Detailed Implementation Explanation of a Pascal Compiler Front-End

Maikel González Lucas González Luis Corrales

June 1, 2025

Abstract

This document provides a comprehensive explanation of the implementation of a Pascal compiler front-end developed in Python using the PLY (Python Lex-Yacc) toolkit. The compiler processes a subset of the Pascal language, performing lexical analysis, syntactic parsing, basic semantic checks, symbol table management, and generating intermediate assembly-like code for a custom stack-based virtual machine. The explanation details the design decisions for each compiler phase, the grammar rules implemented, the strategies for semantic analysis, and the an AST-driven approach to code generation. Key language features supported include variable declarations (integer, boolean, string, arrays), control flow structures (if-then-else, for-to/downto, while), arithmetic and boolean expressions, and I/O operations. The document also outlines the initial steps taken towards parsing function declarations and calls.

Contents

1 Introduction

Compilers are fundamental software tools that translate source code written in high-level programming languages into a lower-level representation, such as machine code or intermediate code, executable by a computer or a virtual machine. This document describes the design and implementation of the front-end of a compiler for a defined subset of the Pascal programming language. The primary goal is to transform Pascal source programs into an assembly-like intermediate code tailored for a specific stack-based virtual machine (VM).

The development of this compiler leverages `PLY`, a widely-used Python library that provides tools for constructing lexers and parsers, inspired by the traditional Lex and Yacc utilities. The compiler is structured into several key phases:

1. **Lexical Analysis:** Breaking down the source code into a stream of tokens.
2. **Syntactic Analysis (Parsing):** Verifying the grammatical structure of the token stream and constructing an Abstract Syntax Tree (AST).
3. **Semantic Analysis (Integrated with Parsing):** Performing basic type checking, scope resolution, and managing a symbol table.
4. **Intermediate Code Generation:** Traversing the AST to produce assembly instructions for the target VM.

This document will elaborate on each of these phases, detailing the specific implementation choices, data structures used (like the symbol table and AST node formats), grammar rules, and the strategies employed for code generation and error handling.

2 Lexical Analysis

Lexical analysis, or scanning, is the initial phase where the raw Pascal source code text is converted into a sequence of tokens. Each token represents a lexeme, which is a sequence of characters that forms a syntactic unit.

2.1 Token Definitions and Reserved Words

The lexer distinguishes between general token types and reserved keywords. Reserved words are specific identifiers that have fixed meanings in Pascal. The core token types include:

- `ID`: For identifiers (variable names, program name, function names).
- `NUMBER`: For integer literals.
- `STRING_LITERAL`: For string literals enclosed in single quotes.
- Operators: Arithmetic (`+`, `-`, `*`), relational (`>`, `<`, `=`, etc.), assignment (`:=`).
- Delimiters and Punctuation: `(`, `)`, `[`, `]`, `;`, `:`, `,`, `.`, `..`

Pascal reserved keywords are managed in a Python dictionary, mapping the lowercase keyword string to its specific token type (e.g., `'program': 'PROGRAM'`). This allows the lexer to identify keywords correctly, even if they appear in mixed case in the source (though Pascal is generally case-insensitive for identifiers and keywords, our lexer normalizes them by checking their lowercase form against the reserved list).

```
# Lex7.py
reserved = {
    'program': 'PROGRAM', 'var': 'VAR', 'integer': 'INTEGER',
    'begin': 'BEGIN', 'end': 'END', 'writeln': 'WRITELN',
    'write': 'WRITE', 'readln': 'READLN', 'if': 'IF',
    'then': 'THEN', 'else': 'ELSE', 'for': 'FOR', 'to': 'TO',
    'do': 'DO', 'while': 'WHILE', 'div': 'DIV', 'mod': 'MOD',
    'array': 'ARRAY', 'of': 'OF', 'boolean': 'BOOLEAN',
    'and': 'AND', 'or': 'OR', 'true': 'TRUE', 'false': 'FALSE',
    'string': 'STRING_TYPE',
    'length': 'LENGTH',
    'downto': 'DOWNTO',
    'function': 'FUNCTION'
}

tokens = [
    'ID', 'NUMBER', 'PLUS', 'MINUS', 'MULT',
    'LPAREN', 'RPAREN', 'SEMI', 'COLON', 'COMMA',
    'DOTDOT', 'DOT', 'ASSIGN', 'STRING_LITERAL',
    'GT', 'LT', 'GE', 'LE', 'EQ', 'NE',
    'LSQUARE', 'RSQUARE'
] + list(reserved.values())
```

Listing 1: Reserved Words and Tokens

2.2 Token Regular Expression Patterns

Each token is defined using a regular expression (or a simple string for fixed tokens) that PLY uses to match lexemes in the input stream.

- **Identifiers (`ID`):** Defined by `r'[a-zA-Z_][a-zA-Z0-9_]*'`. A special handler function `t_ID` checks if the matched identifier is a reserved word; otherwise, it's classified as a generic `ID`.
- **Numbers (`NUMBER`):** Matched by `r'\d+'`. The `t_NUMBER` function converts the matched string of digits into an integer Python object. The `t_STRING_LITERAL` function extracts the content of the string, removing the enclosing single quotes.
- **Operators and Delimiters:** Most are defined by simple string patterns, e.g., `t_ASSIGN = r':='`. Special care is taken for multi-character operators like `DOTDOT` (`..`) to ensure they are tokenized correctly before single-character ones like `DOT` (`.`).
- **Whitespace and Newlines:** Spaces and tabs (`t_ignore = ' \t'`) are skipped. Newlines (`r'\n+'`) are handled by `t_newline` to increment the lexer's line counter, crucial for error reporting.

```
# Lex7.py
def t_STRING_LITERAL(t):
    r'\'[^\']*\''
    t.value = t.value[1:-1] # Remove outer quotes
    return t

def t_ID(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*'
    t.type = reserved.get(t.value.lower(), 'ID') # Check for reserved
    words
    return t

def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)
```

Listing 2: Example Token Definitions in Lexer

2.3 Error Handling in Lexer

If the lexer encounters a character that does not match any defined token rule, the `t_error` function is invoked. This function prints an error message indicating the illegal character and its line number. The lexer then skips the character (`t.lexer.skip(1)`) and attempts to continue tokenizing the rest of the input.

3 Parsing and Abstract Syntax Tree (AST) Construction

The parsing phase takes the stream of tokens from the lexer and verifies if it conforms to the defined grammar of the Pascal subset. If the structure is valid, an Abstract Syntax Tree (AST) is constructed. The AST is a hierarchical tree representation of the source code's structure, which is then used for subsequent phases like code generation. Semantic analysis, including symbol table management and basic type checking, is often interleaved with parsing actions.

3.1 Grammar Rules (BNF-like Productions)

The grammar is defined using PLY's YACC-like syntax, specifying productions for each non-terminal symbol. Each production rule can have an associated Python function (semantic action) that executes when the rule is reduced by the parser. These actions are responsible for constructing AST nodes.

Key syntactic structures defined include:

- `program`: The top-level rule, `PROGRAM ID SEMI block DOT`.
- `block`: Encapsulates declarations and a compound statement: `optional_declarations BEGIN statements_body END`.
- `optional_declarations`: Handles the `VAR` section and, more recently, `FUNCTION` declarations. It can also be empty.

- `var_declaration_list_nonempty`, `var_declaration`: Define how multiple variables of various types are declared.
- `type`, `basic_type`, `array_type`: Define supported data types including `INTEGER`, `BOOLEAN`, `STRING_TYPE`, and one-dimensional arrays.
- `statements_body`, `statement_sequence`, `statement`: Define sequences of statements and individual statement types. Supported statements are assignments, I/O (`writeln`, `readln`), conditionals (`if-then-else`), and loops (`for-to/downto-do`, `while-do`).
- `expression`, `factor`, `variable`: Define arithmetic, boolean, and relational expressions, respecting operator precedence (defined separately using PLY's `precedence` tuple). Factors include numbers, identifiers (variables), boolean literals, string literals, array/string indexed access, and function calls (including 'length').
- `function_declaration`, `optional_parameters`, `formal_parameter_section`: Define the syntax for function declarations, including their name, parameters (name and type), and return type.
- `function_call`, `expr_list_opt`: Define how functions are called with optional arguments.

```
# Lex7.py
def p_if_stmt(p):
    'if_stmt : IF condition THEN statement else_clause'
    # p[0] is the result of the reduction (AST node)
    # p[1] is IF token, p[2] is 'condition' AST, etc.
    p[0] = ('if', p[2], p[4], p[5])
    # AST node: ('if', condition_node, then_statement_node,
   else_node_or_None)

def p_else_clause(p):
    '''else_clause : ELSE statement
                   | empty_stmt_node''' # empty_stmt_node for no else
    if p.slice[1].type == 'ELSE':
        p[0] = p[2] # The statement AST node
    else:
        p[0] = p[1] # The ('empty_node',) AST node
```

Listing 3: Example Grammar Rule for IF statement

3.2 AST Node Structure

AST nodes are represented as Python tuples. The first element of the tuple is a string indicating the node type (e.g., `'assign'`, `'if'`, `'var_decl_group'`), and subsequent elements are children nodes or relevant data (like variable names, literal values, operator types). For instance:

- Assignment: `('assign', ('id', 'x'), ('number', 5))` for `x := 5`.
- Array Declaration: `('var_decl_group', ['myArray'], ('array', 1, 10, 10, ('type', 'integer')))`.
- Function Call: `('function_call', 'MyFunc', [arg1_expr_node, arg2_expr_node])`.

3.3 Symbol Table Management and Scoping

The compiler uses a symbol table mechanism to keep track of declared identifiers (variables, functions) and their attributes (type, scope, memory location/offset).

- **Global Scope:** A `global_symbol_table` (dictionary) stores all globally declared variables and function signatures. A `global_symbol_counter` assigns unique indices for global variables, which map to offsets from the Global Pointer (GP) in the VM.
- **Function Scope:** For functions, a more sophisticated approach to scoping is initiated.
  - When a function declaration is parsed (specifically, when its name is seen via the `seen_func_name` rule), `start_function_scope(func_name)` is called. This sets `current_function_name` and prepares a new entry in `function_scopes`.
  - `function_scopes[func_name]` stores separate lists/maps for parameters (`params`, `param_map`) and local variables (`locals`, `local_map`).
  - `register_param_in_current_scope` and `register_local_var_in_current_scope` are used to add entries to the current function's scope. They use a `next_offset` counter (per function) to assign offsets relative to the Frame Pointer (FP) for use with `PUSHL` and `STOREL` VM instructions.
  - `end_function_scope()` is called after the entire function declaration is parsed to reset `current_function_name`.
  - During parsing of variable declarations (`p_var_declaration`), a check for `current_function_name` determines whether to register the variable as global or local/parameter.
- **Semantic Checks during Parsing:**
  - **Variable Declaration:** The `register_global_variable`, `register_param_in_current_s` and `register_local_var_in_current_scope` functions implicitly check for re-declarations within the same scope.
  - **Array Bounds:** `p_array_type` checks if array bounds define a positive size.
  - **Undeclared Variables (in Code Generation):** Checks for undeclared variables are primarily performed during the code generation phase when an identifier is encountered in an expression or on the left-hand side of an assignment. The generator looks up the ID in the current scope (if in a function) and then the global scope.

The `symbol_table` entry for a variable might look like: `{'index': 0, 'type_info': ('type', 'integer'), 'is_array': False}`. For an array: `{'index': 1, 'type_info': ('array', 1, 5, 5, ('type', 'integer')), 'is_array': True, 'array_info': {...}}`. For a function signature in `global_symbol_table`: `{'is_function': True, 'params_structure': [...], 'return_type_info': ..., 'label': 'FUNC_NAME'}`.

3.4 Error Handling in Parser

The `p_error(p)` function is defined to handle syntax errors. When PLY's parser encounters an unparsable token sequence, it calls `p_error` with the offending token `p`. The function prints an error message including the token's value, type, line number, and position. A global error flag (`parser.error`) is set to prevent subsequent code generation if parsing errors occur.

4 Intermediate Code Generation

After a valid AST is constructed, the `gen_asm_node` function traverses it recursively to produce a sequence of assembly-like instructions for the target stack-based virtual machine.

4.1 Global and Local Variable Handling

- **Global Variables:**
  - At the beginning of the program's code (`START`), space for all global variables is implicitly available (or can be thought of as being at fixed offsets from a global base). The `PUSHI 0` instructions at the start initialize these global slots to 0.
  - Access (read/write) to global variables uses `PUSHG index` and `STOREG index`, where `index` is obtained from the `global_symbol_table`.
  - Global arrays are allocated on the Struct Heap. `gen_declarations_alloc` (called for global declarations) emits `ALLOC size` followed by `STOREG index` to store the returned heap address into the global variable's slot.
- **Function Parameters and Local Variables (Preliminary):**
  - **Prologue:** When a function is called, the `CALL` VM instruction sets the new Frame Pointer (`fp`) to the current Stack Pointer (`sp`), where arguments are assumed to be. The function's prologue then executes `PUSHN num_locals` to reserve space on the stack for its local variables.
  - **Access:**
    - Parameters are accessed using `PUSHL offset` or `STOREL offset`, where `offset` is relative to `fp`. The first parameter is at offset 0, the second at 1, and so on (this convention needs to be strictly followed).
  - **Return Value:** For 'FunctionName := expression;', the expression is evaluated, and its result is left on top of the operand stack. The 'RETURN' VM instruction is then generated.
  - **Epilogue:** The 'RETURN' instruction handles restoring 'sp' to 'fp', and popping the old 'fp' and return address 'pc' from the Call Stack.

4.2 Expression Evaluation

- Literals (`NUMBER`, `STRING_LITERAL`, `TRUE`/`FALSE`) are pushed onto the stack using `PUSHI value`, `PUSHS "string"`, or `PUSHI 1/0`.

- Variable access (`ID`) uses `PUSHG index` for globals or `PUSHL offset` for locals/-params.
- Array element access `array_id[expr]` generates code to:
  1. Push the base address of the array (from `PUSHG` or `PUSHL`).
  2. Evaluate the index `expr`.
  3. Adjust the index by subtracting the array's lower bound (e.g., `PUSHI 1`, `SUB` for 1-based arrays).
  4. Use `LOADN` to load the value from `address[index_vm]`.
- String character access `string_id[expr]` (e.g., for 'bin[i]'):
  1. Push the address of the string (from `PUSHG` or `PUSHL`).
  2. Evaluate the index `expr`.
  3. Adjust to 0-based index (`PUSHI 1, SUB`).
  4. Use `CHARAT` to get the ASCII code of the character.
- 'length(string$_id$)' :
  - Push the address of the string.
  - Use `STRLEN`.

Binary operations (`+`, `-`, `*`, `DIV`, `MOD`, relational operators): Operands are evaluated and pushed, then the corresponding VM instruction (`ADD`, `SUB`, `MUL`, `DIV`, `MOD`, `SUP`, `INF`, `EQUAL`, etc.) is emitted.

Boolean operations (`AND`, `OR`): Implemented with short-circuiting logic using conditional jumps (`JZ`, `JNZ`) and labels.

4.3 Control Flow Statements

- **Assignment (`variable := expression`):**
  - For simple variables: Evaluate expression, then `STOREG index` or `STOREL offset`.
  - For array elements `array_id[index_expr] := value_expr`:
    1. Push base address of array.
    2. Evaluate `index_expr` and adjust to 0-based.
    3. Evaluate `value_expr`.
    4. Use `STOREN` (which expects value, then 0-based index, then base address on stack, in that order from TOS downwards, though the VM doc says "takes a value v, an integer n and an address a", implying v is TOS). The code generation ensures this order: PUSHG addr, (calc index), (calc value), STOREN.
  - For function return value assignment (`FunctionName := expression;`): Evaluate expression, then `RETURN`.

- **If-then-else:**

```
        <c digo para condici n>
        JZ ElseLabel
        <c digo para bloque THEN>
        JUMP EndIfLabel ; (Solo si hay un ELSE)
ElseLabel:
        <c digo para bloque ELSE> ; (Si existe)
EndIfLabel:

```

Listing 4: If-Then-Else Code Structure

- **While-do loops:**

```
LoopStartLabel:
        <c digo para condici n>
        JZ LoopEndLabel
        <c digo para cuerpo del bucle>
        JUMP LoopStartLabel
LoopEndLabel:

```

Listing 5: While-Do Loop Code Structure

- **For-to-do loops:**

```
        <c digo para var_contador := expr_inicio>
LoopStartLabel:
        <c digo para PUSH var_contador>
        <c digo para expr_fin>
        INFEQ ; (var_contador <= expr_fin)
        JZ LoopEndLabel
        <c digo para cuerpo del bucle>
        <c digo para var_contador := var_contador + 1>
        JUMP LoopStartLabel
LoopEndLabel:

```

Listing 6: For-To-Do Loop Code Structure

- **For-downto-do loops:** Similar to `for-to`, but uses `SUPEQ` (>=) for condition and `SUB` for decrementing the loop variable.

4.4 Input/Output Operations

- `readln(variable)`:
  - For `integer`/`boolean` variables: `READ` (pushes address of string read), `ATOI` (converts to int), then `STOREG index` or `STOREL offset`.
  - For `string` variables: `READ` (pushes address of string read), then `STOREG index` or `STOREL offset` (to store the address).
  - For array elements (assumed integer): Similar logic but uses `STOREN`.
- `write`/`writeln`**(expr1, expr2, ...):** For each expression:
  - If string literal: `PUSHS "string"`, `WRITES`.

– If integer/boolean expression: Evaluate expression, `WRITEI`.

`writeln` additionally emits `WRITELN`.

5 Error Handling Strategy

Error detection occurs at multiple stages:

- **Lexical Errors:** Handled by `t_error` in the lexer, reporting illegal characters.
- **Syntax Errors:** Detected by PLY's parser when the token stream violates grammar rules. The `p_error` function reports these errors.
- **Semantic Errors:** Basic semantic checks (e.g., variable re-declaration, array bound issues, type mismatches for specific operations like `LENGTH`) are performed during parsing or code generation, printing error messages.

A flag, `parser.error`, is set upon encountering any parsing or critical semantic error. The main execution workflow checks this flag; if set, code generation is skipped to prevent producing invalid assembly from erroneous Pascal code.

6 Execution Workflow

The compiler operates as follows:

1. Execute lexer7, the others are just sketches
2. The user is prompted to enter a number corresponding to a Pascal source file (e.g., `programa_pascalX.pas`).
3. The specified source file is read.
4. Lexical analysis tokenizes the input. (A debug mode can show these tokens).
5. Syntactic analysis (parsing) builds the AST. PLY's debug output (`parser.out`) can be generated to inspect parsing tables and conflicts.
6. Semantic information is collected in symbol tables (global and per-function).
7. If parsing and initial semantic checks are successful (i.e., `parser.error` is not set):
   (a) The global symbol table and function scope details are printed for inspection.
   (b) The AST is traversed to generate intermediate VM code.
   (c) The generated assembly instructions are written to `codigo_ensamblador.asm`.
8. If errors occurred, an appropriate message is displayed, and code generation is aborted.

7 Conclusion and Future Work

This document has detailed the implementation of a Pascal compiler front-end, covering lexical analysis, parsing with integrated semantic checks, symbol table management for global and function scopes, and intermediate code generation for key Pascal constructs including arrays, strings (with ‘length‘ and character access), and control flow statements. The use of Python and PLY has facilitated a modular and understandable implementation.

The compiler successfully processes several Pascal examples, generating assembly code for a custom stack-based VM. The current implementation provides a solid foundation.

Future enhancements could include:

- **Full Semantic Analysis:** Implementing a comprehensive type system and type checking for all expressions and statements.
- **Procedure Support:** Adding parsing and code generation for Pascal procedures.
- **More Complex Data Types:** Support for records, pointers, and user-defined types.
- **Advanced Scoping:** Full support for nested functions/procedures and more rigorous scope resolution.
- **Code Optimization:** Implementing basic optimization passes on the AST or intermediate code.
- **Error Recovery:** More sophisticated error recovery in the parser to report multiple errors rather than stopping at the first.
- **Directives for VM:** Exploring if the VM supports data definition directives for strings or other constants to optimize ‘PUSHS‘.

This project serves as a practical demonstration of compiler construction principles and provides a functional tool for understanding the translation process from a high-level language to a machine-executable format.