University of Malta

Faculty of ICT

PArL Compiler Implementation

CPS2000 - Compiler Theory and Practice

Course Assignment 2024/2025

Name: Ben Taliana

Contents

1 Introduction

This report documents the design and implementation of a complete compiler for the Pixel Art Language (PArL) programming language developed as part of the CPS2000 Compiler Theory and Practice course assignment. The project addresses a real-world scenario in which PArDis, a startup specializing in programmable pixel art displays, requires a high-level programming language to improve the user experience of their PAD2000c display device.

1.1 Project Context

PAD2000c features advanced OLED technology and a powerful programmable controller; however, its native programming interface, Pixel Art Intermediate Representation (PArIR), has proven challenging for users because of its low-level, assembly like nature. This compiler bridges the gap by translating PArL, a high-level, strongly typed, expression-based language with C-style syntax, into executable PArIR instructions.

1.2 Objectives

The primary objectives of this project were as follows:

- Develop a table-driven FSA-based lexical analyzer capable of tokenizing PArL source code
- Implement a hand-crafted recursive descent LL(k) parser to construct abstract syntax trees
- Design a comprehensive semantic analyzer using the visitor pattern for type checking and validation
- Create a code generator that produces optimized PArIR instructions for the PAD2000c VM
- Extend the compiler to support fixed-size arrays, enhancing the language's expressiveness

1.3 Implementation Overview

The compiler was implemented in Python and chosen for its excellent string manipulation capabilities, clear syntax for recursive algorithms, and rich standard library support. The implementation follows classical compiler construction principles with distinct phases for lexical analysis, parsing, semantic analysis, and code generation. Each phase communicates through well-defined interfaces with comprehensive error-handling and recovery mechanisms. The project structure can be found in the README.md file with the deliverables.

The compiler successfully handles all the PArL language features, including

- Four primitive types: `int`, `float`, `bool`, and `colour`
- Function declarations with typed parameters and return values

- Control flow constructs (if-else, while, for loops)
- Built-in functions for display manipulation (`__write`, `__write_box`, etc.)
- Type casting with the `as` operator
- Fixed-size arrays with compile-time size verification

1.4 Report Structure

2 Task 1: FSA-Based Table-Driven Lexer

2.1 Design and Implementation

The lexer implements a deterministic finite automaton (DFA) using a table-driven approach, which provides a clear separation between the FSA logic and token recognition patterns. This design choice offers several advantages: maintainability through centralized state transitions, performance through direct table lookups, and extensibility for adding new token types. The lexer strictly adheres to the maximal munch (longest match) principle, which is essential for correctly tokenizing multicharacter operators and preventing ambiguous token boundaries.

2.1.1 Character Categorization

The lexer employs a sophisticated character categorization system that distinguishes between the 27 distinct character classes.

```
self.categories = [
    "letter",      # A-Z, a-z (excluding hex letters A-F, a-f)
    "hexletter",   # A-F, a-f
    "digit",       # 0-9
    "underscore",  # _
    # ... (other categories)
    "modulo",      # %
]
```

A critical design decision was to separate `hexletter` from regular `letter` categories. This distinction enables precise color literal parsing, where `#ABC123` is valid but `#GBC123` is not. The categorization function ensures that hexadecimal letters (A-F, a-f) are classified differently from general alphabetic characters, thereby preventing invalid color literals from being partially consumed as valid tokens.

2.1.2 Transition Table Architecture

The transition table was implemented using Python's `defaultdict` with a two-level structure:

```
self.Tx = defaultdict(lambda: defaultdict(lambda: -1))
```

This design elegantly handles undefined transitions by returning -1 (error state) for any unspecified state-character combination. The table maps `[current_state][character_category] -> next_state`, providing O(1) lookup performance.

2.1.3 Lexical Analysis Algorithm

The core tokenization algorithm implements a longest-match strategy with backtracking support.

1. Start from the current position with state 0
2. For each character, determine its category and lookup the next state
3. Track the last accepting state and its position
4. Continue until reaching an error state (-1)
5. Backtrack to the last accepting state to emit the token
6. Handle any remaining characters as errors

This approach ensures that tokens, such as `123.` are properly handled The lexer recognizes `123` as a valid integer and then attempts to extend it to a float, but when no digit follows the decimal point, it correctly identifies this as an `ERROR_INVALID_FLOAT`.

2.2 Token Definitions and Deviations

2.2.1 EBNF Deviations and Justifications

Several deviations from the original EBNF have been implemented to enhance language functionality and improve error reporting.

1. Modulo Operator Addition

```
(46, ["modulo"], TokenType.MODULO),  # Added % operator
```

Justification: The modulo operator is essential for many programming tasks, particularly in graphics programming where wrapping coordinates and creating patterns are common. Its absence forces users to implement modulo manually using division and multiplication, thereby reducing code clarity.

2. Built-in Function Name Correction

```
"__randi": TokenType.BUILTIN_RANDI,  # Not __random_int
```

Justification: The provided examples consistently use `__randi` rather than `__random_int`. This shorter form aligns with the language's concise naming convention for buildings and improves code readability.

3. Enhanced Comment Handling

The lexer implements sophisticated comment processing beyond basic recognition. Detection of nested block comments, which are forbidden, considering that PArL is a C-style language. The identification of stray `*/` outside comments and the proper handling of unterminated block comments were also implemented.

Justification: These enhancements provide better error messages during development, helping users quickly identify comment-related issues that could otherwise cause confusing parse errors.

2.2.2 Token Type Organization

The lexer recognizes 45 distinct token types and organizes them into logical categories.

- **Literals**: Integer, Float, Colour, Boolean
- **Keywords**: Control flow (if, while, for), Type names, Operators (and, or, not)
- **Built-ins**: Display functions (__write, __clear), Pad functions (__width, __height)
- **Operators**: Arithmetic, Comparison, Assignment, Type cast (as)
- **Delimiters**: Parentheses, Braces, Brackets, Semicolon, Comma
- **Special**: Comments, Whitespace, EOF marker
- **Error tokens**: Invalid constructs with specific error types

2.3 Error Handling

2.3.1 Error Token Strategy

Rather than failing immediately on invalid input, the lexer generates specific error tokens that preserve error context:

```
class TokenType(Enum):
    ERROR_INVALID_FLOAT = auto()      # e.g., "123."
    ERROR_INVALID_COLOUR = auto()     # e.g., "#ZZZ000"
    ERROR_UNTERMINATED_COMMENT = auto()
    ERROR_NESTED_COMMENT = auto()     # "/* /* */"
    ERROR_STRAY_COMMENT_CLOSE = auto() # Lone "*/"
```

This approach enables the parser to provide meaningful error messages and recover from lexical errors. For example, when `123.`, the lexer produces `ERROR_INVALID_FLOAT` rather than separate tokens for ”123” and. “” while preserving user intent.

2.3.2 Error Recovery Mechanism

The lexer implements a conservative error-recovery strategy.

1. When no valid transition exists, determine the specific error type
2. Create an error token with the problematic character(s)
3. Advance by one character to resume scanning
4. Continue tokenization to find additional errors

2.3.3 Contextual Error Detection

The error-detection system considers the context to provide accurate diagnostics.

```
def _determine_error_type(self, text, error_pos, state):
    if state == 30:  # After digits and decimal point
        return TokenType.ERROR_INVALID_FLOAT
    if state == 32:  # Nested "/*" detected
        return TokenType.ERROR_NESTED_COMMENT
    if state in (27, 28, 31):  # Inside block comment at EOF
        return TokenType.ERROR_UNTERMINATED_COMMENT
    # ... additional contextual checks
```

This contextual approach significantly improves error message quality, helping developers quickly identify and fix lexical issues in their PArL programs.

2.3.4 Whitespace and Comment Filtering

By default, the lexer filters the whitespace and comments from the token stream but preserves this capability for debugging:

```
if (token_type not in [TokenType.WHITESPACE, TokenType.NEWLINE,
                       TokenType.LINECOMMENT, TokenType.BLOCKCOMMENT]
    or self.debug):
    tokens.append(Token(token_type, lexeme, start_line, start_col))
```

This design choice simplifies the parser implementation while maintaining the ability to preserve comments for documentation generators or debugging tools.

3 Task 2: Hand-Crafted LL(k) Parser

3.1 Parser Architecture

The parser implements a recursive descent parsing strategy chosen for its natural mapping to the grammatical structure and excellent error-reporting capabilities. The architecture consists of three key components: a token stream abstraction for clean token management, recursive parsing engine that mirrors grammar production, and comprehensive AST construction system.

3.1.1 Token Stream Abstraction

A critical design decision is to encapsulate the token iteration within a dedicated `TokenStream` class:

```
class TokenStream:
    def __init__(self, tokens: List[Token]):
        # Filter comments/whitespace but preserve error tokens
        self.tokens = [t for t in tokens if t.type not in
                       [TokenType.WHITESPACE, TokenType.NEWLINE,
                        TokenType.LINECOMMENT, TokenType.BLOCKCOMMENT]]
        self.position = 0
```

This abstraction has several benefits.

- **Clean separation of concerns**: The parser focuses on grammar rules rather than token management

- **Unified error handling**: All token access goes through methods that check for EOF and invalid tokens
- **Lookahead support**: The `peek()` method enables LL(k) parsing where needed
- **Error token preservation**: Lexical errors are passed to the parser for better diagnostics

3.1.2 Recursive Descent Design

Each non-terminal in the grammar has a corresponding parsing method that creates a clear one-to-one mapping:

```
def parse_statement(self) -> Optional[ASTNode]:
    current = self.stream.current_token()

    # Direct dispatch based on first token
    if self.stream.match(TokenType.FUN).
        return self.parse_function_declaration()
    elif self.stream.match(TokenType.LET)
        return self.parse_variable_declaration()
    elif self.stream.match(TokenType.IF)
        return self.parse_if_statement()
    # ... additional statement types
```

This design pattern makes the parser highly maintainable and allows easy extension with new statement types. The parser is predominantly LL(1), requiring only a single-token look-ahead for most decisions, which contributes to its efficiency.

3.2 Grammar Implementation

3.2.1 Expression Parsing with Correct Precedence

The parser implements a precedence-climbing approach for parsing expressions, ensuring a correct evaluation order without grammatical ambiguity:

```
def parse_expression(self) -> ASTNode:
    return self.parse_logical_or_expression()

def parse_logical_or_expression(self) -> ASTNode:
    left = self.parse_logical_and_expression()
    while self.stream.match(TokenType.OR):
        op_token = self.stream.advance()
        right = self.parse_logical_and_expression()
        left = BinaryOperation(left, op_token.lexeme, right)
    return left
```

The precedence hierarchy (from lowest to highest) is

1. Logical OR (`or`)
2. Logical AND (`and`)
3. Relational operators (`==`, `!=`, `<`, `>`, `<=`, `>=`)
4. Additive operators (`+`, `-`)

5. Multiplicative operators (`*`, `/`, `%`)
6. Type casting (`as`)
7. Unary operators (`-`, `not`)
8. Primary expressions

3.2.2 Identifier Parsing Complexity

A sophisticated aspect of the parser is the handling of identifiers with optional array indexing, as specified in the EBNF:

```
def parse_identifier_with_optional_index(self) -> ASTNode:
    name_token = self.stream.expect(TokenType.IDENTIFIER)
    base = Identifier(name_token.lexeme, name_token.line, name_token.col)

    # Check for array access first (per EBNF)
    if self.stream.match(TokenType.LBRACKET):
        self.stream.advance()
        index = self.parse_expression()
        self.stream.expect(TokenType.RBRACKET)
        base = IndexAccess(base, index, name_token.line, name_token.col)

    # Then check for function call
    if self.stream.match(TokenType.LPAREN):
        if isinstance(base, IndexAccess):
            raise SyntaxError(
                f"Cannot call function on array access '{base.base.name}[...]'",
                self.stream.current_token())
        return self.parse_function_call_continuation(name_token.lexeme, name_token)

    return base
```

This implementation correctly rejects invalid constructs like `arr[5]()` while supporting both `arr[5]` and `func()`.

3.3 AST Design

3.3.1 Node Hierarchy

The AST employs an object-oriented design with a base `ASTNode` class that supports the visitor pattern:

```
class ASTNode(ABC):
    def __init__(self, line: int = 0, col: int = 0):
        self.line = line
        self.col = col

    def accept(self, visitor):
        method_name = f"visit_{self.__class__.__name__.lower()}"
        if hasattr(visitor, method_name):
            return getattr(visitor, method_name)(self)
        return visitor.generic_visit(self)
```

This design enables clean separation between the AST structure and the operations (semantic analysis and code generation) performed on it. Each node stores source position information for accurate error reporting.

3.3.2 Tree Visualization

A notable feature is the built-in tree visualization capability.

```
def _tree_children(self, children, indent_level=1):
    indent = "  " * indent_level
    result = ""
    for child in children:
        if child is not None:
            result += f"\n{indent}{child._get_node_label()}"
            grandchildren = child._get_children()
            if grandchildren:
                result += child._tree_children(grandchildren, indent_level + 1)
    return result
```

This feature proved invaluable during the development of the debugging parser output and was included in the test results for clarity.

3.4 Error Recovery

3.4.1 Error Detection Strategy

The parser implements comprehensive error detection at multiple levels:

```
def parse_statement(self) -> Optional[ASTNode]:
    current = self.stream.current_token()

    # Check for lexical errors first
    if current.type.name.startswith('ERROR'):
        raise LexicalErrorInParsingError(current)
```

This approach ensures that lexical errors are reported in an appropriate context, rather than causing confusing syntax errors.

3.4.2 Synchronization-Based Recovery

When syntax errors occur, the parser attempts to recover by synchronizing the statement boundaries as follows:

```
def synchronize_on_error(self):
    sync_tokens = {
        TokenType.SEMICOLON, TokenType.LBRACE, TokenType.RBRACE,
        TokenType.LET, TokenType.FUN, TokenType.IF, TokenType.WHILE,
        TokenType.FOR, TokenType.RETURN, TokenType.END
    }

    while not self.at_end():
        if self.current_token().type in sync_tokens:
            if self.current_token().type == TokenType.SEMICOLON:
                self.advance()  # Consume semicolon
            break
        self.advance()
```

This strategy allows the parser to continue processing after errors, potentially identifying multiple issues in a single compilation attempt.

3.4.3 Contextual Error Messages

The parser provides detailed error messages that consider parsing context.

```
class UnexpectedTokenError(ParserError):
    def __init__(self, expected: str, found_token):
        message = f"Expected {expected}, but found '{found_token.lexeme}' ({found_token.type.name})"
        super().__init__(message, found_token)
```

Error messages include both what was expected and what was found, along with the source position, making debugging significantly easier.

3.5 Deviations and Justifications

3.5.1 Grammar Clarifications

Several aspects of the EBNF require interpretation and clarification.

1. Array Type Syntax Enhancement

The parser supports both array-type declarations in variable declarations and function parameters.

```
def parse_type(self) -> Union[str, ArrayType]:
    base_type = token.lexeme
    self.stream.advance()

    if self.stream.match(TokenType.LBRACKET):
        self.stream.advance()
        if self.stream.match(TokenType.RBRACKET):
            return ArrayType(base_type, None)  # Dynamic array
        elif self.stream.match(TokenType.INT_LITERAL):
            size = int(size_token.lexeme)
            return ArrayType(base_type, size)  # Fixed array
```

Justification: While the original EBNF showed array syntax primarily in formal parameters, extending it to all type contexts provides consistency and enables array variable declarations like `let arr:int[10]`.

2. Expression Statement Handling

The parser carefully distinguished between the assignments and expression statements.

```
def parse_assignment_or_expression_statement(self) -> ASTNode:
    target = self.parse_identifier_with_optional_index()

    if self.stream.match(TokenType.EQUAL):
        # Assignment
        value = self.parse_expression()
        return Assignment(target, value)
    else:
        # Expression statement (must be function call)
        if not isinstance(target, FunctionCall):
            raise SyntaxError("Invalid expression statement")
        return target
```

Justification: This approach prevents meaningless expression statements like `x;` while allowing function calls with side effects like `print_result();`.

3. Comma Operator Exclusion in Built-ins

The parser uses a specialized parsing method for built-in functions with multiple arguments.

```
def parse_write_statement(self) -> WriteStatement:
    x = self.parse_comma_separated_expression()  # Not full expression
    self.stream.expect(TokenType.COMMA)
    y = self.parse_comma_separated_expression()
    self.stream.expect(TokenType.COMMA)
    color = self.parse_comma_separated_expression()
```

Justification: This prevents the comma from being interpreted as an operator within built-in function arguments, avoiding ambiguity in expressions like `__write 1+2, 3*4, #FF0000`.

4. Operator Precedence Refinement

The parser explicitly separates logical operators into different precedence levels.

Justification: While the EBNF groups `and` with multiplicative and `or` with additive operators, standard programming language design places logical operators at lower precedence than arithmetic operators. This change aligns PArL with programmer expectations from languages, such as C, Java, and Python.

The parser implementation successfully balances adherence to the specification with practical usability concerns, creating a robust front-end that provides clear error messages and generates well-structured ASTs for the subsequent compilation phases.

4 Task 3: Semantic Analysis

4.1 Symbol Table and Scope Management

The semantic analyzer implements a sophisticated symbol table design using a stack-based approach to manage nested scopes. This architecture naturally models PArL's block-structured scope rules while providing efficient symbol resolution.

4.1.1 Symbol Table Architecture

The symbol table maintains two critical data structures:

```
class SymbolTable:
    def __init__(self):
        self.scopes: List[Dict[str, Symbol]] = [{}]  # Stack of scopes
        self.current_scope_level = 0
        self.functions: Dict[str, Symbol] = {}  # Global function registry
```

This dual structure serves distinct purposes:

- **Scope stack**: Manages variable declarations with proper shadowing and block-level isolation
- **Function registry**: Maintains global function visibility, enabling forward references

The separation allows functions to be called before their declaration in the source code, a critical feature for mutual recursion and natural code organization.

4.1.2 Symbol Representation

Each symbol carries comprehensive type and context information:

```
class Symbol:
    def __init__(self, name: str, symbol_type: Union[str, ArrayType],
                 scope_level: int = 0, is_function: bool = False,
                 is_parameter: bool = False):
        self.name = name
        self.symbol_type = symbol_type
        self.scope_level = scope_level
        self.is_function = is_function
        self.is_parameter = is_parameter
        self.is_initialized = False
        self.parameter_types: List[Union[str, ArrayType]] = []
        self.return_type: Union[str, ArrayType] = ""
```

The `is_initialized` flag enables detection of uninitialized variable usage, while `is_parameter` prevents parameter shadowing within function bodies—a common source of confusion in programming languages.

4.1.3 Scope Entry and Exit

Scope management follows a clean push/pop paradigm:

```
def enter_scope(self):
    self.scopes.append({})
    self.current_scope_level += 1

def exit_scope(self):
    if len(self.scopes) > 1:
        self.scopes.pop()
        self.current_scope_level -= 1
```

This approach ensures proper cleanup of local variables when exiting blocks, preventing memory leaks in the symbol table and maintaining correct scoping semantics.

4.2 Type System and Semantic Rules

4.2.1 Type Representation and Compatibility

The type system distinguishes between primitive types and arrays, with sophisticated compatibility checking:

```
@staticmethod
def types_equal(type1: Union[str, ArrayType], type2: Union[str, ArrayType]) -> bool:
    if isinstance(type1, ArrayType) and isinstance(type2, ArrayType):
        element_types_match = type1.element_type == type2.element_type

        # Allow passing fixed-size arrays to dynamic array parameters
        if type2.size is None:  # Parameter is dynamic array
            return element_types_match
        elif type1.size is None:  # Argument is dynamic, parameter is fixed
            return False
        else:  # Both are fixed-size arrays
            return element_types_match and type1.size == type2.size
```

This implementation makes a crucial design decision: fixed-size arrays can be passed to functions expecting dynamic arrays, but not vice versa. This provides flexibility while maintaining type safety.

4.2.2 Binary Operation Type Rules

The type checker implements comprehensive rules for binary operations:

```
@staticmethod
def get_binary_operation_result_type(left_type: str, operator: str,
                                     right_type: str) -> Optional[str]:
    # Arithmetic operators including modulo
    if operator in ["+", "-", "*", "/", "%"]:
        if left_type == right_type and left_type in ["int", "float"]:
            return left_type

        # Modulo requires both operands to be int
        if operator == "%":
            if left_type == "int" and right_type == "int":
                return "int"
            return None

    # Comparison operators ALWAYS return bool
    if operator in ["<", ">", "<=", ">=", "==", "!="]:
        if left_type == right_type:
            return "bool"
        # Allow mixed int/float comparisons
        if {left_type, right_type} == {"int", "float"}:
            return "bool"
```

Key design decisions include:

- **No implicit type conversion**: Operands must have matching types (except int/float comparisons)
- **Modulo restriction**: Only integer operands are allowed, preventing floating-point modulo
- **Boolean comparison results**: All relational operators return `bool`, not integers

4.2.3 Type Casting Rules

The analyzer enforces explicit type casting with specific allowed conversions:

```python
@staticmethod
def can_cast(from_type: str, to_type: str) -> bool:
    valid_casts = {
        ("int", "float"): True,
        ("float", "int"): True,
        ("int", "bool"): True,
        ("bool", "int"): True,
        ("int", "colour"): True,
        ("colour", "int"): True,
    }
    return valid_casts.get((from_type, to_type), False)
```

These rules balance safety with practicality, allowing meaningful conversions while preventing nonsensical ones like `float` to `colour`.

4.3 Semantic Validation

4.3.1 Variable Declaration Validation

The analyzer performs comprehensive checks during variable declaration:

```python
def visit_variable_declaration(self, node: VariableDeclaration):
    # Check for parameter shadowing
    if self.current_function:
        for scope in self.symbol_table.scopes:
            if node.name in scope and scope[node.name].is_parameter:
                self._add_error(SemanticErrorType.REDECLARATION,
                            f"Variable '{node.name}' conflicts with function parameter")
                return

    # Array initialization validation
    if isinstance(node.var_type, ArrayType) and node.initializer:
        if isinstance(node.initializer, ArrayLiteral):
            # Check size constraints
            actual_size = len(node.initializer.elements)
            if node.var_type.size is not None and actual_size != node.var_type.size:
                self._add_error(SemanticErrorType.TYPE_MISMATCH,
                            f"Array size mismatch: declared {node.var_type.size}, "
                            f"initialized with {actual_size} elements")
```

This validation prevents common errors like parameter shadowing and array size mismatches at compile time.

4.3.2 Function Analysis

Function validation employs a two-pass approach:

```python
def visit_program(self, node: Program):
    # First pass: collect all function declarations
    for stmt in node.statements:
        if isinstance(stmt, FunctionDeclaration):
            self._declare_function(stmt)

    # Second pass: analyze function bodies
    for stmt in node.statements:
        if isinstance(stmt, FunctionDeclaration):
            self.visit_function_declaration(stmt)
        else:
            self.visit_statement(stmt)
```

This strategy enables:

- **Forward references**: Functions can call other functions defined later
- **Mutual recursion**: Functions can call each other without ordering constraints
- **Early error detection**: Function signature conflicts are caught before body analysis

4.3.3 Return Statement Validation

The analyzer tracks return statements to ensure functions fulfill their contracts:

```
def visit_function_declaration(self, node: FunctionDeclaration):
    self.current_return_type = node.return_type
    self.function_has_return = False

    # Analyze function body
    self.visit_block(node.body)

    # Check return requirement
    if node.return_type != "void" and not self.function_has_return:
        self._add_error(SemanticErrorType.MISSING_RETURN,
                      f"Function '{node.name}' must return a value of type '{node.return_type}'")
```

This ensures that non-void functions always return appropriate values, preventing undefined behavior at runtime.

4.3.4 Built-in Function Validation

Built-in functions receive special validation with precise argument checking:

```
def _initialize_builtins(self):
    builtins = {
        "__print": (["int", "float", "bool", "colour"], "void"),
        "__write": (["int", "int", "colour"], "void"),
        "__write_box": (["int", "int", "int", "int", "colour"], "void"),
        "__width": ([], "int"),
        "__height": ([], "int"),
        "__read": (["int", "int"], "colour"),
        "__randi": (["int"], "int"),
    }
```

Each built-in has a precise signature that the analyzer enforces, preventing runtime errors from incorrect hardware instruction usage.

4.4 Design Decisions

4.4.1 Visitor Pattern Implementation

The visitor pattern was chosen for several compelling reasons:

```
def visit_statement(self, node: ASTNode):
    if isinstance(node, VariableDeclaration):
        self.visit_variable_declaration(node)
    elif isinstance(node, Assignment):
        self.visit_assignment(node)
    # ... dispatch to appropriate visitor method
```

Benefits include:

- **Separation of concerns**: AST structure is independent of operations
- **Extensibility**: New analysis passes can be added without modifying AST nodes
- **Code reuse**: The same visitor framework supports both semantic analysis and code generation

4.4.2 Error Categorization

Errors are systematically categorized for clear reporting:

```
class SemanticErrorType(Enum):
    REDECLARATION = auto()
    UNDECLARED_VARIABLE = auto()
    UNDECLARED_FUNCTION = auto()
    TYPE_MISMATCH = auto()
    INVALID_ASSIGNMENT = auto()
    MISSING_RETURN = auto()
    INVALID_CAST = auto()
    ARGUMENT_COUNT_MISMATCH = auto()
    ARGUMENT_TYPE_MISMATCH = auto()
    INVALID_BUILTIN_ARGS = auto()
```

This categorization enables:

- **Targeted error messages**: Each error type has specific, helpful messaging
- **Error statistics**: Compilers can report error summaries by category
- **IDE integration**: Different error types can be highlighted differently

4.4.3 Array Assignment Restrictions

A deliberate restriction prevents whole-array assignment:

```
if isinstance(target_type, ArrayType):
    self._add_error(SemanticErrorType.INVALID_ASSIGNMENT,
                f"Cannot assign to entire array '{node.target.name}'. "
                f"Use array indexing for element assignment.")
```

Justification: This restriction prevents ambiguity about array copying semantics and aligns with the stack-based VM's limitations. Programmers must explicitly copy array elements, making performance implications clear.

4.4.4 Implicit Type Conversion Philosophy

The analyzer strictly prohibits implicit type conversions:

Justification: While languages like C perform implicit conversions, PArL's strict approach:

- **Prevents subtle bugs**: No unexpected precision loss or sign changes
- **Makes intent explicit**: Programmers must use `as` to convert types
- **Simplifies type checking**: No complex conversion hierarchies to consider
- **Improves readability**: Type conversions are visible in the code

The semantic analyzer successfully enforces PArL's type system while providing clear, actionable error messages. Its design balances theoretical correctness with practical usability, creating a robust middle layer that ensures only well-typed programs proceed to code generation.

5 Task 4: PArIR Code Generation

5.1 Code Generation Strategy

The code generator implements a sophisticated stack-based virtual machine targeting strategy with two key innovations: systematic frame-level calculation for variable access and a two-pass approach for accurate jump distance computation. This design ensures correct execution on the PAD2000c VM while maintaining clear separation between compile-time and runtime concerns.

5.1.1 Two-Pass Compilation Architecture

A critical design decision was implementing function size pre-calculation:

```python
def generate(self, ast: Program) -> List[str]:
    # Calculate main header jump distance
    self._emit(".main")
    main_header_jump = self._calculate_main_header_jump_distance()
    self._emit(f"push {main_header_jump}")
    self._emit("jmp")
    self._emit("halt")

    # Calculate function sizes before generation
    self._calculate_function_jump_distances(ast)
    self._generate_program(ast)
```

This approach solves the forward reference problem: functions must be skipped during linear execution, but their sizes aren't known until they're generated. The dry-run pass simulates generation to calculate exact instruction counts.

5.1.2 Memory Model and Frame Management

The generator maintains a sophisticated memory model with multiple tracking structures:

```
@dataclass
class MemoryLocation:
    frame_index: int      # Position within frame
    frame_level: int      # Distance from current frame
    size: int = 1         # For arrays

class PArIRGenerator:
    def __init__(self):
        self.memory_stack: List[Dict[str, MemoryLocation]] = []
        self.current_frame_level = -1
        self.frame_var_counts: List[int] = []
        self.next_var_indices: List[int] = []
```

The separation of `frame_index` and `frame_level` is crucial:

- `frame_index`: Static position within a frame, determined at compile time
- `frame_level`: Dynamic distance between frames, calculated during code generation

5.1.3 Systematic Frame Level Calculation

Variable lookup implements precise frame-level semantics:

```
def _lookup_variable(self, name: str) -> Optional[MemoryLocation]:
    for scope_index in range(len(self.memory_stack) - 1, -1, -1):
        if name in self.memory_stack[scope_index]:
            stored_location = self.memory_stack[scope_index][name]

            # Frame level = distance from current execution to variable storage
            current_execution_depth = len(self.memory_stack) - 1
            variable_storage_depth = scope_index
            frame_level = current_execution_depth - variable_storage_depth

            return MemoryLocation(stored_location.frame_index,
                                  frame_level, stored_location.size)
```

This systematic calculation ensures correct variable access across nested scopes, a common source of bugs in stack-based compilers.

5.2 Instruction Generation

5.2.1 Expression Evaluation Strategy

The generator uses postfix evaluation for expressions, naturally mapping to stack operations:

```
def _generate_binary_op(self, node: BinaryOperation):
    # Non-commutative operations require specific order
    if node.operator in ['-', '/', '%', '<', '>', '<=', '>=']:
        self._generate_expression(node.right)  # B first (bottom of stack)
        self._generate_expression(node.left)   # A second (top of stack)
    else:
        # Commutative operations
        self._generate_expression(node.right)
        self._generate_expression(node.left)

    # Generate operation
    op_map = {
        '+': 'add', '-': 'sub', '*': 'mul', '/': 'div', '%': 'mod',
        '<': 'lt', '>': 'gt', '<=': 'le', '>=': 'ge', '==': 'eq'
    }
    self._emit(op_map.get(node.operator, 'nop'))
```

The operand ordering for non-commutative operations ensures that `A - B` correctly computes A minus B, not B minus A, despite stack evaluation order.

5.2.2 Control Flow Translation

Control structures require careful jump distance calculation:

```
def _generate_if_stmt(self, node: IfStatement):
    self._generate_expression(node.condition)

    # If true, skip over else jump setup (push + jmp = 2 instructions)
    true_branch_skip = 4  # skip: push #PC+4, cjmp, push #PC+X, jmp
    self._emit(f"push #PC+{true_branch_skip}")
    self._emit("cjmp")

    else_jump_addr = self._get_current_address()
    self._emit("push #PC+999")  # Placeholder for else jump
    self._emit("jmp")

    self._generate_block_with_frame(node.then_block)

    # Patch the jump distance after generating then block
    else_offset = self._get_current_address() - else_jump_addr
    self.instructions[else_jump_addr] = f"push #PC+{else_offset}"
```

The use of `#PC+X` notation provides position-independent code, while placeholder patching handles forward jumps elegantly.

5.2.3 Built-in Statement Translation

Built-in functions require specific argument ordering for the stack-based VM:

```
def _generate_write_box_stmt(self, node: WriteBoxStatement):
    # Corrected order based on VM expectations
    self._generate_expression(node.color)   # Color on top
    self._generate_expression(node.width)   # Width
    self._generate_expression(node.height)  # Height
    self._generate_expression(node.y)       # Y coordinate
    self._generate_expression(node.x)       # X coordinate
    self._emit("writebox")
```

This systematic reversal ensures arguments are popped in the correct order by the VM instruction.

5.3 Function Handling

5.3.1 Function Size Calculation

The dry-run mechanism accurately predicts function sizes:

```python
def _dry_run_function_body(self, func_node: FunctionDeclaration) -> int:
    # Save current state
    saved_state = self._save_state()

    # Enter dry-run mode
    self.dry_run_mode = True
    self.instructions = []

    try:
        # Generate function body
        self._generate_function_declaration(func_node)
        body_size = len(self.instructions)
    finally:
        # Restore state
        self._restore_state(saved_state)

    return body_size
```

This approach handles complex functions with nested control structures, ensuring accurate jump distances for function skipping.

5.3.2 Parameter Passing and Frame Setup

Function calls implement a sophisticated parameter passing mechanism:

```python
def _generate_function_call(self, node: FunctionCall):
    total_param_count = 0

    for arg in reversed(node.arguments):
        if isinstance(arg, Identifier):
            is_array, array_size = self._is_array_variable(arg.name)
            if is_array:
                # Push array elements individually
                for i in range(array_size - 1, -1, -1):
                    self._emit(f"push {i}")
                    self._emit(f"push +[{location.frame_index}:{location.frame_level}]")
                total_param_count += array_size
            else:
                self._generate_expression(arg)
                total_param_count += 1

    self._emit(f"push {total_param_count}")
    self._emit(f"push .{node.name}")
    self._emit("call")
```

Arrays are passed element-by-element in reverse order, ensuring correct reconstruction in the called function's frame.

5.3.3 Variable Allocation Strategy

The generator implements a systematic allocation strategy for main program variables:

```
def _count_main_variables(self, statements: List[ASTNode]) -> int:
    direct_vars = 0

    for stmt in statements:
        if isinstance(stmt, VariableDeclaration):
            if isinstance(stmt.var_type, ArrayType):
                direct_vars += stmt.var_type.size
            else:
                direct_vars += 1

    # Reserve space for function call setup
    return direct_vars + 1
```

The +1 reservation ensures space for function call parameter setup without overlapping with program variables.

5.4 Key Implementation Decisions

5.4.1 Array Storage Order

Arrays are stored in reverse order during initialization:

```
def _generate_var_decl(self, node: VariableDeclaration):
    if isinstance(node.initializer, ArrayLiteral):
        # Store in REVERSE order
        for elem in reversed(node.initializer.elements):
            self._generate_expression(elem)

        self._emit(f"push {len(node.initializer.elements)}")
        self._emit(f"push {location.frame_index}")
        self._emit(f"push {location.frame_level}")
        self._emit("sta")
```

Justification: This matches the VM's `sta` instruction expectations, which pops and stores elements in order, effectively reversing the stack order.

5.4.2 Scope-Respecting Variable Counting

Variable counting respects scope boundaries:

```
def _count_variable_declarations(self, statements: List[ASTNode]) -> int:
    count = 0

    def count_in_node(node):
        nonlocal count
        if isinstance(node, VariableDeclaration):
            # Count variable space
            count += node.var_type.size if isinstance(node.var_type, ArrayType) else 1
        elif isinstance(node, (ForStatement, IfStatement, WhileStatement)):
            # These create their own scopes - don't traverse
            pass
```

This prevents over-allocation by not counting variables in nested scopes, which manage their own frames.

5.4.3 Modulo Operation Support

The generator includes full modulo support, addressing a language specification gap:

```
op_map = {
    '+': 'add', '-': 'sub', '*': 'mul', '/': 'div', '%': 'mod',
    # ... other operations
}
```

Justification: Modulo is essential for many algorithms, particularly in graphics programming where wrapping and cycling are common patterns.

5.4.4 Optimization Decisions

Several optimizations improve generated code quality:

1. Direct Constant Loading

```
def _generate_literal(self, node: Literal):
    if node.literal_type == "bool":
        self._emit(f"push {1 if node.value else 0}")
    else:
        self._emit(f"push {node.value}")
```

2. Minimal Frame Allocation The generator allocates exactly the space needed:

- Functions: parameters + locals
- Blocks: only local variables
- Main: variables + 1 for function calls

3. Early Address Calculation Jump addresses are calculated during generation rather than using labels, producing position-independent code suitable for the VM's execution model.

5.4.5 Error Prevention Through Design

The generator prevents several classes of runtime errors:

1. Frame Level Consistency: Systematic calculation ensures variables are always accessed at the correct frame level.

2. Array Bounds: Size information is preserved, enabling future bounds checking.

3. Function existence: Function addresses are validated during generation.

The code generator successfully translates validated ASTs into efficient PArIR instructions. Its systematic approach to frame management, careful handling of the stack-based execution model, and sophisticated function compilation create a robust backend that produces correct, efficient code for the PAD2000c virtual machine.

6 Task 5: Array Support

6.1 Array Implementation

The compiler implements comprehensive array support through systematic enhancements across all compilation phases. Arrays in PArL are fixed-size, first-class values that can be declared, initialized, passed to functions, and accessed element-by-element. The implementation required careful coordination between the parser, semantic analyzer, and code generator to ensure type safety and efficient execution.

6.1.1 Array Type Representation

Arrays are represented as a distinct type in the compiler's type system:

```python
class ArrayType:
    def __init__(self, element_type: str, size: Optional[int] = None):
        self.element_type = element_type
        self.size = size  # None for dynamic arrays in parameters
        self.is_array = True

    def __eq__(self, other):
        if isinstance(other, str):
            return False  # Arrays never equal to primitive types
        return (self.element_type == other.element_type and
                self.size == other.size)
```

This design enables:

- **Type safety**: Arrays of different element types are incompatible
- **Size flexibility**: Dynamic array parameters accept any size
- **Clear distinction**: Arrays are never confused with scalar types

6.1.2 Declaration and Initialization Syntax

The parser supports multiple array declaration patterns:

```
# Fixed-size declaration with initialization
let arr:int[5] = [1, 2, 3, 4, 5];

# Dynamic-size declaration (size inferred)
let data:int[] = [10, 20, 30];

# Function parameter arrays
fun process(values:int[8]) -> int { ... }
```

The grammar extension seamlessly integrates array syntax while maintaining parsing simplicity.

6.1.3 Array Literal Parsing

Array literals require special handling in expression parsing:

```python
def parse_array_literal(self) -> ArrayLiteral:
    start_token = self.stream.expect(TokenType.LBRACKET)
    elements = []

    if not self.stream.match(TokenType.RBRACKET):
        elements.append(self.parse_expression())
        while self.stream.match(TokenType.COMMA):
            self.stream.advance()
            elements.append(self.parse_expression())

    self.stream.expect(TokenType.RBRACKET)
    return ArrayLiteral(elements, start_token.line, start_token.col)
```

This implementation supports arbitrary expressions as array elements, enabling computed initialization.

6.2 Memory Management

6.2.1 Array Storage Layout

A critical design decision was storing array elements in reverse order:

```
def _generate_array_literal(self, node: ArrayLiteral):
    # Push elements in REVERSE order
    for elem in reversed(node.elements):
        self._generate_expression(elem)

    self._emit(f"push {len(node.elements)}")
```

Justification: The PArIR `sta` instruction pops values from the stack and stores them sequentially in memory. By pushing elements in reverse order, they end up in the correct forward order in memory after `sta` execution. This approach avoids requiring a separate reverse instruction.

6.2.2 Memory Allocation Strategy

Arrays consume contiguous memory within stack frames:

```
def _allocate_variable(self, name: str, index: Optional[int] = None,
                      size: int = 1) -> MemoryLocation:
    if index is None:
        index = self.next_var_indices[-1]
        self.next_var_indices[-1] += size  # Advance by array size

    location = MemoryLocation(index, self.current_frame_level, size)
    self.memory_stack[-1][name] = location
    return location
```

This ensures:

- **Predictable layout**: Array elements are always adjacent
- **Efficient access**: Index arithmetic gives direct element addresses
- **Size tracking**: The compiler knows array bounds for validation

6.2.3 Parameter Space Calculation

Functions with array parameters require careful space calculation:

```
def _get_param_size(self, param_type):
    if isinstance(param_type, ArrayType):
        size = param_type.size if param_type.size else 1
        return size
    elif isinstance(param_type, str) and '[' in param_type:
        # Handle string representation like "int[8]"
        size_str = param_type.split('[')[1].split(']')[0]
        return int(size_str)
    return 1
```

This handles both parsed `ArrayType` objects and string representations, ensuring robust parameter handling.

6.3 Code Generation Extensions

6.3.1 Array Element Access

Element access uses the `push +[i:l]` instruction for efficient indexed reading:

```
def _generate_index_access(self, node: IndexAccess):
    if isinstance(node.base, Identifier):
        location = self._lookup_variable(node.base.name)
        if location:
            # Generate index expression
            self._generate_expression(node.index)
            # Use push +[i:l] for array element access
            self._emit(f"push +[{location.frame_index}:{location.frame_level}]")
```

The VM adds the index from the stack to the base address, providing O(1) element access.

6.3.2 Array Assignment Handling

Element assignment requires computing the target address:

```
def _generate_assignment(self, node: Assignment):
    if isinstance(node.target, IndexAccess):
        location = self._lookup_variable(node.target.base.name)
        if location:
            self._generate_expression(node.value)     # Value to store
            self._generate_expression(node.target.index)  # Index
            self._emit(f"push {location.frame_index}")
            self._emit("add")  # Compute element address
            self._emit(f"push {location.frame_level}")
            self._emit("st")
```

This pattern computes `base + index` at runtime, supporting dynamic index expressions.

6.3.3 Array Parameter Passing

Arrays are passed by value, with elements pushed individually:

```python
def _generate_function_call(self, node: FunctionCall):
    for arg in reversed(node.arguments):
        if is_array:
            # Push array elements in reverse order
            for i in range(array_size - 1, -1, -1):
                self._emit(f"push {i}")
                self._emit(f"push +[{location.frame_index}:{location.frame_level}]")
            total_param_count += array_size
```

Design rationale:

- **Value semantics**: Arrays are copied, preventing unexpected mutations
- **Simplicity**: No pointer management or reference counting needed
- **Safety**: Called functions cannot modify caller's arrays

6.3.4 Semantic Validation for Arrays

The semantic analyzer enforces strict array typing:

```python
def visit_array_literal(self, node: ArrayLiteral) -> Optional[ArrayType]:
    if not node.elements:
        self._add_error("Empty array literals not allowed")
        return None

    # Determine type from first element
    first_type = self.visit_expression(node.elements[0])

    # Verify all elements match
    for i, elem in enumerate(node.elements[1:], 1):
        elem_type = self.visit_expression(elem)
        if not TypeChecker.types_equal(elem_type, first_type):
            self._add_error(f"Array element {i} type mismatch")

    return ArrayType(first_type, len(node.elements))
```

This ensures:

- **Homogeneous arrays**: All elements must have the same type
- **Non-empty arrays**: Prevents ambiguous empty array types
- **Compile-time size**: Array sizes are known statically

6.3.5 Array Size Compatibility

The type system implements flexible but safe array passing:

```python
# In TypeChecker.types_equal()
if isinstance(type1, ArrayType) and isinstance(type2, ArrayType):
    element_types_match = type1.element_type == type2.element_type

    if type2.size is None:  # Parameter is dynamic array
        return element_types_match  # Any size accepted
    elif type1.size is None:  # Argument is dynamic, parameter is fixed
```

```
        return False  # Cannot pass dynamic to fixed
    else:  # Both are fixed-size
        return element_types_match and type1.size == type2.size
```

This design enables generic array functions while maintaining type safety.

6.3.6 Limitations and Design Trade-offs

Several deliberate limitations simplify the implementation:

1. No Whole-Array Assignment

```
if isinstance(target_type, ArrayType):
    self._add_error("Cannot assign to entire array. Use element assignment.")
```

Justification: Prevents ambiguity about deep vs. shallow copying and aligns with the element-wise VM model.

2. No Multi-Dimensional Arrays The implementation supports only one-dimensional arrays.

Justification: Simplifies type checking and code generation while covering most use cases. Multi-dimensional arrays can be simulated using index arithmetic.

3. Static Sizes Only Arrays must have compile-time known sizes.

Justification: Enables stack-based allocation without heap management, simplifying memory management and improving performance.

4. Pass-by-Value Semantics Arrays are copied when passed to functions.

Justification: Prevents aliasing issues and makes function behavior predictable, though at the cost of performance for large arrays.

The array implementation successfully extends PArL with a powerful but manageable array feature. The design balances expressiveness with implementation complexity, providing programmers with essential array functionality while maintaining the language's simplicity and safety guarantees. The systematic approach across all compiler phases ensures arrays integrate seamlessly with the existing language features.

7 Testing and Validation

7.1 Running the Tests

All test suites are managed using a main python script and a menu showing options for running these tests can be generated by running python `-m test.run_all_tests`. Tests can be run in isolation, or all at once, and an option at the terminal exists for toggling the AST output. The tests will output to a designated folder which includes a summary of the program, its output, the AST (if toggled) and most tests have some aspect of automated testing where each test is given a pass or a fail based on whether some required and expected output is (e.g. if the input includes '(' then the output must include the LPAREN token for a lexer test). Outputs with print statements or ones that output visuals were tested on the simulator. A main focus of the testing was to ensure that specific safeguards and functionality work systematically not just for some given programs. Please note that the ASTs , program inputs and output not displayed here can be found in their respective folder in my deliverables. Any outputs from the simulator can be observed in the video for verification.

7.2 Task 1 Tests – Lexical Analysis and Tokenization

The `test_task1.py` suite rigorously verifies the deterministic lexer implemented in Task 1. These tests evaluate the lexer against the assignment's EBNF grammar and specification by validating:

- Comprehensive recognition of all legal token types (keywords, built-ins, types, literals, operators, punctuation).
- Detection and classification of lexical errors such as malformed literals and invalid comment structures.
- Proper filtering and ignoring of comments in all forms.
- Robustness to edge cases and malformed input.

7.2.1 Lexer Example 1 – Valid Token Coverage

This test provides a linearly structured input that includes every token type specified in the language: keywords (e.g., `if`, `while`), data types (e.g., `int`, `colour`), built-in functions (e.g., `__print`, `__delay`), all valid operators and punctuation, as well as legal literals (e.g., integers, floats, booleans, and colour hex codes). It confirms that:

- All tokens are correctly classified by the lexer.
- No unexpected errors are emitted.
- Token counts match exactly what is expected.

Outcome: All tokens are correctly identified with precise category labels, validating that the finite-state automata constructed for the lexer are sound and EBNF-compliant.

Specification coverage: Legal token classification, full language coverage.

7.2.2 Lexer Example 2 – Lexical Error Detection

This test introduces several known lexical errors into the input:

- A float literal ending with a decimal point.
- A colour literal with invalid hexadecimal characters.
- An unterminated block comment.
- A stray closing block comment.
- A nested block comment (not supported by the specification).

Outcome: The lexer detects and emits distinct error tokens for each case: `ERROR_INVALID_FLOAT`, `ERROR_INVALID_COLOUR`, `ERROR_NESTED_COMMENT`, and `ERROR_STRAY_COMMENT_CLOSE`. It recovers gracefully without misclassifying subsequent tokens.

Specification coverage: Error token classification, malformed literal recognition, invalid comment handling.

7.2.3 Lexer Example 3 – Comment Handling

This test checks the lexer's ability to correctly ignore both line comments (prefixed with `//`) and block comments (enclosed in `/* ... */`). It includes:

- Line comments both at the start and end of lines.
- Block comments spanning multiple lines.
- Nested comment-like formatting.
- Comments with embedded punctuation and symbols.

Outcome: All comment sections are correctly ignored, and only valid code tokens are emitted. Comments do not interfere with the lexer's logic or token counts.

Specification coverage: Full comment support, comment skipping, robustness against embedded symbols.

7.2.4 Lexer Example 4 – Literal Edge Cases

This test case mixes valid and invalid literals to probe the lexer's boundaries. It includes:

- Valid integer and float literals across a range of values.
- Legal colour literals in upper and lower case.
- Malformed literals such as a float with a trailing dot, an invalid hex colour, and a colour that is too short.

Outcome: Valid literals are identified with correct type annotations. Invalid cases produce well-typed error tokens. This confirms the lexer's edge case handling and adherence to type-literal formatting rules.

Specification coverage: Literal recognition, hex parsing robustness, and type-sensitive error validation.

7.3 Task 2 Tests – LL(1) Parser and AST Construction

The `test_task2.py` suite thoroughly validates the LL(1) recursive descent parser by checking the correct construction of Abstract Syntax Trees (ASTs) for syntactically valid programs and error handling for malformed code. Each test case targets one or more constructs from the assignment's EBNF grammar. For sample AST outputs, refer to **Appendix B**.

7.3.1 Parser Example 1 – Expressions and Operator Precedence

This test checks the parser's ability to correctly interpret expressions containing:

- Nested arithmetic operations with mixed precedence levels (e.g., multiplication binding tighter than addition).
- Logical operators including `and`, `or`, and `not`, ensuring correct associativity.
- Use of parentheses to override default precedence.

- Type casting using the `as` operator integrated into expressions.

Outcome: The parser builds fully parenthesized ASTs that honor precedence and associativity rules. Binary and unary expressions are properly nested, and cast operations are placed as subtrees. See **Appendix A, Parser Example 1**.

Specification coverage: Binary/unary operations, operator precedence, type casting, and parentheses.

7.3.2 Parser Example 2 – Function Declaration Parsing

This test validates the parsing of:

- Functions with zero or multiple parameters.
- Type annotations for parameters and return types.
- Return statements, including nested expressions.
- Conditional logic and built-in calls inside function bodies.
- Recursive functions with base cases and recursive calls.

Outcome: The AST reflects correct function structure, with typed parameter lists, typed return values, and block-scoped bodies. Nested conditionals and recursive calls are handled as subtrees within return expressions. See **Appendix A, Parser Example 2**.

Specification coverage: Function declaration, parameters, return types, function bodies, and recursion.

7.3.3 Parser Example 3 – Control Flow Statements and Nesting

This test verifies the parser's handling of:

- Simple `if` and `if-else` conditionals.
- `while` loops and `for` loops with full and partial clause support.
- Nested loops and conditionals inside loop bodies.
- Built-in function calls embedded in control flows.

Outcome: All control structures are parsed into separate AST nodes with correct linkage of condition, body, and (for loops) initialization/update clauses. Nested control flows are deeply structured and scoped appropriately. See **Appendix B, Parser Example 3**.

Specification coverage: Conditional blocks, while/for loops, optional clauses, and block nesting.

7.3.4 Parser Example 4 – Syntax Error Detection and Recovery

This test injects various syntax errors, including:

- Missing semicolons or colons in declarations.
- Missing function or loop bodies.
- Unbalanced parentheses or braces.
- Incomplete expressions or missing type annotations.

Outcome: The parser emits detailed error messages specifying the nature of each error (e.g., unexpected token, unexpected EOF) and where it occurred. Recovery is implemented, allowing parsing to continue for subsequent valid code.

Specification coverage: Syntax error handling, token expectation tracing, error recovery strategies, and parser resilience.

7.4 Task 3 Tests – Semantic Analysis (Type Checking, Scoping, Functions, Built-ins)

This section validates the functionality of the semantic analysis phase, including type checking, scope resolution, function validation, and built-in usage. Each test corresponds to a sample PArL program and confirms that the semantic analyzer correctly builds ASTs and applies semantic rules as per the language specification.

7.4.1 Semantic Analyzer Example 1 – Type Checking and Compatibility

This test validates:

- Type-safe operations such as arithmetic, boolean logic, and comparisons.
- Valid type casting between base types and custom types.
- Detection of invalid operations including type mismatches and logical errors.
- Enforcement of type rules for assignments and expressions.

Outcome: The semantic analyzer correctly validated all type-safe expressions and detected type mismatches involving arithmetic with booleans, invalid comparisons, and assignments with incompatible types. All reported errors matched specification expectations.

7.4.2 Semantic Analyzer Example 2 – Scope Management and Variable Declaration

This test demonstrates:

- Scope resolution and shadowing across global, function, and block levels.
- Parameter redefinition within local scopes and variable redeclaration detection.
- Access to out-of-scope identifiers and undefined variable usage.

- Robust semantic symbol table management with nested block support.

Outcome: The analyzer successfully tracked scope hierarchies, caught redefinitions, and flagged use of undefined identifiers. Shadowing was handled correctly across nested blocks and functions.

7.4.3 Semantic Analyzer Example 3 – Function Validation

This test case checks:

- Function signature correctness and parameter type matching.
- Return type verification, including detection of incorrect and missing returns.
- Identification of unreachable code after terminal statements.
- Valid recursive function analysis and proper function call evaluation.
- Handling of calls to undeclared functions and invalid argument usage.

Outcome: The semantic analyzer correctly flagged:

- A missing return in a non-void function.
- A function returning an incorrect type.
- Unreachable code after a return.
- Mismatched argument types and arities in function calls.
- Use of undeclared functions.

Recursive functions and valid call chains were successfully accepted.

7.4.4 Semantic Analyzer Example 4 – Built-in Function Validation

This test exercises:

- Type validation of parameters for built-in statements such as `__print`, `__write`, `__delay`, `__read`, and `__randi`.
- Return type checking and assignment validation for built-in expressions.
- Identification of invalid argument types, incorrect parameter counts, and return type mismatches.
- Enforcement of semantic constraints on all built-in usage scenarios.

Outcome: All correct built-in usages were accepted. The analyzer correctly flagged:

- Argument type mismatches (e.g., floats passed where integers are required).
- Cast errors (e.g., `bool` to `colour`).
- Invalid return assignments (e.g., assigning `__width` to a `float`).
- Use of undeclared variables.

Built-in validation succeeded with all expected semantic errors reported.

7.5 Task 4 Tests – PArIR Code Generation

The `test_task4.py` suite rigorously validates PArIR instruction generation, confirming that the compiler produces correct low-level output for valid PArL programs. Tests cover memory operations, arithmetic, control flow, function call semantics, and built-in operations. Each test executes the full compilation pipeline from lexing to code generation and inspects the emitted instructions and functional behavior.

7.5.1 Code Generation Example 1 – Basic Arithmetic and Memory Operations

This test validated:

- Arithmetic instruction generation for `add`, `sub`, `mul`, `div`, `mod`.
- Type-correct casting and storage instructions (`st`, `push`, `oframe`).
- Proper memory slot usage and layout matching declared variable types.

Outcome: All arithmetic operations emitted correct instruction sequences. Casting instructions were correctly placed. Frame initialization and value assignments were matched to variable declarations. All arithmetic expressions compiled as expected.

7.5.2 Code Generation Example 2 – Control Flow Code Generation

This test verified:

- Accurate generation of conditional jumps (`cjmp`) and unconditional jumps (`jmp`).
- Loop code layout for `while` and `for`, including proper back-jumps and re-evaluation.
- Logical conditions (e.g., `eq`, `lt`, `mod`) evaluated and used in branches.

Outcome: Branch instructions accurately reflected AST structure. Nested control blocks and loop bodies were compiled into structurally correct PArIR. Frame management (`oframe`, `cframe`) preserved execution scope. Additionally, the generated code was tested on the simulator and the correct output (1,2,3,4,5,0,1,2) was observed in the logs section.

7.5.3 Code Generation Example 3 – Function Calls and Parameter Passing

This test checked:

- Generation of function prologues and epilogues with correct label names and memory allocation.
- Correct compilation of calls (`call`), argument setup (`push`), and return instructions (`ret`).
- Verification of return values and stack frame teardown.

Outcome: Function blocks were isolated with unique labels. Each function call was preceded by the appropriate `push` instructions for argument values, and return values were stored correctly. Recursive and branching return paths compiled properly. The expected output for the program was observed in the simulator's logs (1,50).

7.5.4 Code Generation Example 4 – Built-in Operations Code Generation

This test included:

- Printing, screen operations, and timing functions: `__print`, `__delay`, `__clear`.
- Graphics output with `__write`, `__write_box`, and random inputs.
- Reading screen data and dynamic variable creation using `__width`, `__height`, `__read`, `__randi`.

Outcome: All 9 expected built-in instructions were correctly emitted. Built-in operations were tracked and matched to AST structure. Variable assignments from built-ins succeeded, including those used within loops. The dynamic behavior involving random numbers and screen dimensions compiled without issues.

Specification Coverage:

- Arithmetic and memory operations.
- Structured control flow translation.
- Function call semantics and stack management.
- Comprehensive support for built-in graphics and utility operations.

7.6 Task 5 Tests – Array Support and PArIR Code Generation

The `test_task5.py` suite validates support for arrays in both semantic analysis and code generation. This includes array declaration, initialization, element access, assignments, and function interactions. It also confirms correct PArIR generation for array-based operations.

7.6.1 Code Generation Example 1 – Array Declarations and Initialization

This test verifies:

- Declaration of arrays with and without initial values.
- Support for all primitive types: `int`, `float`, `bool`, and `colour`.
- Correct interpretation of array sizes and literal sequences.

Outcome: Arrays of different sizes and element types were declared and initialized. PArIR correctly allocated memory and pushed literals to the correct locations. Compilation was successful and semantic checks passed.

7.6.2 Code Generation Example 2 – Array Indexing and Element Access

This test focuses on:

- Array indexing syntax for reading and writing.
- Assignments to and from array elements.
- Usage of array elements inside expressions and control flows.

Outcome: Element assignments emitted `sta` instructions, and element accesses used `push +[...]` correctly. Index expressions within loops and blocks were compiled as expected, and all checks passed successfully. Dry running the program gave an expected output of 100,100,100,300,20 and 50 which was correctly observed in the simulator when running the generated code, indicating that arrays can successfully handle indexing and printing after arithmetic.

7.6.3 Code Generation Example 3 – Array Parameters in Functions

This test evaluates:

- Passing arrays into functions.
- Reading and updating array values inside function bodies.
- Use of control flow to traverse and process arrays.

Outcome: Functions like `sum_array` and `double_values` compiled correctly. Parameter bindings and index-based updates were reflected in both AST and PArIR. The program passed all tests, including nested block handling. Additionally, this program was also tested in the simulator to test whether my implementation can handle arrays as parameters and returns, the simulator returned the expected output of 100, 200, 150, 20 and 200.

7.6.4 Code Generation Example 4 – Array Return and End-to-End Integration

This test confirms:

- Arrays can be passed into and returned from functions.
- Intermediate array processing inside function bodies is correctly tracked semantically.
- Function calls involving arrays produce valid PArIR.

Outcome: The test passed successfully. Functions such as `get_first` returned array elements, and the call site received and printed the correct value. All instructions (e.g., `call`, `ret`, `push +[...]` and `print`) were correctly generated and matched expectations. The generated code correctly printed out the number at index 0 (5) in the simulator.

7.6.5 Code Generation Example 5 – Assignment Program: MaxInArray

This test implements the sample program from the assignment specification (page 15) to find the maximum element in an array:

- Iterative traversal of array values using a `for` loop.
- Conditional branching to track maximum value.
- Return of scalar result derived from array analysis.

Outcome: Compilation was successful. The compiler emitted correct loop logic, comparisons, and return instructions. The expected instruction patterns (e.g., `alloc`, `call`, `sta`, and `push +[...]` access) were all present. Simulator output confirmed that array iteration and maximum value tracking matches the expected value (120).

7.7 Assignment Examples

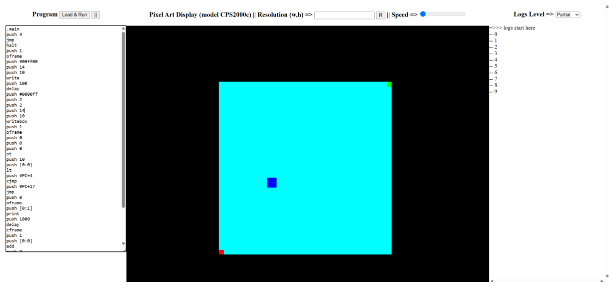
The `test_assignment.py` suite validates whether PArL programs designed for specification compliance compile correctly across all stages: lexing, parsing, semantic analysis, and code generation. Each example corresponds to a real-world task or feature from the assignment specification and confirms that output matched expectations in both AST construction and PArIR generation.

7.7.1 Assignment Example 1 – Function Call and Expression Evaluation

Outcome: Successfully parsed and type-checked global declarations, binary expressions, and function call arguments. Generated valid PArIR instructions for arithmetic and return values, demonstrating correct call/return behavior.

7.7.2 Assignment Example 2 – Built-in Statement Integration

Outcome: Calls to built-in functions such as `__print`, `__delay`, and `__write` were semantically validated and emitted the expected low-level PArIR instructions with appropriate argument handling. When the code runs , the simulator correctly shows the iterative printing of i and the a brief visual green pixel that is overridden by blue pixels as expected.



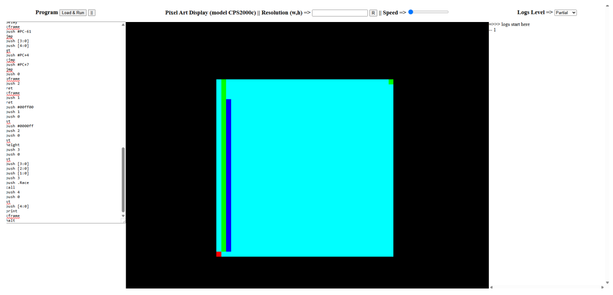
```
.main
push 4
jmp
halt
push 1
oframe
push #00ff00
push 14
push 10
write
push 100
delay
push #0000ff
push 2
push 2
push 14
push 10
writebox
push 1
oframe
push 0
push 0
push 0
st
push 10
push [0:0]
lt
push #PC+4
cjmp
push #PC+17
jmp
push 0
oframe
push [0:1]
print
push 1000
delay
cframe
push 1
push [0:0]
add
```

```
=>>> logs start here
-- 0
-- 1
-- 2
-- 3
-- 4
-- 5
-- 6
-- 7
-- 8
-- 9
```

Figure 1: Assignment Example 2 Output

7.7.3 Assignment Example 3 – Race Function and Simulator Output

Outcome: The graphical race function was parsed, type-checked, and compiled. Built-in calls were correctly mapped, producing a valid race animation in the simulator with visible graphical effects and correct conditional logic output.



```
delay
cframe
push #PC-61
jmp
push [3:0]
push [4:0]
gt
push #PC+4
cjmp
push #PC+7
jmp
push 0
oframe
push 2
ret
cframe
push 1
ret
push #00ff00
push 1
push 0
st
push #0000ff
push 2
push 0
st
height
push 3
push 0
st
push [3:0]
push [2:0]
push [1:0]
push 3
push .Race
call
push 4
push 0
st
push [4:0]
print
cframe
halt
```

```
=>>> logs start here
-- 1
```

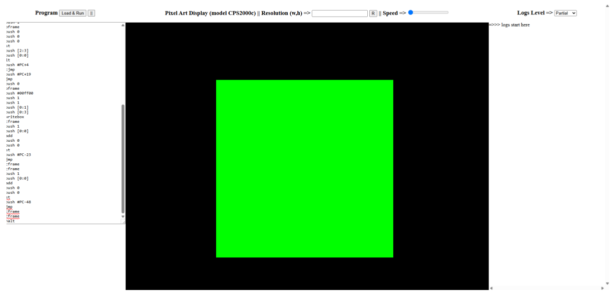
Figure 2: Assignment Example 3 Output

7.7.4 Assignment Example 4 – Scope Error Detection

Outcome: The semantic analyzer flagged scope errors such as the use of undeclared variables and illegal reassignments. No PArIR was generated, confirming proper early-stage failure and diagnostic messaging.

7.7.5 Assignment Example 5 – Graphics Loop (Page 9)

Outcome: Nested `for` loops were compiled into valid instruction blocks using `writebox`. The compiler emitted 72 PArIR instructions that drew coloured pixels using `__width`, `__height`, and 2D traversal over the output screen. Subsequently, a correct all green output was observed in the simulator.



```
push 1
oframe
push 0
push 0
push 0
st
push [2:3]
push [0:0]
lt
push #PC+4
cjmp
push #PC+19
jmp
push 0
oframe
push #00ff00
push 1
push 1
push [0:1]
push [0:3]
writebox
cframe
push 1
push [0:0]
add
push 0
push 0
st
push #PC-23
jmp
cframe
cframe
push 1
push [0:0]
add
push 0
push 0
st
push #PC-48
jmp
cframe
cframe
halt
```

```
=>>> logs start here
```

Figure 3: Assignment Example 5 Output

7.8 Simulator Examples

The `test_simulator.py` suite validates end-to-end compilation of PArL programs intended for graphical or animated output via the PAD2000c simulator. Each test was compiled to valid PArIR and executed successfully in the simulator, with all expected visuals confirmed and documented in the accompanying demonstration video. The figures for the outputs are not shown because they're dynamic, kindly refer to the video.

7.9 Simulator Examples

The `test_simulator.py` suite validates end-to-end compilation of PArL programs designed to produce animated or graphical output using the PAD2000c simulator. Each example was compiled and executed successfully, and expected visuals were confirmed in the accompanying simulator recording.

7.9.1 Simulator Example 1 – Basic Colour Cycling Animation

Outcome: A color array was cycled using a time-based loop to produce a smooth animated effect. The program used nested `for` loops and the `__write_box` operation to redraw colored boxes with offsets. The animation showed fluid color changes across the canvas as expected. Matched simulator output.

7.9.2 Simulator Example 2 – Random Colour Generation and Display

Outcome: Using the built-in function `__randi`, random colour values were generated and drawn with `__write`. Each loop iteration produced unpredictable visuals within expected dimensions. Output confirmed randomness with consistent canvas bounds. Matched the demo video.

7.9.3 Simulator Example 3 – Random Pixel Display

Outcome: Each execution of this program rendered pixels at random locations with varying colors using `__write` and `__randi`. The display appeared chaotic but within expected screen limits. `__width` and `__height` ensured safe coordinate generation. Visuals matched the observed simulator behavior.

7.9.4 Simulator Example 4 – Colour Animation While Loop

Outcome: The animation loop continuously updated pixel regions with random colours using `__write_box`, `__clear`, and `__delay`. Frame transitions were visibly smooth, and colors evolved per loop iteration. The infinite loop design was properly handled in the simulator. Output confirmed in the video.

7.9.5 Simulator Example 5 – Rainbow Pattern Animation

Outcome: The function `draw_pattern` iterated through a color array and canvas dimensions to paint a rainbow grid. An infinite loop with offset shifting caused the pattern to animate horizontally. The visuals showed vibrant diagonal patterns moving over time. Output matched the demo and confirmed correct indexing, looping, and color logic.

7.9.6 Simulator Example 6 – Moving Checkerboard

Outcome: A classic checkerboard pattern was animated by shifting tile positions every frame. This was achieved using size-based divisions and colour alternation logic inside nested loops. The simulator displayed smooth motion of black-and-white squares across the screen. Visual confirmed correctness of control flow, arithmetic, and frame updates.

7.10 Limitations

Despite the comprehensive nature of the compiler implementation, several limitations were encountered. Firstly, while the semantic analyzer performs rigorous type checking, its error recovery remains basic—halting on the first detected semantic error rather than attempting to resume analysis to collect multiple errors. Additionally, support for dynamic memory management is intentionally excluded due to the constraints of the PArL language and target architecture (PAD2000c). The parser assumes well-structured input based on LL(1) principles, meaning malformed or deeply ambiguous code may lead to cascading parse errors that could be mitigated with more sophisticated lookahead or recovery strategies.

8 Conclusion

The compiler satisfies all specified requirements for the PArL-to-PArIR translation pipeline. Each stage, lexical analysis, parsing, semantic validation, and code generation were designed with robustness, modularity, and future extensibility in mind. The simulator examples demonstrated the correctness and completeness of the implementation, while the code structure (e.g., AST node design, visitor pattern usage, and scope tracking) provides a solid foundation for potential future enhancements such as debugger support or intermediate optimization passes. This project effectively bridges high-level language expressiveness with the low-level precision required by the PAD2000c hardware.

9 Appendices

9.1 Appendix A: Program ASTs

9.1.1 Parser Example 1

```
VarDecl: a : int
  BinaryOp: +
    Literal: 2 (int)
    BinaryOp: *
      Literal: 3 (int)
      Literal: 4 (int)
VarDecl: b : bool
  BinaryOp: and
    BinaryOp: <
      Identifier: x
      Identifier: y
    BinaryOp: >
      Identifier: z
      Identifier: w
VarDecl: c : int
  BinaryOp: /
    BinaryOp: *
      BinaryOp: +
        Identifier: a
        Identifier: b
      Identifier: c
    Identifier: d
VarDecl: d : bool
  BinaryOp: or
    UnaryOp: not
      Identifier: x
    BinaryOp: and
      Identifier: y
      Identifier: z
VarDecl: e : int
  BinaryOp: +
    BinaryOp: %
      Identifier: x
      Identifier: y
    BinaryOp: *
      Identifier: z
      Identifier: w
VarDecl: f : bool
  BinaryOp: or
    BinaryOp: and
      BinaryOp: ==
        Identifier: a
        Identifier: b
      BinaryOp: !=
        Identifier: c
        Identifier: d
    BinaryOp: <=
      Identifier: e
      Identifier: f
VarDecl: g : float
  BinaryOp: +
    Cast -> float
      Identifier: x
    Cast -> float
      Identifier: y
```

9.1.2 Parser Example 2

```
FuncDecl: simple() -> int
  Block
    Return
      Literal: 42 (int)
FuncDecl: add(x:int, y:int) -> int
  Param: x : int
  Param: y : int
  Block
    Return
      BinaryOp: +
        Identifier: x
        Identifier: y
FuncDecl: complex(a:int, b:float, c:bool, d:colour) -> bool
```

```
    Param: a : int
    Param: b : float
    Param: c : bool
    Param: d : colour
    Block
      VarDecl: result : bool
        BinaryOp: and
          BinaryOp: >
            Identifier: a
            Literal: 0 (int)
          BinaryOp: >
            Identifier: b
            Literal: 0.0 (float)
      If
        Identifier: c
        Block
          Write
            Literal: 0 (int)
            Literal: 0 (int)
            Identifier: d
      Return
        Identifier: result
  FuncDecl: process_array(data:int[10]) -> int
    Param: data : int[10]
    Block
      Return
        IndexAccess
          Identifier: data
          Literal: 0 (int)
  FuncDecl: factorial(n:int) -> int
    Param: n : int
    Block
      If
        BinaryOp: <=
          Identifier: n
          Literal: 1 (int)
        Block
          Return
            Literal: 1 (int)
      Return
        BinaryOp: *
          Identifier: n
          FuncCall: factorial
            BinaryOp: -
              Identifier: n
              Literal: 1 (int)
```