- **Javascript on the web**: scripts embedded in web pages, executed on the client
  - **Mobile code**: code is executed remotely (at the client side)

Figure 27: 412x186

- Javascript on the server: node.js
  - Renders support for **asynchronous I/O** on files and sockets
  - **Asynchronous I/O**: IO operations are non-blocking
  - **Node.js**: web and network application server

    - **No execution in a sandbox**
- JavaScript code is **embedded in a host environment**:
  - If embedded **within browser host**: access to DOM and Local storage
  - If embedded **within server host**: access to network and file IO
- JS is a **scripting language**: embedded and used in a host environment
  - Host environment: a bit more specific than "OS where the program runs" → designed to only exist inside some other program
    - E.g. event loop is offered by nodeJS, not within the scripts themselves
  - Compare with *Flash games*: support was dropped (no host environment) → game scripts could not run anymore.

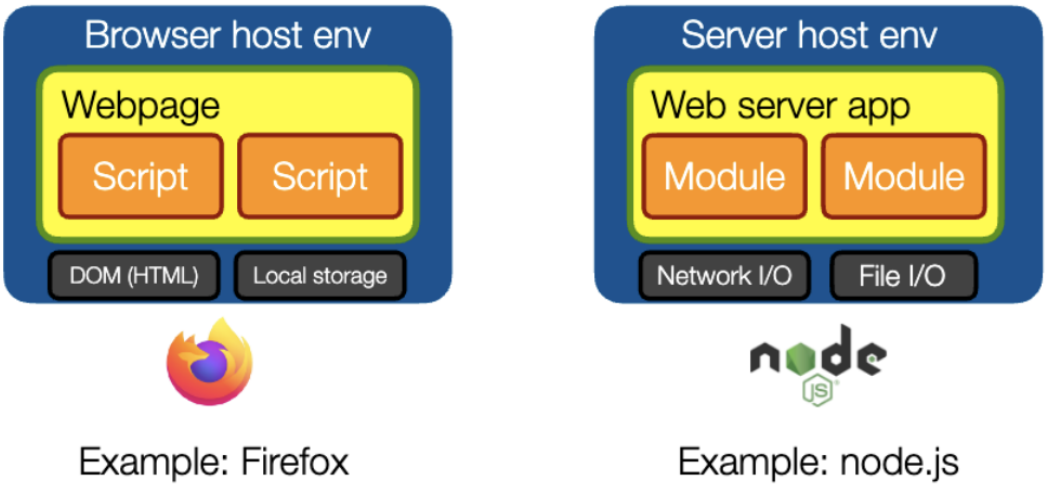


Figure 28: 535x249

Three most important values in JS programs:

- **Object**: mapping from a key to a value
  - Dynamic collection of properties
    - Can add/remove properties **at runtime**
    - Can lookup properties based on **computed key**
    - Can turn all properties of an object into an array and iterate over them
  - **Property**: key-value pair
    - Property values can be functions
  - **Object literal:** expressions that evaluate to a fresh object
    - Can be nested
- **Arrays**: sequences of values (similar to Python or Java lists)
  - `length` property: a **computed property**: returns current number of elements
  - Can access elements from index 0 to index `length - 1`
    - Out of range → return `undefined`
  - You can dynamically grow/shrink to add/remove elements
  - Arrays are objects with many utility functions (`forEach`, `map`, ...)
- **Functions**: functions are values
  - Functions are **First-class** values → like arrays and objects
  - Must use explicit `return` or else returns `undefined`
  - **Higher-order** functions: can take other functions as input or return other functions as output
  - Functions may use variables from their lexically enclosing scope (= closures)

- **Arrow functions**: notational shorthand
  - Always anonymous
  - Function body: expression
    - Without `{ }`: no `return` statement required
    - If enclosed with `{ }`: statement (`return` statement required if you want to require something)
  - **Method**: a function-valued property of an object that can be called with `object.method(arg)`
  - Functions **can be used as class constructors**: returns new object on each function call

example JS code

Figure 29: example JS code

Class syntax

- Class syntax = syntactic sugar: still a function at runtime
  - use the `new` keyword if you want to create an object from the class

comparison class syntax vs plain function syntax

Figure 30: comparison class syntax vs plain function syntax

- Javascript uses **prototypal inheritance**
- Javascript is dynamically typed:
  - **Dynamically typed:** values have a runtime type, but variables or object properties do not have a static type
- **TypeScript**: dialect of JS that allows for (optional) static type annotations
  - TypeScript supports **Type Inference**: we can deduce the type of a variable based on how it's used in the program

"JavaScript is a List in C's clothing". Why?

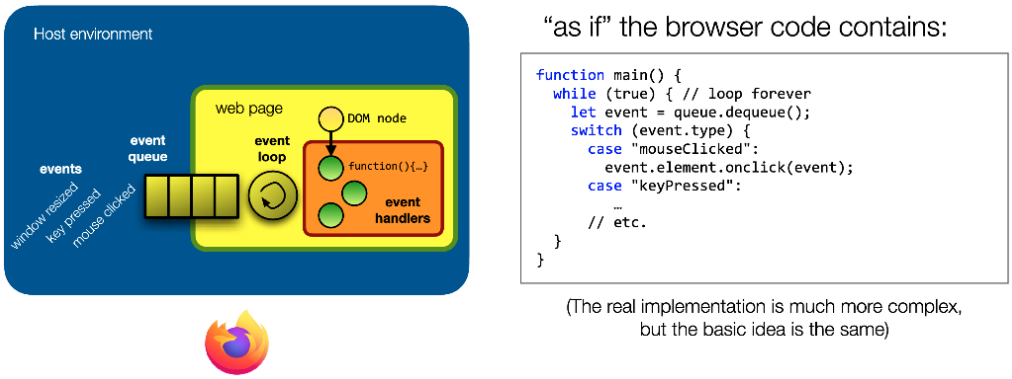


Figure 31: JS code execution on browser + idea of event loop|510x192

- **Event loop**: invisible infinite loop that calls JS code
  - Executed by a **single thread** of control
  - Events are processed **one at a time**
  - When an event occurs: it gets enqueued in **event queue**

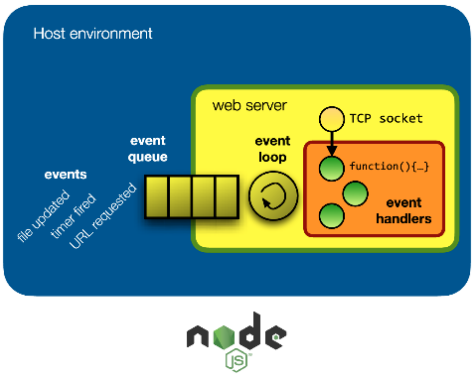


Figure 32: JS code execution for nodejs|243x191

  - If there are are no events to process: event loop is **idle**

  - UI rendering is performed by the same event loop while no events are being processed
  - ***NEVER* block the event loop!**

- To respond to an event: register an **event handler**

- **Callback:** function that acts as the event handler

Callback functions

All code in `<script></script>` elements share the **same** event loop
A node.js process provides a single event loop for its code

Events in javascript make use of inversion of control

Inversion of control: "Don't call us, we'll call you"

```
let button = document.getElementById("button-id")

button.onclick = function(event) {
  window.alert("button was clicked")
}
```

Figure 33: example in JS where the continuation (event handler) is called after input (event) is obtained (occurs)|410x104

IO in event loops

- **XMLHTTPRequest**: a legacy browser API that allows HTTP requests to be made within JS code
- More modern alternatives exist now

- **XHR**: the HTTP requests made by JS Scripts in browser to a server
  - (?) Refers to the actual requests, not the API?

  - Asynchronous: allows for processing other events while the request is pending
    * If XHR was synchronous then the entire event loop would be **blocked** and the web page would be **unresponsive**

callback hell - **Callback hell**: deeply nested code that we obtain because we the rest of the work to be done in callback functions.

- Exceptions don't work for asynchronous operations

- Caller has already returned when the operation is executed

- Common pattern: pass error object as argument to callback function
  - If success: error is `undefined`
  - If failed: error contains `Error` object with details
- The JS code **still runs on a single main thread**, only one task at a time!
  - Longer processing is delegated to OS or browser, callback is used to let JS handle the result again
- **Call stack:** live runtime structure of *active* function calls

- **Stack trace:** *snapshot* of a call stack at a particular moment (e.g. after error)

Event loops and non-blocking IO:

- Benefits
  - **Run-to-completion:** functions are never pre-empted while running
    - New events gets pushed onto the event queue and the event loop pops them one by one
  - **Better resource utilisation**: event loop never blocks on external I/O → user application remains responsive to user events
- Drawbacks
  - **Inversion of control**: more difficult to understand the control flow.
    - Rest of the computation provided as callback → callback hell
  - **No call stack to unwind**:
    - **Harder to debug**: stack traces in event handlers don't reveal context
      - Error happen at a different place than where the callback is fired
      - E.g. `fileIO` error "file not found" in OS, callback (event handler) has no knowledge of context for this error)
    - No exception handling

Promises

- **Promise**: placeholder for a value that may only be available in the future
  - Most asynchronous APIs return a `Promise` instead of taking a `callback` function as extra argument

promises vs callbacks

Figure 34: promises vs callbacks

- Promise is an *object* that can be in one of three states:
  - Pending: initial state
  - Fulfilled (with a value)
  - Rejected (with an error)

Once a promise is either fulfilled or rejected, it remains in that state ### Promise chaining

- A call to `then` returns a chained promise
  - Success and failure callbacks passed to then may return a `value` or throw an `Exception`
    - If returns a value: fulfill the chained promise
    - If exception is thrown: reject the chained promise

Promise chaining solves the problem of callback hell

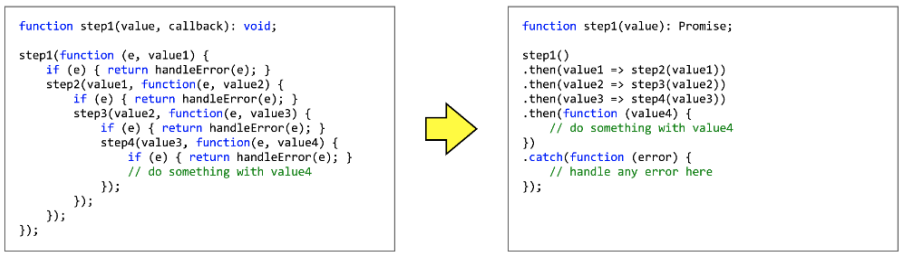


Figure 35: Promise chaining solves callback hell|452x127

Resolving a promise `p1` with another promise `p2` causes `p1` to eventually become fulfilled/rejected with the same value/error as `p2`

```
let promise = readFile("hello.txt");
let p2 = promise.then(function (content) {
  // decode may throw
  return decode(content);
}, function (err) {
  // fall back to another file
  return readFile("default.txt");
});
```

Figure 36: promise.then is another promise that gets resolved/rejected|384x173

CPL Summary 2026-01-21 10.27.18.excalidraw

Figure 37: CPL Summary 2026-01-21 10.27.18.excalidraw

- `Promise.all`:
  - Fufills if: **all** of the promises fulfil
  - Rejects if: **any** of the promises reject
  - Fulfilled value: all of the fulfilled promises' values (`.then(vals => ...)`)
- 'Promise.any:
  - Fulfills if: **any** of the promises fulfill
  - Rejects if: **all** of the promises reject
  - Fulfilled value: value of the first promise to be fulfilled (`.then(val => ...)`)
    - what if all promises reject?
- Concepts related to Promises exist in other languages as well: **futures**, **deferreds**, **tasks**
  - Many differences in terms of API
    - **opaque vs transparent**: is `Promise<T>` a subtype of `T`?
      - E.g. is `Promise<int>` a subtype of `int`? (Transparent?)
    - **read-only vs read-write**: does having access to the promise object mean you can also set its value?
      - E.g. change return value of promise from $3 \to 5$
    - **blocking vs non-blocking**: does accessing the Promise's value suspend the calling thread until the value is available?
- **Terminology** of Promises is *inconsistent* across languages
  - E.g. Promise in scala $\neq$ Promise in JavaScript
- Compared to callbacks, promises make **delayed computation explicit as data**
  - Promise = object = data
  - Benefits:
    - Delayed computation can be **composed** through standard function composition (`.then().then()...`)

    - **Automatic propagation of errors**: Promise objects distinguish success paths from failure paths
        - If `p2` is chained on `p1`: if `p1` rejects, then `p2` also rejects if it does not handle the error

  - Drawbacks:
    - Delayed computation **must still be wrapped in nested functions**
        - `promise.then(() => ...)` → the callback `() => ...` still has to be provided as nested function

    - We cannot use sequential control flow constructs: `while`, `return`, `try-catch`, ...
      ## Async functions Async await example
- `Async`: a **modifier** that is used to mark a function as an async function
  - Async function: function that always returns promises

- `Await expr`: wraps **expression** into a promise where the continuation of `Await expr` is configured to be a delayed computation on `p`.
  - The async function immediately returns p

  - Can only be used inside an async function

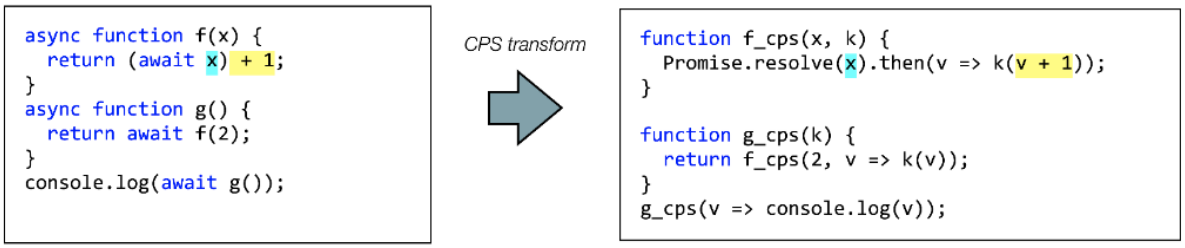


Figure 38: (Simplified) CPS transform from async/await to normal functions|595x124

Chapter 9: TypeScript