Calculator

Class outline:

- Programming languages
- Parsing a language
- The Calculator language
- Evaluating a language
- Interactive interpreters

Programming languages

Levels of languages

High-level programming language
(Python, C++, JavaScript)

⬇

Assembly language
(Hardware-specific)

⬇

Machine language
(Hardware-specific)

Machine language

The language of the machine is all 1s and 0s, often specifying the action and the memory address to act on:

```
00000100 10000010   # Load data in 10000010
00000001 10000001   # Subtract data at 10000001
00000101 10000100   # Store result in 10000100
00001011 10000100   # Etc..
00001101 00010000
00010100 00000010
00000101 10000011
00001111 00000000
00010100 00000011
00000101 10000011
```

Code is executed directly by the hardware.

Assembly language

Assembly language was introduced for (slightly) easier programming.

Machine code	Assembly code
00000100 10000010	LOD Y
00000001 10000001	SUB X
00000101 10000100	STO T1
00001011 10000100	CPL T1
00001101 00010000	JMZ 16
00010100 00000010	LOD #2
00000101 10000011	STO Z
00001111 00000000	HLT
00010100 00000011	LOD #3
00000101 10000011	STO Z

Assembly still has a 1:1 mapping with machine language, however.

Higher-level languages

Higher level languages:

- provide means of abstraction such as naming, function definition, and objects
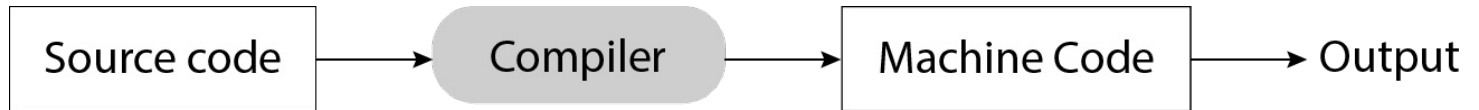- abstract away system details to be independent of hardware and operating system
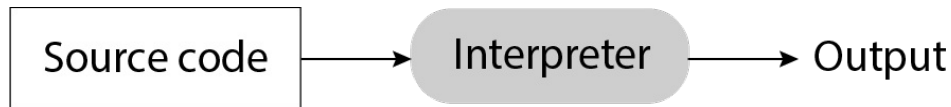
```
if x > y:
    z = 2
else:
    z = 3
```

Statements & expressions are either **interpreted** by another program or **compiled** (translated) into a lower-level language.

Compilers & Interpreters

A **compiler** translates source code into machine code, so that the machine code can be distributed and run repeatedly.
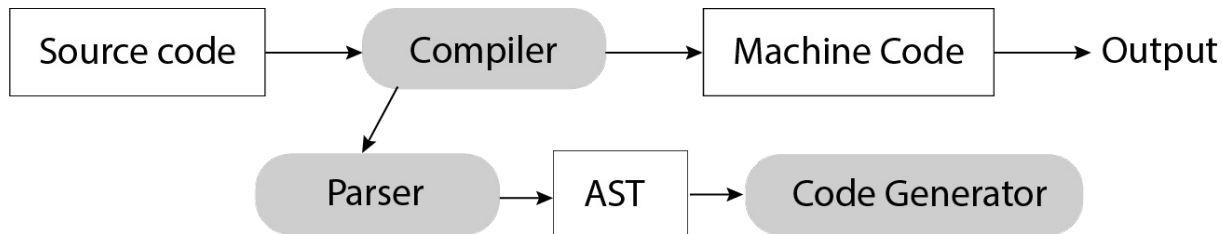
```
Source code  ──▶  Compiler  ──▶  Machine Code  ──▶ Output
```

An **interpreter** runs source code directly, without first compiling it into machine code.

```
Source code  ──▶  Interpreter  ──▶ Output
```

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 Scheme

Parsing: turning a string representation of an expression into a structured object representing the expression.

Reminder: Scheme Lists

A Scheme list is written as elements in parentheses:

```
(<element_0> <element_1> ... <element_n>)
```

Each <element> can be a combination or primitive.

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

Parsing

A parser takes text and returns an expression object.

Text	Lexical Analysis	Tokens	Syntactic Analysis	Expression
`'(+ 1'`	→	`'('`, `'+'`, `1`	→	`Pair('+', Pair(1, ...))`
`' (- 23)'`	→	`'('`, `'-'`, `23`, `')'`		printed as
`' (* 4 5.6))'`	→	`'('`, `'*'`, `4`, `5.6`, `')'`, `')'`		(+ 1 (- 23) (* 4 5.6))

Lexical analysis

`' (* 4 5.6))'` → `'('`, `'*'`, `4`, `5.6`, `')'`, `')'`

- Iterative process
- Checks for malformed tokens
- Determines types of tokens
- Processes one line at a time

Syntactic analysis

`'('`, `'+'`, `1`, … → `Pair('+', Pair(1, ...))`

- Tree-recursive process
- Balances parentheses
- Returns tree structure
- Processes multiple lines

In scheme_reader.py, each call to `scheme_read` consumes the input tokens for exactly one expression.

- Base case:
- Recursive case:

Syntactic analysis

`'('`, `'+'`, `1`, … → `Pair('+', Pair(1, ...))`

- Tree-recursive process
- Balances parentheses
- Returns tree structure
- Processes multiple lines

In scheme_reader.py, each call to `scheme_read` consumes the input tokens for exactly one expression.

- Base case: symbols and numbers
- Recursive case:

Syntactic analysis

`'('`, `'+'`, `1`, … → `Pair('+', Pair(1, ...))`

- Tree-recursive process
- Balances parentheses
- Returns tree structure
- Processes multiple lines

In scheme_reader.py, each call to `scheme_read` consumes the input tokens for exactly one expression.

- Base case: symbols and numbers
- Recursive case: read subexpressions and combine them

Pair class

The Pair class represents Scheme lists.

```python
class Pair:
    def __init__(self, first, second):
        self.first = first
        self.second = second
```

Well-formed list: (second element is either a pair or `nil`)

```python
s = Pair(1, Pair(2, Pair(3, nil)))
print(s)
len(s)
```

Improper lists:

```python
print(Pair(1, 2))
print(Pair(1, Pair(2, 3)))
len(Pair(1, Pair(2, 3)))
```

Pair class

The Pair class represents Scheme lists.

```python
class Pair:
    def __init__(self, first, second):
        self.first = first
        self.second = second
```

Well-formed list: (second element is either a pair or `nil`)

```python
s = Pair(1, Pair(2, Pair(3, nil)))
print(s)  # (1 2 3)
len(s)
```

Improper lists:

```python
print(Pair(1, 2))
print(Pair(1, Pair(2, 3)))
len(Pair(1, Pair(2, 3)))
```

Pair class

The Pair class represents Scheme lists.

```python
class Pair:
    def __init__(self, first, second):
        self.first = first
        self.second = second
```

Well-formed list: (second element is either a pair or `nil`)

```python
s = Pair(1, Pair(2, Pair(3, nil)))
print(s)   # (1 2 3)
len(s)     # 3
```

Improper lists:

```python
print(Pair(1, 2))
print(Pair(1, Pair(2, 3)))
len(Pair(1, Pair(2, 3)))
```

Pair class

The Pair class represents Scheme lists.

```python
class Pair:
    def __init__(self, first, second):
        self.first = first
        self.second = second
```

Well-formed list: (second element is either a pair or `nil`)

```python
s = Pair(1, Pair(2, Pair(3, nil)))
print(s)   # (1 2 3)
len(s)     # 3
```

Improper lists:

```python
print(Pair(1, 2))          # (1 . 2)
print(Pair(1, Pair(2, 3)))
len(Pair(1, Pair(2, 3)))
```

Pair class

The Pair class represents Scheme lists.

```python
class Pair:
    def __init__(self, first, second):
        self.first = first
        self.second = second
```

Well-formed list: (second element is either a pair or `nil`)

```python
s = Pair(1, Pair(2, Pair(3, nil)))
print(s)   # (1 2 3)
len(s)     # 3
```

Improper lists:

```python
print(Pair(1, 2))          # (1 . 2)
print(Pair(1, Pair(2, 3))) # (1 2 . 3)
len(Pair(1, Pair(2, 3)))
```

Pair class

The Pair class represents Scheme lists.

```python
class Pair:
    def __init__(self, first, second):
        self.first = first
        self.second = second
```

Well-formed list: (second element is either a pair or `nil`)

```python
s = Pair(1, Pair(2, Pair(3, nil)))
print(s)   # (1 2 3)
len(s)     # 3
```

Improper lists:

```python
print(Pair(1, 2))           # (1 . 2)
print(Pair(1, Pair(2, 3)))  # (1 2 . 3)
len(Pair(1, Pair(2, 3)))    # Error!
```

The Calculator Language

What's in a language?

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 need either one of these:

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

Calculator language syntax

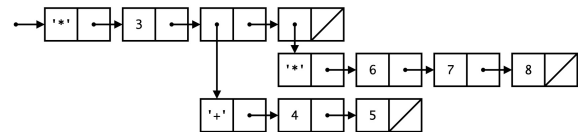The Calculator language has primitive expressions and call expressions. (That's it!)
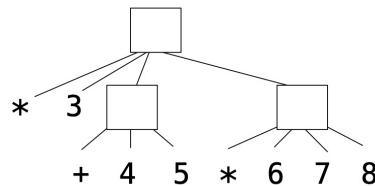
A **primitive expression** is a number: `2` `-4` `5.6`

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

Expression	Expression tree	Representation as pairs

Calculator language 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	Expression Tree

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

Evaluating Calculator

Evaluation: Turning a structured representation of a program into the expected program output according to the language semantics.

The eval function

The eval function computes the value of an expression.

It is a generic function that behaves according to the type of the expression (primitive or call).

Implementation

```python
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
```

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	**Language semantics**

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

+
Sum of the arguments

-
…

*
…

/
…

Interactive interpreters

REPL: 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 can be raised during 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")`

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.