STRUCTURE OF PROGRAMMING LANGUAGES - FINAL EXAM STUDY GUIDE

TOPIC 1: INTEGERS AND BASIC EXPRESSIONS

Core Concepts:

- Basic arithmetic expressions: addition (+), subtraction (-), multiplication (*), division (/)
- Operator precedence (multiplication/division before addition/subtraction)
- Parentheses for grouping expressions
- Tokenization: breaking source code into tokens (numbers, operators, parentheses)
- Parsing: converting tokens into Abstract Syntax Trees (AST)
- Evaluation: traversing AST to compute results

Grammar (EBNF):

```
factor = <number> | "(" expression ")"
term = factor { "*"|"/" factor }
expression = term { "+"|"-" term }
program = expression
```

Key Implementation Details:

- Parser uses recursive descent parsing
- AST represented as nested dictionaries with 'tag' and 'value' fields
- Evaluator recursively processes AST nodes

TOPIC 2: PROGRAMS (MULTIPLE STATEMENTS)

Core Concepts:

- Programs as sequences of statements separated by semicolons
- Print statements for output
- Multiple expressions evaluated in sequence
- Statement terminators and separators

Grammar Extension:

```
statement = <print> expression | expression
program = statement { ";" statement }
```

Key Implementation Details:

- Parse multiple statements separated by semicolons
- Execute statements sequentially
- Program returns value of last expression
- Handle optional trailing semicolons

TOPIC 3: ENVIRONMENTS AND VARIABLES

Core Concepts:

- Variable assignment and storage
- Identifiers as variable names
- Environment: mapping from variable names to values
- Variable lookup and retrieval
- Scope basics

Grammar Extension:

```
factor = <number> | <identifier> | "(" expression ")"
```

Key Implementation Details:

- Environment implemented as dictionary
- Assignment statement: identifier = expression
- Variable lookup in environment during evaluation
- Undefined variables raise errors
- Variables can be reassigned

TOPIC 4: OPERATORS AND ASSIGNMENTS

Core Concepts:

- Relational operators: ==, !=, <, >, <=, >=
- Logical operators: && (and), || (or), ! (not)
- Unary operators: negation (-), logical not (!)
- Boolean values: true, false
- Operator precedence hierarchy
- Short-circuit evaluation for logical operators

Grammar Extension:

```
factor = <number> | <identifier> | "(" expression ")" | "!" factor | "-"
factor
arithmetic_expression = term { "+"|"-" term }
relational_expression = arithmetic_expression { ("<" | ">" | "<=" | ">=" |
"==" | "!=") arithmetic_expression }
```

```
logical_factor = relational_expression
logical_term = logical_factor { "&&" logical_factor }
logical_expression = logical_term { "||" logical_term }
expression = logical_expression
```

Key Implementation Details:

- Multiple levels of precedence in parser
- Boolean evaluation returns true/false
- Comparison operators work on numbers
- Logical operators work on boolean values
- Short-circuit: && stops at first false, || stops at first true

---

TOPIC 5: CONTROL STRUCTURES

Core Concepts:

- If statements with optional else clause
- While loops for iteration
- Statement blocks with curly braces
- Break and continue statements
- Control flow and branching

Grammar Extension:

```
statement_block = "{" statement { ";" statement } "}"
if_statement = "if" "(" expression ")" statement_block [ "else"
statement_block ]
while_statement = "while" "(" expression ")" statement_block
```

Key Constructs:

```
if (condition) { statements }
if (condition) { statements } else { statements }
while (condition) { statements }
break    - exit current loop
continue - skip to next iteration
```

Key Implementation Details:

- Conditional expression must evaluate to boolean
- Statement blocks create sequences of statements
- Nested control structures supported
- Break/continue affect innermost loop only

TOPIC 6: GRAMMAR VERIFICATION AND STRINGS

Core Concepts:

- String literals and string operations
- String concatenation with + operator
- String comparison
- Print statements with strings
- Grammar correctness and validation
- Parser error handling

String Operations:

- String concatenation: "Hello" + "World" → "HelloWorld"
- String repetition: "dog" * 3 → "dogdogdog"
- String comparison: "abc" == "abc" → true

Key Implementation Details:

- String tokens recognized by tokenizer
- String values in AST
- Type checking during evaluation
- Grammar rules must be unambiguous

TOPIC 7: RETURNING STATUS (BREAK, CONTINUE, ASSERT)

Core Concepts:

- Break statement to exit loops early
- Continue statement to skip to next iteration
- Assert statement for testing and validation
- Return status from statement execution
- Control flow interruption

Key Constructs:

```
break                           - exit loop
continue                        - skip to next iteration
assert condition                - verify condition is true
assert condition, "message"     - assertion with error message
```

Key Implementation Details:

- Break/continue require special return values from evaluator
- Assert raises error if condition false
- Status propagation through nested statements
- Control flow managed through return values or exceptions

WHY RETURN STATUS WITH VALUES?

The Problem:

- Normal expression evaluation returns a value
- Control flow statements (break, continue) need to interrupt normal flow
- How do we signal "break out of loop" while in nested expression evaluation?
- Example: `while (true) { x = x + 1; if (x > 5) { break } }`
  - The 'break' is deep inside: while → block → if → block → break
  - Must propagate "break signal" back up through all levels

The Solution: Status + Value Return Pattern

- Evaluator returns BOTH a status and a value: (status, value)
- Status types:
  - "normal" - continue normal execution
  - "break" - exit current loop
  - "continue" - skip to next loop iteration
  - "return" - return from function (if functions supported)

Example Return Values:

```
2 + 3                  → ("normal", 5)
break                  → ("break", None)
continue               → ("continue", None)
x = 5                  → ("normal", 5)
```

Status Propagation Rules:

1. Expressions: evaluate and propagate any exceptional status upward
2. Statement sequences: stop at first exceptional status
3. Loops: clear "break" and "continue" at the loop level
4. Blocks: pass through all statuses unchanged
5. If statements: pass through status from executed branch

Clearing Exceptional Status:

CRITICAL PRINCIPLE: Clear status at the level capable of handling it

Example - While Loop (handles break and continue):

```
def evaluate_while(ast, env):
    while evaluate(condition, env)[1]:  # check condition value
        status, value = evaluate(body, env)

        if status == "break":
            return ("normal", None)      # CLEAR break status here
        if status == "continue":
            continue                     # CLEAR continue, start next
iteration
        if status != "normal":
            return (status, value)       # propagate other statuses up

    return ("normal", None)
```

Why This Works:

- 'break' only makes sense inside a loop → loop clears it
- 'continue' only makes sense inside a loop → loop clears it
- Other statuses (like 'return') propagate to outer levels
- Each construct handles only what it understands

Example Trace:

```
while (x < 10) {
    x = x + 1;
    if (x == 5) { break }
}

Evaluation sequence:
1. while evaluates condition: ("normal", true)
2. block evaluates x = x + 1: ("normal", 5)
3. if evaluates condition: ("normal", true)
4. if body evaluates break: ("break", None)
5. if returns: ("break", None)          ← propagates up
6. block returns: ("break", None)       ← propagates up
7. while sees "break": clears to ("normal", None)
8. while exits normally
```

Benefits of Status Pattern:

- Clean separation of control flow and data flow
- Exceptional conditions propagate automatically
- Each construct handles only its own concerns
- No need for exceptions or global state
- Easy to add new control flow constructs
- Explicit and predictable behavior

Alternative Approaches (and why they're worse):

1. Exceptions: heavyweight, non-local control flow, harder to reason about
2. Global flags: shared mutable state, not thread-safe, confusing
3. Special return values: can't distinguish from normal values
4. Continuation passing: complex, hard to understand

The status pattern is elegant, explicit, and maintainable.

TOPIC 8: COMPLEX EXPRESSIONS

Core Concepts:

- Nested expressions and complex ASTs
- Multiple operators in single expression
- Precedence and associativity rules
- Expression composition
- Parenthesized sub-expressions

Key Areas:

- Arithmetic with multiple operations: 2 + 3 * 4 - 5
- Logical expressions: (x > 5) && (y < 10) || (z == 0)
- Mixed type expressions
- Deep nesting with parentheses

Key Implementation Details:

- Parser handles arbitrary nesting depth
- Correct precedence through grammar structure
- AST depth reflects expression complexity
- Left-to-right associativity for same precedence

TOPIC 9: COMPLEX ASSIGNMENTS (LISTS AND OBJECTS)

Core Concepts:

- List data structure: ordered collection of elements
- List literals: [1, 2, 3]
- List indexing: list[index]
- List concatenation: list1 + list2
- Object/dictionary data structure: key-value pairs
- Object literals: {"name": "Alice", "age": 30}
- Object member access: object["key"]
- Nested data structures

List Operations:

- Creation: `x = [1, 2, 3]`
- Access: `x[0]` → 1
- Assignment: `x[1] = 27`
- Concatenation: `[1, 2] + [3, 4]` → [1, 2, 3, 4]
- Nested lists: `[[1, 2], [3, 4]]`

Object Operations:

- Creation: `person = {"name": "Alice", "age": 30}`
- Access: `person["name"]` → "Alice"
- Assignment: `person["age"] = 31`
- Adding fields: `person["city"] = "New York"`
- Nested objects: `obj["nested"]["inner"]`

Key Implementation Details:

- Lists stored as Python lists
- Objects stored as Python dictionaries
- Index out of bounds errors
- Type checking for indexing operations

TOPIC 10: FUNCTIONAL PROGRAMMING

Core Concepts:

- Pure functions and immutability
- Recursion as primary iteration method
- Higher-order functions
- List processing with head/tail pattern
- Divide and conquer algorithms

Key Functions Demonstrated:

- head(list): returns first element
- tail(list): returns all but first element
- Recursive list filtering: less_than, greater_than, equal_to
- Quicksort implementation using recursion

Quicksort Algorithm:

1. If list is empty, return empty list
2. Choose pivot (first element)
3. Partition into: less than pivot, equal to pivot, greater than pivot
4. Recursively sort partitions
5. Concatenate: sorted(less) + equal + sorted(greater)

Key Principles:

- No mutation of data structures
- Functions return new values
- Recursion instead of loops
- Pattern matching on list structure
- Base case and recursive case

TOPIC 11: CLOSURES AND SCOPE

Core Concepts:

- Closures: functions that capture variables from enclosing scope
- Lexical scope: variable binding based on code structure
- Static binding: variable references resolved at compile/parse time
- Dynamic binding: variable references resolved at runtime
- First-class functions: functions as values
- Function factories: functions that return functions

Static vs Dynamic Binding:

STATIC BINDING (Lexical Scope):

- Variable references determined by code structure
- Resolved at parse/compile time
- Most modern languages use static binding
- Example: Python, JavaScript, Java, C++
- Predictable: same name always refers to same variable

DYNAMIC BINDING (Dynamic Scope):

- Variable references determined by call stack
- Resolved at runtime based on execution context
- Less common in modern languages
- Example: early Lisp, some shell scripts
- Unpredictable: same name can refer to different variables

Example Comparison:

```
x = "global"

function foo() {
    return x
}

function bar() {
    x = "local"
    return foo()
}

Static binding: bar() returns "global" (x from foo's definition scope)
Dynamic binding: bar() returns "local" (x from call stack)
```

Closures Explained:

- A closure is a function bundled with its lexical environment
- Captures variables from enclosing scope at creation time
- Retained even after outer function returns

- Enables data hiding and encapsulation

Closure Example:

```
function make_counter() {
    count = 0
    function counter() {
        count = count + 1
        return count
    }
    return counter
}

c1 = make_counter()
c1() → 1
c1() → 2
c1() → 3
```

Key Points:

- 'count' is captured by closure
- Each call to make_counter() creates new closure
- Different counters have independent state
- 'count' persists across counter() calls

Closure with Parameters:

```
function make_multiplier(factor) {
    function multiplier(x) {
        return x * factor
    }
    return multiplier
}

double = make_multiplier(2)
triple = make_multiplier(3)
double(5) → 10
triple(4) → 12
```

Implementation Considerations:

- Captured variables must persist after function returns
- Environment must be saved with function reference
- Garbage collection needed for cleanup

Use Cases for Closures:

- Private state/data hiding

- Function factories and partial application
- Callbacks with state
- Iterator patterns
- Event handlers

TOPIC 12: LOGIC PROGRAMMING (PROLOG)

Core Concepts:

- Facts: assertions about the world
- Rules: logical implications (if-then)
- Queries: asking questions
- Unification: bidirectional pattern matching that binds variables to make terms match
- Backtracking: searching for solutions
- Declarative programming paradigm

Prolog Syntax:

- Facts: `person(greg).`
- Rules: `grandchild(X, Y) :- child(X, Z), child(Z, Y).`
- Queries: `?- grandchild(greg, Who).`
- Variables: uppercase (X, Y, Z)
- Constants: lowercase (greg, susan)

Key Constructs:

- Predicates define relationships
- `:-` means "if" (implication)
- `,` means "and" (conjunction)
- `;` means "or" (disjunction)
- Underscore _ is anonymous variable

Logic Programming Paradigm:

- Define what is true, not how to compute
- Prolog searches for solutions automatically
- Multiple solutions through backtracking
- Recursive rules for transitive relationships

Unification Explained:

- Bidirectional pattern matching (unlike assignment which is one-way)
- Attempts to make two terms identical by finding variable bindings
- Both sides can contain variables
- Example: `[H|T] = [1, 2, 3]` binds H to 1 and T to [2, 3]
- Fails if terms cannot be made to match
- Fundamental operation in Prolog execution

Example - Family Relationships:

```
child(X, Y) :- son(X, Y).
child(X, Y) :- daughter(X, Y).
grandchild(X, Y) :- child(X, Z), child(Z, Y).
```

Example - Quicksort in Prolog:

```
qsort([], []).
qsort([V|Rest], Sorted) :-
  lower(Rest, V, Lower),
  equal([V|Rest], V, Equal),
  upper(Rest, V, Upper),
  qsort(Lower, SortedLower),
  qsort(Upper, SortedUpper),
  append(SortedLower, Equal, SortedUpper, Sorted).
```

COMPREHENSIVE TOPICS SUMMARY

1. LEXICAL ANALYSIS (Tokenization)

- Breaking source code into tokens
- Token types: numbers, identifiers, operators, keywords, punctuation

2. SYNTAX ANALYSIS (Parsing)

- Grammar specification (EBNF/BNF)
- Recursive descent parsing
- Abstract Syntax Trees (AST)
- Parser error handling

3. SEMANTIC ANALYSIS (Type Checking)

- Type compatibility
- Variable scope
- Undefined variable detection

4. RUNTIME EXECUTION (Evaluation)

- AST interpretation
- Environment management
- Control flow execution
- Function call semantics

5. PROGRAMMING PARADIGMS

- Imperative: statements, variables, loops
- Functional: recursion, pure functions, immutability
- Logic: facts, rules, queries, unification

6. DATA STRUCTURES

- Primitive types: numbers, booleans, strings
- Composite types: lists, objects/dictionaries
- Nested structures

7. LANGUAGE FEATURES

- Expressions and operators
- Variables and assignment
- Control structures (if, while, break, continue)
- Functions and recursion
- Closures and lexical scope
- Static vs dynamic binding
- Assertions for testing

KEY IMPLEMENTATION CONCEPTS

Parser Design:

- Each grammar rule becomes a parse function
- Recursive calls for nested structures
- Token consumption and lookahead
- Error reporting with position information

Evaluator Design:

- Pattern matching on AST node tags
- Recursive evaluation of sub-expressions
- Environment passing for variable lookup
- Return value propagation

Status and Value Return Pattern:

- Evaluator returns tuple: (status, value)
- Status types: "normal", "break", "continue", "return"
- Normal execution has status "normal"
- Exceptional control flow uses other statuses
- Each construct clears status it can handle:
    - Loops clear "break" and "continue"
    - Functions clear "return" (if supported)
    - Other constructs propagate status upward
- Prevents need for exceptions or global flags
- Makes control flow explicit and traceable

Environment Model:

- Dictionary mapping names to values
- Nested scopes (local, global)
- Variable shadowing
- Closure capture (if supported)

Static vs Dynamic Scoping:

Static (Lexical) Scoping:

- Variable binding determined by source code structure
- Function sees variables from where it was DEFINED
- Resolved at parse time
- Enables closures
- More predictable and maintainable

Dynamic Scoping:

- Variable binding determined by call stack
- Function sees variables from where it was CALLED
- Resolved at runtime
- No closures possible
- Can be confusing and error-prone

Closure Implementation:

- Store environment reference with function
- Capture variables from enclosing scope
- Retained environment frames
- Reference counting or garbage collection
- Each closure gets its own environment instance

Testing Strategy:

- Unit tests for each component
- Grammar verification tests
- Integration tests for full programs
- Edge cases and error conditions

EXAM PREPARATION TIPS

1. Understand the progression of language features across topics
2. Know the grammar rules and how they translate to parser code
3. Be able to trace AST construction from source code
4. Understand evaluation order and operator precedence
5. Know how environments track variable bindings
6. Understand control flow execution paths

7. Compare imperative, functional, and logic programming paradigms
8. Practice writing and tracing recursive functions
9. Understand closures and how they capture variables
10. Know the difference between static and dynamic binding
11. Be able to trace variable resolution in nested scopes
12. Understand the status + value return pattern for control flow
13. Know where exceptional statuses get cleared (loops clear break/continue)
14. Be able to trace status propagation through nested structures
15. Understand Prolog unification and backtracking
16. Review error handling and edge cases