CS351 Midterm Exam Study Guide

Exam Format

- **Date**: Monday, Oct 27
- **Format**: In-class Lockdown Browser exam
- **Topics**: Slides 0-3 (Through V6 - Define)
- **Types of Questions**: Multiple choice concepts, grammar writing, expression evaluation, coding semantics, environment diagrams

Key Concepts to Master

1. Lexical, Syntactic, and Semantic Analysis

Definitions (MEMORIZE THESE!)

- **Lexical Analysis**: The process of converting a stream of characters to a stream of tokens
  - Input: String of characters
  - Output: Stream of tokens
  - Tool: Scanner/Lexer
- **Syntactic Analysis**: The process of analyzing a stream of tokens to ensure they adhere to a set of grammar rules
  - Input: Stream of tokens
  - Output: Parse tree
  - Tool: Parser
- **Semantic Analysis**: The process of checking that a sentence in a language is meaningful and determining that meaning
  - Input: Parse tree
  - Output: Result/behavior of the program
  - Tool: Interpreter/Evaluator

Processing Pipeline

```
Source Code → [Lexical Analysis] → Tokens → [Syntactic Analysis] → Parse Tree →
[Semantic Analysis] → Result
```

2. PLCC Grammar Rules and Java Class Generation

Grammar Rule Types

1. **Simple Rule**: `<name> ::= ...`

    - Generates: `Name` class with fields for each component

2. **Named Rule**: `<name>:ClassName ::= ...`

    - Generates: `ClassName` class with specified fields

3. **Repeating Rule**: `<name> **= ITEM`

    - Generates: `Name` class containing list attributes (if captured)

Class Generation Examples

Example 1: Named rule with captured fields

```
<stmt>:Assignment ::= LET <var>v <EQUALS> <expr>e1 SEMICOLON
```

Generates: `Assignment(Var v, Token equals, Expr e1)`

- LET is a token (not captured as field)
- *v becomes field v of type Var*
- becomes field equals of type Token
- e1 becomes field e1 of type Expr
- SEMICOLON is a token (not captured as field)

Example 2: Kleene star rule

```
<params> **= <ident> +COMMA
```

Generates: `Params(List<Ident> identList)`

Field Naming Rules

- **Tokens** (all caps like LET, SEMICOLON, COMMA) are NOT included as fields unless explicitly captured
- **Non-terminals with variable names** become fields (e.g., `<var>v` → field `v` of type `Var`, `<expr>e1` → field `e1` of type `Expr`)
- **Non-terminals without names** use their type name lowercased (e.g., `<ident>` → field `ident`, `<EQUALS>` → field `equals`)
- **Repeating Rule** (**=) creates a class with List instance variables
- **Separators in repeating rules** (like `+COMMA`) indicate tokens that separate list items but are not captured

3. Regular Expressions in PLCC

Common Regex Patterns (KNOW THESE!)

- `\s+` - One or more whitespace characters
- `\s*` - Zero or more whitespace characters

- `\d` - Single digit
- `\d\d` - Exactly two digits
- `\w` - Word character (letter, digit, underscore)
- `[abc]` - Character class (a OR b OR c)
- `a+` - One or more 'a'
- `a*` - Zero or more 'a'

Lexical Specification Example

```
TWOD '\d\d'        # Matches exactly two digits
COLON ':'          # Matches colon character
skip WHITESPACE '\s+'  # Skip one or more whitespace
```

4. Expression Evaluation

Basic Operations

- `+(3, 4)` evaluates to `7`
- `add1(3)` evaluates to `4`
- `sub1(5)` evaluates to `4`
- `*(2, 3)` evaluates to `6`
- `-(5, 2)` evaluates to `3`
- `zero?(0)` evaluates to `1`
- `zero?(5)` evaluates to `0`

If Expressions

- In PLCC languages, non-zero numbers are true, 0 is false
- `if 3 then 4 else 5` evaluates to `4` (3 is true)
- `if 0 then 4 else 5` evaluates to `5` (0 is false)
- `if add1(0) then 42 else 24` evaluates to `42` (1 is true)
- `if sub1(1) then 1 else 0` evaluates to `0` (0 is false)

5. Let Expressions and Environments

Basic Let Expression

```
let
  x = 3
in
  x
```

Evaluates to: `3`

Multiple Bindings

```
let
  x = 3
  y = 5
in
  *(x, y)
```

Evaluates to: `15`

Nested Let Expressions

```
let
  x = 3
in
  let
    y = 5
  in
    -(y, x)
```

Evaluates to: `2`

Variable Shadowing

```
let
  x = 3
in
  let
    x = 5    # This shadows the outer x
    y = x    # y gets 5 (the inner x)
  in
    +(x, y)  # 5 + 5
```

Evaluates to: `10`

6. Free Variables in Procedures

Key Concept

A variable is **free** in a procedure if it's used but not:

1. A formal parameter of that procedure
2. Bound in a let/letrec within that procedure

Example Analysis

```
let
  f = proc(x) {
    if zero?(x) then 0
    else *(2, .f(-(x, y)))
  }
in
  let
    y = 2
  in
    .f(y)
```

Free variables in the proc:

- `y` is free (used but not a parameter)
- `f` is free (used but not a parameter in let)
- `x` is NOT free (it's a formal parameter)

Let vs Letrec Difference

- **let**: The binding is NOT available in its own definition
  - `f` is free in `proc(x) { ... .f(...) }` when using let
- **letrec**: The binding IS available in its own definition
  - `f` is NOT free in `proc(x) { ... .f(...) }` when using letrec

7. Environment Diagrams

Drawing Rules

1. Each let/letrec creates a new environment frame
2. Draw arrows from new frame to parent environment
3. Procedures capture the environment where they're defined
4. Show bindings as `var | value` in each frame
5. When applying a procedure, extend its captured environment

Example Environment Diagram

```
let x = 5           # Creates Env1: [x→5] → empty
in
  let z = x         # Creates Env2: [z→5] → Env1
  in
    let z = 4       # Creates Env3: [z→4, f→proc] → Env2
    f = proc(y) +(z, +(x, y))
    in
      .f(z)         # Looks up f in Env3, creates Env4: [y→4] → Env2 for applying
body of f with z=5
```

8. Procedures (V4)

Syntax

- Definition: `proc(x, y) +(x, y)`
- Application: `.foo(3, 5)`
- Anonymous: procedures don't have names unless bound

Important Notes

- Procedures evaluate to ProcVal
- Must use dot (.) for application
- Procedures capture their defining environment (closure)
- Applying a procedure extends the captured environment

9. SeqExp - Sequential Expressions (V4)

Syntax

```
{exp1; exp2; exp3}
```

- Evaluates each expression in order
- Returns the value of the LAST expression
- Example: `{3; 4; 5}` evaluates to `5`

10. Letrec - Recursive Bindings (V5)

Key Difference from Let

- **letrec** allows recursive definitions
- The binding is available in its own definition

Example

```
letrec
  fact = proc(n)
    if zero?(n) then 1
    else *(n, .fact(sub1(n)))
in
  .fact(5)
```

Evaluates to: `120`

11. Define Expressions (V6)

Syntax

```
define x = 3
define f = proc(y) +(x, y)
.f(5)
```

Key Points

- Creates global bindings
- Each define modifies the initial (outer-most) environment
- Later defines can overwrite earlier ones

Practice Problems

Problem Set 1: Definitions

1. What process converts "3 + 4" into tokens [NUM(3), PLUS, NUM(4)]?
2. What process checks if tokens follow grammar rules?
3. What process evaluates a parse tree to get a result?

Problem Set 2: PLCC Classes

Given these grammar rules, what Java classes are generated?

1. `<expr>:Add ::= PLUS <left> <right>`

2. `<items> **= ITEM +COMMA`

3. `<data>:Record ::= ID COLON <value>val SEMI`

Problem Set 3: Expression Evaluation

Evaluate these expressions:

1. `+(if 0 then 3 else 5, 7)`
2. `let x = 4 in let y = let z = 2 in +(x, z) in *(x, y)`
3. `if sub1(1) then 100 else 200`

Problem Set 4: Free Variables

Identify free variables in these procedures:

1. `proc(x) +(x, y)`
2. `proc(x, y) +(x, *(y, z))`
3. `letrec f = proc(x) if zero?(x) then 0 else .f(sub1(x)) in .f(5)`
4. `let f = proc(x) .f(x) in .f(5)`

Practice Problem Solutions

Problem Set 1 Solutions: Definitions

1. **Lexical Analysis** - converts characters to tokens
2. **Syntactic Analysis** - checks if tokens follow grammar rules
3. **Semantic Analysis** - evaluates parse tree to get a result

Problem Set 2 Solutions: PLCC Classes

1. `Add(Left left, Right right)`

2. `Items(List<Token> itemList)`

3. `Record(Value val)`

Problem Set 3 Solutions: Expression Evaluation

1. `+(if 0 then 3 else 5, 7)` → **12** (0 is false, so 5 + 7)
2. `let x = 4 in let y = let z = 2 in +(x, z) in *(x, y)` → **24** (y = 6, then 4 × 6)
3. `if sub1(1) then 100 else 200` → **200** (sub1(1) = 0 which is false)

Problem Set 4 Solutions: Free Variables

1. `proc(x) +(x, y)` → Free: **y**
2. `proc(x, y) +(x, *(y, z))` → Free: **z**
3. `letrec f = proc(x) if zero?(x) then 0 else .f(sub1(x)) in .f(5)` → Free: **none** (f bound by letrec)
4. `let f = proc(x) .f(x) in .f(5)` → Free: **f** (let doesn't allow self-reference)

Course content developed by Declan Gray-Mullen for WNEU with Claude