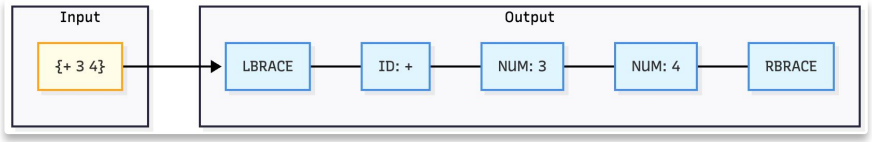
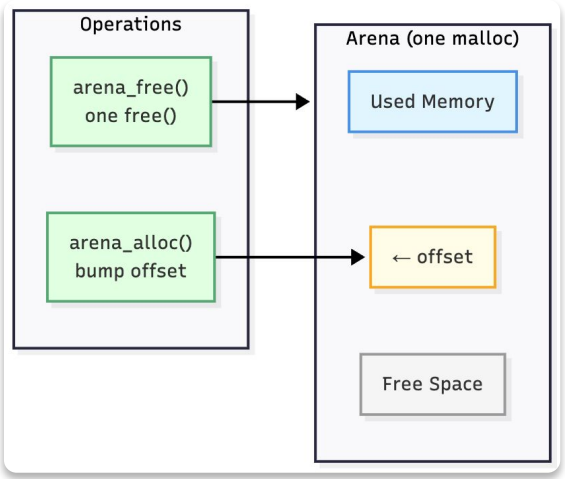
SHEQ4: A C Interpreter

Kevin Rutledge
James Hazelwood
Weston Patrick
Bryce Uota

Programming Languages Project

Tokenizing Input





- Racket handles memory automatically, C requires explicit allocation
- Arena allocator allows one malloc at start, bump pointer for each allocation, single free at end
- Scanner walks input string character by character, producing token array

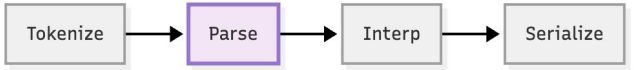
Tokenize

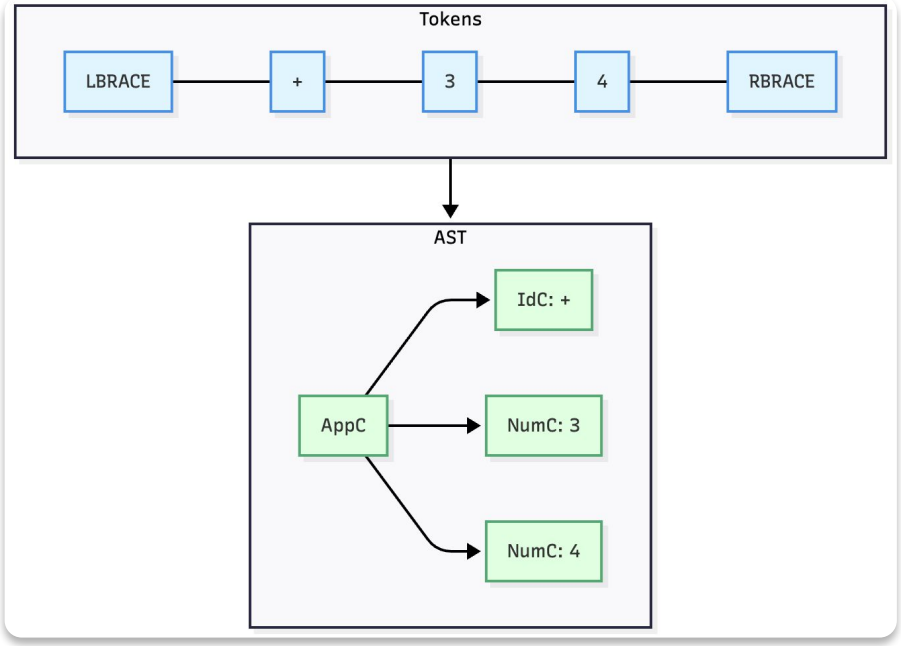

```
// Single-character token handling
else if (input[pos] == '{') { tok.type = TOK_LBRACE; tok.text = "{"; pos++; col++; }
else if (input[pos] == '}') { tok.type = TOK_RBRACE; tok.text = "}"; pos++; col++; }
else if (input[pos] == '(') { tok.type = TOK_LPAREN; tok.text = "("; pos++; col++; }
else if (input[pos] == ')') { tok.type = TOK_RPAREN; tok.text = ")"; pos++; col++; }

// Keyword vs identifier
if (strcmp(tok.text, "if") == 0) tok.type = TOK_IF;
else if (strcmp(tok.text, "lambda") == 0) tok.type = TOK_LAMBDA;
else if (strcmp(tok.text, "let") == 0) tok.type = TOK_LET;
else tok.type = TOK_ID;
```

- Single characters like braces and parens map directly to token types
- Identifiers get compared against reserved words (if, lambda, let)
- Anything not a keyword becomes TOK_ID

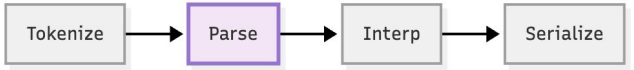
Parsing Tokens





- Recursive descent parser mirrors the grammar structure from Racket's SHEQ4
- Each token type triggers a different parsing path
- Output is a tree of ASTNode structs matching Racket's ExprC variants

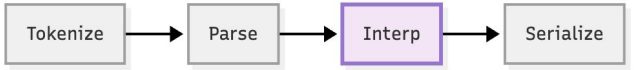
Parse

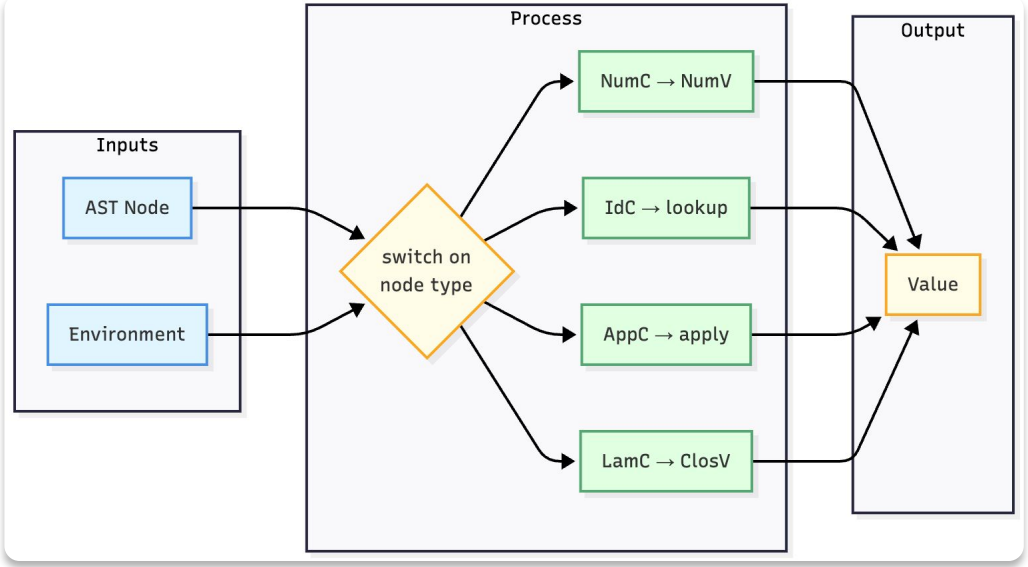



```c
// Token router
ASTNode *parse_expr(Parser *parser) {
    Token tok = peek(parser);
    switch (tok.type) {
        case TOK_LBRACE:
            return parse_braced(parser);
        case TOK_NUMBER:
            advance(parser);
            return make_num(parser->arena, strtod(tok.text, NULL));
        case TOK_ID:
            advance(parser);
            return make_id(parser->arena, tok.text);
    }
}
```

- peek() looks at current token without consuming it
- Switch dispatches to parse_braced() for compound expressions
- Literals and identifiers get wrapped in AST nodes directly

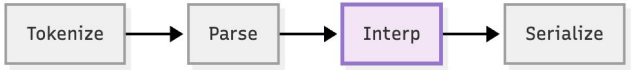
Interpreting the AST





- Tree-walking interpreter recursively evaluates each node type
- Environment is a linked list of bindings with parent pointer for scope
- Closures capture their defining environment, enabling lexical scoping

Interp



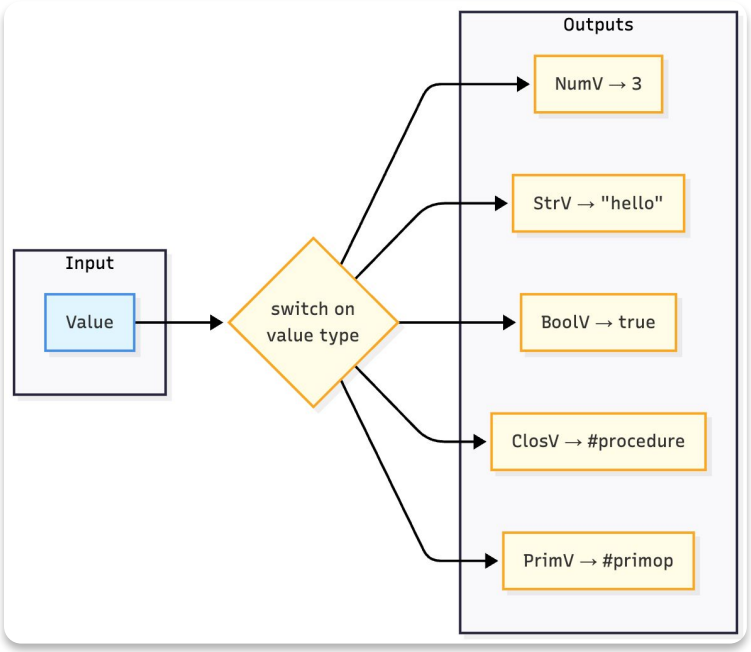

```c
// Evaluation switch
switch (node->type) {
    case NODE_NUMC:
        out->type = VAL_NUMV;
        out->as.num = node->as.num_val;
        return out;

    case NODE_IDC:
        Value *val = lookup(env, node->as.var);
        return val;

    case NODE_LAMC:
        out->type = VAL_CLOSV;
        out->as.clos.params = node->as.lam_node.params;
        out->as.clos.body = node->as.lam_node.body;
        out->as.clos.env = env;  // captures current environment
        return out;
}
```

- NumC wraps the literal value in a NumV struct
- IdC walks the environment chain looking for the binding
- LamC stores params, body, and current env together as a closure

Serializing Output



- Converts runtime Value back to printable string representation
- Static 4KB buffer avoids extra allocation for output formatting
- Each value type has its own string format (numbers, strings, booleans, procedures)

Serialize


```c
// Value to string conversion
char *serialize(Value *val) {
    static char buf[4096];
    switch (val->type) {
        case VAL_NUMV:
            snprintf(buf, sizeof(buf), "%.15g", val->as.num);
            break;
        case VAL_BOOLV:
            snprintf(buf, sizeof(buf), "%s",
                     val->as.boolval ? "true" : "false");
            break;
        case VAL_CLOSV:
            snprintf(buf, sizeof(buf), "#<procedure>");
            break;
        case VAL_PRIMV:
            snprintf(buf, sizeof(buf), "#<primop>");
            break;
    }
    return strdup(buf);
}
```

- snprintf formats into the static buffer based on value type
- Numbers use %.15g to preserve precision without trailing zeros
- Closures and primops print opaque representations since they can't be serialized

Running SHEQ4


```
$ ./sheq4 '{* {+ 1 2} {- 6 4}}'
6
$ ./sheq4 '{if {<= 1 2} "yes" "no"}'
"yes"
$ ./sheq4 '{{lambda (x y) : {* x y}} 3 4}'
12
$ ./sheq4 '{{lambda (x) : {{lambda (y) : {+ x y}} 1}} 2}'
3
$ ./sheq4 '{{let {[x = 7]} in {lambda (y) : {+ x y}} end} 4}'
11
$ ./sheq4 '{/ 3 0}'
SHEQ: division by zero
$ ./sheq4 '{+ "hello" 2}'
SHEQ: + expects number, got string
$ ./sheq4 '{{lambda (x) : x} 1 2}'
SHEQ: arity mismatch: want 1, got 2
$ ./sheq4 '{{lambda (x) : {+ x 1}'
SHEQ: expected '}' at line 1 col 23
SHEQ: unexpected token at line 1 col 23
$ ./sheq4 '{{lambda (x : {+ x 1}}'
SHEQ: expected param name at line 1 col 13
```

- **Lambda application extends the closure's captured environment**
- **Type errors and arity mismatches produce descriptive messages**

Questions?

Kevin Rutledge
James Hazelwood
Weston Patrick
Bryce Uota