L11: Metalinguistic Abstraction

CS1101S: Programming Methodology

Boyd Anderson

October 26, 2022

Module Overview

- Unit 1—Functions (textbook Chapter 1)
  - Getting acquainted with programming, using functional abstraction
  - Learning to read programs, statically, and using the substitution model
  - Example applications: Runes, curves
- Unit 2—Data (textbook Chapter 2)
  - Getting familiar with data: pairs, lists, trees
  - Searching in lists and trees, sorting of lists
  - Example application: sound processing

Module Overview, continued

- Unit 3—State (parts of textbook Chapter 3)
  - Programming with stateful abstractions
  - Arrays, loops, searching in and sorting of arrays
  - Reading programs using the **environment model**
  - Example applications: robots, video processing
- Unit 4—Beyond (parts of textbook Chapters 3 and 4)
  - Streams (example application: video stream processing)
  - **Metalinguistic abstraction: understanding environment model by programming it**

CP3108: A project module on "Source Academy"

Time and workload: Sem 2 AY2022/23; CP3108A/B (2/4 MC)

CP3108: A project module on "Source Academy"

Time and workload: Sem 2 AY2022/23; CP3108A/B (2/4 MC)

Application: from Friday Week 11 to Friday Week 13

CP3108: A project module on "Source Academy"

Time and workload: Sem 2 AY2022/23; CP3108A/B (2/4 MC)

Application: from Friday Week 11 to Friday Week 13

Interviews: 14/11–2/12/2022

CP3108: A project module on "Source Academy"

Time and workload: Sem 2 AY2022/23; CP3108A/B (2/4 MC)

Application: from Friday Week 11 to Friday Week 13

Interviews: 14/11–2/12/2022

Scope: topics listed in application form; look out for announcements

Avenger recruitment

Avenger recruitment

Applications: from 11/11/2022 (Friday Week 13) to 9/1/2023 (Monday Week 1)

Avenger recruitment

Applications: from 11/11/2022 (Friday Week 13) to 9/1/2023 (Monday Week 1)

Interviews: throughout Sem 2

Avenger recruitment

Applications: from 11/11/2022 (Friday Week 13) to 9/1/2023 (Monday Week 1)

Interviews: throughout Sem 2

Selection: first round before end of Sem 2; waiting list

Why a metacircular evaluator?

Recall: programming is...

Why a metacircular evaluator?

Recall: programming is...

...communicating computational processes

Why a metacircular evaluator?

Recall: programming is...

...communicating computational processes

By now...

Why a metacircular evaluator?

Recall: programming is...

...communicating computational processes

By now...

...we are quite effective in using Source for communicating computational processes

Why a metacircular evaluator?

Recall: programming is...

...communicating computational processes

By now...

...we are quite effective in using Source for communicating computational processes

Consider the environment model...

Why a metacircular evaluator?

Recall: programming is...

...communicating computational processes

By now...

...we are quite effective in using Source for communicating computational processes

Consider the environment model...

The model describes a computational process (frames etc) that unfolds *whenever any* program runs.

Why a metacircular evaluator?

Recall: programming is...

...communicating computational processes

By now...

...we are quite effective in using Source for communicating computational processes

Consider the environment model...

The model describes a computational process (frames etc) that unfolds *whenever any* program runs.

Our goal today

Can we use Source to better understand the environment model?

Importance of metalinguistic abstraction

Importance of metalinguistic abstraction

The most fundamental idea in programming

The evaluator, which determines the meaning of statements and expressions in a programming language, is just another program.

Importance of metalinguistic abstraction

The most fundamental idea in programming

The evaluator, which determines the meaning of statements and expressions in a programming language, is just another program.

- Symbolic evaluator/differentiator in Lecture L6 is just another program!

Importance of metalinguistic abstraction

The most fundamental idea in programming

The evaluator, which determines the meaning of statements and expressions in a programming language, is just another program.

- Symbolic evaluator/differentiator in Lecture L6 is just another program!
- Source Academy where your programs run is just another program!

Importance of metalinguistic abstraction

The most fundamental idea in programming

The evaluator, which determines the meaning of statements and expressions in a programming language, is just another program.

- Symbolic evaluator/differentiator in Lecture L6 is just another program!
- Source Academy where your programs run is just another program!
- Firmware that you uploaded to the EV3 LEGO Mindstorms computers is just another program!

Importance of metalinguistic abstraction

The most fundamental idea in programming

The evaluator, which determines the meaning of statements and expressions in a programming language, is just another program.

- Symbolic evaluator/differentiator in Lecture L6 is just another program!
- Source Academy where your programs run is just another program!
- Firmware that you uploaded to the EV3 LEGO Mindstorms computers is just another program!
- Environment visualizer is just another program!

The most beautiful program ever written

Gerald Jay Sussman, CS1101S Guest Lecture 2020/21, 25:35

"The most beautiful program ever written, in my opinion, is the Eval/Apply Interpreter."

The most beautiful program ever written

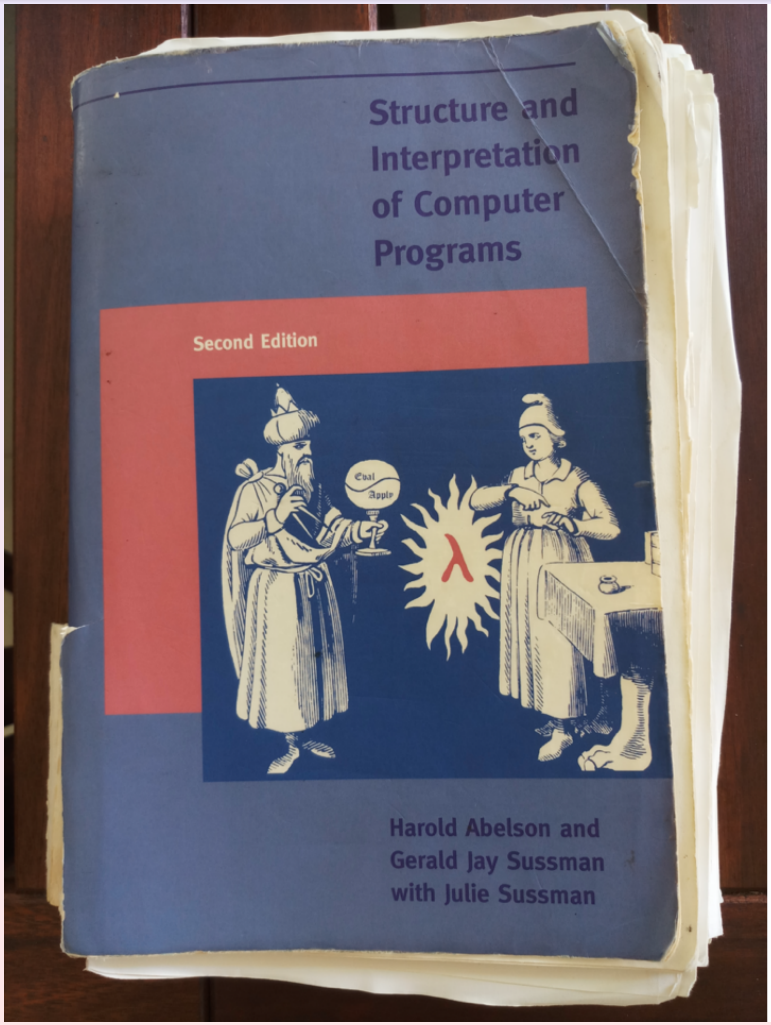
Gerald Jay Sussman, CS1101S Guest Lecture 2020/21, 25:35

"The most beautiful program ever written, in my opinion, is the Eval/Apply Interpreter."

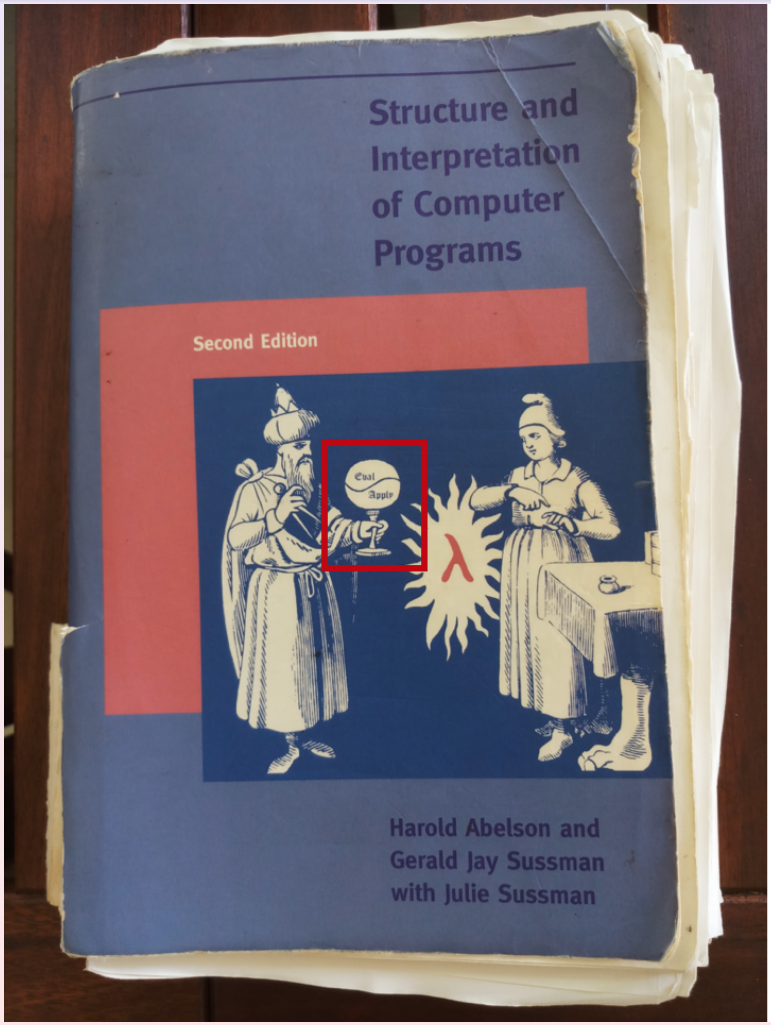
Why is this beautiful?

It is beautiful in the same way as Maxwell's Equations on electromagnetism are beautiful: It conveys deep insights in a few lines.

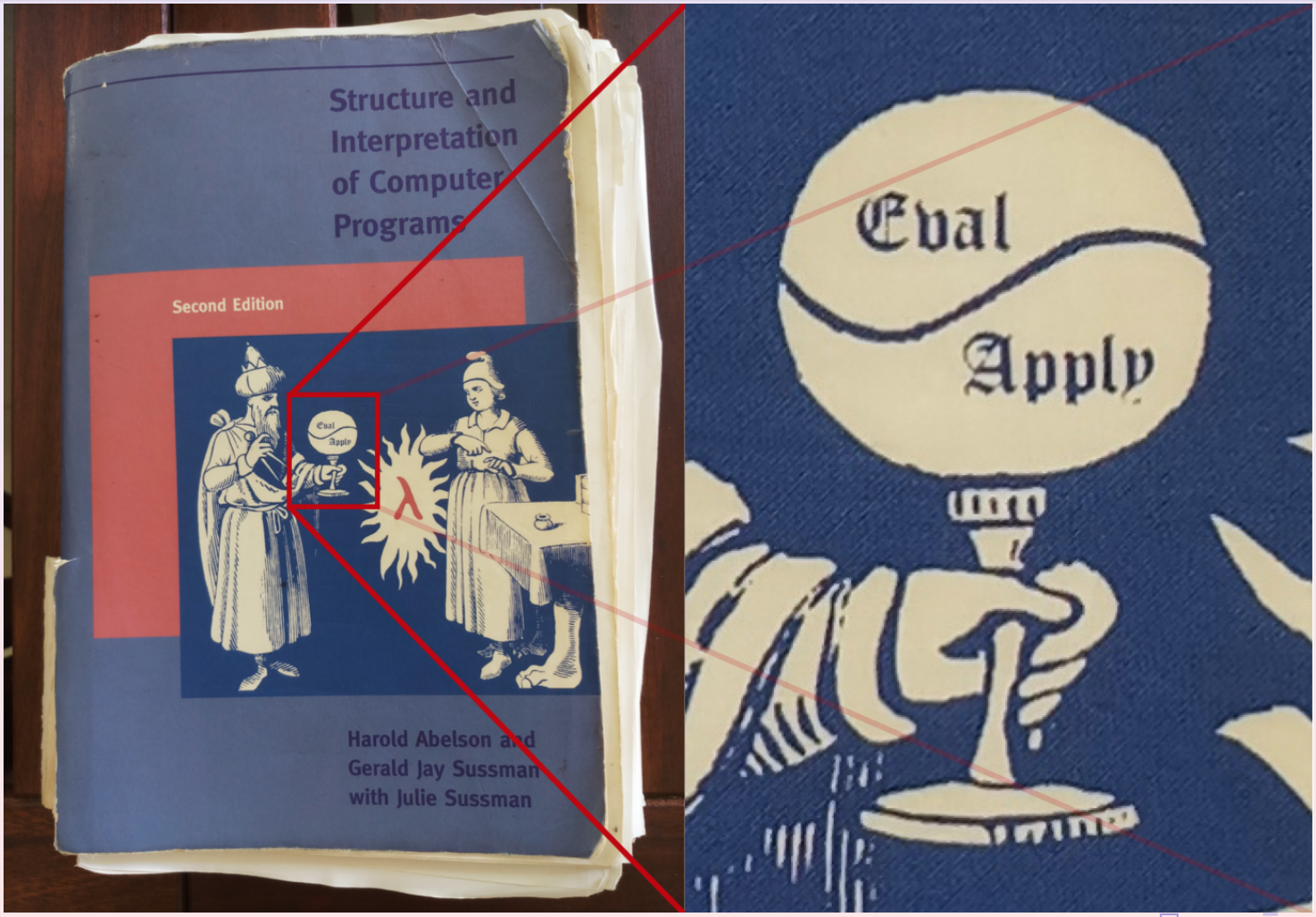
SICP


Structure and Interpretation of Computer Programs
Second Edition
Harold Abelson and
Gerald Jay Sussman
with Julie Sussman

SICP: What is that in the middle?

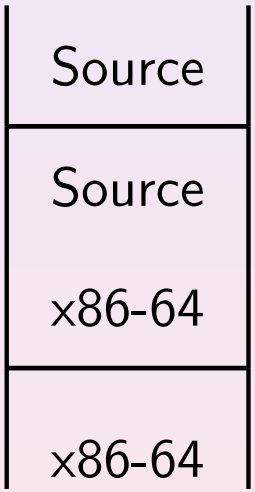


SICP: The importance of Eval/Apply

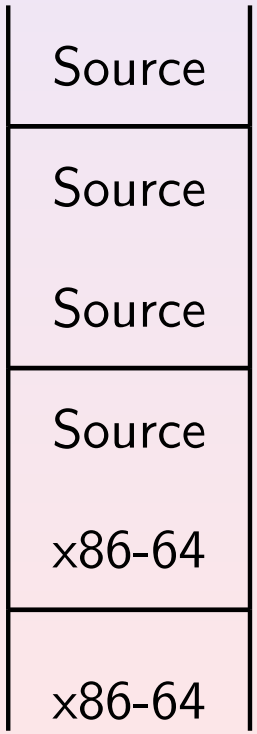


MetaCircular evaluator (MCE) as a language processor

`factorial` in Source Academy



`factorial` in MCE



Our task and the objective

Task

Our task and the objective

Task

We want to write an evaluator in Source that can execute programs that are written in Source

Our task and the objective

Task

We want to write an evaluator in Source that can execute programs that are written in Source

Why?

Our task and the objective

Task

We want to write an evaluator in Source that can execute programs that are written in Source

Why?

There is no *practical* use for such a *metacircular evaluator*.

Our task and the objective

Task

We want to write an evaluator in Source that can execute programs that are written in Source

Why?

There is no *practical* use for such a *metacircular evaluator*.
After all, in order to run the evaluator, you need to have already a working implementation of Source!

Our task and the objective

Task

We want to write an evaluator in Source that can execute programs that are written in Source

Why?

There is no *practical* use for such a *metacircular evaluator*.
After all, in order to run the evaluator, you need to have already a working implementation of Source!

Objective

Our task and the objective

Task

We want to write an evaluator in Source that can execute programs that are written in Source

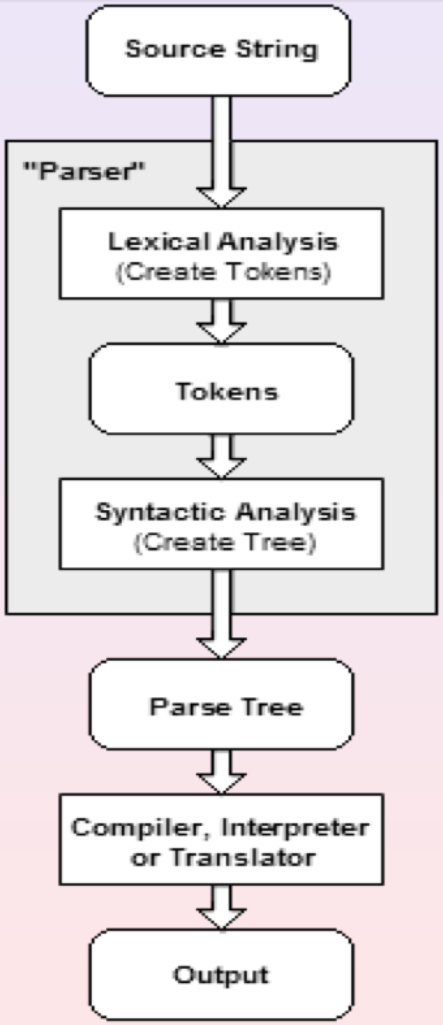
Why?

There is no *practical* use for such a *metacircular evaluator*. After all, in order to run the evaluator, you need to have already a working implementation of Source!

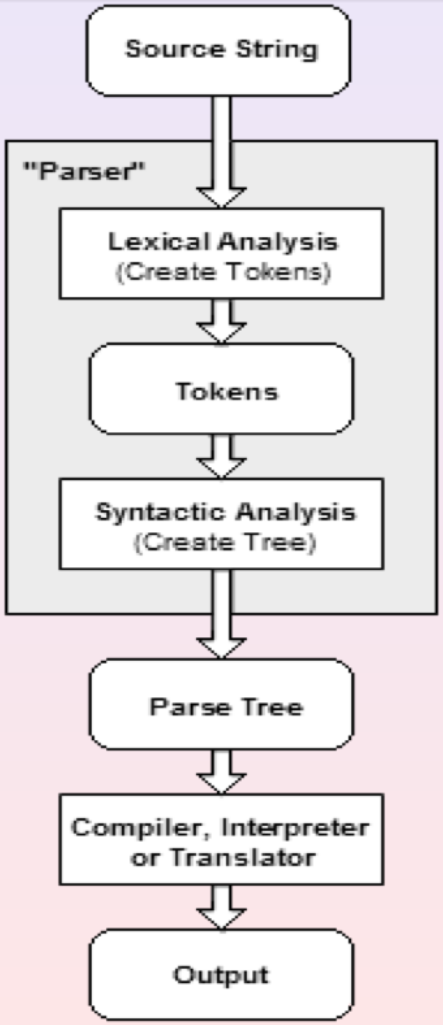
Objective

By writing a metacircular evaluator for Source, we hope to gain new insights into the language and a deeper understanding of the environment model.

First step for evaluator: parsing

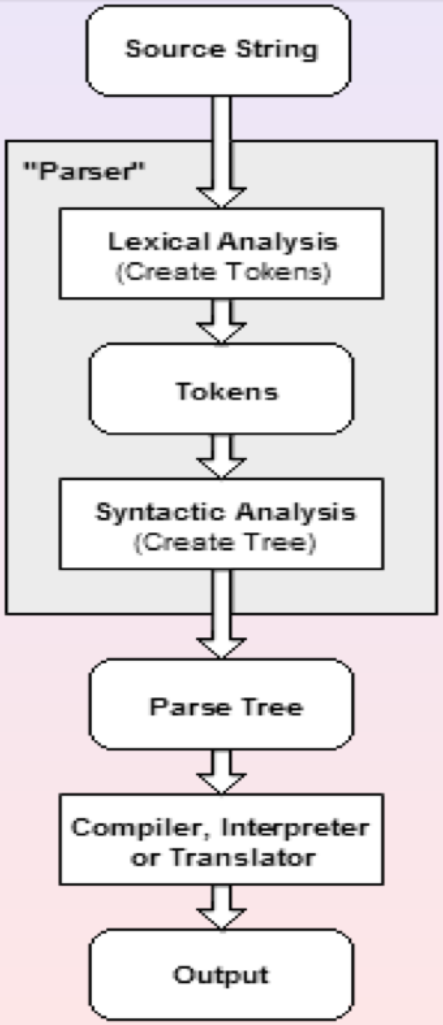


First step for evaluator: parsing



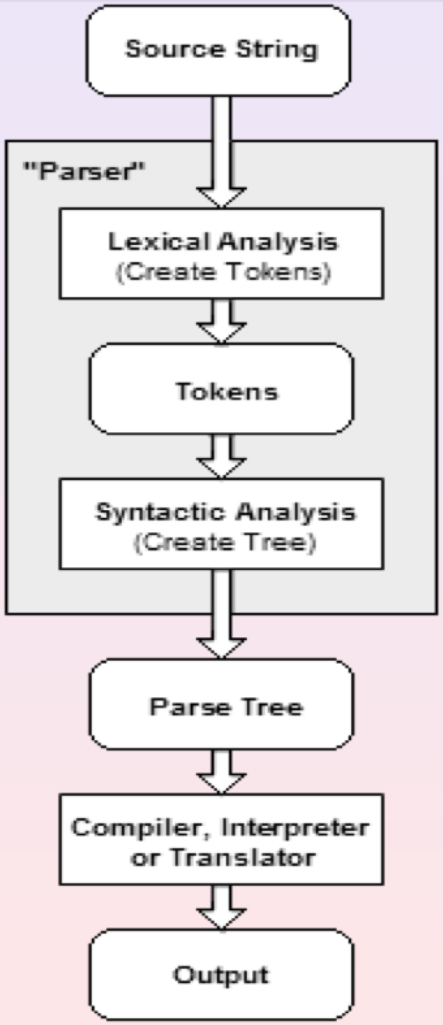
- Lexical analysis

First step for evaluator: parsing



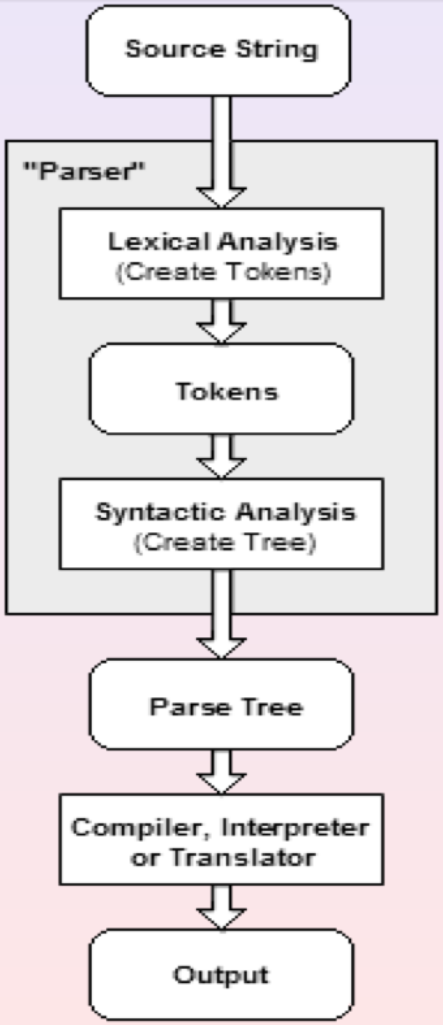
- Lexical analysis
  - Input characters are split into meaningful symbols (tokens) defined using *regular expressions*

First step for evaluator: parsing



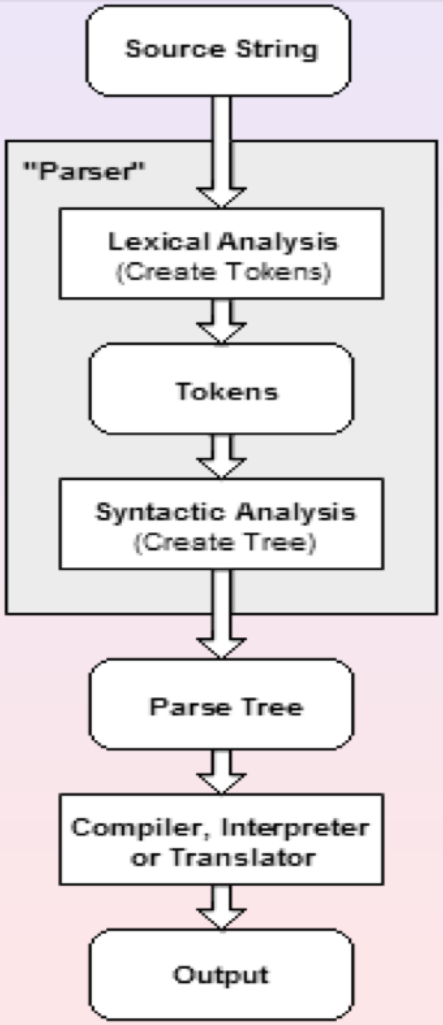
- Lexical analysis
  - Input characters are split into meaningful symbols (tokens) defined using *regular expressions*
  - Example: `"8 + 34;"` is split into tokens `8` `+` `34` `;`

First step for evaluator: parsing



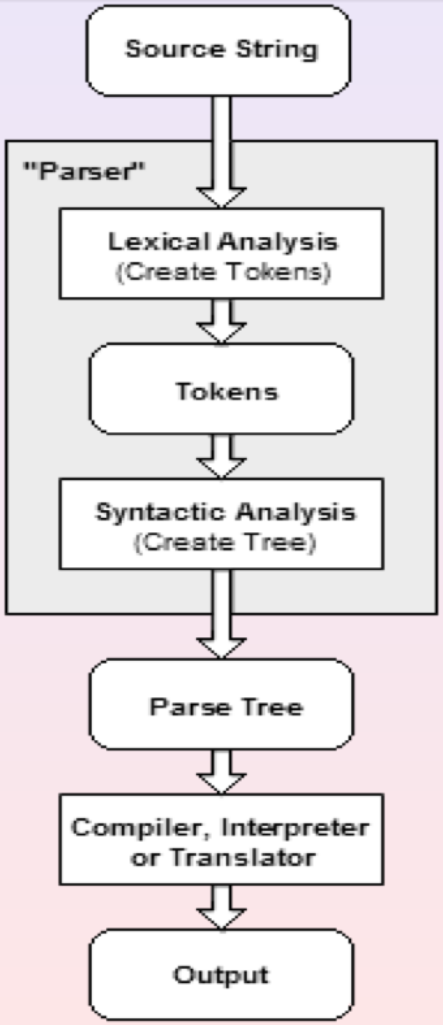
- Lexical analysis
  - Input characters are split into meaningful symbols (tokens) defined using *regular expressions*
  - Example: `"8 + 34;"` is split into tokens `8` `+` `34` `;`
- Parsing

First step for evaluator: parsing



- Lexical analysis
  - Input characters are split into meaningful symbols (tokens) defined using *regular expressions*
  - Example: `"8 + 34;"` is split into tokens `8` `+` `34` `;`
- Parsing
  - Checks that the tokens form allowable components

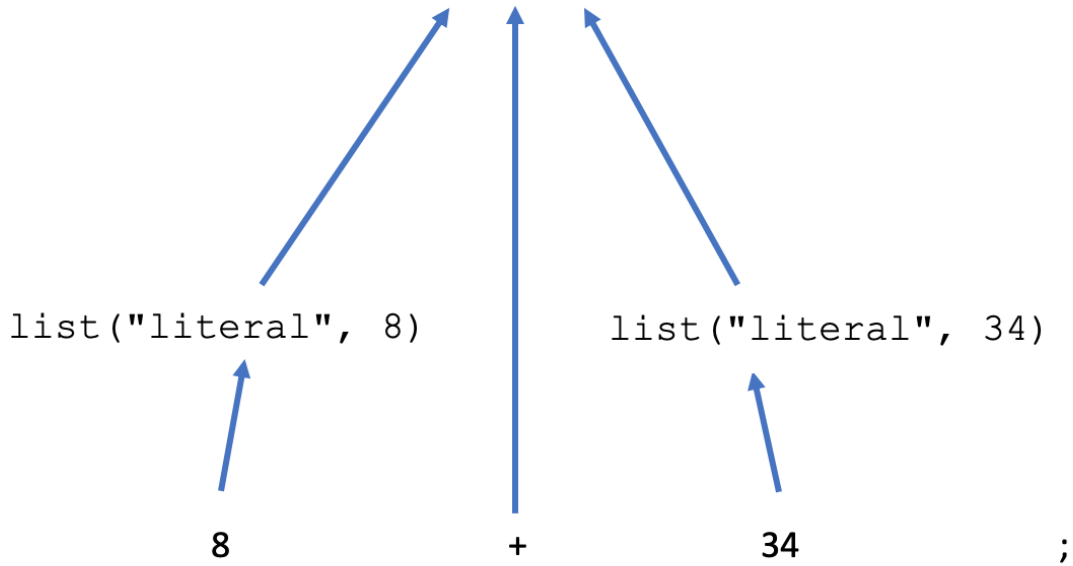
First step for evaluator: parsing



- Lexical analysis
  - Input characters are split into meaningful symbols (tokens) defined using *regular expressions*
  - Example: `"8 + 34;"` is split into tokens `8` `+` `34` `;`
- Parsing
  - Checks that the tokens form allowable components
  - Produces symbolic representation of the program, a *syntax tree*

Syntax tree in graphical notation

```
list(
 "binary_operator_combination","+", list("literal", 8), list("literal", 34))
```



Support for parsing in Source §4

```
const syntax_tree
    = parse("8 + 34;");
```

generates the syntax tree as Source list:

```
display_list(syntax_tree);
```

displays

```
list("binary_operator_combination", "+",
     list("literal", 8), list("literal", 34))
```

Scope of L11 (also for Mission, PA, Final)

L11: evaluator for Source §1 without `return`

Sequence of evaluators describing the structure and interpretation of a simplified Source §1, starting with a "Calculator language"

Scope of L11 (also for Mission, PA, Final)

L11: evaluator for Source §1 without `return`

Sequence of evaluators describing the structure and interpretation of a simplified Source §1, starting with a "Calculator language"

Does not include `return`, pairs, assignment, loops; lazy evaluation

Scope of L11 (also for Mission, PA, Final)

L11: evaluator for Source §1 without `return`

Sequence of evaluators describing the structure and interpretation of a simplified Source §1, starting with a "Calculator language"

Does not include `return`, pairs, assignment, loops; lazy evaluation

Reading

- SICP JS 4.1: L11

Calculator language in Backus-Naur Form (BNF)

stmt	::=	*expr* ;	expression statement
expr	::=	*expr bin-op expr*	binary operator combination
	|	*number*	number expression
	|	(*expr*)	parenthesised expression
bin-op	::=	+ | - | * | /	binary operator

Calculator language in Backus-Naur Form (BNF)

stmt	::=	*expr* ;	expression statement
expr	::=	*expr bin-op expr*	binary operator combination
	|	*number*	number expression
	|	(*expr*)	parenthesised expression
bin-op	::=	+ | - | * | /	binary operator

Example:

```
1.4 + 2.3 / 70.4;
```

Parsing the calculator language

To run the program

```
1.4 + 2.3 / 70.4;
```

we first parse it:

```
parse("1.4 + 2.3 / 70.4;");
```

returns the syntax tree

```
list("binary_operator_combination",
     "+",
     list("literal", 1.4),
     list("binary_operator_combination", "/",
          list("literal", 2.3),
          list("literal", 70.4)))
```

Syntax of literals

Example: parse("...38..."); ⇒
...list("literal", 38)...

```
// syntax predicate (in the following in maroon)
function is_literal(comp) {
  return is_tagged_list(comp, "literal");
}
function is_tagged_list(comp, the_tag) {
  return is_pair(comp) && head(comp) === the_tag;
}
// selector (in the following in blue)
function literal_value(comp) {
  return head(tail(comp));
}
```

Syntax of operator combinations

```
parse("... 2.3 + 70.4 ...");
```
$\Rightarrow$
```
... list("binary_operator_combination", "+",
         list("literal", 2.3),
         list("literal", 70.4)) ...
```

Syntax of operator combinations

```
parse("... 2.3 + 70.4 ...");⇒

... list("binary_operator_combination", "+",
         list("literal", 2.3),
         list("literal", 70.4)) ...

// syntax predicate (parsing details in 4.1.2)
const is_operator_combination = comp =>
is_tagged_list(comp,"binary_operator_combination"
// selectors
const op_comb_symbol      = comp => list_ref(comp,1);
const op_first_operand    = comp => list_ref(comp,2);
const op_second_operand = comp => list_ref(comp,3);
```

Evaluating the calculator language

```
function evaluate(expr) {
  return is_literal(expr)
    ? literal_value(expr)
    : is_operator_combination(expr)
    ? apply(op_comb_operator_symbol(expr),
        list_of_values(
          list(op_comb_first_operand(expr),
               op_comb_second_operand(expr))))
    : error(expr, "Unknown expression:");
}
```

Evaluating the calculator language

```
function evaluate(expr) {
  return is_literal(expr)
    ? literal_value(expr)
    : is_operator_combination(expr)
    ? apply(op_comb_operator_symbol(expr),
        list_of_values(
          list(op_comb_first_operand(expr),
               op_comb_second_operand(expr))))
    : error(expr, "Unknown expression:");
}

function list_of_values(exprs) {
  return map(evaluate, exprs);
}
```

Applying operator combinations

Example: `2.3 / 70.4`

```
function apply(operator, operands)
{
    const first_op = head(operands);
    const second_op = head(tail(operands));
    return operator === "+"
           ? first_op + second_op
           : operator === "-"
           ? first_op - second_op
           : operator === "*"
           ? first_op * second_op
           : operator === "/"
           ? first_op / second_op : error(); }
```

Adding booleans, conditionals, and sequences

stmt	::=	$\cdots$	
	$\mid$	*stmt*$_1$ ... *stmt*$_n$	statement sequence
expr	::=	$\cdots$	
	$\mid$	`true` $\mid$ `false`	boolean literals
	$\mid$	*expr*$_1$? *expr*$_2$: *expr*$_3$	conditional expression

Example:

```
8 + 34; true ? 1 + 2 : 17;
```

Evaluating the language

```
function evaluate(comp) {
  return ... // handling literals, operator comb
    : is_conditional(comp)
    ? eval_conditional(comp)
    : is_sequence(comp)
    ? eval_sequence(sequence_statements(comp))
    : error(comp, "Unknown component:");
}
```

Parsing conditionals: an example

```
parse("true ? 42 : 17;");
```

returns

```
list("conditional_expression",
     list("literal", true),
     list("literal", 42),
     list("literal", 17))
```

Parsing conditionals: an example

```
parse("true ? 42 : 17;");
```

returns

```
list("conditional_expression",
     list("literal", true),
     list("literal", 42),
     list("literal", 17))
```

Syntax predicate: `is_conditional`
Selectors: `cond_expr_pred`, `cond_expr_cons`, `cond_expr_alt`

Evaluating conditionals

Example: `true ? 42 : 17;`

```
function eval_conditional(comp) {
    return is_truthy(evaluate(cond_expr_pred(comp)))
           ? evaluate(cond_expr_cons(comp))
           : evaluate(cond_expr_alt(comp));
}
```

Evaluating conditionals

Example: `true ? 42 : 17;`

```
function eval_conditional(comp) {
    return is_truthy(evaluate(cond_expr_pred(comp)
           ? evaluate(cond_expr_cons(comp))
           : evaluate(cond_expr_alt(comp));
}
const is_truthy = x => is_boolean(x) ? x : error(
```

Evaluating conditionals

Example: `true ? 42 : 17;`

```
function eval_conditional(comp) {
    return is_truthy(evaluate(cond_expr_pred(comp)
           ? evaluate(cond_expr_cons(comp))
           : evaluate(cond_expr_alt(comp));
}
const is_truthy = x => is_boolean(x) ? x : error(
```

We explain conditionals in the interpreted language using conditional expressions in the interpreting language.

Evaluating conditionals

Example: `true ? 42 : 17;`

```
function eval_conditional(comp) {
    return is_truthy(evaluate(cond_expr_pred(comp)
           ? evaluate(cond_expr_cons(comp))
           : evaluate(cond_expr_alt(comp));
}
const is_truthy = x => is_boolean(x) ? x : error(
```

We explain conditionals in the interpreted language using conditional expressions in the interpreting language.

A cop-out? A stroke of genius? You decide!

Parsing sequences: an example

```
parse("1; 2; 3;");
```

returns

```
list("sequence",
     list(list("literal", 1),
          list("literal", 2),
          list("literal", 3))
```

Parsing sequences: an example

```
parse("1; 2; 3;");
```

returns

```
list("sequence",
     list(list("literal", 1),
          list("literal", 2),
          list("literal", 3))
```

Syntax predicate: `is_sequence`
Selectors: `sequence_statements`, `first_statement`, `rest_statements`
More syntax predicates: `is_empty_sequence`, `is_last_statement`

Evaluating sequences

Example: `8 + 34; true ? 1 + 2 : 17;`

```
function eval_sequence(stmts)
{
  if (is_empty_sequence(stmts)) {
    return undefined;
  } else if (is_last_statement(stmts)) {
    return evaluate(first_statement(stmts));
  } else {
    const ignore =
             evaluate(first_statement(stmts));
    return eval_sequence(
                    rest_statements(stmts)); }
}
```

Evaluating sequences

Example: `8 + 34; true ? 1 + 2 : 17;`

```
function eval_sequence(stmts)
{
  if (is_empty_sequence(stmts)) {
    return undefined;
  } else if (is_last_statement(stmts)) {
    return evaluate(first_statement(stmts));
  } else {
    const ignore =
             evaluate(first_statement(stmts));
    return eval_sequence(
                    rest_statements(stmts)); }
}
```

Another cop-out/stroke-of-genius: We explain sequences in the

Adding blocks and declarations

stmt	::=	⋯	
	|	{ *stmt* }	block
	|	`const` *name* = *expr* ;	constant declaration
expr	::=	⋯	
	|	*name*	name

Example:

```
const y = 4;
{
    const x = y + 7;
    x * 2;
}
```

Evaluating the language

```
function evaluate(comp, env) {
  return ...
         : is_name(comp)
         ? lookup_symbol_value(symbol_of_name(com
                               env)
         : is_block(comp)
         ? eval_block(comp, env)
         : is_declaration(comp)
         ? eval_declaration(comp, env)
         : error(comp, "Unknown component:");
}
```

Frames and environments

```
function make_frame(names, values) {
    return pair(names, values);
}
const frame_names = head;
const frame_values = tail;

function extend_environment(ns, vs, e) {
    return pair(make_frame(ns, vs), e);
}
const the_empty_environment = null;
const the_global_environment
                    = the_empty_environment;
```

Evaluating blocks

Example of block statement:

```
{ const x = y + 7; x * 2; }
```

```
function eval_block(component, env) {
    const body = block_body(component);
    const locals = scan_out_declarations(body);
    const unassigneds = list_of_unassigned(locals
    return evaluate(body,
                    extend_environment(locals,
                                       unassigned
                                       env));
}
```

Evaluating blocks

Example of block statement:

```
{ const x = y + 7; x * 2; }
```

```
function eval_block(component, env) {
    const body = block_body(component);
    const locals = scan_out_declarations(body);
    const unassigneds = list_of_unassigned(locals
    return evaluate(body,
                    extend_environment(locals,
                                       unassigned
                                       env));
}
const list_of_unassigned =
 symbols => map(symbol => "*unassigned*", symbols
```

Scanning out declarations

Example of block statement:

```
{ const x = y; { const y = 2; } const z = x/2; x;
```

```
function scan_out_declarations(comp) {
    return is_sequence(comp)
           ? accumulate(
                 append,
                 null,
                 map(scan_out_declarations,
                     sequence_statements(comp)))
           : is_declaration(comp)
           ? list(declaration_symbol(comp))
           : null;
}
```

Evaluating declarations

Example: `const x = y + 7;`

```
function eval_declaration(comp, env) {
    assign_symbol_value(
      declaration_symbol(comp),
      evaluate(declaration_value_expression(comp),
               env),
      env);
    return undefined;
}
```

Assign a name to a value

Example: `const x = 8 + 34;`

$\Rightarrow$ `assign_symbol_value("x", 42, env)`

```
function assign_symbol_value(symbol, val, env) {
  function scan_write(symbols, vals) {
    return symbol === head(symbols)
        ? set_head(vals, val) // was: "*unassigned*"
        : scan_write(tail(symbols), tail(vals));
  }
  const frame = first_frame(env);
  return scan_write(frame_symbols(frame),
                    frame_values(frame));
}
```

Some more environment functions

```
function first_frame(env) {
    return head(env);
}

function enclosing_environment(env) {
    return tail(env);
}

function is_empty_environment(env) {
    return is_null(env);
}
```

Looking up a symbol

```
function lookup_symbol_value(name, env)
{
  function env_loop(env) {
    function scan_read(names, vals) {...}
    if (is_empty_environment(env)) {
        error(name, "Unbound name:");
    } else {
      const frame = first_frame(env);
      const value = scan_read(frame_names(frame),
                              frame_values(frame));
      if (value === "*unassigned*") {
        error(name, "Used before declared:");
      } else { return value;
} } }
```

Adding compound functions

stmt	::=	···	
	|	`function` *name* `(` *params* `)` *block*	function declaration
expr	::=	···	
	|	*expr*`(` *expr*$_1$`,` ... `,` *expr*$_n$`)`	function application
	|	*params* `=>` *block*	lambda expression

Example:

```
function fact(n) { // no need for return
    n === 1 ? 1 : n * fact(n - 1);
}
fact(5); // result: 120
```

Evaluating compound functions

```
function evaluate(comp, env) {
 return ...
   : is_lambda_expression(comp)
   ? make_function(lambda_parameter_symbols(comp),
                   lambda_body(comp), env)
   : is_function_declaration(comp)
   ? evaluate(function_decl_to_constant_decl(comp),
              env)
   : is_application(comp)
   ? apply(evaluate(function_expression(comp), env),
           list_of_values(arg_expressions(comp), env))
   : error(comp, "Unknown component:");
}
```

Function objects

Example: `x => x + y`

```
function make_function(parameters, body, env) {
    return list("compound_function",
                parameters, body, env);
}
```

Transformation of function declaration

```
function fact(n) {
    n === 1 ? 1 : n * fact(n - 1);
}
// becomes
const fact = n => { n===1 ? 1 : n * fact(n - 1);

function function_decl_to_constant_decl(comp) {
  return make_constant_declaration(
            function_declaration_name(comp),
            make_lambda_expression(
              function_declaration_parameters(comp
              function_declaration_body(comp)));
}
```

Function application

Example: `fact(n - 1)`

```
function apply(fun, args) {
  return is_primitive_function(fun)
         ? apply_primitive_function(fun, args)
         : is_compound_function(fun)
         ? evaluate(function_body(fun),
                    extend_environment(
                       function_parameters(fun),
                       args,
                       function_environment(fun))
         : error(fun, "Unknown function type:");
}
```

Declaring primitive functions

```
const primitive_functions =
  list(pair("math_sin", math_sin),
       ...
```

Declaring primitive functions

```
const primitive_functions =
  list(pair("math_sin", math_sin,
            ...
            // operator combinations can be treated
            // as primitive function applications
            pair("+"          , (x, y) => x + y),
            pair("-"          , (x, y) => x - y),
            ...));
```

Declaring primitive functions

```
const primitive_functions =
  list(pair("math_sin", math_sin,
        ...
        // operator combinations can be treated
        // as primitive function applications
        pair("+"          , (x, y) => x + y),
        pair("-"          , (x, y) => x - y),
        ...);
function setup_environment () {
    ...primitive_functions...
}
const the_global_environment =
    setup_environment();
```

Applying primitive functions

```
function apply_primitive_function(fun, arglist) {
    return apply_in_underlying_javascript(
                primitive_implementation(fun),
                arglist);
}
```

Return statements in JavaScript

The Problems

- Functions that do not evaluate a return statement must return `undefined`

Return statements in JavaScript

The Problems

- Functions that do not evaluate a return statement must return `undefined`
- Evaluation of a return statement *anywhere* in the function body will return from the function with the result of evaluating the return expression

Adding return statements

$$\begin{array}{rcll} \textit{stmt} & ::= & \cdots & \\ & & \mid \texttt{return}\ \textit{expr}\ \texttt{;} & \text{return statement} \end{array}$$

Example:

```
function fact(n) {
    return n === 1 ? 1 : n * fact(n - 1);
}
fact(5);
```

Evaluating compound functions

```
function evaluate(comp, env) {
 return ...
   : is_return_statement(comp)
   ? eval_return_statement(component, env)
   : error(comp, "Unknown component:");
}
```

Handling return statements

Example: `return n * fact(n - 1);`

```
function eval_return_statement(stmt, env) {
    return make_return_value(
        evaluate(return_statement_expression(stmt),
                 env));
}
function make_return_value(content) {
    return list("return_value", content);
}
function is_return_value(value) {
    return is_tagged_list(value, "return_value");
}
```

Handling return values in sequences

```
function eval_sequence(stmts, env)                    {
  if (is_empty_sequence(stmts)) {
    return undefined;
  } else if (is_last_statement(stmts)) {
    return evaluate(first_statement(stmts), env);
  } else {
    const first_stmt_value =
            evaluate(first_statement(stmts),env);
    if (is_return_value(first_stmt_value)) {
      return first_stmt_value;
    } else {
      return eval_sequence(
                    rest_statements(stmts),env);
  } }                                                  }
```

Handling return values in apply

```
function apply(fun, args) {
  ...
  const result = evaluate(...body...);
  if (is_return_value(result)) {
    return return_value_content(result);
  } else {
    return undefined;
  } ...
}
```

Summary

Summary

- Metacircular evaluator:
  communicating the computational process called
  "environment model"

Summary

- Metacircular evaluator:
  communicating the computational process called "environment model"
- Frames and environment, clarified as *data types*

Summary

- Metacircular evaluator:
  communicating the computational process called "environment model"
- Frames and environment, clarified as *data types*
- Demystify environments: lists of pairs!

Summary

- Metacircular evaluator:
  communicating the computational process called "environment model"
- Frames and environment, clarified as *data types*
- Demystify environments: lists of pairs!
- Demystify function objects: lists!

Summary

- Metacircular evaluator:
  communicating the computational process called "environment model"
- Frames and environment, clarified as *data types*
- Demystify environments: lists of pairs!
- Demystify function objects: lists!
- `evaluate`—`apply` loop