Parser Implementation

From Tokens to Structure

Jian Weng

CEMSE, KAUST

Week 4, Session 2

Today's Story

1. We finished lexer: now we have tokens
2. Parser purpose: organize tokens into structure
3. Target structure first: what AST do we want?
4. CST vs AST: trace vs final data structure
5. Theory: context-free grammar / context-free language
6. Grammar is recursive expansion
7. Recursion maps naturally to functions
8. Classic tools: Flex + Bison
9. Why we choose hand-written recursive descent

Recap: What Lexer Gives Us

```
Input:
  fn add(x, y) { x + y; }

Lexer Output:
  FN IDENT LPAREN IDENT COMMA IDENT RPAREN LBRACE IDENT PLUS IDENT SEMI RBRACE
```

- Lexer solved tokenization
- But token list alone has no hierarchy

Parser Purpose

- Parser consumes lexer tokens
- Parser builds a structured data model
- That model is the AST (Abstract Syntax Tree)
- AST is what semantic analysis and codegen actually need

```
Tokens -> Parser -> AST
```

Target Data Structure (First)

```
struct FuncDecl : ASTNode {
  string id;
  vector<Argument*> args;
  CompoundStatement* body;
};
```

- Parsing is essentially organizing tokens into this kind of structure
- We design target node shapes before parser function details

CST vs AST

- CST (Concrete Syntax Tree):
  trace of grammar traversal and parsing function calls
- AST (Abstract Syntax Tree):
  simplified final data structure for later compiler stages

```
Tokens -> [CST-like parsing trace] -> AST
```

In this course, AST is the final parser output.

Why Tokens Are Not Enough

```
IDENT LPAREN IDENT COMMA IDENT RPAREN
```

- Is this function call?
- Is this declaration?
- Are arguments complete?
- Where is body / scope?

Parser resolves these structural questions.

Formal Language View

- In theory, parser works on a formal language definition
- We describe syntax with CFG (Context-Free Grammar)
- The language described by CFG is CFL (Context-Free Language)

```
Grammar (rules) defines legal programs
Parser checks and reconstructs those rules from tokens
```

CFG Example for Declarations

```
func_decl          := "fn" identifier "(" argument_list? ")" compound_statement
argument_list      := argument ("," argument)*
compound_statement := "{" statement_list? "}"
statement_list     := statement (";" statement)*
```

- Non-terminals: `func_decl`, `argument_list`, ...
- Terminals: `fn`, `identifier`, `(`, `)`, `{`, `}`

Grammar Is Recursive Expansion

```
func_decl
  -> "fn" identifier "(" argument_list ")" compound_statement

argument_list
  -> argument
  -> argument "," argument_list
```

- A rule expands into smaller rules
- Some rules call themselves recursively

Key Insight

Each recursive grammar rule can map to one function.

```
func_decl        -> parse_func_decl()
argument_list    -> parse_argument_list()
statement_list   -> parse_statement_list()
```

This gives us the most intuitive parser implementation strategy.

Algorithm: Recursive Descent

```
For each non-terminal:
  write one parser function
```

- Top-down parsing
- Human-readable control flow
- Easy to debug in class and in assignment

Classic Parser Generators

- Historically: `flex` for lexer + `bison` for parser
- Very easy to write and still used in production systems: e.g. Verilator

Why Not Bison/Flex for This Class?

- ~~You have more code to write now to practice your vibe coding~~
- We want parser logic that is transparent and easy to evolve
- Hand-written recursive descent gives direct control over rule dispatch
- Easier to inject context-sensitive checks during parsing when needed

```
This course focus: clarity + flexibility + engineering control
```

Do you know what is PoD?

- PoD = Plain old Data
- Bison was ok, but it only supports PoD types for AST nodes

```
union ASTNode {
  Argument* args;
  CompoundStatement* body;
  FuncDecl* func;
  // ...
};
```

Parser State and API

```
Token Parser::peek(int k = 0) const;
bool  Parser::match(TokenKind kind);
Token Parser::expect(TokenKind kind, const string& message);
```

- `peek` : lookahead without consuming
- `match` : consume if next token matches
- `expect` : enforce syntax rule or throw parser error

Lookahead Tokens

- Lookahead means "inspect upcoming token(s) first"
- Usually one-token lookahead is enough for our grammar
- This is how parser decides which rule to enter

```
if peek() == FN      -> parse_func_decl()
if peek() == STRUCT  -> parse_struct_decl()
```

Rule Dispatch in Practice

```
Decl* Parser::parse_decl() {
  if (peek().kind == FN) return parse_func_decl();
  if (peek().kind == STRUCT) return parse_struct_decl();
  error("expected declaration");
}
```

- Dispatch is the central switchboard of recursive descent

Implement `parse_func_decl`

```
FuncDecl* Parser::parse_func_decl() {
  expect(FN, "expected 'fn'");
  string id = expect(IDENT, "expected function name").lexeme;
  expect(LPAREN, "expected '('");
  auto args = parse_argument_list();
  expect(RPAREN, "expected ')'");
  auto body = parse_compound_statement();
  return new FuncDecl{id, args, body};
}
```

Implement `parse_argument_list`

```
vector<Argument*> Parser::parse_argument_list() {
  vector<Argument*> args;
  if (peek().kind == RPAREN) return args;
  args.push_back(parse_argument());
  while (match(COMMA)) args.push_back(parse_argument());
  return args;
}
```

- Zero args: immediate return
- N args: parse first, then loop on comma

AST Is Built During Descent

- Parsing functions do two things together:
  - verify syntax
  - construct AST nodes
- Example: `parse_func_decl` consumes tokens and returns `FuncDecl*`
- Semantic analysis should read AST only, not parser cursor state

Error Handling

- Syntax errors must include token + location
- Continue when possible to find more errors
- Recovery points: `;`, `}`, next declaration keyword

```
error: expected ')' after argument list at line 12, column 19
```

End-to-End Parsing Flow

```
parse_program()
  -> parse_decl()
       -> parse_func_decl()
            -> parse_argument_list()
            -> parse_compound_statement()
```

This is exactly grammar expansion, but implemented as function calls.

LLM-Assisted Parser Development

Bottom-Up Workflow

- Do not ask the model to "build the full parser" in one session
- Start from small leaf-level parse functions
- Then compose them into higher-level parse functions
- Bottom-up generation improves correctness and debuggability

Why Bottom-Up Works Better

- Smaller tasks produce more reliable model outputs
- Each parser node has a clear contract
- Unit tests can lock behavior before composition
- Composition bugs are easier to isolate

```
parse_statement() first
parse_compound_statement() second
parse_func_decl() after that
```

Step 1: Generate Leaf Parser Nodes

Prompt the model to implement one node at a time.

```
Implement parse_statement() for this grammar subset.
Use Parser::peek/match/expect only.
Return Statement* AST nodes.
Do not modify other parser functions.
Also generate unit tests for valid and invalid statements.
```

- Keep scope narrow
- Force AST return type
- Always request tests in the same prompt

Step 2: Compose Higher-Level Nodes

After `parse_statement()` is stable, generate `parse_compound_statement()`.

```
Implement parse_compound_statement():
compound_statement := "{" statement_list? "}"
Reuse parse_statement().
Stop on RBRACE.
Return CompoundStatement*.
Add tests for empty body and multiple statements.
```

- Composition happens only after child nodes are tested
- Reuse already-validated functions

Step 3: Build Composition Rules

- Ask the model to wire parent-child relationships explicitly
- Keep function boundaries strict
- Avoid hidden behavior in helper utilities

```
parse_statement_list()
  - repeatedly call parse_statement()
  - collect nodes into vector<Statement*>
  - enforce semicolon policy based on grammar
```

Then use this in `parse_compound_statement()` and later in `parse_func_decl()`.

Prompt Pattern for Reliable Output

```
Task: Implement <one parser function>.
Grammar: <exact rule>.
Input API: peek/match/expect.
Output: <specific AST type>.
Constraints: no unrelated refactors.
Tests: include valid + invalid cases.
```

- Deterministic prompt structure reduces hallucination
- One function per prompt keeps review manageable

Verification Loop with LLM

1. Generate one parser node
2. Run parser tests
3. Fix failures only for that node
4. Commit
5. Move one level up in the grammar tree

This is bottom-up parser construction with continuous validation.

Checkpoint #2 (Assignment)

Goal: implement parser stage for the language subset in this course.

- Input: token sequence from your lexer
- Output: AST for declarations and statements
- Method: recursive descent with lookahead dispatch

Checkpoint #2 Requirements

1. Implement parser helper APIs (`peek` , `match` , `expect`)
2. Implement declaration dispatch (`parse_decl`)
3. Implement `parse_func_decl` and `parse_compound_statement`
4. Construct AST nodes correctly
5. Add parser tests for valid and invalid inputs

Required Syntax Features (Checkpoint #2)

Your parser must support:

1. Function parsing: declarations, names, parameter list, and body
2. Compound statements: `{ ... }` blocks with nested statements
3. Complex expressions with precedence/associativity (e.g., `a * b + c`)
4. Array access expressions (e.g., `arr[i]`, `matrix[i][j]`)
5. Loop statements: `for (...) { ... }` and `while (...) { ... }`
6. `return` statements (with and without return value)

Suggested Test Cases for These Features

1. Function with empty body and function with multiple statements
2. Nested compound statements
3. Expression precedence and parentheses:
   `a * b + c`, `a * (b + c)`, `x + y * z`
4. Array indexing and chained indexing:
   `a[i]`, `a[i + 1]`, `m[i][j]`
5. `for` / `while` parsing with block bodies
6. `return;` and `return expr;`