Syntax Analysis

Mini-C Compiler

CS4031 - Compiler Construction | Fall 2025

Team Members

Subhan (22K-4316)

Sadiq (22K-4303)

Muhammad Ahmed Haque (22K-4232)

Muhammad Irtiza (22K-4638)

Umer (22K-4160)

Junaid (22K-4369)

December 2025

1. Overview

The parser (syntax analyzer) is the second phase of the Mini-C compiler. It takes a stream of tokens from the lexer and constructs an Abstract Syntax Tree (AST) that represents the hierarchical structure of the program.

Location: src/frontend/parser/parser.py

Parsing Technique: Recursive Descent Parser with Precedence Climbing for expressions

2. Grammar Summary

2.1 Top-Level Productions

```
Program        -> Declaration*
Declaration    -> FunctionDecl | VariableDecl | StructDecl
FunctionDecl   -> TypeSpec Identifier '(' ParamList ')' (Block | ';')
VariableDecl   -> TypeSpec Declarator ['=' Initializer] ';'
StructDecl     -> 'struct' Identifier '{' MemberDecl* '}' ';'
```

2.2 Statement Productions

```
Statement      -> CompoundStmt | ExprStmt | SelectionStmt | IterationStmt | JumpStmt
CompoundStmt   -> '{' BlockItem* '}'
SelectionStmt  -> 'if' '(' Expression ')' Statement ['else' Statement]
IterationStmt  -> WhileStmt | DoWhileStmt | ForStmt
WhileStmt      -> 'while' '(' Expression ')' Statement
ForStmt        -> 'for' '(' ForInit [Expr] ';' [Expr] ')' Statement
JumpStmt       -> 'return' [Expression] ';' | 'break' ';' | 'continue' ';'
```

2.3 Expression Productions

```
Expression     -> AssignmentExpr
AssignmentExpr -> ConditionalExpr | UnaryExpr '=' AssignmentExpr
ConditionalExpr-> LogicalOrExpr ['?' Expression ':' ConditionalExpr]
LogicalOrExpr  -> LogicalAndExpr ('||' LogicalAndExpr)*
LogicalAndExpr -> EqualityExpr ('&&' EqualityExpr)*
EqualityExpr   -> RelationalExpr (('==' | '!=') RelationalExpr)*
RelationalExpr -> AdditiveExpr (('<' | '>' | '<=' | '>=') AdditiveExpr)*
AdditiveExpr   -> MultExpr (('+' | '-') MultExpr)*
MultExpr       -> CastExpr (('*' | '/' | '%') CastExpr)*
UnaryExpr      -> PostfixExpr | UnaryOp CastExpr | 'sizeof' UnaryExpr
PostfixExpr    -> PrimaryExpr (Subscript | FuncCall | MemberAccess)*
PrimaryExpr    -> Identifier | Constant | '(' Expression ')'
```

3. AST Node Types

3.1 Declaration Nodes

Node	Fields	Description
Program	function_definition: List	Root of AST
FunDecl	name, params, type, body	Function declaration
VarDecl	name, init, var_type	Variable declaration
StructDecl	tag, members	Struct type declaration

3.2 Statement Nodes

Node	Fields	Description
Return	exp	Return statement
If	exp, then, _else	If-else statement
While	exp, body	While loop
For	init, cond, post, body	For loop
Compound	block	Block statement
Break	-	Break statement
Continue	-	Continue statement

3.3 Expression Nodes

Node	Fields	Description
Binary	operator, left, right	Binary operation
Unary	operator, expr	Unary operation
Constant	value	Literal constant
Var	identifier	Variable reference
Assignment	left, right	Assignment
Conditional	cond, exp2, exp3	Ternary operator
FunctionCall	identifier, args	Function call
Subscript	array, index	Array subscript

4. Parse Tree Derivations

4.1 Example 1: Simple Return Statement

Source Code:

```
int main(void) {
    return 42;
}
```

Parse Tree Derivation:

```
Program
 +-- FunctionDecl
      +-- type: Int
      +-- name: "main"
      +-- params: [void]
      +-- body: Block
           +-- Return
                +-- Constant
                     +-- value: 42
```

4.2 Example 2: If-Else Statement

Source Code:

```
int max(int a, int b) {
    if (a > b)
        return a;
    else
        return b;
}
```

Parse Tree Derivation:

```
Program
 +-- FunctionDecl
      +-- type: Int
      +-- name: "max"
      +-- params: [(Int, "a"), (Int, "b")]
      +-- body: Block
           +-- If
                +-- condition: Binary(GT)
                |    +-- left: Var("a")
                |    +-- right: Var("b")
                +-- then: Return
                |    +-- Var("a")
                +-- else: Return
                     +-- Var("b")
```

4.3 Example 3: For Loop with Array

Source Code:

```
int sum(int arr[5]) {
    int total = 0;
    for (int i = 0; i < 5; i = i + 1) {
        total = total + arr[i];
    }
    return total;
}
```

Parse Tree Derivation:

```
Program
 +-- FunctionDecl
      +-- type: Int
      +-- name: "sum"
      +-- params: [(Array[5] of Int, "arr")]
      +-- body: Block
           +-- VarDecl(name: "total", type: Int, init: 0)
           +-- For
           |    +-- init: VarDecl(Int, "i", 0)
           |    +-- condition: Binary(LT, Var("i"), Const(5))
           |    +-- post: Assignment(Var("i"), Binary(ADD, Var("i"), Const(1)))
           |    +-- body: Block
           |         +-- Assignment
           |              +-- left: Var("total")
           |              +-- right: Binary(ADD)
           |                   +-- Var("total")
           |                   +-- Subscript(Var("arr"), Var("i"))
           +-- Return(Var("total"))
```

5. Operator Precedence Parsing

The parser uses precedence climbing for expression parsing:

Precedence	Operators	Associativity
50	* / %	LEFT
45	+ -	LEFT
35	< > <= >=	LEFT
30	== !=	LEFT
10	&&	LEFT
5	||	LEFT
3	?:	RIGHT
1	=	RIGHT

6. Error Handling

Error	Example	Message
Missing semicolon	int x = 5	Expected ';', got ...
Unmatched paren	if (x > 5	Expected ')', got ...
Invalid decl	int 123abc;	Expected identifier
Missing brace	int f() { return 1;	Expected '}'