Meta-Circuit Evaluator

Expressions & Statements

Literals: list("literal", 5)
is_literal (predicate) — literal_value (selector)

list("literal", true)

Bin Op Comb: → is_bin_op_combination
list("binary_operator_combination",
"+", → operator_symbol
E.g. 5+7 — list(...), → first_operand
list(...)) → second_operand

Sequence:
list("sequence", → is_sequence
list(list(...), → sequence_statements
list(...))

Conditional:
list("conditional", → is_conditional
list("literal", true), → cond_predicate
list(...), → cond_consequent
list(...)) → cond_alternative

Block: list("block", → is_block
list(...)) → block_body

Application:
list("application", → is_application
list("name", "myFunc1"), → function_expression
list("literal", 10)) → arg_expressions

Function Declaration:
list("function_declaration", → is_function_declaration
list("name", "myFunc1"), → function_declaration_name
list(list("name", "x"), → function_declaration_parameters
list("name", "y")),
list("return_statement", ...)) → function_declaration_body

Declaration: list("declaration", → is_declaration
E.g. const x = y+7; "x", → declaration_symbol
list("lambda_expression", ...))
↳ declaration_value_expression
(can be lambda, bin op comp, literal, name etc.)

Name: list("name", → is_name
"x") → symbol_of_name

Assignment: E.g. x = y+7;
list("assignment", → is_assignment
"x", → assignment_symbol
list("...", ...)) → assignment_value_expressions

Lambda Expression:
list("lambda_expression",
list("x", "y"), → lambda_parameter_symbols
list("return_statement", → lambda_body
list("binary_operator_combination", ...))
is_return_statement — return_expression

Constant Declaration
list("constant_declaration", → is_constant_declaration
list("name", "myFunc"), → declaration_symbol
list("lambda_expression", ...)) → declaration_value expression

Simple Function:
list("simple_function",
list("x", "y", "z"), → function_parameters
list("literal", ...), → function_body
list(frame, env...)) → function_environment

Environment: pair(frame, environment)
first_frame — enclosing_environment

Frame: pair(names, values)
Names: list("x", "y", ...)
Values: list(5, 6, ...)
frame_symbols = head   frame_values = tail
the_empty_environment = null
extend_environment(ns, vs, e)
- Return pair(make_frame(ns, vs), e)

Instructions

Pop Instr: list("pop_instruction") — is_pop_instr
Bin Op Instr: list("binop", "+")
is_bin_op_instr — op_instr_symbol
Unary Op Instr: list("unop", "!")
is_unary_operator_instruction — operator_instruction_symbol
Branch Instr:
list("branch", → is_branch_instr
list(...), → branch_instr_consequent
list(...)) → branch_instr_alternative

Assignment Instruction:
list("asgn", "x")
↙ assign_instruction_symbol
is_assign_instr
Env Instruction: list("env", → is_env_instruction
list(frame, env...))
env_instruction_environment
Call Instruction: list("call", 2)
is_call_instruction — call_instruction_arity

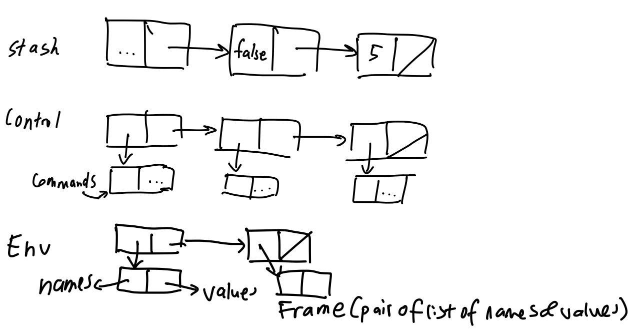
Functions

scan_out_declarations(comp)
- For declarations: returns list(decl_symbol(comp))
- For sequences: accumulate over each statement and scans its declaration
- Not decl or sequence: returns null

Returns: list("x")
list("x", "y", ...)
or null

assign_symbol_value(symbol, val, env)
- If symbols(names) in current env is null, search in enclosing env
- If symbol matches current head of symbols, set head of values in current env to val
- Else scan tail of symbols & values until you hit one of the other 2 base cases

lookup_symbol_value(symbol, env)
- Similar to assign_symbol_value, but returns the value in the frame instead of setting it



Literal → put on top of stash

Binary op combination → put first operand, second operand, symbol on top of control (first operand at the top)

Unary op combination → same as bin op combi but only one operand

Sequence → if no statements, put undefined on top of control
if only one statement, put it on top of control
if >1 statement, put it on top of control, put a pop instr after it, then put the remaining sequence on control

Conditional → put the predicate command on top of control, followed by a branch instruction to either consequent or alternative

Block → scan out declarations in the body
all of them have a value of *unassigned* at the start
put block body on top of control, followed by env instr to return to current env
extend current environment with the new names & values (if there are no declarations then names & values is null)

Fn declaration → Change to const decl, put it on top of control
(Assigns a name to a lambda expr with the params & body)

Constant decl → Put assignment from symbol to value on top of command

Name → Search for a symbol in the current env (look upwards if not found in current env), place it on the stash

Assignment → Put the value expr on top of control, followed by an asgn instruction with the symbol to be assigned to

Lambda expr → Make a simple function, then put it on top of the stash.

Application → Get the list of arguments, put the fn expression on top of control, followed by the args, followed by a call instr. (Fn expr is a name command)

Pop instr → Pop head of stash

Binary op instr → Get the two operands from stash, apply the binary op, place resulting value on stash
(same for unop)

Asgn instr → Keep trying to find matching name/value pair in current env. If found, set the corresponding value to val. If not found, look up in env. If still not found, error

Env instr → Set environment to specified env

Call instr → get the arity first. then take that many values from the stash into an args list.
If fn is a primitive fn, apply it directly.
If fn is a simple fn, put the fn body on top of control, make an env instr to return to current env, then extend current env to include the function parameters as names, the arguments as values.