Interpreters

July 26, 2022
Laryn Qi

Announcements

Programming Languages

Levels of Languages

High-level language

(Python, Scheme, SQL, Java, C)

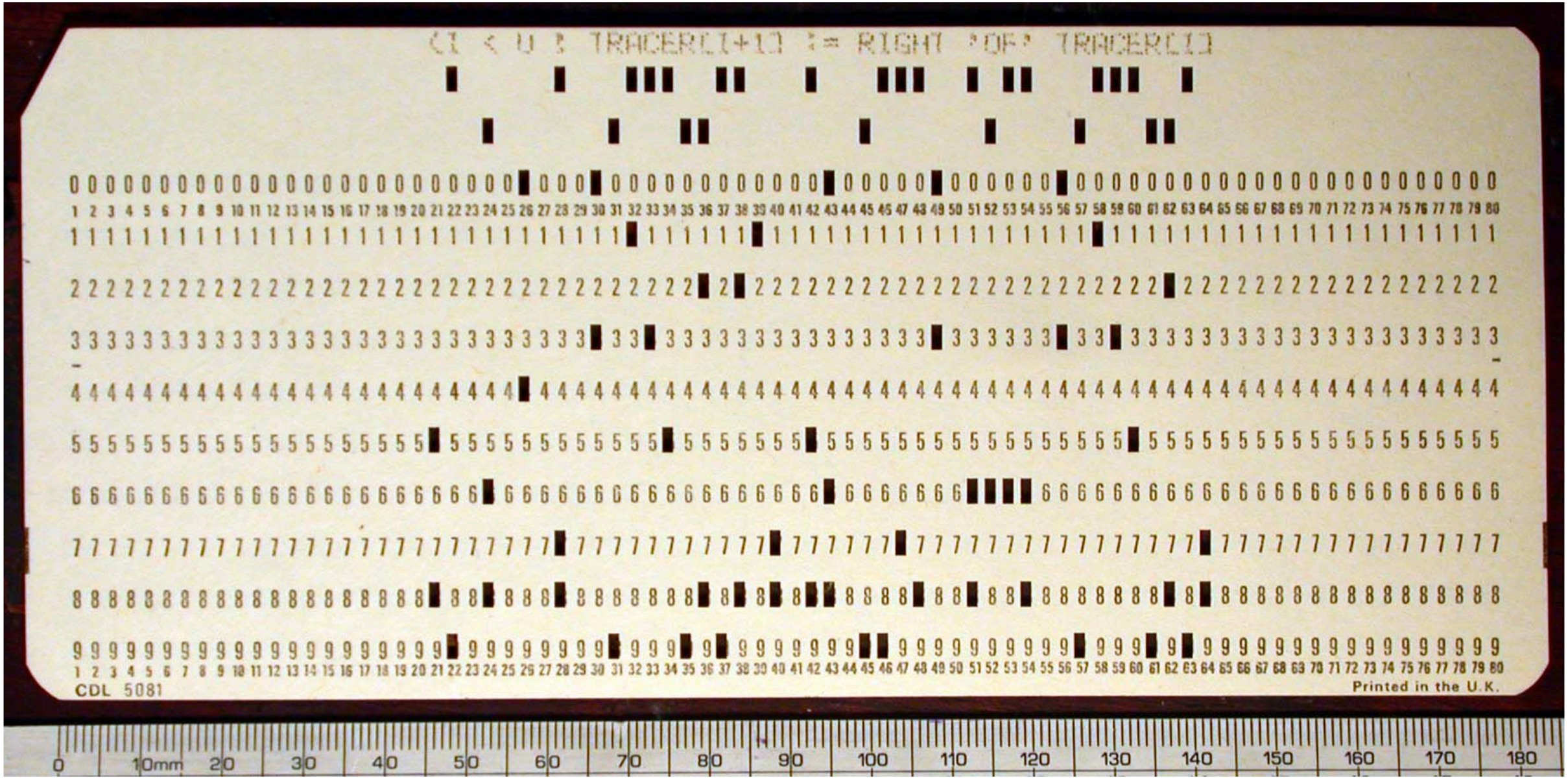
Assembly language

(RISC-V Assembly, x86 Assembly)

Machine language

(RISC-V Instruction Set, x86 Instruction Set)

https://go.cs61a.org/riscv


(I < U): TRACER(I+1) := RIGHT 'OF' TRACER(I)
CDL 5081
Printed in the U.K.

Programming Languages

A computer typically executes programs written in many different programming languages

Machine languages: statements are interpreted by the hardware itself

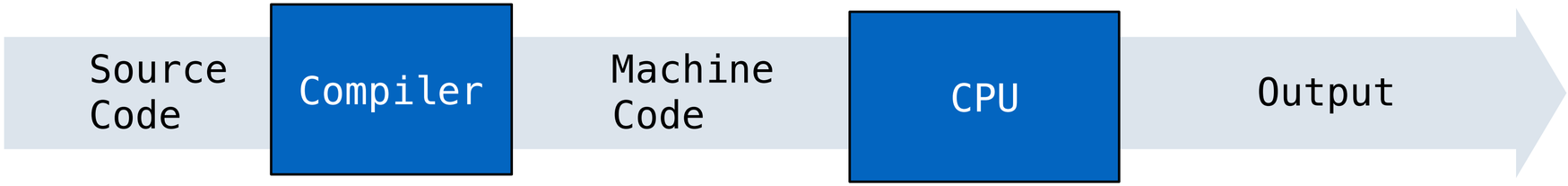
- A fixed set of instructions invoke operations implemented by the circuitry of the central processing unit (CPU)
- Operations refer to specific hardware memory addresses; no abstraction mechanisms

High-level languages: statements & expressions are interpreted by another program into values or compiled (translated) into another language

- Provide means of abstraction such as naming, function definition, and objects
- Abstract away system details to be independent of hardware and operating system

Compilers & Interpreters

Compilers: translate source code into machine code so that the machine code can be distributed and run repeatedly.



Interpreters: run source code directly producing an output/value, without first compiling it into machine code.

Programming Languages as Abstraction

Defining a new programming language that is tailored to a particular type of application or problem domain is a powerful form of abstraction.

Type of application: Go was designed for concurrent programs. It has built-in elements for expressing concurrent routines. It is used, for example, to implement chat servers, livestreams, and multiplayer game servers, with many simultaneous connections.

Problem domain: LaTeX was designed for generating static technical & scientific documentation. It has built-in elements for text formatting, mathematical expressions, and code blocks. It is used, for example, to write research papers.

A programming language has:

- **Syntax:** The legal statements and expressions in the language
- **Semantics:** The execution/evaluation rule for those statements and expressions

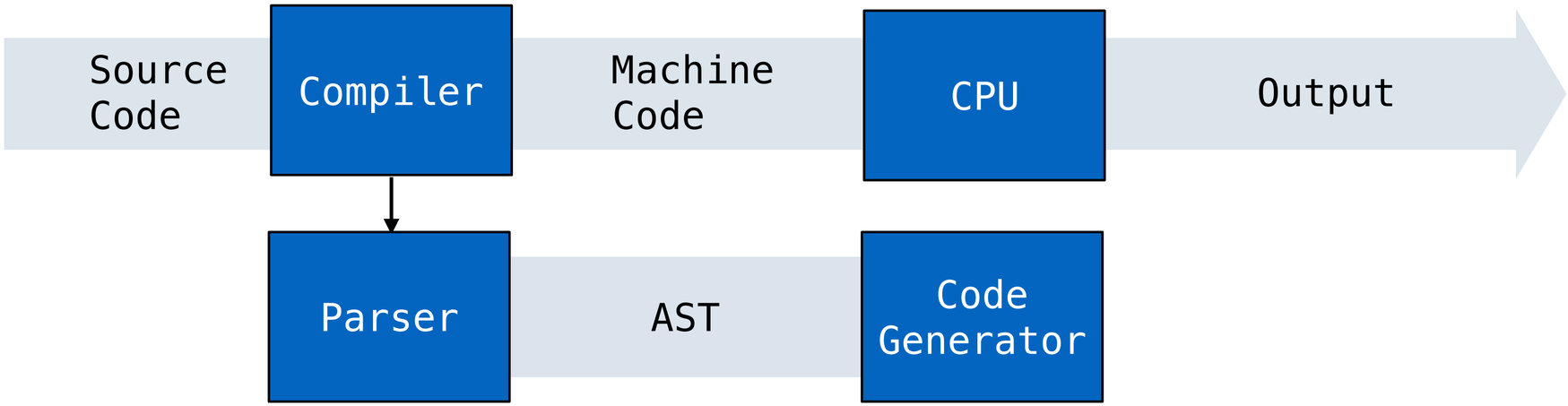
To create a new programming language, you either need a:

- **Specification:** A document describe the precise syntax and semantics of the language
- **Canonical Implementation:** An interpreter or compiler for the language

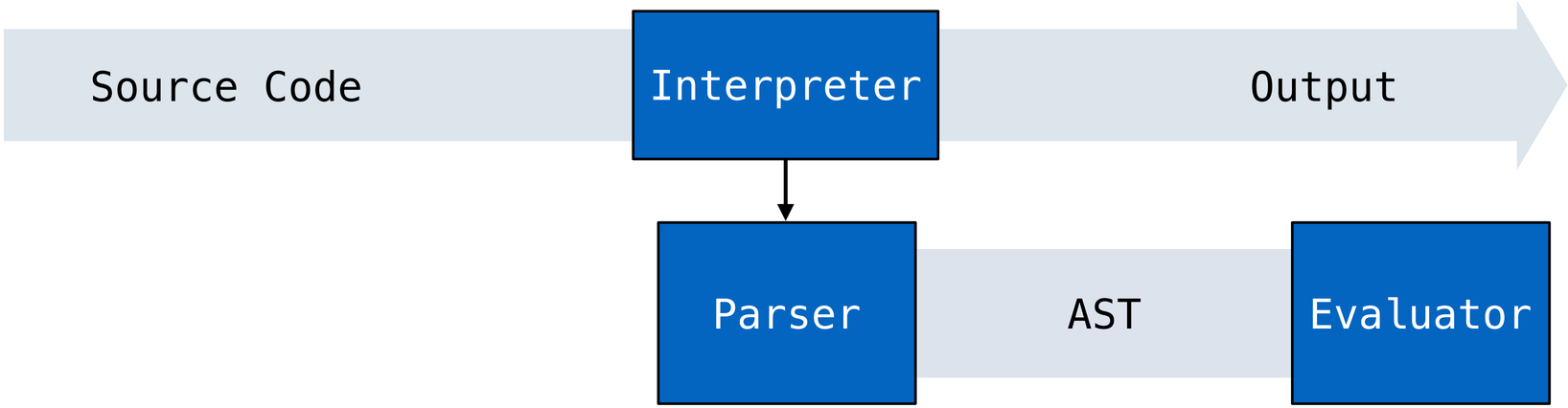
Understanding Source Code

In order to either interpret or compile source code, a **parser** must be written to understand that source code.

Compilers have parsers:



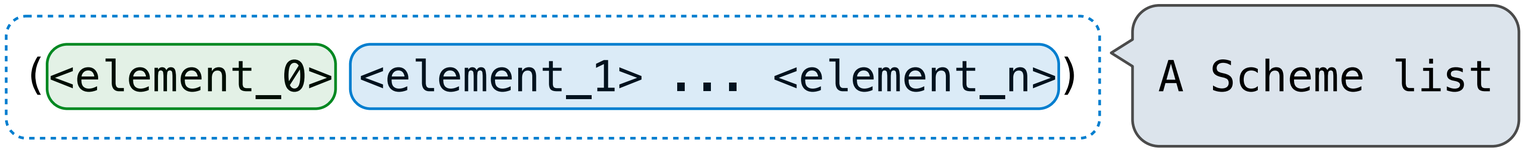
So do interpreters!



Parsing

Reading Scheme Lists

A Scheme list is written as elements in parentheses:

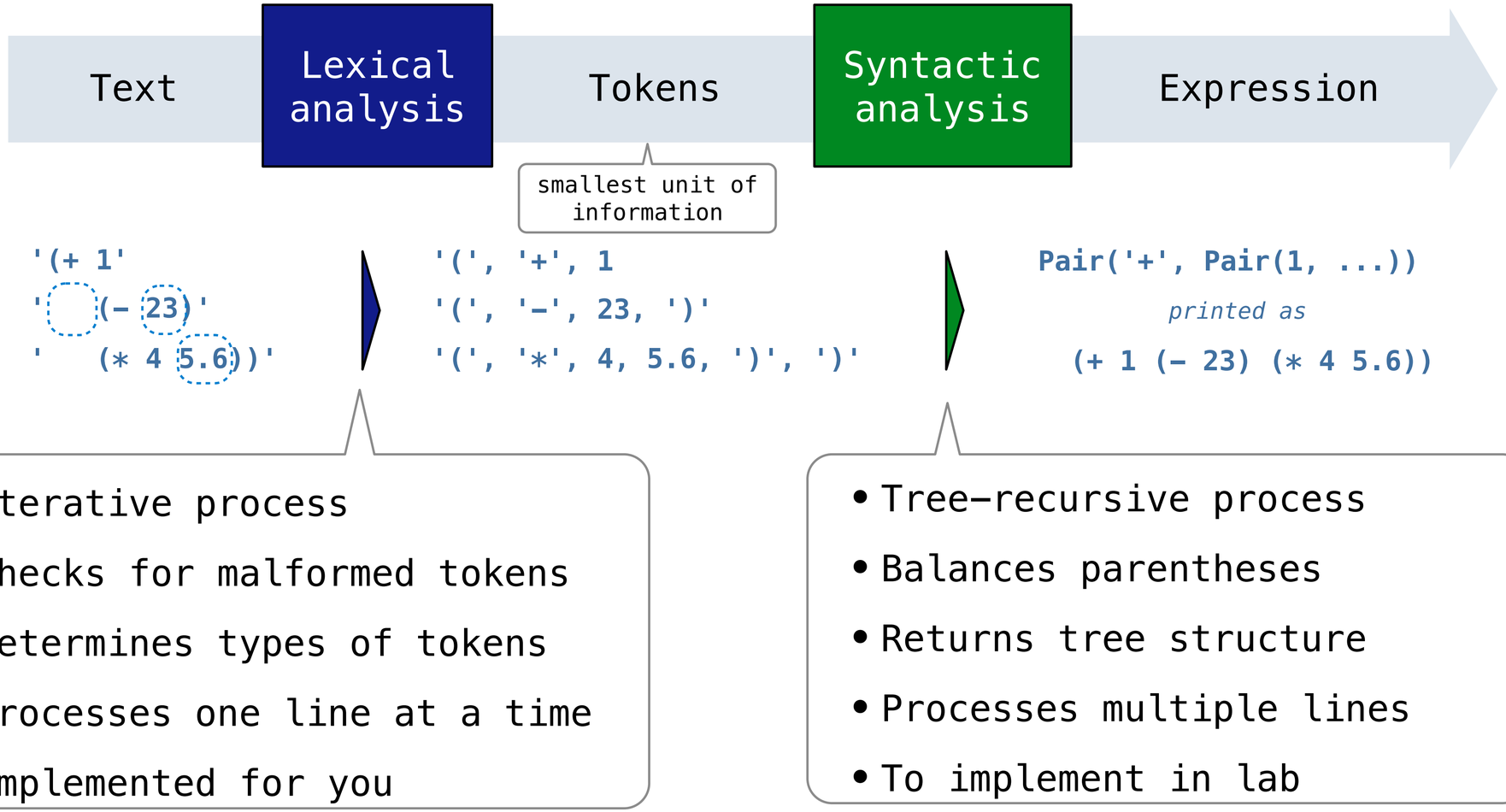


Each <element> can be a combination or primitive

```
(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))
```

Parsing

Parsing: turning a string representation of an expression into a structured object representing the expression. A parser takes text and returns an expression.



Syntactic Analysis

Syntactic analysis identifies the hierarchical structure of an expression, which may be nested

Each call to scheme_read consumes the input tokens for exactly one expression

Base case: symbols and numbers

Recursive call: scheme_read sub-expressions and combine them — scheme_read calls read_tail and read_tail calls scheme_read

(Demo)

Pair

(Demo)

The Calculator Language

Calculator Syntax

The Calculator language has primitive expressions and call expressions. (That's it!)

A primitive expression is a number: 2 -4 5.6

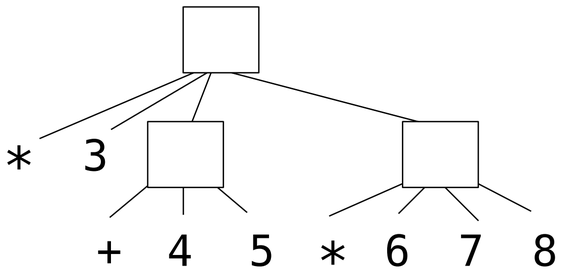
A call expression is a combination that begins with an operator (+, -, *, /) followed by 0 or more expressions: (+ 1 2 3) (/ 3 (+ 4 5))

Expressions are represented as Scheme lists (Pair instances) that encode tree structures.

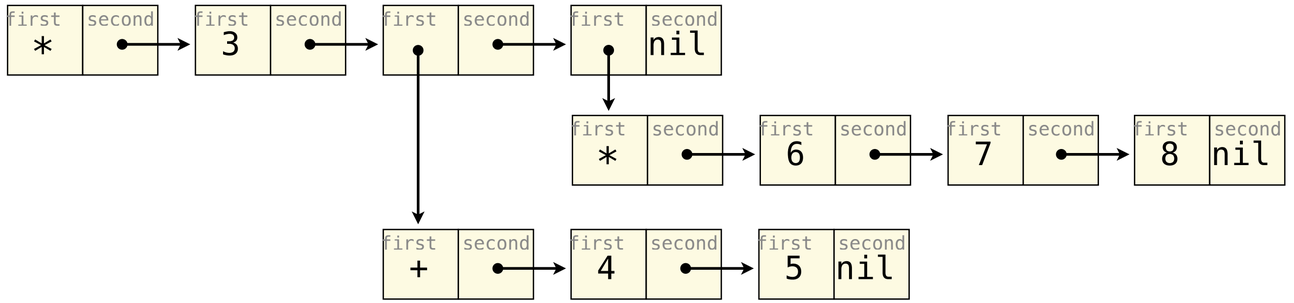
Expression

```
(* 3
   (+ 4 5)
   (* 6 7 8))
```

Expression Tree



Representation as Pairs



Calculator Semantics

The value of a calculator expression is defined recursively.

Primitive: A number evaluates to itself.

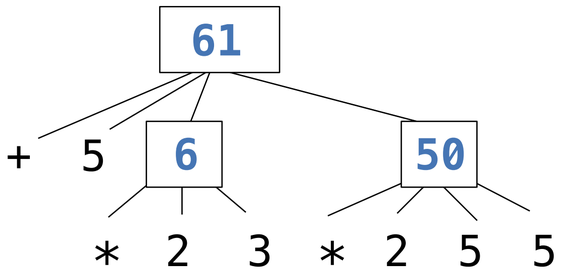
Call: A call expression evaluates to its argument values combined by an operator.

- **+**: Sum of the arguments
- *: Product of the arguments
- **-**: If one argument, negate it. If more than one, subtract the rest from the first.
- /: If one argument, invert it. If more than one, divide the rest from the first.

Expression

```
(+ 5
   (* 2 3)
   (* 2 5 5))
```

Expression Tree



Evaluation

The Eval Function

The eval function computes the value of an expression, which is always a number

It is a generic function that dispatches on the type of the expression (primitive or call)

Implementation

```
def calc_eval(exp):
    if isinstance(exp, (int, float)):
        return exp
    elif isinstance(exp, Pair):
        arguments = exp.second.map(calc_eval)
        return calc_apply(exp.first, arguments)
    else:
        raise TypeError
```

Recursive call returns a number for each operand

'+', '-', '*', '/'

A Scheme list of numbers

Language Semantics

A number evaluates...

to itself

A call expression evaluates...

to its argument values

combined by an operator

Applying Built-in Operators

The apply function applies some operation to a (Scheme) list of argument values

In calculator, all operations are named by built-in operators: +, -, *, /

Implementation

```
def calc_apply(operator, args):
    if operator == '+':
        return reduce(add, args, 0)
    elif operator == '-':
        ...
    elif operator == '*':
        ...
    elif operator == '/':
        ...
    else:
        raise TypeError
```

Language Semantics

+:
Sum of the arguments
-:
...
...

(Demo)

Interactive Interpreters

Read-Eval-Print Loop

The user interface for many programming languages is an interactive interpreter

1. Print a prompt
2. **Read** text input from the user
3. Parse the text input into an expression
4. **Evaluate** the expression
5. If any errors occur, report those errors, otherwise
6. **Print** the value of the expression and repeat

Raising Exceptions

Exceptions are raised within lexical analysis, syntactic analysis, eval, and apply

Example exceptions

- **Lexical analysis:** The token 2.3.4 raises ValueError("invalid numeral")
- **Syntactic analysis:** An extra) raises SyntaxError("unexpected token")
- **Eval:** An empty combination raises TypeError("() is not a number or call expression")
- **Apply:** No arguments to - raises TypeError("- requires at least 1 argument")

(Demo)

Handling Exceptions

An interactive interpreter prints information about each error

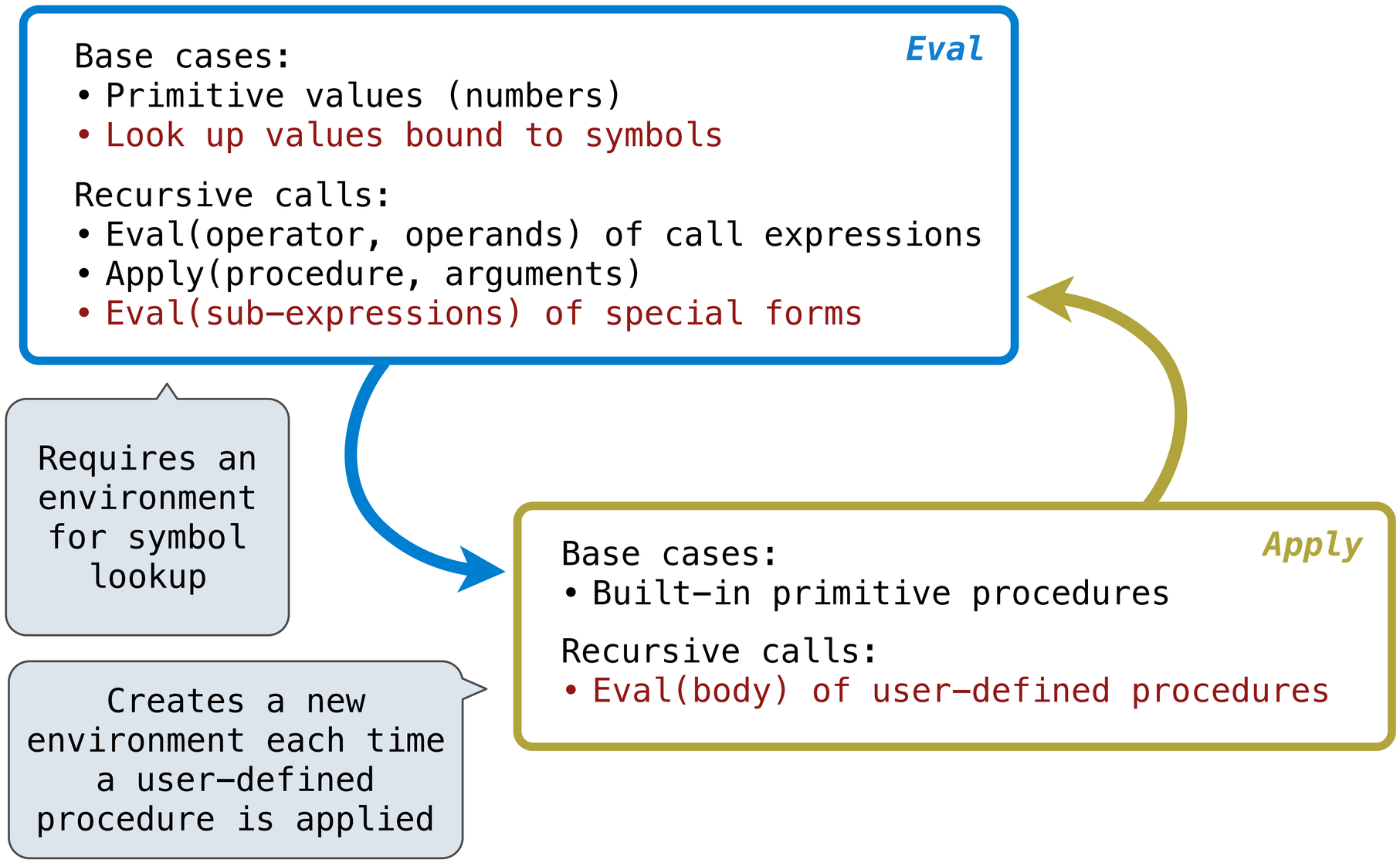
A well-designed interactive interpreter should not halt completely on an error, so that the user has an opportunity to try again in the current environment

Break


scm> (cons 1 (cons 2 nil))
>>> Pair(1, Pair(2, nil))
Always has been
Wait, it's all just Python?

Interpreting Scheme

The Structure of an Interpreter

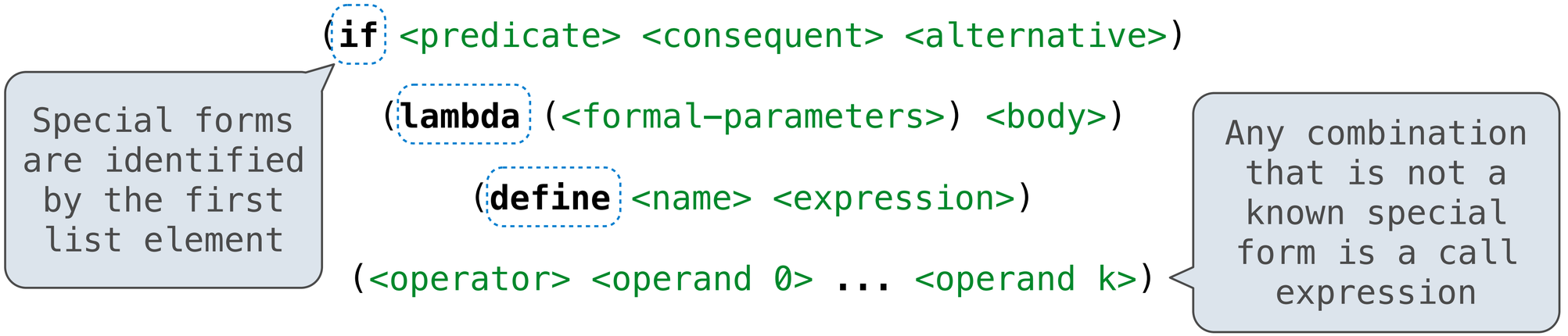


Special Forms

Scheme Evaluation

The scheme_eval function choose behavior based on expression form:

- Symbols are looked up in the current environment
- Self-evaluating expressions are returned as values
- All other legal expressions are represented as Scheme lists, called combinations




```
(define (demo s) (if (null? s) '(3) (cons (car s) (demo (cdr s))) ))

(demo (list 1 2))
```

Logical Forms

Logical Special Forms

Logical forms may only evaluate some sub-expressions

- **If** expression: (**if** <predicate> <consequent> <alternative>)
- **And** and **or**: (**and** <e1> **...** <en>), (**or** <e1> **...** <en>)
- **Cond** expression: (**cond** (<p1> <e1>) **...** (<pn> <en>) (else <e>))

The value of an if expression is the value of a sub-expression:

- Evaluate the predicate
- Choose a sub-expression: <consequent> or <alternative>
- Evaluate that sub-expression to get the value of the whole expression

do_if_form

(Demo)

Quotation

Quotation

The quote special form evaluates to the quoted expression, which is not evaluated

`(quote <expression>)` `(quote (+ 1 2))`  `(+ 1 2)`

The <expression> itself is the value of the whole quote expression

'<expression> is shorthand for (quote <expression>)

`(quote (1 2))` is equivalent to `'(1 2)`

The scheme_read parser converts shorthand ' to a combination that starts with quote

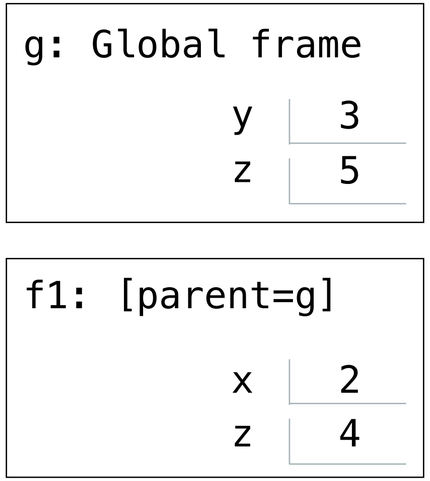
(Demo)

Frames and Environments

A frame represents an environment by having a parent frame

Frames are Python instances with methods **lookup** and **define**

In Project 4, Frames do not hold return values



(Demo)

Define Expressions

Define Expressions

Define binds a symbol to a value in the first frame of the current environment.

```
(define <name> <expression>)
```

1. Evaluate the <expression>
2. Bind <name> to its value in the current frame

```
(define x (+ 1 2))
```

Procedure definition is shorthand of define with a lambda expression

```
(define (<name> <formal parameters>) <body>)
```

```
(define <name> (lambda (<formal parameters>) <body>))
```

Lambda Expressions

Lambda Expressions

Lambda expressions evaluate to user-defined procedures

```
(lambda (<formal-parameters>) <body>)
```

```
(lambda (x) (* x x))
```

```
class LambdaProcedure:
    def __init__(self, formals, body, env):
        self.formals = formals ..... A scheme list of symbols
        self.body = body ..... A scheme list of expressions
        self.env = env ..... A Frame instance
```

Applying User-Defined Procedures

To apply a user-defined procedure, create a new frame in which formal parameters are bound to argument values, whose parent is the **env** attribute of the procedure

Evaluate the body of the procedure in the environment that starts with this new frame

```
(define (demo s) (if (null? s) '(3) (cons (car s) (demo (cdr s)))))

(demo (list 1 2))
```