Compiler Report

Parl Program

Jake Carabott

Introduction

This report outlines the design and implementation of a compiler for the PArL programming language, developed as part of the CPS2000 Compiler Theory and Practice course. PArL is a high-level, strongly typed language that compiles down to PArIR, an intermediate representation designed to run on the PAD2000c pixel-art display simulator.

The compiler was implemented in Python and is structured in a modular fashion, with each compilation stage handled by its own component. The main components of the project are:

* **Lexer (Scanner):** A table-driven finite state automaton that converts the raw input into a sequence of tokens.
* **Parser:** A hand-crafted top-down LL(1) parser that builds an abstract syntax tree (AST) from the token stream.
* **Semantic Analyzer:** Implements a visitor pattern to traverse the AST and perform scope-based type checking, ensuring semantic correctness of the input program.
* **Code Generator:** Another AST visitor that translates the program into PArIR instructions, adhering to the PAD2000c virtual machine architecture.
* **Array Support:** The compiler was extended to support fixed-size arrays, both as variables and as function parameters.

The project includes support for expressions, conditionals, loops, function declarations and calls, built-in PAD2000c instructions (like \_\_print, \_\_write, \_\_delay), and array manipulation. Most features of the language are implemented successfully, including proper scope management and type checking for array structures.

While most language features work as intended, function calls with parameters are not fully supported in code generation due to unresolved stack frame binding issues. Nevertheless, all other compiler components including parsing and semantic validation of such calls are functioning correctly.

The source code is divided into several key files:

* Lexer.py: Tokenization logic and regex-based classification of input.
* Parser.py: Recursive descent parser producing AST nodes.
* AST.py: Definitions of AST node classes and the visitor interface.
* SymbolTable.py: Scope and symbol management logic.
* TypeCheckVisitor.py: Type-checking visitor using the symbol table.
* CodeGenVisitor.py: PArIR code generation visitor.
* main.py: Entry point with example programs and test harness.

The remaining sections of this report will describe each compiler stage (Tasks 1–5) in detail, showing the logic used, representative code snippets, and test results observed via the PAD2000c simulator.

Task 1 – Lexer

Task 1 involved implementing the lexer, which is the first stage of the compiler pipeline. Its purpose is to scan the source code character by character and convert it into a linear stream of tokens that the parser can consume. The lexer is implemented in the Lexer.py file. It uses regular expressions to match input lexemes and maps them to their corresponding token types, defined in an Enum called TokenType. This enum includes over 50 entries and classifies tokens into categories such as keywords (e.g., let, fun, if, return), literals (e.g., integers, floats, booleans, colours), operators (e.g., +, ==, as), delimiters (e.g., (, {, ,), and special built-in functions like \_\_print and \_\_write.

The Lexer class maintains several important attributes. The self.code attribute stores the raw input string. self.position keeps track of the current position during scanning, and self.tokens holds the resulting list of tokens. The class also includes self.current\_token\_index, which is used to iterate through the token stream, and is updated by the GetNextToken() method. The core of the lexical analysis happens in the tokenize() method, which loops through the source string character by character and constructs tokens based on matching patterns. When a token is identified, the method uses get\_token\_type() to determine its TokenType.

class Lexer:

    def \_\_init\_\_(self, code):

        self.code = code

        self.position = 0

        self.tokens = []

        self.current\_token\_index = 0

        self.tokens = self.tokenize()

A notable part of the design is the use of mapping dictionaries to classify keywords, operators, and delimiters quickly. For example, the keyword\_map converts a string like "let" to TokenType.let. This same strategy is used for operator and delimiter detection. These maps simplify classification logic and allow new keywords or symbols to be added easily.

keyword\_map = {

    "fun": TokenType.fun,

    "if": TokenType.if\_,

    "else": TokenType.else\_,

    "while": TokenType.while\_,

    "for": TokenType.for\_,

    "return": TokenType.return\_,

    "let": TokenType.let,

    "int": TokenType.int\_,

    "float": TokenType.float\_,

    "bool": TokenType.bool,

    "colour": TokenType.colour,

    "true": TokenType.true,

    "false": TokenType.false

}

The tokenize() method is the main loop that drives the lexical analysis. It handles whitespace and comment skipping, detects compound tokens like ->, matches identifiers and keywords using regular expressions, and parses numeric and colour literals. Each recognized element is converted into a Token object and appended to the tokens list. When no known pattern is matched, the lexer produces an ERROR token to flag invalid input.

def tokenize(self):

        tokens = []

        buffer = ""

        while self.position < len(self.code):

            char = self.code[self.position]

            if char.isspace():

                if char == '\n':

                    tokens.append(Token(TokenType.NEWLINE, '\n'))

                else:

                    tokens.append(Token(TokenType.WHITESPACE, char))

                self.position += 1

                continue

            # Handle single-line comments

            if self.code[self.position:self.position+2] == "//":

                comment = ""

                self.position += 2

                while self.position < len(self.code) and self.code[self.position] != '\n':

                    comment += self.code[self.position]

                    self.position += 1

                tokens.append(Token(TokenType.COMMENT, comment))

                continue .....

The get\_token\_type() method is used internally to classify a matched lexeme. If the lexeme matches a keyword, it returns the corresponding TokenType from the map. If it matches a literal such as an integer or float, it assigns the appropriate literal token type using regex checks. This method ensures each lexeme is assigned a consistent and unambiguous type before being added to the final token stream.

 def get\_token\_type(self, lexeme):

        if lexeme in keyword\_map:

            return Token(keyword\_map[lexeme], lexeme)

        elif lexeme in builtin\_map:

            return Token(builtin\_map[lexeme], lexeme)

        elif lexeme in operator\_map:

            return Token(operator\_map[lexeme], lexeme)

        elif lexeme in delimiters\_map:

            return Token(delimiters\_map[lexeme], lexeme)

        elif re.fullmatch(r"#[0-9a-fA-F]{6}", lexeme):

            return Token(TokenType.COLOUR\_LITERAL, lexeme)

        elif re.fullmatch(r"\d+", lexeme):

            return Token(TokenType.INTEGER\_LITERAL, lexeme)

        elif re.fullmatch(r"\d+\.\d+", lexeme):

            return Token(TokenType.FLOAT\_LITERAL, lexeme)

        else:

            return Token(TokenType.IDENTIFIER, lexeme)

To validate the lexer, I tested it using the following PArL program stored as code1 in main.py. This program declares an integer variable and prints its value.

code = '''

    let x:int = 5;

    \_\_print x;

'''

Which the lexer processes its tokens as:

TOKENS:

TokenType.NEWLINE '

'

TokenType.WHITESPACE ' '

TokenType.WHITESPACE ' '

TokenType.WHITESPACE ' '

TokenType.WHITESPACE ' '

TokenType.let 'let'

TokenType.WHITESPACE ' '

TokenType.IDENTIFIER 'x'

TokenType.COLON ':'

TokenType.int\_ 'int'

TokenType.WHITESPACE ' '

TokenType.OP\_ASSIGN '='

TokenType.WHITESPACE ' '

TokenType.INTEGER\_LITERAL '5'

TokenType.SEMICOLON ';'

TokenType.NEWLINE '

'

TokenType.WHITESPACE ' '

TokenType.WHITESPACE ' '

TokenType.WHITESPACE ' '

TokenType.WHITESPACE ' '

TokenType.PRINT '\_\_print'

TokenType.WHITESPACE ' '

TokenType.IDENTIFIER 'x'

TokenType.SEMICOLON ';'

TokenType.WHITESPACE ' '

TokenType.NEWLINE '

'

TokenType.EOF 'EOF'

This simple example was chosen to ensure the lexer correctly identifies variable declarations, type annotations, assignments, literals, and built-in functions. The lexer correctly generated the following token stream: TokenType.let, TokenType.IDENTIFIER, TokenType.COLON, TokenType.int\_, TokenType.OP\_ASSIGN, TokenType.INTEGER\_LITERAL, TokenType.SEMICOLON, followed by TokenType.PRINT, TokenType.IDENTIFIER, and another TokenType.SEMICOLON. This output confirms that the lexical analysis stage is working as expected, accurately recognizing all token types involved in variable declarations and basic output.

One slight deviation from the original EBNF was the inclusion of as as a casting operator. This was added to the operator\_map and handled explicitly in later stages. The EBNF also did not mention PAD2000c built-in functions like \_\_print and \_\_write, so I chose to include them as special token types rather than regular identifiers. This simplifies later stages of compilation, especially semantic analysis and code generation.

Overall, Task 1 successfully provides a robust lexical foundation for the rest of the compiler pipeline. The regular-expression-driven tokenization and dictionary-based classification allow for easy extensibility, and the output integrates seamlessly with the parser.

Task 2 – Parser/AST

Task 2 focused on implementing the parser, which takes the token stream generated by the lexer and constructs a structured abstract syntax tree (AST) representation of the input program. This was accomplished using a hand-written recursive descent parser defined in the Parser.py file. The parser supports all core language constructs from the EBNF, including variable declarations, expressions, conditionals, loops, function definitions, and calls. The parser is structured around a series of parse\_\* methods, each of which corresponds to a rule in the language grammar. It maintains internal state via the current\_token and current\_index attributes, and it advances through the token list by calling advance() when a token has been successfully consumed.

The parser starts by invoking parse\_program(), which repeatedly calls parse\_statement() until the end of the input is reached. Each type of statement is dispatched to its corresponding parser function, such as parse\_variable\_declaration() for variable declarations, parse\_function() for function definitions, or parse\_expression\_statement() for standalone expressions. A notable design choice was the use of predictive lookahead with the peek() method, which allows the parser to disambiguate statements like assignments vs. function calls by inspecting upcoming tokens without consuming them prematurely.

def parse\_program(self):

        program\_node = ASTProgramNode()

        while self.current\_token.type != TokenType.EOF:

            stmt = self.parse\_statement()

            program\_node.add\_statement(stmt)

        return program\_node

Another important section is the parsing of expressions, which is done via a hierarchy of methods: parse\_expression() handles comparisons and casts, parse\_simple\_expr() handles +, -, and or, and parse\_term() handles \*, /, and and. This breakdown mirrors standard operator precedence and ensures correct grouping of subexpressions in the AST.

def parse\_expression(self):

        expr = self.parse\_simple\_expr()

        while self.current\_token.type in {

            TokenType.OP\_LT, TokenType.OP\_GT, TokenType.OP\_LE, TokenType.OP\_GE,

            TokenType.OP\_EQ, TokenType.OP\_NEQ

        }:

            op = self.current\_token.value

            self.advance()

            right = self.parse\_simple\_expr()

            expr = ASTBinaryOpNode(expr, op, right)

        while self.current\_token.type == TokenType.OP\_CAST:

            self.match(TokenType.OP\_CAST)

            if self.current\_token.type not in {

                TokenType.int\_, TokenType.float\_, TokenType.bool, TokenType.colour

            }:

                raise SyntaxError(f"Invalid cast target type: {self.current\_token.value}")

            target\_type = self.current\_token.value

            self.advance()

            expr = ASTCastNode(expr, target\_type)

        return expr

The method parse\_primary() plays a central role in parsing the most atomic expressions in the language. It is responsible for handling literals, identifiers, function calls, and array accesses. When an identifier is encountered, the parser checks whether it is followed by a ( to determine if it is a function call, or a [ to parse it as an array access. If neither is present, it is simply returned as an ASTIdentifierNode. Literals such as integers and booleans are parsed directly into ASTLiteralNode objects. This function ensures that low-level syntactic units are correctly parsed and that higher-level expressions can build on top of them.

def parse\_primary(self):

        if self.current\_token.type == TokenType.LPAREN:

            self.match(TokenType.LPAREN)

            expr = self.parse\_expression()

            self.match(TokenType.RPAREN)

            return expr

        elif self.current\_token.type in {

            TokenType.IDENTIFIER,

            TokenType.PRINT, TokenType.DELAY, TokenType.WRITE, TokenType.WRITE\_BOX,

            TokenType.CLEAR, TokenType.RANDOM\_INT, TokenType.READ,

            TokenType.WIDTH, TokenType.HEIGHT

        }:

            name = self.current\_token.value

            self.advance()

            if self.current\_token.type == TokenType.LPAREN:

                self.match(TokenType.LPAREN)

                args = self.parse\_argument\_list()

                return ASTFunctionCallNode(name, args)

            elif self.current\_token.type == TokenType.LBRACKET:

                self.match(TokenType.LBRACKET)

                index\_expr = self.parse\_expression()

                self.match(TokenType.RBRACKET)

                return ASTArrayAccessNode(name, index\_expr)

            return ASTIdentifierNode(name)

Syntax constructs like for, while, and return are each parsed by specialized methods. For instance, parse\_for\_statement() handles full for loop declarations, supporting optional initialization, condition, and increment clauses.

def parse\_for\_statement(self):

        self.match(TokenType.for\_)

        self.match(TokenType.LPAREN)

        # init

        init\_stmt = None

        if self.current\_token.type == TokenType.let:

            init\_stmt = self.parse\_variable\_declaration\_no\_semicolon()

        elif self.current\_token.type != TokenType.SEMICOLON:

            init\_stmt = self.parse\_assignment\_no\_semicolon()

        self.match(TokenType.SEMICOLON)

        # condition

        condition\_expr = None

        if self.current\_token.type != TokenType.SEMICOLON:

            condition\_expr = self.parse\_expression()

        self.match(TokenType.SEMICOLON) .....

def parse\_if(self):

        self.match(TokenType.if\_)

        self.match(TokenType.LPAREN) ...

def parse\_while(self):

        self.match(TokenType.while\_)

        self.match(TokenType.LPAREN)....

def parse\_return(self):

        self.match(TokenType.return\_)

        expr = self.parse\_expression()

These are parsed using helper methods such as parse\_variable\_declaration\_no\_semicolon() and parse\_assignment\_no\_semicolon() which are variants of their standard counterparts that omit semicolon parsing. This allows the parser to consume entire for loop headers like for (let i: int = 0; i < 8; i = i + 1) as a single unit.

def parse\_variable\_declaration\_no\_semicolon(self):

        self.match(TokenType.let)

        name = self.current\_token.value

        self.match(TokenType.IDENTIFIER)

        self.match(TokenType.COLON) ....

def parse\_assignment\_no\_semicolon(self):

        name = self.current\_token.value

        self.match(TokenType.IDENTIFIER)

        self.match(TokenType.OP\_ASSIGN)

        expr = self.parse\_expression()

        return ASTAssignmentNode(name, expr)....

The full version of variable declarations, parse\_variable\_declaration(), is responsible for recognizing both scalar and array declarations. It starts by matching the let keyword, variable name, type, and optional assignment. If a square bracket is detected, it switches into array parsing logic via parse\_array\_declaration() and expects an initializer list enclosed in brackets. This modularity ensures that both regular variables and arrays are handled correctly and distinctly.

ef parse\_variable\_declaration(self):

        self.match(TokenType.let)

        name = self.current\_token.value

        self.match(TokenType.IDENTIFIER)

        self.match(TokenType.COLON)

        var\_type = self.current\_token.value

        self.advance()

        if self.current\_token.type == TokenType.LBRACKET:

            return self.parse\_array\_declaration(name, var\_type)

        expr = None

        if self.current\_token.type == TokenType.OP\_ASSIGN:

            self.match(TokenType.OP\_ASSIGN)

            expr = self.parse\_expression()

        self.match(TokenType.SEMICOLON)

        return ASTVariableDeclarationNode(var\_type, name, expr)

Assignments are parsed by parse\_assignment(), which distinguishes between regular variable assignments like x = 5; and array assignments such as arr[i] = 10;. It first matches an identifier and checks whether it is followed by a bracket ([) to determine if it is an array access. Based on this, it either constructs an ASTAssignmentNode or an ASTArrayAssignmentNode.

def parse\_assignment(self):

        if self.current\_token.type != TokenType.IDENTIFIER:

             raise SyntaxError("Expected identifier on left-hand side of assignment")

        name = self.current\_token.value

        self.match(TokenType.IDENTIFIER)

        # Array assignment like x[i] = ...

        if self.current\_token.type == TokenType.LBRACKET:

            self.match(TokenType.LBRACKET)

            index\_expr = self.parse\_expression()

            self.match(TokenType.RBRACKET)

            self.match(TokenType.OP\_ASSIGN)

            value\_expr = self.parse\_expression()

            self.match(TokenType.SEMICOLON)

            return ASTArrayAssignmentNode(name, index\_expr, value\_expr) ...

The function declaration parser, parse\_function(), is one of the more complex routines in the parser. It handles the full syntax of function definitions, including the parameter list, optional array types, return type, and function body. Parameters are parsed as (type name) pairs, with support for fixed-size array parameters like int[8]. After parsing the parameter list and return type, the function body is parsed using parse\_block() and encapsulated in an ASTFunctionNode.

def parse\_function(self):

        self.match(TokenType.fun)

        name = self.current\_token.value

        self.match(TokenType.IDENTIFIER)

        self.match(TokenType.LPAREN)

        params = []

        if self.current\_token.type != TokenType.RPAREN:

            while True:

                param\_name = self.current\_token.value

                self.match(TokenType.IDENTIFIER)

                self.match(TokenType.COLON)

                # Base type (e.g., int, float)

                base\_type = self.current\_token.value

                self.advance().....

Each control structure for, while, if, return has a dedicated method that matches the syntax exactly as expected by the grammar, and builds corresponding AST nodes like ASTForNode, ASTWhileNode, and ASTReturnNode. These methods are called from parse\_statement(), the central dispatcher that inspects the current token and forwards it to the correct parser based on its type.

This architecture closely mirrors the structure of the original EBNF, with only slight enhancements. These include array syntax and support for built-in function tokens like \_\_print, which the grammar treats as identifiers but which are matched using their own token types in the lexer. This simplifies later stages of the compiler by letting the semantic and codegen stages treat built-ins differently.

All parsing results are converted into abstract syntax trees using custom ASTNode classes defined in AST.py. To verify correctness, I used the simple test program:

code3 = '''

fun giga()-> int {

    return 5;

    }

let x:int = giga();

\_\_print x;

'''

Which gets parsed into

AST:

Program:

Function: giga -> int

Block:

Return:

Literal (int): 5

VarDecl: int x

Expr:

FunctionCall: giga

Print:

Identifier: x

All parsing results are converted into abstract syntax trees composed of custom node classes defined in AST.py. This file serves as the backbone of the entire compiler architecture, providing structured node types for every language construct from ASTLiteralNode for constants, to ASTFunctionNode, ASTArrayAccessNode, and ASTReturnNode. Each node class implements an accept() method to support the visitor pattern, allowing the later stages of semantic analysis and code generation to traverse the tree in a uniform way.

class ASTFunctionNode(ASTStatementNode):

    def \_\_init\_\_(self, name, params, return\_type, body):

        self.name = name

        self.params = params

        self.return\_type = return\_type

        self.body = body

    def accept(self, visitor):

        return visitor.visit\_function\_node(self)

class ASTFunctionCallNode(ASTExpressionNode):

        def \_\_init\_\_(self, name, arguments):

            self.name = name

            self.arguments = arguments

        def accept(self, visitor):

            return visitor.visit\_function\_call\_node(self)

class ASTVariableDeclarationNode(ASTStatementNode):

    def \_\_init\_\_(self, var\_type, name, expr=None):

        self.var\_type = var\_type

        self.name = name

        self.expr = expr

    def accept(self, visitor):

        return visitor.visit\_variable\_declaration\_node(self)

class ASTAssignmentNode(ASTStatementNode):

    def \_\_init\_\_(self, name, expr):

        self.name = name

        self.expr = expr

    def accept(self, visitor):

        return visitor.visit\_assignment\_node(self) ......

Task 3 Type Checking – Semantic Analysis

Task 3 involved implementing the semantic analysis phase of the compiler, which is responsible for enforcing type rules and scope correctness across the abstract syntax tree (AST). This phase was implemented using a visitor pattern, specifically in the TypeCheckVisitor.py file. The visitor traverses the AST and performs checks for type compatibility, undeclared variables, incorrect return types, and misuse of arrays. This task ensures that the program is semantically valid before it proceeds to code generation.

At the heart of semantic analysis is the SymbolTable, implemented in SymbolTable.py. The symbol table is designed as a stack of dictionaries, with each dictionary representing a lexical scope. As the visitor enters a new block or function, it pushes a new scope onto the stack using push\_scope(). When the block ends, it pops the scope off. This mechanism ensures that variables are only visible within their declared scope and prevents redeclaration in the same block. The declare() method adds a new variable or function to the current scope, while lookup() searches outward through the nested scopes to find an existing identifier.

class SymbolTable:

    def \_\_init\_\_(self):

        self.scopes = [{}]  # stack of dictionaries

    def push\_scope(self):

        self.scopes.append({})

    def pop\_scope(self):

        if len(self.scopes) > 1:

            self.scopes.pop()

        else:

            raise Exception("Cannot pop global scope")

    def declare(self, name, var\_type):

        if name in self.scopes[-1]:

             raise TypeError(f"Variable '{name}' already declared in this scope")

        self.scopes[-1][name] = var\_type

    def lookup(self, name):

        for scope in reversed(self.scopes):

            if name in scope:

                return scope[name]

        raise Exception(f"Undeclared identifier: '{name}'")

    def \_\_str\_\_(self):

        return str(self.scopes)

The visit\_variable\_declaration\_node() method checks that the declared type matches the type of the expression assigned to the variable. If the expression is not present, the variable is just registered in the symbol table. If the type does not match, a TypeError is raised.

def visit\_variable\_declaration\_node(self, node):

        if node.expr:

            expr\_type = node.expr.accept(self)

            if expr\_type != node.var\_type:

                raise TypeError(f"Type mismatch: Cannot assign {expr\_type} to {node.var\_type}")

        self.symbols.declare(node.name, node.var\_type)

Function declarations are handled in visit\_function\_node(), where the return type and parameter types are registered in the symbol table. The method also checks that the function contains a matching return statement if its return type is not void, and verifies that return expressions match the declared type.

def visit\_function\_node(self, node):

        param\_types = []

        for (ptype, pname) in node.params:

            if "[" in ptype:

                element\_type = ptype.split("[")[0]

                size = int(ptype.split("[")[1].rstrip("]"))

                param\_type = {

                    "type": "array",

                    "element\_type": element\_type,

                    "size": size

                }

            else:

                param\_type = ptype

            param\_types.append(param\_type)

        self.symbols.declare(node.name, (node.return\_type, param\_types))....

Control structures like if, for, and while are semantically validated by their respective visitor methods. For example, visit\_if\_node() and visit\_while\_node() ensure that the condition expression returns a boolean type. The visit\_for\_node() method checks the initialization, condition, and increment expressions for type correctness and ensures that the loop body has no violations.

def visit\_while\_node(self, node):

        cond\_type = node.condition.accept(self)

        if cond\_type != "bool":

            raise TypeError("Condition in 'while' must be boolean")

        node.body.accept(self)

Array declarations and assignments are also semantically verified. visit\_array\_declaration\_node() ensures that all initializer values match the declared element type and that the size of the array matches the number of values given. The visit\_array\_assignment\_node() checks that the index is an integer and the assigned value has the correct element type.

The following is an incorrect syntax test code:

code3 = '''

fun giga()-> int {

    return "hello";

    }

let x:int = giga();

\_\_print x;

'''

SyntaxError: Unexpected primary expression: TokenType.ERROR '"'

code3 = '''

fun giga()-> int {

    return 5;

    }

let x:int = giga();

\_\_print y;

'''

Exception: Undeclared identifier: 'y'

This was processed by the TypeCheckerVisitor without error, confirming that the variable declaration and usage were consistent with the language’s type rules.

Task 4 – Code Gen

Task 4 focused on code generation: the process of converting a semantically valid abstract syntax tree (AST) into PArIR assembly instructions suitable for execution on the virtual machine. This was implemented in the CodeGenVisitor.py file, which uses the visitor pattern to walk the AST and emit low-level instructions for each node. Each visit method handles a specific language construct, translating high-level operations into corresponding stack-based VM commands such as push, pop, add, call, and ret.

The code generator maintains a list of emitted instructions, a symbol table-like structure for variable/frame indexing, and a counter to generate unique labels for control flow constructs. When entering a new scope (such as the global block, function body, or loop), the generator uses push\_scope() to open a new frame with push N followed by oframe. When exiting, pop\_scope() emits cframe to clean up. This mimics how stack frames are managed in a real VM.

def push\_scope(self, size=0):

        self.emit(f"push {size}")

        self.emit("oframe")

        self.symbols.append({})

        self.frame\_index = 0

        self.scope\_level += 1

def pop\_scope(self):

        if self.scope\_level > 0:

            self.emit("cframe")

            self.symbols.pop()

            self.scope\_level -= 1

Variable declarations are assigned frame slots based on the current scope level. When a variable is declared, it is registered with an index and level. On assignment, the expression is evaluated and stored in that frame slot using st. For example, declaring let x:int = 5; results in instructions like push 5, push 0, push 0, st, which means "push the value 5, then store it in frame slot 0 at scope level 0".

Function declarations are handled in visit\_function\_node(). The generator emits a .label for the function name, manually binds parameters to slots in the newly entered frame, and generates the body. A notable detail is that the VM *automatically* pushes a new frame during call, so the generator must not emit oframe again inside the function. Parameters are bound by retrieving their values from the caller’s frame using push [i:level], then storing them in local slots with st.

def visit\_function\_node(self, node):

        old\_emit = self.emit

        self.emit = self.function\_instructions.append

        self.emit(f".{node.name}")

        self.symbols.append({})

        self.scope\_level += 1

        self.frame\_index = 0

        for i, (\_, pname) in enumerate(node.params):

            idx = self.frame\_index

            lvl = self.scope\_level

            self.symbols[-1][pname] = (idx, lvl)

            self.frame\_index += 1

            self.emit(f"push [{i}:{lvl}]")

            self.emit(f"push {idx}")

            self.emit(f"push {lvl}")

            self.emit("st")

Function calls themselves are emitted using call, preceded by argument pushes. Arguments are pushed in reverse order to match the expected stack layout of the VM. After arguments, the function label is pushed and the call is made.

def visit\_function\_call\_node(self, node):

        for arg in reversed(node.arguments):

            arg.accept(self)

        self.emit(f"push .{node.name}")

        self.emit("call")

Unfortunately, I was not able to fully implement support for **function parameters** — especially with multiple arguments or arrays. While simple scalar functions can be emitted, such as those with no parameters, attempts to handle parameter passing at runtime led to incorrect stack behavior and broken results. This means functions like add(a: int, b: int) could not be reliably called, and functions receiving arrays like MaxInArray(x: int[8]) fail during execution, even though they are parsed and type-checked correctly. The code generator successfully emits call and labels, but does not reliably bind or retrieve parameter values at runtime.

code3 = '''

fun giga()-> int {

    return 5;

    }

let x:int = giga();

\_\_print x;

'''

Generated ParIR:

.main

push 1

oframe

push .giga

call

push 0

push 0

st

push [0:0]

print

halt

.giga

push 5

ret

This output demonstrates that basic function calls without parameters are supported correctly. The value returned from giga() is pushed onto the stack, stored into x, and printed. The .main block wraps the global logic in a frame, and the .giga label contains the expected body of the function.

While full function parameter handling could not be completed in time, the rest of the code generator behaves as expected for variable declarations, assignments, control flow, arrays, and built-in functions like \_\_print, \_\_write, and \_\_clear. Labels and jumps are correctly emitted for if, while, and for constructs, with unique label generation using a label\_counter to avoid conflicts. The overall structure of the code generator is sound, and the output integrates cleanly with the PAD2000c simulator for testable cases.

Task 5 – Arrays

Task 5 extended the compiler to support fixed-size arrays, both as variables and as parameters in functions. This required changes across multiple stages of the compiler: parsing, semantic analysis, and code generation. The parser was updated to recognize array declarations, array accesses, and array assignments, while the semantic analyzer was extended to validate array types, bounds, and element types. Finally, the code generator was modified to emit specialized instructions for storing, accessing, and modifying arrays in the virtual machine stack model.

Array declarations were implemented in the method parse\_array\_declaration() within the parser. This supports syntax such as let x: int[5] = [10, 20, 30, 40, 50];, and constructs an ASTArrayDeclarationNode. The array size can be inferred from the initializer or specified explicitly. The values are parsed as a list of ASTLiteralNode elements and stored in the node.

def parse\_array\_declaration(self, name, element\_type):

        self.match(TokenType.LBRACKET)

        size\_expr = None

        if self.current\_token.type == TokenType.INTEGER\_LITERAL:

            size\_expr = ASTLiteralNode(self.current\_token.value, "int")

            self.match(TokenType.INTEGER\_LITERAL)

        self.match(TokenType.RBRACKET)

        self.match(TokenType.OP\_ASSIGN)

        self.match(TokenType.LBRACKET) ...

The semantic analysis of arrays was handled in visit\_array\_declaration\_node() and visit\_array\_assignment\_node() within the TypeCheckVisitor. The declaration method checks that all initializer values match the declared element type, and that the number of values matches the declared or inferred size. The assignment method validates that the array index is an integer and that the value assigned to a given index is type-compatible.

def visit\_array\_declaration\_node(self, node):

        array\_size = len(node.values)

        if node.size\_expr:

            size\_type = node.size\_expr.accept(self)

            if size\_type != "int":

                raise TypeError("Array size must be an integer")

            declared\_size = int(node.size\_expr.value)

            if declared\_size != array\_size:

                raise TypeError(f"Array initializer has {array\_size} values but declared size is {declared\_size}")

        else:

            declared\_size = array\_size  # infer ...

Arrays are treated structurally in the type checker, meaning an array is defined by its element type and size. This enables support for arrays as function parameters, though in practice, parameter binding for arrays was not fully implemented in code generation due to limitations in the function calling mechanism.

In CodeGenVisitor, array support was added through several new node handlers. visit\_array\_declaration\_node() pushes the array’s values onto the stack in reverse order and emits an sta (store array) instruction with the index, level, and total size.

def visit\_array\_declaration\_node(self, node):

        size = len(node.values) if node.size\_expr is None else int(node.size\_expr.value)

        self.declare\_variable(node.name)

        index, level = self.lookup\_variable(node.name)

        for val\_node in reversed(node.values):

            val\_node.accept(self)

        self.emit(f"push {size}")

        self.emit(f"push {index}")

        self.emit(f"push {level}")

        self.emit("sta")

To test array support, I used the program in code5 from main.py:

code5 = '''

let nums: int[5] = [10, 20, 30, 40, 50];

let x: int = nums[2];

\_\_print x;

'''

Which outputs

AST:

Program:

ArrayDecl: int nums

Size:

Literal (int): 5

Initial Values:

Literal (int): 10

Literal (int): 20

Literal (int): 30

Literal (int): 40

Literal (int): 50

VarDecl: int x

Expr:

ArrayAccess: nums

Literal (int): 2

Print:

Identifier: x

Generated ParIR:

.main

push 6

oframe

push 50

push 40

push 30

push 20

push 10

push 5

push 0

push 0

sta

push 2

push +[0:0]

push 1

push 0

st

push [1:0]

print

halt

This program declares an array of five integers, accesses the third element (index 2), stores it in variable x, and prints it. The semantic analyzer confirmed that the array is declared correctly and indexed using an integer. The generated PArIR pushes the array values onto the stack, stores them using sta, accesses index 2, stores the result into x, and prints it. This confirms that array declaration, access, and assignment all work together successfully through the entire compilation pipeline.

Test codes and VM Output

A screenshot of a computer program

AI-generated content may be incorrect.code5 = '''

let nums: int[5] = [10, 20, 30, 40, 50];

let x: int = nums[2];

\_\_print x;

'''

Array Test

A computer screen shot of a program

AI-generated content may be incorrect.

code3 = '''

fun giga()-> int {

    return 5;

    }

let x:int = giga();

\_\_print x;

'''

Function Test

A screenshot of a computer program

AI-generated content may be incorrect.code6 = '''

let a: int = 10 + 5 \* 2;

let b: int = (10 + 5) \* 2;

let c: int = a - b;

let d: int = a / 3 + b \* 2 - c;

\_\_print d;

'''

General Maths test

A screen shot of a computer program

AI-generated content may be incorrect.code7 = '''

let a: int = 1;

let b: int = 2;

if (a < b) {

    \_\_print a;

} else {

    \_\_print b;

}

'''

Condition test

A screen shot of a computer

AI-generated content may be incorrect.

code4 = '''

fun add(a: int, b: int) -> int {

    return a + b;

}

let result: int = add(10, 32);

\_\_print result;

'''

Function with parameters failed on code Generation

Parses well though.

Conclusion

This project presents a mostly functional implementation of a compiler for the PArL programming language, developed as part of the CPS2000 module. The compiler processes source code through lexical analysis, recursive-descent parsing, semantic checking with scoped symbol tables, and partial PArIR code generation. Core features such as variable declarations, arithmetic expressions, if statements, arrays, and built-in functions like \_\_print and \_\_write are implemented and were tested using small example programs. The compiler adheres closely to the provided EBNF, with minor extensions to support array syntax and casting.

Although most language constructs are handled successfully, full code generation support for function calls with parameters could not be completed. These functions parse and type-check correctly, but issues with stack management and parameter binding at runtime caused incorrect execution. Despite this, the compiler successfully generates and runs valid PArIR for many common use cases and provides a solid framework that could be extended to complete the remaining features.