CS 61A SU26

Lecture 20: Interpreters

July 27, 2026

Rebecca Dang

Join pollev.com/rebeccadang831 (or scan QR code) on your phone or laptop

We'll begin at Berkeley time (10 minutes after), as per tradition!

Potpourri

5 min

- Announcements
- Computing in the News
- WWSD: Scheme Lists

Announcements

- Quiz 3 grades released
- Updated Lecture 19 slide 19 to explicitly point out that this is likely a bug in 61A's Scheme interpreter
- Quiz 5 reservations are open
  - Last quiz of the summer 🎉
- More grades (and regrade requests) released
- How to check your discussion attendance score Ed post
- Lecture 21 has been updated on the course website
  - Tail Calls and **Macros**
  - Lecture video playlist also updated to include macros videos

Computing in the News

- US accuses American of allegedly wiping his phone using a 'duress' password during border search (TechCrunch)
  - "The U.S. Justice Department is prosecuting an American for allegedly providing U.S. border authorities with a passcode that wiped the contents of his phone, according to an indictment and media reports."
  - "The case centers on a feature included in GrapheneOS, a custom Android operating system that runs in place of the software on most modern Google Pixel devices."
- Can you have both security and privacy? What are the tradeoffs?
- What are privacy-preserving alternatives to big tech? Often open-source. Examples: GrapheneOS, Signal, DuckDuckGo, BitWarden
- Does everyone have a human right to privacy?
- Should companies provide backdoors to governments in the name of national security? (Apple famously says no)

When poll is active respond at **PollEv.com/rebeccadang831**

Activate this Poll with the Poll Everywhere Live app.

If video conferencing, you must be sharing your full screen to share the activated poll.

Scheme Lists: What would Scheme display?

Interpreters

50 min

- Why are we learning Scheme again?
- Programming Languages
- Parsing
- REPLs
- Scheme Calculator
- Raising Exceptions
- Define Special Form
- Eval-Apply Mutual Recursion
- Lexical vs. Dynamic Scope (`mu`)
- If you enjoyed this lecture, you might like...

Why are we learning Scheme again?

- Aside from teaching you a new programming paradigm (functional), **Scheme is a powerful language with few rules**
- This allows us to **write our own Scheme interpreter in Python** (your final project), **with a relatively "small" amount of code**
  - Ex: Compare this to CPython, the reference implementation of Python in the C programming language, which has ~350,000 lines of C code!
- But first, we have to learn how interpreters work
  - "The Structure and **Interpretation** of Computer Programs"

Programming Languages (1 of 3)

A computer typically executes programs written in many different programming languages. 2 overarching types:

1. **Machine languages:** statements are interpreted by the hardware itself
   a. Fixed set of instructions invoke operations implemented in the circuitry of the central processing unit (CPU)
   b. Operations refer to specific hardware addresses; no abstraction mechanisms
   c. See CS 61C
2. **High-level languages:** statements and expressions are interpreted by another program or compiled (translated) into another language
   a. Provide means of abstraction such as naming, function definition, and objects
   b. Abstract away system details to be independent of hardware and operating system

Programming Languages (2 of 3)

Demo: Disassembling Python into bytecode

```
>>> def square(x):
...     return x * x
...
>>> from dis import dis
>>> dis(square)
  1           RESUME                   0

  2           LOAD_FAST_LOAD_FAST      0 (x, x)
              BINARY_OP                5 (*)
              RETURN_VALUE
```

Programming Languages (3 of 3)

- Programming languages are often created to be tailored to a particular domain/problem/application
  - Ex: Erlang for concurrent programs
  - See also: Domain-specific languages (DSLs)
- A programming language has:
  - **Syntax**: The legal statements and expressions in the language ("grammar" or "structure" of a language)
  - **Semantics:** The execution/evaluation rule for those statements and expressions ("meaning" of a language)
- To create a new programming language, you need at least one of the following:
  - **Specification:** Document to describe the syntax and semantics
  - **Canonical/Reference Implementation**: An interpreter or compiler for the language
  - Python first had a reference implementation, then a specification. Most popular languages have both

Parsing (1 of 2)

The task of **parsing** a language involves coercing a string representation of an expression to the expression itself (so it can be executed).

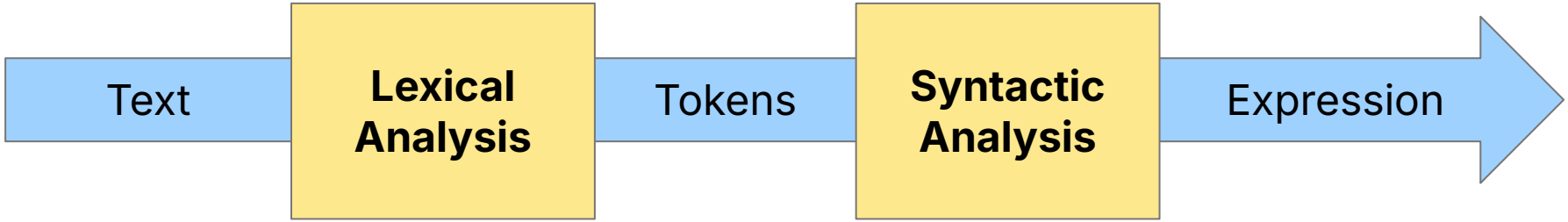
The cool thing about Scheme is that Scheme code is itself written as a Scheme list!

Ex: `(+ 1 2 3)` is really just a Scheme list `(list '+ 1 2 3)`

We can therefore parse all Scheme code into some linked list data structure in Python in order to interpret it!

Parsing (2 of 2)

How do we go from the code (text) into something a computer can understand and execute?




```
'(+ 1'
'    (- 23)'
'    (* 4 5.6))'
```

```
'(', '+', 1
'(', '-', 23, ')'
'(', '*', 4, 5.6, ')', ')'
```

```
Link('+', Link(1, ...))
```

printed as

```
(+ 1 (- 23) (* 4 5.6))
```

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

- Tree-recursive process
- Balances parentheses
- Returns an abstract syntax tree (AST)
- Processes multiple lines

Demo: Scheme Tokenizer and Reader

- `scheme_tokens.py` performs **lexical analysis** (it is a **tokenizer**, or a program that converts text into tokens)
- `scheme_reader.py` performs **syntactic analysis**
  - **Base case:** Symbols and numbers are self-evaluating
  - **Recursive case:** Call `scheme_read` on sub-expressions and combine them
    - Ex: In the expression below, the first call to `scheme_read` starts at the leftmost parentheses and ends at the rightmost parentheses

```
(+ 1 (- 23) (* 4 5.6))
```

    - **Where do the recursive calls happen?**

REPLs

- REPL stands for **R**ead-**E**val-**P**rint-**L**oop
  - Ex: The Python interpreter or the Scheme interpreter in 61A Code
- It is a user interface for programming languages that involves:
  - Printing a prompt
  - **Read**ing input text from the user
  - Parsing the text into an expression
  - **Eval**uating the expression
  - If there are errors, report them. Otherwise
  - **Print** the value of the expression
  - Repeat all of the above

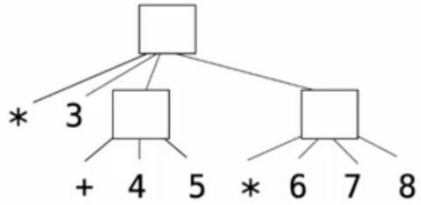
Scheme Calculator (1 of 2)

- Today, let's implement a simple interpreter for a subset of the Scheme language: the **Scheme Calculator** language!
- This language only has primitive and call expressions (no special forms)
  - Primitives are numbers only
  - A call expression is a combination that begins with an arithmetic operator (`+`, `-`, `*`, `/`) followed by 0 or more expressions, e.g. (+ 1 2 3) or (/ 3 (+ 4 5))

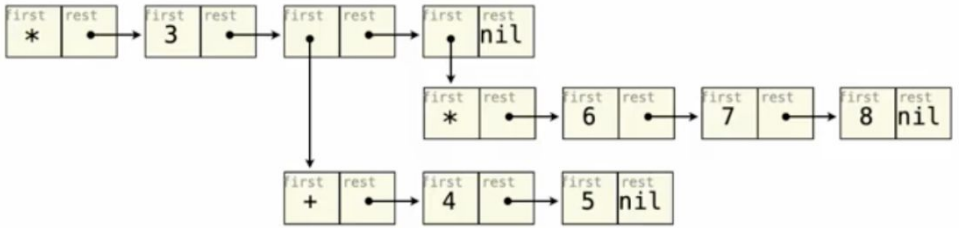
Expression

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

Expression Tree



Representation as Links



Scheme Calculator (2 of 2)

Language semantics:

- `+`: Sum of the arguments
- `*`: Product of the arguments
- `-`: If there is only one argument, negate it. Otherwise, subtract rest from the first.
- `/`: If there is only one argument, invert it (e.g. `1 / x`). Otherwise, divide the rest from the first.

Demo Repository

1. Go to https://github.com/phrdang/cs61a-su26-lec20
2. Click on the green dropdown and download a .zip of the repository
3. Unzip the file
4. Open the folder in VS Code
5. Follow the Installation instructions (see the `README.md` file)

Practice: REPLs

- Implement `read_eval_print_loop`
  - **Hint**: You do not need to define your own functions. Call existing ones that are either imported or defined in the `calc.py` file.
- Test if it works by trying out Scheme Calculator expressions. Start the interpreter by running: `uv run python3 calc.py`
- 2 min: Try implementing it yourself
- 2 min: Discuss with someone next to you and compare your implementations
  - What worked?
  - What didn't?
  - Why?

Practice: Basic Arithmetic

- Implement the 4 basic arithmetic operations (`+`, `*`, `-`, `/`) in `calc_apply`
  - **Hint**: The `Link` class provided in `scheme_reader.py` implements `__len__` and `__getitem__`, which means you can call `len` on `Link` objects and use indexing to access elements of the linked list
  - **Hint:** How can you use the `reduce` function as well as the imported functions on line 34 in `calc.py`?
- Test if it works by trying out Scheme Calculator expressions. Start the interpreter by running: `uv run python3 calc.py`
- 5 min: Try implementing it yourself
- 5 min: Discuss with someone next to you and compare your implementations
  - What worked?
  - What didn't?
  - Why?

Practice: Quotient Procedure

- Implement the `quotient` procedure in `calc_apply`
  - **Recall:** Quotient takes in exactly 2 arguments and performs floor division
  - **Hint:** How can you use an imported function on line 34 of `calc.py`?
- Test if it works by trying out Scheme Calculator expressions. Start the interpreter by running: `uv run python3 calc.py`
- 5 min: Try implementing it yourself
- 5 min: Discuss with someone next to you and compare your implementations
  - What worked?
  - What didn't?
  - Why?

Raising Exceptions

Within a REPL, exceptions can be raised at many different stages. For example:

- **Lexical analysis:** The token `2.3.4` raises a `ValueError("invalid numeral")`
- **Syntactic analysis:** An extra `)` raises a `SyntaxError("unexpected token")`
- **Apply:** Division by zero raises a `ZeroDivisionError`

A well-designed interpreter should not halt completely when an exception occurs, and instead it should report it and handle it gracefully.

What other things might go wrong in our current Scheme Calculator interpreter?

Possible responses:

- *Not enough arguments to `-` or `/` operator*
- *Invalid operator*
- *Too many arguments to `quotient` procedure*

Practice: Raising Exceptions

- Raise a `TypeError` (and provide a human-readable error message) in each situation in `calc_apply`:
  - Not enough or too many arguments
  - Invalid operator
- Test if it works by trying out Scheme Calculator expressions. Start the interpreter by running: `uv run python3 calc.py`
- 2 min: Try implementing it yourself
- 2 min: Discuss with someone next to you and compare your implementations
  - What worked?
  - What didn't?
  - Why?

Define Special Form

- Let's extend our Scheme Calculator language so that we can define variables!
- Recall that the `define` special form for variables looks like this:

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

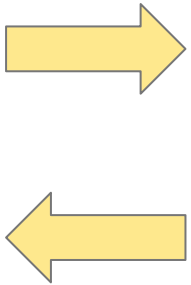
Practice: Define Special Form

- Implement the `define` special form functionality and variable lookup by implementing `do_define_form` and editing `calc_eval`
  - You should raise a `NameError` if the variable doesn't exist
- Test if it works by trying out Scheme Calculator expressions. Start the interpreter by running: `uv run python3 calc.py`
- Test your entire implementation: `uv run python3 -m doctest calc.py`
- 5 min: Try implementing it yourself
- 5 min: Discuss with someone next to you and compare your implementations
  - What worked?
  - What didn't?
  - Why?
  - **How might we extend our design so that we can support multiple frames and parent frames?**

Eval-Apply Mutual Recursion (Full Scheme Interpreter)

Eval

- Base cases:
  - Primitive values
  - Look up values bound to symbols → requires an environment
- Recursive cases:
  - Eval(operator, operands) of call expressions
  - Apply(procedure, arguments)
  - Eval(subexpressions) of special forms



Apply

- Base case:
  - Built-in primitive procedures
- Recursive case:
  - Eval(body) of user-defined procedures → creates a new frame

When poll is active respond at **PollEv.com/rebeccadang831**

Activate this Poll with the Poll Everywhere Live app.

If video conferencing, you must be sharing your full screen to share the activated poll.

Scope: What would Scheme display?

Lexical vs. Dynamic Scope (`mu`) (1 of 2)

The way in which names are looked up in Python and Scheme is called **lexical scope** (or static scope).

- **Lexical scope:** The parent of a frame is the environment in which a procedure was ***defined***
- **Dynamic scope:** The parent of a frame is the environment in which a procedure was ***called***

Lexical vs. Dynamic Scope (`mu`) (2 of 2)

Enter the `mu` special form:

```
(mu ([param] ...) <body> ...)
```

Creates a new `mu` procedure with `params` as its parameters and the `body` expressions as its body. When the procedure this form creates is called, the call frame will extend the environment the `mu` is called in.

When poll is active respond at **PollEv.com/rebeccadang831**

Activate this Poll with the Poll Everywhere Live app.

If video conferencing, you must be sharing your full screen to share the activated poll.

Mu: What would Scheme display?

If you enjoyed this lecture, you might like...

- CS 164: Programming Languages and Compilers
- CS 294-184: Building User-Centered Programming Tools (last offered SP26)
- Prof. Sarah Chasins is very cool! Talk to her if you're interested in PL (programming languages) and HCI (human-computer interaction) research
- Prof. Amy J. Ko at UW is also very cool! She does CS Education, HCI, and PL research