CSS125

notes and stuff

Petcham

Table of contents

Racket Functional Programming

Data Types

```
42, -5          ; integer
3.14            ; float
1/3             ; rational
#t, #f          ; boolean
"text"          ; string
'symbol         ; symbol
'(1 2 3)        ; list
#(1 2 3)        ; vector
```

Variables

- **Types**: Implicit
- **Typing**: Dynamic

```
(define x 10)                  ; global binding
(let ([a 5] [b 10]) (+ a b))   ; local binding
```

Control Structures

```
; Branching
(if (> x 0) "pos" "neg")
(cond [(> x 0) "pos"] [(< x 0) "neg"] [else "zero"])

; Iteration
(define (countdown n)            ; recursion
  (if (= n 0) 'done (countdown (- n 1))))
(for ([i (range 1 6)]) (display i))     ; loop
```

Functions

```
(define (name param1 param2) body)       ; definition
(lambda (x) (* x x))                     ; anonymous
```

Exercises

1. Hello World

```
(display "Hello, World!")
```

2. Read/Echo Integer

```
(display "Enter an integer: ")
(define input (read))
(printf "You entered ~a.~n" input)
```

3. Prime Checker

```
(define (is-prime? n)
  (cond [(<= n 1) #f] [(= n 2) #t] [(even? n) #f]
        [else (prime-helper n 3)]))

(define (prime-helper n div)
  (cond [(> (* div div) n) #t] [(= (remainder n div) 0) #f]
        [else (prime-helper n (+ div 2))]))

(display "Enter integer: ")
(define num (read))
(printf "~a is ~a prime.~n" num (if (is-prime? num) "" "not"))
```

4. Math Expression: $x^2 + (\log_{10}(x) - \sin(x))/\sqrt{x}$

```
(define (calc x)
  (+ (expt x 2) (/ (- (log x 10) (sin x)) (sqrt x))))

(display "Enter float x: ")
(define x (read))
(if (<= x 0)
    (display "Error: x must be positive")
    (printf "Result: ~a~n" (calc x)))
```

Lexical Analysis

Learning Objectives

- Understand the role of the lexical analyzer in compilation
- Learn about tokens, patterns, and lexemes
- Explore lexical error handling
- Master regular expressions for token specification

Guide Questions

1. **What is the role of the lexical analyzer?**
2. **Why should parsing be separated from lexical analysis?**
3. **What are tokens, patterns, and lexemes and what are the differences between these three?**
4. **What are lexical errors and how are they handled?**
5. **How is a lexical definition of a language specified?**

The Role of the Lexical Analyzer

The lexical analyzer serves as the **first phase of the compiler** and performs several crucial functions:

- **Converts raw text into tokens** - Transforms the source code from a stream of characters into meaningful units
- **Strips out whitespace and comments** - Removes unnecessary formatting and documentation that doesn't affect program logic
- **Correlates error messages** - Links compiler errors back to specific locations in the source code
- **Tracks position information** - Maintains line numbers and column numbers for debugging purposes

Example Flow

```
Source Code: "int x = 42; // this is a variable"
↓
Lexical Analyzer
↓
Tokens: [INT_KEYWORD, IDENTIFIER(x), ASSIGN_OP, NUMBER(42), SEMICOLON]
```

Issues in Lexical Analysis

Why Separate Parsing from Tokenization?

1. Simplicity in Design

• Each phase has a single, well-defined responsibility

• Lexical analyzer focuses only on character-level recognition

• Parser focuses only on grammatical structure

2. Improvement in Efficiency

• Specialized algorithms for each phase

• Better performance through focused optimization

• Reduced complexity in each component

3. Compiler Portability Enhanced

• Easy to adapt lexical analyzer for different character sets

• Grammar rules remain unchanged across platforms

• Modular design allows independent updates

Tokens, Patterns, and Lexemes

Definitions

- **Token** - A category or label for lexical units (like a part of speech in grammar)
- **Pattern** - The rule or regular expression that defines how to recognize a token
- **Lexeme** - The actual text from source code that matches a token's pattern

Example Breakdown

Component	Value	Description
Token	`identifier`	Category name
Pattern	`/[A-Za-z_][A-Za-z0-9_]*`	Rule for recognition
Lexeme	`"nCount"`	Actual text found

Fortran Statement Example

For the statement: `E = M * C ** 2`

Lexeme	Token	Attributes
`E`	`<id, pointer to symbol-table entry for E>`	Variable identifier
`=`	`<assign_op>`	Assignment operator
`M`	`<id, pointer to symbol-table entry for M>`	Variable identifier
`*`	`<mult_op>`	Multiplication operator
`C`	`<id, pointer to symbol-table entry for C>`	Variable identifier
`**`	`<exp_op>`	Exponentiation operator
`2`	`<number, integer value 2>`	Numeric literal

Important Notes

- **Tokens serve as terminal symbols** in a language's grammar
- **Reserved strings** are lexemes with specific meaning (keywords like `if`, `while`)
- **Attributes** provide additional information when multiple lexemes match the same token pattern

---

Lexical Errors

What Are Lexical Errors?

- Occur when **no pattern matches** the current sequence of characters
- Few errors are detectable at lexical level alone
- Example: `fi(a == f (x))` - `fi` might be a typo for `if`

Error Recovery Strategies

1. Panic Mode

- Delete characters until a well-formed token can be created
- Simple but may skip over multiple errors

2. Deletion

- Remove one character from the problematic lexeme
- Try to match again with shortened string

3. Insertion

- Add a character to the current lexeme
- Attempt to create valid token

4. Replacement

- Replace one character in the current lexeme
- Most common for typos

5. Transposition

- Swap adjacent characters
- Handles common typing mistakes

Specifications of Tokens

Tokens are specified using **regular expressions** - mathematical formulas that describe text patterns.

Basic Concepts

Alphabet (Σ)

- Finite set of symbols
- Examples: ASCII (128 characters), Extended ASCII (256 characters)

String

- Finite sequence of symbols from an alphabet
- Each symbol must be from the defined alphabet Σ

String Length

- Number of symbols in string S
- Denoted by |S|
- Empty string (ε) has length 0

Language

- Set of strings defined over an alphabet
- If language L is over alphabet Σ, then $L \subseteq \Sigma^*$
- Σ^* = set of all possible strings from symbols in Σ

String Set Operations

Union ($L_1 \cup L_2$)

- **Definition**: Set of strings in either L_1 or L_2 (or both)
- **Example**: If L_1 = {cat, dog} and L_2 = {bird, cat}, then $L_1 \cup L_2$ = {cat, dog, bird}

Concatenation (L_1L_2)

- **Definition**: Set of strings formed by concatenating each string in L_1 with each string in L_2
- **Example**: If L_1 = {a, b} and L_2 = {c, d}, then L_1L_2 = {ac, ad, bc, bd}

Kleene Closure (L^*)

- **Definition**: Set containing ε and all strings formed by concatenating zero or more strings from L
- **Example**: If L = {a, b}, then L^* = {ε, a, b, aa, ab, ba, bb, aaa, aab, ...}

Positive Closure (L^+)

- **Definition**: Set of all strings formed by concatenating one or more strings from L
- **Example**: If L = {a, b}, then L^+ = {a, b, aa, ab, ba, bb, aaa, aab, ...}
- **Note**: L^+ = LL *(L concatenated with L)*

Regular Expressions

Regular expressions are patterns that match strings in a language. They use special symbols to describe text patterns.

Basic Symbols

Symbol	Meaning	Example
`a`	Literal character 'a'	Matches exactly 'a'
`ε`	Empty string	Matches nothing (zero length)
`∅`	Empty set	Matches no strings

Operators

<table>
<tr><th>Operator</th><th>Name</th><th>Description</th><th>Example</th></tr>
<tr><td><code>|</code></td><td>Union/OR</td><td>Either pattern</td><td><code>a|b</code> matches 'a' or 'b'</td></tr>
<tr><td><code>·</code> or concatenation</td><td>Concatenation</td><td>One after another</td><td><code>ab</code> matches 'a' followed by 'b'</td></tr>
<tr><td><code>*</code></td><td>Kleene Star</td><td>Zero or more</td><td><code>a*</code> matches '', 'a', 'aa', 'aaa'...</td></tr>
<tr><td><code>+</code></td><td>Plus</td><td>One or more</td><td><code>a+</code> matches 'a', 'aa', 'aaa'...</td></tr>
<tr><td><code>?</code></td><td>Question</td><td>Zero or one</td><td><code>a?</code> matches '' or 'a'</td></tr>
<tr><td><code>[]</code></td><td>Character class</td><td>Any character inside</td><td><code>[abc]</code> matches 'a', 'b', or 'c'</td></tr>
<tr><td><code>[^]</code></td><td>Negated class</td><td>Any character NOT inside</td><td><code>[^abc]</code> matches any char except 'a', 'b', 'c'</td></tr>
</table>

Common Patterns

Pattern	Regular Expression	Description
Identifier	`[a-zA-Z_][a-zA-Z0-9_]*`	Letter/underscore followed by letters/digits/underscores
Integer	`[0-9]+`	One or more digits
Float	`[0-9]+\.[0-9]+`	Digits, dot, digits
Whitespace	`[ \t\n\r]+`	One or more space characters

Examples in Action

```
Pattern: [0-9]+
Matches: "42", "123", "7"
Doesn't match: "abc", "12.5"

Pattern: [a-zA-Z][a-zA-Z0-9]*
Matches: "x", "variable1", "myVar"
Doesn't match: "123abc", "_private"
```

Operator Precedence

When building regular expressions, operators have different precedence levels (like mathematical operations).

Precedence Order (Highest to Lowest)

1. **Parentheses** `()` - Highest precedence
2. Used for grouping
3. Example: `(a|b)*` vs `a|b*`
4. **Closure operators** `*`, `+`, `?`
5. Apply to immediately preceding element
6. Example: `ab*` means `a` followed by zero or more `b`s
7. **Concatenation (implicit)**
8. Items next to each other are concatenated
9. Example: `abc` means `a` then `b` then `c`
10. **Union/OR** `|` - Lowest precedence
11. Separates alternative patterns
12. Example: `a|bc` means `a` OR `bc`

Examples Showing Precedence

Expression	Interpretation	Matches
`a|b*`	`a` OR (zero or more `b` s)	"a", "", "b", "bb", "bbb"...
`(a|b)*`	Zero or more of (`a` OR `b`)	"", "a", "b", "ab", "ba", "abb"...
`ab|c`	(`ab`) OR `c`	"ab", "c"
`a(b|c)`	`a` followed by (`b` OR `c`)	"ab", "ac"

Best Practice

- **Use parentheses liberally** to make intentions clear
- **When in doubt, add parentheses** to avoid confusion
- Example: `(a|b)*(c|d)+` is clearer than `a|b*c|d+`

Quick Reference

Token Recognition Process

1. **Read characters** from source code
2. **Apply patterns** using regular expressions
3. **Match longest possible** lexeme
4. **Generate token** with attributes
5. **Handle errors** if no pattern matches
6. **Repeat** until end of input

Common Token Types

- **Keywords**: `if`, `while`, `class`, `public`
- **Identifiers**: Variable and function names

- **Literals**: Numbers, strings, booleans
- **Operators**: `+`, `-`, `*`, `/`, `==`, `!=`
- **Delimiters**: `(`, `)`, `{`, `}`, `;`, `,`
- **Whitespace**: Spaces, tabs, newlines (usually ignored)

Guide Questions - Complete Answers

1. What is the role of the lexical analyzer?

The lexical analyzer serves as the **first phase of the compiler** with four main roles:

- **Tokenization**: Converts raw source code text into meaningful tokens
- **Cleaning**: Strips out whitespace, comments, and other non-essential characters
- **Position tracking**: Maintains line and column numbers for error reporting
- **Error correlation**: Links compiler messages back to specific source code locations

2. Why should parsing be separated from lexical analysis?

Separation provides three key benefits:

- **Simplicity in design**: Each phase has a single, focused responsibility - lexical analysis handles character recognition, parsing handles grammar structure
- **Improved efficiency**: Specialized algorithms can be optimized for each specific task
- **Enhanced portability**: The lexical analyzer can be easily adapted for different character sets while grammar rules remain unchanged across platforms

3. What are tokens, patterns, and lexemes and what are the differences between these three?

Concept	Definition	Example
Token	A category or label for lexical units (like grammatical parts of speech)	`identifier`, `number`, `operator`
Pattern	The rule or regular expression defining how to recognize a token	`/[A-Za-z_][A-Za-z0-9_]*/`
Lexeme	The actual text from source code that matches a token's pattern	`"count"`, `"42"`, `"+"`

Key difference: Tokens are categories, patterns are recognition rules, lexemes are actual text instances.

4. What are lexical errors and how

are they handled?

Lexical errors occur when no defined pattern matches the current sequence of characters in the source code.

Five handling strategies:

1. **Panic mode**: Delete characters until a valid token can be formed
2. **Deletion**: Remove one character from the problematic lexeme
3. **Insertion**: Add a character to create a valid token
4. **Replacement**: Replace one character (common for typos)
5. **Transposition**: Swap adjacent characters (handles typing mistakes)

5. How is a lexical definition of a language specified?

Lexical definitions use **regular expressions** built on these foundations:

- **Alphabet (Σ)**: Finite set of symbols (e.g., ASCII characters)
- **Strings**: Finite sequences of symbols from the alphabet
- **Languages**: Sets of strings defined over an alphabet
- **Regular expressions**: Mathematical patterns using operators like `*` (zero or more), `+` (one or more), `|` (union), and `[]` (character classes)

Example specification:

```
identifier: [a-zA-Z_][a-zA-Z0-9_]*
number: [0-9]+
whitespace: [ \t\n\r]+
```

Remember: The lexical analyzer is your first line of defense in compilation - it transforms messy text into clean, categorized tokens that the parser can work with!

Lexical Analysis Implementation

Overview

Lexical Analysis is the first phase of a compiler that breaks down source code into meaningful tokens. Think of it as reading a sentence and identifying each word, punctuation mark, and symbol.

Key Concepts:

- **Token**: A meaningful unit (like keywords, identifiers, operators)
- **Lexeme**: The actual string that forms a token (e.g., "if", "x", "+")
- **Pattern**: Rules that describe how tokens are formed

Recognition of Tokens

Basic Assumptions

1. **Reserved words cannot be used as identifiers**
   - Example: You can't name a variable "if" or "while"
2. **Whitespace is ignored**
   - Spaces, tabs, and newlines don't affect token recognition

Token Recognition Process

1. **Order token patterns by precedence**
   - Keywords before identifiers
   - Longer operators before shorter ones (">=" before ">")

2. **Use two pointers:**
    - **Lexeme-beginning pointer**: Marks start of current token
    - **Forward pointer**: Scans ahead through input
3. **Recognition steps:**

```
Input: "if (x >= 10)"

Step 1: Scan "if" → Recognized as KEYWORD
Step 2: Skip whitespace
Step 3: Scan "(" → Recognized as LEFT_PAREN
Step 4: Skip whitespace
Step 5: Scan "x" → Recognized as IDENTIFIER
Step 6: Skip whitespace
Step 7: Scan ">=" → Recognized as GTE_OPERATOR
Step 8: Skip whitespace
Step 9: Scan "10" → Recognized as NUMBER
Step 10: Scan ")" → Recognized as RIGHT_PAREN
```

Transition Diagrams

A **transition diagram** is a visual representation of how a lexical analyzer recognizes tokens.

Elements of a Transition Diagram:

1. **States** (circles): Represent current position in recognition
2. **Edges/Transitions** (arrows): Show movement between states
3. **Symbols** (labels on arrows): Characters that trigger transitions
4. **Start state**: Where recognition begins
5. **Accepting states**: Where valid tokens are recognized
6. **Determinism**: Each state has at most one transition for each input symbol

Example: Recognizing Identifiers

```
Input: Letters followed by letters/digits

    letter
[Start] ───────→ [State1] ───────→ [Accept]
                   ↑        letter/digit
                   └─────────────────┘
```

Example: Recognizing Numbers

```
Input: Digits, possibly with decimal point

    digit        digit
[Start] ───────→ [State1] ───────→ [Accept]
                   |
                   | '.'
                   ↓        digit
                 [State2] ───────→ [State3] → [Accept]
                            ↑
                            └──── digit
```

Implementation Details

Core Functions:

- `nextchar()`: Reads next character, advances forward pointer
- **State management**: Each state has its own code segment
- **Backtracking**: When recognition fails, return to beginning

Implementation Algorithm:

```
function getNextToken():
    lexeme_begin = forward_pointer

    while (not end_of_input):
        current_char = nextchar()

        if (valid_transition_exists):
            move_to_next_state()
        else:
            if (current_state_is_accepting):
                retract_one_character()
                return create_token(current_lexeme)
            else:
                reset_to_beginning()
                try_next_pattern()

    return END_OF_FILE_TOKEN
```

Symbol Table Integration:

- **Identifiers** must be stored in symbol table during lexical analysis
- **Keywords** can be stored in a hash table for quick lookup
- This reduces the number of states needed in transition diagrams

---

Finite State Automata (FSA)

An **FSA** is a mathematical model for recognizing patterns in input.

Components:

1. **States**: Finite set of conditions
2. **Alphabet**: Set of input symbols
3. **Transition function**: Maps (state, symbol) → state
4. **Start state**: Initial state

5. **Accepting states**: States that recognize valid input

Example: Binary strings ending with "01"

```
Input alphabet: {0, 1}

       0        0
[q0] ────→ [q1] ────→ [q1]
  |          |
  |1         |1
  ↓          ↓
[q0] ←──── [q2] ──── ((q3))
      0         1
```

- q0: Start state
- q3: Accepting state (double circle)
- Accepts: "01", "001", "101", "1001", etc.

Guide Questions Answered

1. How are tokens recognized by a lexical analyzer?

Tokens are recognized through a **pattern matching process** using:

- **Transition diagrams** or **finite state automata** that define valid token patterns
- **Two-pointer technique** (lexeme-beginning and forward pointers) to scan input
- **Priority-based matching** where patterns are tried in order of precedence
- **Backtracking** when invalid patterns are encountered

Example Process:

```
Input: "count = 42"

1. Scan "count" → matches identifier pattern → TOKEN: ID("count")
2. Skip whitespace
3. Scan "=" → matches assignment operator → TOKEN: ASSIGN
4. Skip whitespace
5. Scan "42" → matches number pattern → TOKEN: NUM(42)
```

2. What are the elements of a transition diagram?

The key elements are:

1. **States** (nodes/circles): Represent current position in pattern recognition
2. **Edges/Transitions** (arrows): Show valid moves between states based on input
3. **Symbols** (edge labels): Characters or character classes that trigger transitions
4. **Start state**: Initial state where recognition begins
5. **Accepting states**: Final states that indicate successful token recognition
6. **Determinism**: Property ensuring unique transitions (no ambiguity)

Visual Example:

```
Recognizing floating-point numbers:

    digit     digit        '.'       digit
[Start] → [Integer] → [DecPoint] → [Float]
           ↑                         ↑
           └── digit ────────────────┘
```

3. How are lexical analyzers implemented?

Lexical analyzers are implemented using:

A. State-driven approach:

- Each state corresponds to a code segment
- Transitions change program state
- `nextchar()` function advances through input

B. Table-driven approach:

- Transition table maps (current_state, input_symbol) → next_state
- Generic driver reads table and processes input

C. Key implementation strategies:

- **Error handling**: Backtrack when invalid patterns are found
- **Symbol table integration**: Store identifiers as they're recognized
- **Keyword handling**: Pre-populate reserved words to distinguish from identifiers
- **Buffer management**: Efficiently handle large input files

Sample Implementation Structure:

```
class LexicalAnalyzer:
    input_buffer
    forward_pointer
    lexeme_begin
    symbol_table
    keyword_table

    function getToken():
        skip_whitespace()
        lexeme_begin = forward_pointer

        # Try each token pattern in priority order
        if (try_keyword_pattern()):
            return keyword_token
        elif (try_identifier_pattern()):
            return identifier_token
        elif (try_number_pattern()):
            return number_token
        # ... other patterns
        else:
            return error_token
```

FSA Activities Solutions

Activity 1: FSA that accepts strings ending with "01"

```
States: {q0, q1, q2, q3}
Start state: q0
Accepting state: q3
Alphabet: {0, 1}

Transitions:
       0        1
q0 → q1,    q0 → q0
q1 → q1,    q1 → q2
q2 → q1,    q2 → q0
q3 → q1,    q3 → q0 (q3 is accepting)

Diagram:
       0        0
[q0] ────→ [q1] ────→ [q1]
 | ↑         |
 | |0        |1
 |1└─── [q2] ──────→ ((q3))
 ↓            1
[q0]

Actually, let me correct this:

       0        0
[q0] ────→ [q1] ────→ [q1]
 |           |
 |1          |1
 ↓           ↓
[q0] ←──── [q2] ──── ((q3))
      0         1
```

Test cases: - "01" ✓ (accepts) - "001" ✓ (accepts) - "101" ✓ (accepts) - "10" ✗ (rejects) - "11" ✗ (rejects)

Activity 2: FSA that accepts strings with an even number of a's

```
States: {q0, q1}
Start state: q0 (also accepting - 0 is even)
Accepting state: q0
Alphabet: {a, b}

Transitions:
q0 → q1 (on 'a'), q0 → q0 (on 'b')
q1 → q0 (on 'a'), q1 → q1 (on 'b')

Diagram:
        a
((q0)) ↔ [q1]
  ↑        ↑
  |b       |b
  └────────┘
```

Test cases: - "" ✓ (0 a's - even) - "aa" ✓ (2 a's - even) - "bab" ✗ (1 a - odd) - "baab" ✓ (2 a's - even)

Activity 3: FSA that accepts strings with an odd number of b's

```
States: {q0, q1}
Start state: q0
Accepting state: q1
Alphabet: {a, b}

Transitions:
q0 → q0 (on 'a'), q0 → q1 (on 'b')
q1 → q1 (on 'a'), q1 → q0 (on 'b')

Diagram:
         b
[q0] ↔ ((q1))
 ↑        ↑
 |a       |a
 └────────┘
```

Test cases: - "b" ✓ (1 b - odd) - "bb" ✗ (2 b's - even) - "aba" ✓ (1 b - odd) - "abab" ✗ (2 b's - even)

Activity 4: FSA that accepts strings containing "ab" as a substring

```
States: {q0, q1, q2}
Start state: q0
Accepting state: q2
Alphabet: {a, b}

Transitions:
q0 → q1 (on 'a'), q0 → q0 (on 'b')
q1 → q1 (on 'a'), q1 → q2 (on 'b')
q2 → q2 (on 'a'), q2 → q2 (on 'b')

Diagram:
       a        b
[q0] ────→ [q1] ────→ ((q2))
 │ ↑                    ↑
 │b│a                   │a,b
 ↓ └────────────────────┘
[q0]
```

Test cases: - "ab" ✓ (contains "ab") - "aab" ✓ (contains "ab") - "abbb" ✓ (contains "ab") - "ba" ✗ (doesn't contain "ab") - "a" ✗ (doesn't contain "ab")

Summary

Lexical analysis is the foundation of compilation, transforming raw source code into meaningful tokens through:

- **Pattern matching** using FSAs and transition diagrams
- **Systematic scanning** with pointer-based techniques
- **Priority-based recognition** to handle conflicts
- **Integration with symbol tables** for identifier management