Project report

MiniPL Interpreter

Compilers CSM14204

Harri Kähkönen

July 31, 2023

Faculty of Science

University of Helsinki

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

Contents

1 Documentation

This is the documentation report for the Compilers course project. In the project, I implemented an interpreter for the Mini-PL programming language following the syntax and semantics of Mini-PL document. [1]. The document is attached to Appendix B. The interpreter implemented seems to recognize correctly all valid Mini-PL programs. In addition, it recognizes syntax and lexical errors during tokenization and parsing. The parser constructs an Abstract Syntax Tree (AST) according to the Mini-PL language grammar. If during lexical analysis, during tokenization and parsing, errors are found, they are prompted, and after recovery, the parsing and analysis continue. If the program is found error-free, the semantic analysis is run to check that expressions are valid. Especially the semantic analysis checks that there are not any type incompatibility issues. If also semantic analysis shows the program is error-free, the program is interpreted.

This report is structured as follows. In the beginning, there are first instructions on how to use the interpreter in Section 1.1. Then, in Section 1.2, regular expressions of the Mini-PL token patterns are shown, including also a discussion of the string literals and escape characters. In Section 1.3, the modifications needed to the grammar given in the Mini-PL language definition are discussed to make the grammar suitable for recursive descent parsing. Continuing the topic, in Section 1.4 the Abstract Syntax Tree (AST) created by the parser is discussed. Section 1.5 concentrates on explaining the error-handling approach in each phase of the compilation and execution. Finally, to complete the requirements, the work-hour log of the project is shown.

1.1 How to use the interpreter

To interpret a program, give the file path of the source code file as the first parameter.

For example:

```
python3 minipl.py sample_programs/sample6.mpl
```

The program should be runnable using the development tools available at the Computer Science Department of University of Helsinki, because it is written and tested using the computer managed by the CS department. In the computer, the newest version of Python is Python 3.6.9, and it is used for running the program.

In some circumstances, Python 3 might be set as a default. Then, instead of `python3` it might be enough to use a command `python`.

In addition to giving the file path of the source code file, you can use the following parameters. The parameters and instructions are also shown to a user, if only `python3 minipl.py` is executed, without any parameters.

Parameter	Description
`--print-tokens`	To print tokens during parsing.
`--print-ast`	To print AST.
`--print-debug-info`	To print information during debugging.
`--print-symbol-table`	To print symbol table after parsing, semantic analysis, and execution.

For example, to pretty print AST of the mini-PL program reachable in `sample_programs/sample6.mpl`, write the following.

```
python3 minipl.py sample_programs/sample6.mpl --print-ast
```

1.2 Mini-PL token patterns

The Mini-PL token patterns are recognized by the regular expressions shown on Table 1.2. Let us first define the following notations on Table 1.1.

Notation	Description
`[0-9]`	Digit from 0 to 9.
`[a-z]`	Letter from a to z.
`[A-Z]`	Letter from A to Z.
`[a|b]`	Symbol a or symbol b
`[ ]*`	Zero or more occurrences of symbols inside [ ].
`[ ]*`	One or more occurrences of symbols inside [ ].
`^a`	Not symbol a.
`^[.]`	Not symbols defined inside [].
`\n`	newline
α	Any character, except newline

Table 1.1: Basic notations.

The following Table 1.2 shows the regular expressions used to recognize token patterns during the scanning phase.

<table>
<tr><th>Token</th><th>Regular expression</th></tr>
<tr><td>+</td><td>+</td></tr>
<tr><td>-</td><td>-</td></tr>
<tr><td>*</td><td>*</td></tr>
<tr><td>/</td><td>/^[/|*]</td></tr>
<tr><td>=</td><td>=</td></tr>
<tr><td><</td><td><</td></tr>
<tr><td>&</td><td>&</td></tr>
<tr><td>!</td><td>!</td></tr>
<tr><td>:</td><td>!^=</td></tr>
<tr><td>:=</td><td>:=</td></tr>
<tr><td>;</td><td>;</td></tr>
<tr><td>..</td><td>..</td></tr>
<tr><td>;</td><td>;</td></tr>
<tr><td>(</td><td>(</td></tr>
<tr><td>)</td><td>)</td></tr>
<tr><td>int</td><td>int^[[a-z]|[A-Z]|[0-9]|_]</td></tr>
<tr><td>string</td><td>string^[[a-z]|[A-Z]|[0-9]|_]</td></tr>
<tr><td>bool</td><td>bool^[[a-z]|[A-Z]|[0-9]|_]</td></tr>
<tr><td>var</td><td>var^[[a-z]|[A-Z]|[0-9]|_]</td></tr>
<tr><td>for</td><td>for^[[a-z]|[A-Z]|[0-9]|_]</td></tr>
<tr><td>end</td><td>end^[[a-z]|[A-Z]|[0-9]|_]</td></tr>
<tr><td>in</td><td>in^[[a-z]|[A-Z]|[0-9]|_]</td></tr>
<tr><td>do</td><td>do^[[a-z]|[A-Z]|[0-9]|_]</td></tr>
<tr><td>read</td><td>read^[[a-z]|[A-Z]|[0-9]|_]</td></tr>
<tr><td>print</td><td>print^[[a-z]|[A-Z]|[0-9]|_]</td></tr>
<tr><td>assert</td><td>assert^[[a-z]|[A-Z]|[0-9]|_]</td></tr>
<tr><td>if</td><td>if^[[a-z]|[A-Z]|[0-9]|_]</td></tr>
<tr><td>else</td><td>else^[[a-z]|[A-Z]|[0-9]|_]</td></tr>
<tr><td>Integer literal</td><td>[0-9]+</td></tr>
<tr><td>String literal</td><td>"[α|[\"]]*"</td></tr>
<tr><td>Identifier</td><td>[[a-z]|[A-Z]][[a-z]|[A-Z]|[0-9]|_]* (More below.)</td></tr>
</table>

Table 1.2: Regular expressions for token patterns.

While scanning, a lexeme starting with a letter is first tested for being an identifier by the regular expression rules shown in Table 1.2. After it is found to match the rules of an identifier, it is tested if it is a proper keyword.

Inside string literals all characters are allowed, except a newline character. The following escape characters, listed in Python language reference [5], are recognized as escape characters and interpreted similarly as in C language strings.

Character	Description
`\\`	Backslash (\)
`\'`	Single quote (')
`\"`	Double quote (")
`\a`	ASCII Bell (BEL)
`\b`	ASCII Backspace (BS)
`\f`	ASCII Formfeed (FF)
`\n`	ASCII Linefeed (LF)
`\r`	ASCII Carriage Return (CR)
`\t`	ASCII Horizontal Tab (TAB)
`\v`	ASCII Vertical Tab (VT)

Table 1.3: Escape characters recognized in string literals.

The following escape sequences are not recognized: `\newline` (Backslash and newline ignored), `\ooo` (Character with octal value ooo), `\xhh` (Character with hex value hh).

Comments are recognized as follows. While scanning, the newlines, whitespaces, tabs, multiline comments with nested multiline comments, and rest-of-the-line comments are skipped, in this order. There is not, however, any preprocessing phase. The scanner is managed by the parser. The parser asks one token at the time from the scanner, and the scanner goes through the raw source code in one pass. The scanner recognizes characters of the raw source code character by character and mostly decides what to do based on the current character, and in some circumstances by deciding after peeking the next character.

A start of a multiline comment is recognized from the character stream when a character `/` is followed by `*`. The skipping of multiline comments keeps track of the depth of the multiline comment blocks, to correctly recognize when each nested block ends. The ending of the comment block is recognized when a character `*` is followed by `/`. The rest-of-the-line comments are recognized if a character `/` is followed by `/`.

1.3 Modified context-free grammar

This section shows the modified context-free grammar suitable for recursive-descent parsing. In the following grammar, LL(1) violations are eliminated, without affecting the language accepted.

```
<prog>             ::= <stmts>
<stmts>            ::= <stmt> ";" ( <stmt> ";" )*
<stmt>             ::= "var" <var_ident> ":" <type> [ ":=" <expr> ]
                    |  <var_ident> ":=" <expr>
                    |  "for" <var_ident> "in" <expr> ".." <expr>
                       "do" <stmts> "end" "for"
                    |  "read" <var_ident>
                    |  "assert" "(" <expr> ")"
                    |  "print" <expr>
                    |  "if" <expr> "do" <stmts> [ "else" <stmts> ]
                       "end" "if"

<expr>             ::= <opnd> <expr_tail> | <unary_op> <opnd>
<expr_tail>        ::= [ <op> <opnd> ]

<opnd>             ::= <int>
                    |  <string>
                    |  <var_ident>
                    |  "(" <expr> ")"

<type>             ::= "int" | "string" | "bool"
<var_ident>        ::= <ident>
<op>               ::= "+", "-", "*", "/", "+", "&", "=", "<"
<unary_op>         ::= "!"
<reserved_keyword> ::= "var" | "for" | "end" | "in" | "do" | "read"
                    |  "print" | "int" | "string" | "bool" | "assert"
                    |  "if" | "else"
```

The modified grammar shown has the following modifications. The rule for the assertion is added, although it is not mentioned in the grammar definition document, to make it possible to run example codes having the keyword `assert`. In addition, the name of the non-terminal <unary_opnd> is changed to <unary_op>, since it means unary operation, not unary operand. For completeness, the rules for <op> and <unary_op are also included.

To make a grammar LL(1) compatible, left recursion and common prefixes must be removed. In the grammar, there are not any left recursion. However, in the original rule for expression, shown below, there is a common prefix problem.

```
<expr>              ::= <opnd> <op> <opnd> | [ <unary_op> ] <opnd>
```

When parsing the expression, the right-hand side (RHS) of the rule allows one to choose either `<opnd> <op> <opnd>` or `[ <unary_op> ] <opnd>`. Since `<unary_op>` has the brackets, it effectively means one can choose either `<opnd> <op> <opnd>` or `<opnd>`. Therefore, since RHS of the `<opnd>` makes it possible to choose from, for example, two equal integer literals, there exists a common prefix problem, making the grammar ambiguous.

The common prefix problem is solved in the modified grammar as shown below.

```
<expr>              ::= <opnd> <expr_tail> | <unary_op> <opnd>
<expr_tail>         ::= [ <op> <opnd> ]
```

This way the RHS of the `<expr>` rule has not any common prefix anymore. The `<opnd>` starts either with a digit (if integer literal), a double quote (if string literal), a letter (if identifier), or with a left parenthesis. Correspondingly, `<unary_op>` starts with a different symbol, an exclamation mark. If the `<opnd>` is followed by `<op>`, the `<expr_tail>` rule is followed.

1.4 Abstract Syntax Tree (AST) spesification

In this section, the AST created by parser is discussed. First, the UML diagram of the AST nodes is shown in Figure 1.3. The diagram can be seen as a scalable SVG picture in the permalink shown here [3]. For easier reading, there are also the UML tables of the AST classes. Also, printouts of the ASTs created by the parser for sample programs are shown. In the end of the section is also UML diagram of the whole interpreter program, and of the subset of classes.

AST Classes

Here are the UML table diagrams of the AST classes. The names of the classes are self-explanatory. However, the naming of the class fields regarding AST nodes is not necessarily the most successful part of the program. To make it easier to notice which fields are AST nodes, and which are lists of AST nodes, the notation `(AST)` and `AST[]` is added to the corresponding fields.

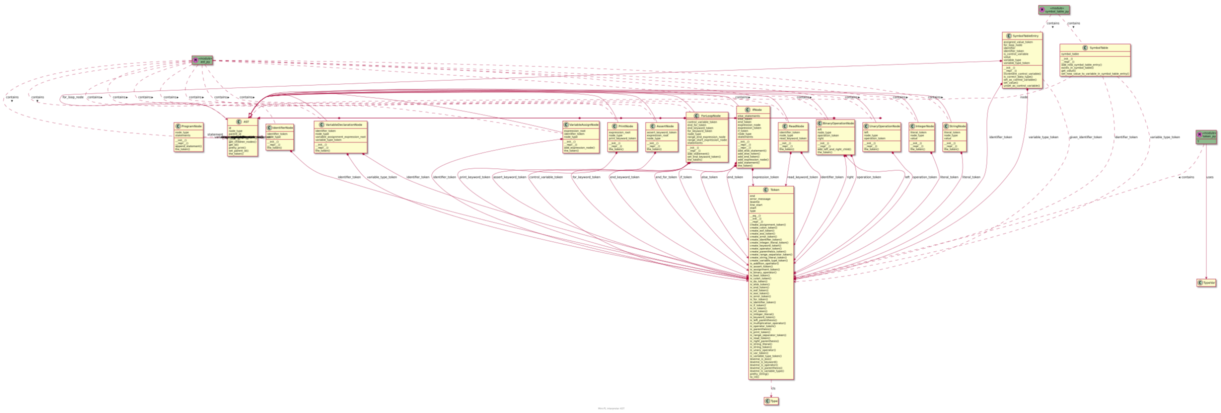

Figure 1.1: UML diagram of AST nodes of Mini-PL interpreter. See the scalable SVG picture in the web address in [3].

AST
id node_type parent_id
__init__() get_children_nodes() get_id() pretty_print() set_parent_id() the_token()

VariableDeclarationNode
identifier_token node_type variable_assignment_expression_root (`AST`) variable_type_token
__init__() __repr__() the_token()

ProgramNode
node_type statements (`AST[]`)
__init__() __repr__() append_statement() the_token()

VariableAssignNode
expression_root (`AST`) identifier_token node_type
__init__() __repr__() add_expression_node() the_token()

IntegerNode
literal_token
node_type
value
__init__()
__repr__()
the_token()

StringNode
literal_token
node_type
value
__init__()
__repr__()
the_token()

IdentifierNode
identifier_token
node_type
__init__()
__repr__()
the_token()

BinaryOperationNode
left (`AST`)
node_type
operation_token
right (`AST`)
__init__()
__repr__()
add_left_and_right_child()
the_token()

UnaryOperationNode
left (`AST`)
node_type
operation_token
__init__()
__repr__()
the_token()

PrintNode
expression_root (`AST`)
node_type
print_keyword_token
__init__()
__repr__()
the_token()

ReadNode
identifier_token
node_type
read_keyword_token
__init__()
__repr__()
the_token()

AssertNode
assert_keyword_token
expression_root (`AST`)
node_type
__init__()
__repr__()
the_token()

ForLoopNode
control_variable_token
end_for_token
end_keyword_token
for_keyword_token
node_type
range_end_expression_node (`AST`)
range_start_expression_node (`AST`)
statements (`AST[]`)
__init__()
__repr__()
add_statement()
set_end_keyword_token()
the_token()

IfNode
else_statements (`AST[]`)
else_token
end_token
expression_node `AST`
expression_token
if_token
node_type
statements (`AST[]`)
__init__()
__repr__()
__repr__()
add_else_statement()
add_else_token()
add_end_token()
add_expression_node()
add_statement()
the_token()

Example programs and pretty printed ASTs

To illustrate ASTs the interpreter implemented produces, here are four simple Mini-PL programs with their ASTs pretty printed. The source code files of the programs can be found in the `sample_programs` folder residing in the project's root folder. The ASTs are printed given a parameter `--print-ast`.

```
// Try also to remove an end of statement symbol (;). It will give an error while parsing, and starts recovering process. The parsing will start after the next semicolon.
var X : int := 4 + (6 * 2);
assert (X=16);
```

```
print X;
print "\n";
```

Function 1.1: Program sample_1.mpl

```
PROGRAM id: 1 parent_id: None. Token: None
- VARIABLE_DECLARATION id: 7 parent_id: 1. Token: type ’IDENTIFIER’, lexeme ’X’, line 2
- - BINARY_OPERATION id: 6 parent_id: None. Token: type ’PLUS’, lexeme ’+’, line 2
- - - INTEGER_LITERAL id: 2 parent_id: 6. Token: type ’INTEGER␣LITERAL’, lexeme ’4’, line 2
- - - BINARY_OPERATION id: 4 parent_id: 6. Token: type ’MUL’, lexeme ’*’, line 2
- - - - INTEGER_LITERAL id: 3 parent_id: 4. Token: type ’INTEGER␣LITERAL’, lexeme ’6’, line 2
- - - - INTEGER_LITERAL id: 5 parent_id: 4. Token: type ’INTEGER␣LITERAL’, lexeme ’2’, line 2
- ASSERT id: 11 parent_id: 1. Token: type ’KEYWORD’, lexeme ’assert’, line 3
- - BINARY_OPERATION id: 10 parent_id: 11. Token: type ’EQUAL’, lexeme ’=’, line 3
- - - IDENTIFIER id: 8 parent_id: 10. Token: type ’IDENTIFIER’, lexeme ’X’, line 3
- - - INTEGER_LITERAL id: 9 parent_id: 10. Token: type ’INTEGER␣LITERAL’, lexeme ’16’, line 3
- PRINT id: 13 parent_id: 1. Token: type ’KEYWORD’, lexeme ’print’, line 4
- - IDENTIFIER id: 12 parent_id: 13. Token: type ’IDENTIFIER’, lexeme ’X’, line 4
- PRINT id: 15 parent_id: 1. Token: type ’KEYWORD’, lexeme ’print’, line 5
- - STRING_LITERAL id: 14 parent_id: 15. Token: type ’STRING␣LITERAL’, lexeme ’
’, line 5
```

Function 1.2: AST of the program sample_1.mpl

The reason why there is a line break after `lexeme ’` in the last AST node printed in AST printout 1.2, is because the statement is `print "\n";`. Thus, when the string literal lexeme is printed, the newline acts as meant to, printing a line break.

```
var nTimes : int := 0;
print "How many times?";
read nTimes;
var x : int;
for x in 0..nTimes-1 do
print x;
print " : Hello, World!\n";
end for;
if x = nTimes do
print "x is equal to nTimes\n";
end if;
```

Function 1.3: Program sample_2.mpl

The code in Program **??** is a corrected version of the one shown on the Mini-PL definition document. In the original version, there is a statement beginning with `if x = ntimes do`, which produces an error because variable `ntimes` is not declared. In this version, `ntimes` is changed to `nTimes`.

```
PROGRAM id: 1 parent_id: None. Token: None
- VARIABLE_DECLARATION id: 3 parent_id: 1. Token: type 'IDENTIFIER', lexeme 'nTimes', line 1
- - INTEGER_LITERAL id: 2 parent_id: None. Token: type 'INTEGER␣LITERAL', lexeme '0', line 1
- PRINT id: 5 parent_id: 1. Token: type 'KEYWORD', lexeme 'print', line 2
- - STRING_LITERAL id: 4 parent_id: 5. Token: type 'STRING␣LITERAL', lexeme 'How␣many␣times?', line 2
- READ id: 6 parent_id: 1. Token: type 'KEYWORD', lexeme 'read', line 3
- VARIABLE_DECLARATION id: 7 parent_id: 1. Token: type 'IDENTIFIER', lexeme 'x', line 4
- FOR_LOOP id: 12 parent_id: 1. Token: type 'KEYWORD', lexeme 'for', line 5
- - INTEGER_LITERAL id: 8 parent_id: None. Token: type 'INTEGER␣LITERAL', lexeme '0', line 5
- - BINARY_OPERATION id: 11 parent_id: None. Token: type 'SUB', lexeme '-', line 5
- - - IDENTIFIER id: 9 parent_id: 11. Token: type 'IDENTIFIER', lexeme 'nTimes', line 5
- - - INTEGER_LITERAL id: 10 parent_id: 11. Token: type 'INTEGER␣LITERAL', lexeme '1', line 5
- - PRINT id: 14 parent_id: 12. Token: type 'KEYWORD', lexeme 'print', line 6
- - - IDENTIFIER id: 13 parent_id: 14. Token: type 'IDENTIFIER', lexeme 'x', line 6
- - PRINT id: 16 parent_id: 12. Token: type 'KEYWORD', lexeme 'print', line 7
- - - STRING_LITERAL id: 15 parent_id: 16. Token: type 'STRING␣LITERAL', lexeme '␣:␣Hello,␣World!
', line 7
- IF id: 20 parent_id: 1. Token: type 'KEYWORD', lexeme 'if', line 9
- - BINARY_OPERATION id: 19 parent_id: 20. Token: type 'EQUAL', lexeme '=', line 9
- - - IDENTIFIER id: 17 parent_id: 19. Token: type 'IDENTIFIER', lexeme 'x', line 9
- - - IDENTIFIER id: 18 parent_id: 19. Token: type 'IDENTIFIER', lexeme 'nTimes', line 9
- - PRINT id: 22 parent_id: 20. Token: type 'KEYWORD', lexeme 'print', line 10
- - - STRING_LITERAL id: 21 parent_id: 22. Token: type 'STRING␣LITERAL', lexeme 'x␣is␣equal␣to␣nTimes
', line 10
```

Function 1.4: AST of the program sample_2.mpl

The following Program **??** is also from the Mini-PL definition document.

```
print "Give a number: ";
var n : int;
read n;
var v : int := 1;
var i : int;
```

```
for i in 1..n do
v := v * i;
end for;
print "The result is: ";
print v;
print "\n";
```

Function 1.5: Program sample_3.mpl

```
PROGRAM id: 1 parent_id: None. Token: None
- PRINT id: 3 parent_id: 1. Token: type ’KEYWORD’, lexeme ’print’, line 1
- - STRING_LITERAL id: 2 parent_id: 3. Token: type ’STRING␣LITERAL’, lexeme ’Give␣a␣number:␣’, line 1
- VARIABLE_DECLARATION id: 4 parent_id: 1. Token: type ’IDENTIFIER’, lexeme ’n’, line 2
- READ id: 5 parent_id: 1. Token: type ’KEYWORD’, lexeme ’read’, line 3
- VARIABLE_DECLARATION id: 7 parent_id: 1. Token: type ’IDENTIFIER’, lexeme ’v’, line 4
- - INTEGER_LITERAL id: 6 parent_id: None. Token: type ’INTEGER␣LITERAL’, lexeme ’1’, line 4
- VARIABLE_DECLARATION id: 8 parent_id: 1. Token: type ’IDENTIFIER’, lexeme ’i’, line 5
- FOR_LOOP id: 11 parent_id: 1. Token: type ’KEYWORD’, lexeme ’for’, line 6
- - INTEGER_LITERAL id: 9 parent_id: None. Token: type ’INTEGER␣LITERAL’, lexeme ’1’, line 6
- - IDENTIFIER id: 10 parent_id: None. Token: type ’IDENTIFIER’, lexeme ’n’, line 6
- - VARIABLE_ASSIGN id: 12 parent_id: 11. Token: type ’IDENTIFIER’, lexeme ’v’, line 7
- - - BINARY_OPERATION id: 14 parent_id: 12. Token: type ’MUL’, lexeme ’*’, line 7
- - - - IDENTIFIER id: 13 parent_id: 14. Token: type ’IDENTIFIER’, lexeme ’v’, line 7
- - - - IDENTIFIER id: 15 parent_id: 14. Token: type ’IDENTIFIER’, lexeme ’i’, line 7
- PRINT id: 17 parent_id: 1. Token: type ’KEYWORD’, lexeme ’print’, line 9
- - STRING_LITERAL id: 16 parent_id: 17. Token: type ’STRING␣LITERAL’, lexeme ’The␣result␣is:␣’, line 9
- PRINT id: 19 parent_id: 1. Token: type ’KEYWORD’, lexeme ’print’, line 10
- - IDENTIFIER id: 18 parent_id: 19. Token: type ’IDENTIFIER’, lexeme ’v’, line 10
- PRINT id: 21 parent_id: 1. Token: type ’KEYWORD’, lexeme ’print’, line 11
- - STRING_LITERAL id: 20 parent_id: 21. Token: type ’STRING␣LITERAL’, lexeme ’
’, line 11
```

Function 1.6: AST of the program sample_3.mpl

The fourth of the sample programs, `sample_6.mpl`, is written for testing purposes by the author. Especially, this is to test the correctness of the use of if and for statements. In the program, there is an if statement with else block, and two nested for loops. The Mini-PL interpreter created in this work interprets it correctly.

```
print "Give a number in range [3, 20]: ";
var count : int;
read count;
```

```

if (!(count < 3)) & (count < 21) do
  var e : int;
  var p : int;
  for e in 1..count do
    for p in 1..e do
      print "*";
    end for;
    print "\n";
  end for;
  else print "You gave too low or high number. Goodbye!\n";
end if;

assert (e = (count + 1));
assert (p = (count + 1));
```

Function 1.7: Program sample_6.mpl

```
PROGRAM id: 1 parent_id: None. Token: None
- PRINT id: 3 parent_id: 1. Token: type ’KEYWORD’, lexeme ’print’, line 1
- - STRING_LITERAL id: 2 parent_id: 3. Token: type ’STRING␣LITERAL’, lexeme ’Give␣a␣number␣in␣range␣[3,␣20]:␣’, line 1
- VARIABLE_DECLARATION id: 4 parent_id: 1. Token: type ’IDENTIFIER’, lexeme ’count’, line 2
- READ id: 5 parent_id: 1. Token: type ’KEYWORD’, lexeme ’read’, line 3
- IF id: 14 parent_id: 1. Token: type ’KEYWORD’, lexeme ’if’, line 5
- - BINARY_OPERATION id: 13 parent_id: 14. Token: type ’AND’, lexeme ’&’, line 5
- - - UNARY_OPERATION id: 9 parent_id: 13. Token: type ’NOT’, lexeme ’!’, line 5
- - - - BINARY_OPERATION id: 8 parent_id: 9. Token: type ’SMALLER’, lexeme ’<’, line 5
- - - - - IDENTIFIER id: 6 parent_id: 8. Token: type ’IDENTIFIER’, lexeme ’count’, line 5
- - - - - INTEGER_LITERAL id: 7 parent_id: 8. Token: type ’INTEGER␣LITERAL’, lexeme ’3’, line 5
- - - BINARY_OPERATION id: 12 parent_id: 13. Token: type ’SMALLER’, lexeme ’<’, line 5
- - - - IDENTIFIER id: 10 parent_id: 12. Token: type ’IDENTIFIER’, lexeme ’count’, line 5
- - - - INTEGER_LITERAL id: 11 parent_id: 12. Token: type ’INTEGER␣LITERAL’, lexeme ’21’, line 5
- - VARIABLE_DECLARATION id: 15 parent_id: 14. Token: type ’IDENTIFIER’, lexeme ’e’, line 6
- - VARIABLE_DECLARATION id: 16 parent_id: 14. Token: type ’IDENTIFIER’, lexeme ’p’, line 7
- - FOR_LOOP id: 19 parent_id: 14. Token: type ’KEYWORD’, lexeme ’for’, line 8
- - - INTEGER_LITERAL id: 17 parent_id: None. Token: type ’INTEGER␣LITERAL’, lexeme ’1’, line 8
- - - IDENTIFIER id: 18 parent_id: None. Token: type ’IDENTIFIER’, lexeme ’count’, line 8
- - - FOR_LOOP id: 22 parent_id: 19. Token: type ’KEYWORD’, lexeme ’for’, line 9
- - - - INTEGER_LITERAL id: 20 parent_id: None. Token: type ’INTEGER␣LITERAL’, lexeme ’1’, line 9
- - - - IDENTIFIER id: 21 parent_id: None. Token: type ’IDENTIFIER’, lexeme ’e’, line 9
- - - - PRINT id: 24 parent_id: 22. Token: type ’KEYWORD’, lexeme ’print’, line 10
- - - - - STRING_LITERAL id: 23 parent_id: 24. Token: type ’STRING␣LITERAL’, lexeme ’*’, line 10
- - - PRINT id: 26 parent_id: 19. Token: type ’KEYWORD’, lexeme ’print’, line 12
- - - - STRING_LITERAL id: 25 parent_id: 26. Token: type ’STRING␣LITERAL’, lexeme ’
’, line 12
- - PRINT id: 28 parent_id: 14. Token: type ’KEYWORD’, lexeme ’print’, line 14
- - - STRING_LITERAL id: 27 parent_id: 28. Token: type ’STRING␣LITERAL’, lexeme ’You␣gave␣too␣low␣or␣high␣number.␣Goodbye!
’, line 14
- ASSERT id: 34 parent_id: 1. Token: type ’KEYWORD’, lexeme ’assert’, line 17
- - BINARY_OPERATION id: 33 parent_id: 34. Token: type ’EQUAL’, lexeme ’=’, line 17
- - - IDENTIFIER id: 29 parent_id: 33. Token: type ’IDENTIFIER’, lexeme ’e’, line 17
- - - BINARY_OPERATION id: 32 parent_id: 33. Token: type ’PLUS’, lexeme ’+’, line 17
- - - - IDENTIFIER id: 30 parent_id: 32. Token: type ’IDENTIFIER’, lexeme ’count’, line 17
- - - - INTEGER_LITERAL id: 31 parent_id: 32. Token: type ’INTEGER␣LITERAL’, lexeme ’1’, line 17
- ASSERT id: 40 parent_id: 1. Token: type ’KEYWORD’, lexeme ’assert’, line 18
- - BINARY_OPERATION id: 39 parent_id: 40. Token: type ’EQUAL’, lexeme ’=’, line 18
- - - IDENTIFIER id: 35 parent_id: 39. Token: type ’IDENTIFIER’, lexeme ’p’, line 18
- - - BINARY_OPERATION id: 38 parent_id: 39. Token: type ’PLUS’, lexeme ’+’, line 18
- - - - IDENTIFIER id: 36 parent_id: 38. Token: type ’IDENTIFIER’, lexeme ’count’, line 18
- - - - INTEGER_LITERAL id: 37 parent_id: 38. Token: type ’INTEGER␣LITERAL’, lexeme ’1’, line 18
```

Function 1.8: AST of the program sample_6.mpl

More sample programs can be found in the `sample_programs` and `example_codes` folder, residing in the project root folder. Especially the programs in `example_codes` were extensively used while implementing the lexical analyzer (scanner), parser, semantic analyzer, and interpreter. Some of the programs have intentionally written with syntax and semantical errors, to test that the errors are found by the interpreter program.

Modularity of the Mini-PL interpreter program

The interpreter is written to be modular enough. File handling and I/O primitives for reading input from the user, and printing are abstracted. The class structure can be examined in the UML diagram in Figure 1.2. Since the figure is so large, and embedded in this report as a png picture, which does not scale well, it might be hard to see the details. The scalable SVG figure is available in the web address given in the reference [2].

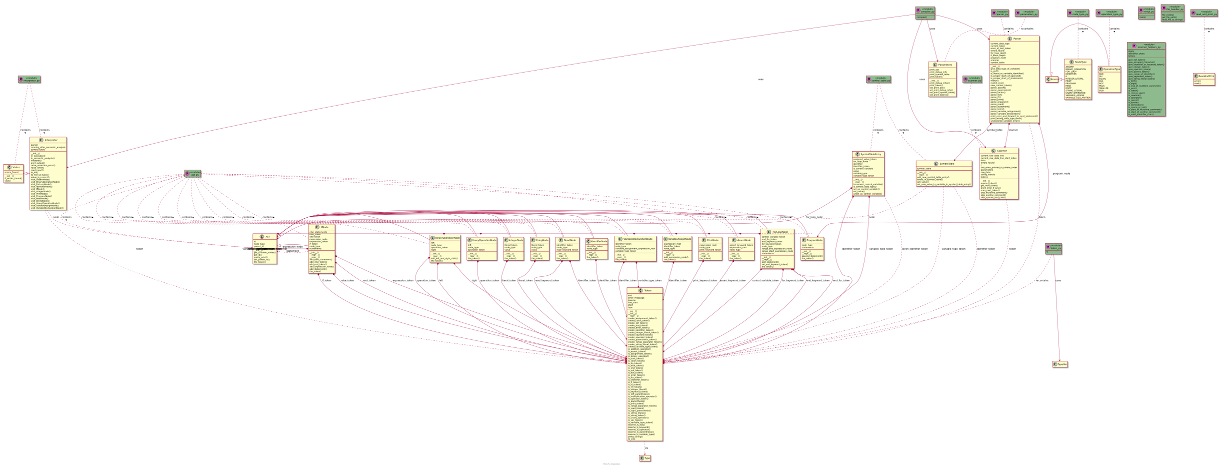

Figure 1.2: UML diagram of the Mini-PL interpreter. See the scalable SVG picture in the web address in [2].

In Figure 1.3 there is also a UML diagram of the AST nodes, symbol table, and Tokens are shown.

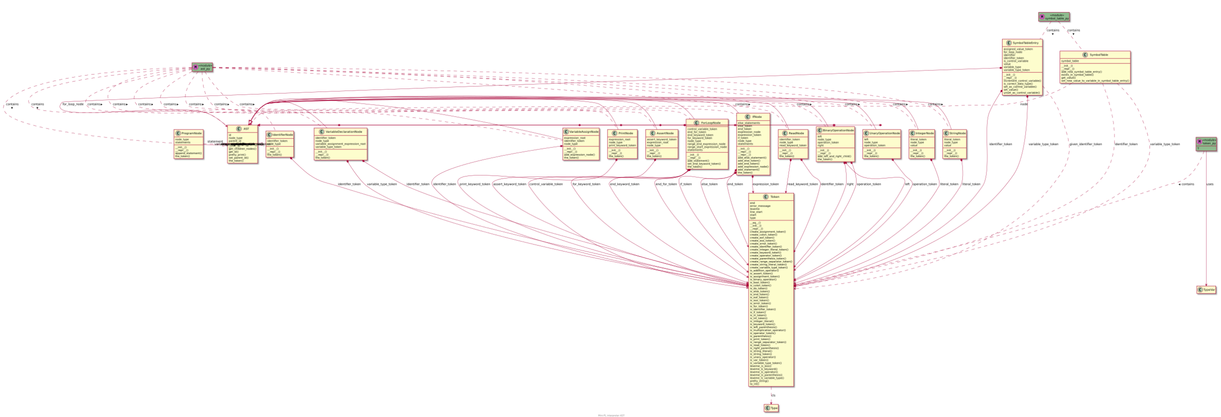

Figure 1.3: UML diagram of AST, Symbol Table, and Token of Mini-PL interpreter. See the scalable SVG picture in the web address in [4].

1.5 Error handling approach

In this section, the error handling approach and solutions used in the implementation of the scanner, parser, semantic analyzer, and interpreter are discussed.

1.5.1 Lexical analysis in Scanner and Parser

In the implementation, lexical analysis is implemented in cooperation with the scanner and parser.

The scanner is managed by the parser. The parser asks next token from the parser. While generating the token, the scanner recognizes all correct tokens. If the token is not part of the language, an error is found. If an error is found, the scanner creates an error token and returns it to the parser similarly to the syntactically correct tokens.

For example, if in a for loop, there are three dots between the range expressions, the scanner notices the correct two dots token and gives it to the parser. But when the next token is asked, the scanner notices a lonely dot, prompts an error message, and returns an error token to the parser.

When the error is such that the parser does not know what is meant in the erroneous code, the parser recovers from the error using panic mode recovery. First, it prompts an error message with information about the incorrect token, such as a line number and erroneous token lexeme. Or, if the token was a proper token of the language, but was not expected regarding the rules of the language, the parser prompts an illustrative message.

For example, if after the keyword `var` the next token is not an identifier, the parser prompts that error is found in the line where the error was found, and "There should be a variable identifier after keyword var, but found INTEGER 12", assuming that the token expected to be an identifier is an integer literal and the lexeme is 12.

Then, it recovers by asking for new tokens from the scanner, until the end of the statement (EOS) token is found. The parsing starts again after the EOS token. However, if the end of the file (EOF) is reached while in panic mode recovery, it prompts about it, and ends the parsing.

If errors were found, after parsing the message is prompted to tell that the program is not executed due the errors.

The parser is responsible for putting the symbols in the symbol table with the information of an identifier and type (int, string, or bool). Thus, if a variable is already declared, the error message is prompted. In this case, however, the parsing continues without entering the panic mode recovery, because there is not any need to forward to the next syntactically correct statement. After parsing is completed, however, the message "Since the program is not error-free, it is not interpreted." is prompted and the execution of the interpreter program is ended.

So, there are two kinds of error recovery during scanning and parsing. If the parser does not know what to do, for example, because the token is not part of the language, or because the order of tokens does not follow the grammar, the parser enters panic mode recovery. If, however, the error is such that there was not any syntactical problem, such as in the re-declaration of the variable, the parser just prompts the error, adds the counter of the errors, and continues parsing.

1.5.2 Semantical analysis before interpretation

The semantical analysis is implemented in the Interpreter class. The class travels the AST, starting from the ProgramNode, and interprets the program. However, before interpretation, a class field variable is used to determine if the state of the program is semantic analysis or interpretation.

During semantic analysis, all the same actions are done as during interpretation, but with the following differences. The values are not stored in the symbol table, to make sure that the interpretation will start with correct values. Also, the analyzer does not read values from the user but instead uses just some proper values based on the type of the variable in which the value is read. The analyzer does not print anything. The analyzer goes through the whole codebase. It is worth mentioning that in the if-else statements, the analyzer visits both blocks, regardless of the condition's truthfulness.

During the analysis, any incorrect values in expressions such as mismatching types are found. In addition, if the control variable of the for-loop is tried to be changed during the for-loop, it causes an error. After exiting the for-loop, the variable set as the control variable is unset since it has no anymore role as a control variable.

Although not connected to the error handling, it is worth mentioning that the for-loop expressions are evaluated only at the start of the for-loop, as expected based on the Mini-PL definition document.

If the semantical analyzer did not find any errors, the program is immediately executed by the interpreter.

1.5.3 Runtime error handling

Since the semantic analyzer and interpreter use the same code to visit the AST nodes and even statements in both if and else blocks are evaluated during analysis, there should not be any semantical errors during runtime. However, there could still happen at least divide by zero error.

For example, consider the integer value is read from the user to a variable x. The user enters 0. Then there is an expression where the variable x is the right-hand-side operand of the division. The interpreter checks the divisor before dividing, and if the divisor is zero, the interpreter prompts about dividing by zero and ends the execution of the program.

Another run-time error is the assertion error, which is useful for the programmer to check the correctness of her own code. If this error occurs, the interpreter prompts an error message and stops the execution of the program.

1.5.4 Testing the correctness of the interpreter program

During and after the development, the interpreter program is tested by running 23 different Mini-PL programs. Three of the programs are given in the Mini-PL definition document [1]. All programs in the project source code folders `example_codes` and `sample_programs` are used for testing.

Most of the 23 Mini-PL programs are carefully written by the author to be error-free, to test that the interpreter program recognizes and executes valid Mini-PL programs. Some of the programs are written such that they purposedly contain lexical, syntax, or semantical errors, to test that the interpreter recognizes invalid programs, and gives descriptive enough error messages.

The interpreter recognized and executed correctly all valid programs. In addition, all invalid programs were recognized, with error messages. To conclude, it seems that the interpreter works correctly.

1.6 Work Hour Log

The following work hour log shows the number of hours used working on the project, including descriptions of the work done.

Date	Hours	Description
8.6.2023	5	Building the basic structure and file handling.
9.6.2023	9	Implementing the scanner.
10.6.2023	5	Implementing the scanner continues.
24.6.2023	7	Implementing the scanner continues.
25.6.2023	9	Scanner completed. Started to build the parser.
13.7.2023	6	Implementing Symbol Table. Starting to write the report.
17.7.2023	4	Implementing the parser.
21.7.2023	8	Implementing the parser.
22.7.2023	7	Implementing the parser.
25.7.2023	5	Implementing the parser. Enhancements to the code base.
26.7.2023	4	Implementing the parser. Refactoring.
27.7.2023	8	AST nodes, creating the nodes during parsing.
28.7.2023	10	Finishing the creation of the AST nodes during parsing. Pretty printing of AST.
29.7.2023	12	Implementing the Interpreter.
30.7.2023	12	Implementing Semantic Analyzer, corrections, completion of the program. Writing the report.
31.7.2023	8	Finalizing the report.
Total hours	119	

Table 1.4: Work hour log.

Bibliography

[1] U. of Helsinki. *Syntax and semantics of Mini-PL*. https://moodle.helsinki.fi/course/view.php?id=56500. 2023.

[2] H. Kähkönen. *Mini-PL interpreter UML diagram*. 2023. URL: https://plantuml.
atug.com/svg/lHfTR-Csybt0_WS3VTaMpHxQnyKYWFVY10QIEESu0Uw902pQ9Y9BgYXl4lJwtply7dv8IPp
69fPZYS6SuNIQqhAiHapslULgHzcNpxjc_AGyLEJ-5nyldgrwPRxtZvymG0FvtHxl6MqOkEiNhoPp61tyd9Y
_hdugrWhXkjct_AATHzXQNaejWNcU79oEhcX7TspddLYydLo44ma5BThMjUi6q830p6CdTV0kkta18v4QNRF
RpiZAkgxf1DaqXxfK04B0b3qX8BHyvHk2IGpqX-IXrJ9SyNIQSwl4Ku_TeQvvlILq0sWYs9xMFLyJMjFgHN2
g6fI4yVAfOLf2RicQqFFMzg3qfuEPGHnxot7K18S-ZRWmLrb0j6G7RMIPvQpG0NG1WjGI0_
l6_W5Rr83oJWWlrBowoLQ7LWX0koRhgEhNjiF39L4ryt9Gl0ttbDkvT5opegfKhmvqrtsJHj0f-
N3EomuQnBKPiK1ERPPaXlf9aoq6OBlfh0bs2J17ZR0uzj0huRL-kUTbokF5Y5RuVHPmoyBM017cWT-
jl_JhOR5jWbltmvdQoQHmZHhx1Oy9zWfxShmDN0rNWqzmJlJG8WL3JwonSNTryXBzGzP1inESsGXN6BIBLK6
S7SfFYmPovw6p9RrXC5RMc_mrwlWE3XGgqiof5o_QTrgPp61dtdWq8EDgbFaIjc83dHmVX-
cwOCm5w141c3eJIBH8Iuxz5RoHuEMuZdJOE0IKRIOEK-8fUYObuYWAjn1YB0rawE1EOv92WiKuBKac6Jjqxv
mA5afWHCvHFzBHdUs2KNLf2EjPsxfMsBGC_T6FoCqkBoAI4fYwm63xjE4052YqaAdVPNVFFoA-
2uv6zF3LTYOZ6zHWGsCvUE35-aIs5JQ2PgVA4YTYsMGYArTjrF4gPSB8YcljDyWwKdxSB_
iCmaEq6r3Uvqf1ymP_X5LgFhcdwa4wE_EWg3aXT36apReetZSh9FpfGk_zRp1CjXMyKR2ERCUU7NOgM9HQaz
A_hlbH6de12YHBe6DMf9XuL4lKtHirPPdjeEBxoFAAGWNQD5q4IGqYQL9eL3relOL-i4xS7M6XI7Lz5l--
sbccXLEMyKJuyVL0fYgms08L6R3U87rgga7IrDO9gLxgCUED3icjngWutXamsKYNeNU6DcDKWf5e9HKizAzi
iiDhOp9afIJ8q47767pDc3yOsARNA9jB6aQ7qxljtrRqm3agtQqAb1UWSmRkoCY4LExC10Ilf1AcsCSHhN-
IhbNc7r2f7b5HlFB49wKFB-DJYgmAKJ5nbPOfCDaRoY9Jg0ikfns5ZIQgI5hWuVsGNu2GiyUUWeWGn6L7-
WzV-HrCzNeAb7vDHYKd7CaOeP1PjxlQPeL-zHUCywGEBsVmFuQDSrG3MqLNY573NAWnhOJdE007Lf0nIuf0u
4hy5pZszFRszbofHw_pw-do_-InSriELtD5zSAUBvOail5uaQzpCzcrwlv-NosrA_d-
kzwDRkO0Tby1ICl5Unc2L3pjBXOJg_Kyt8sFTCFYuN6tgw0y28OUfUHudwwd4-_Nyu8Y3k_
Y61ditzVJWqnUFYiFknLN5bDZU8kPe-8cc0icDWPCMDWjupOojTQxxFvlVgV_UVEQ7jrf_-
lZK5kBkzkjMMkffUNnj2tTzy3pO8fpBGexnwdOye050sLoP-s2dS3j3vqwcIkf3trwT52KICfxraWLP9AUcq
PtlBXwfPFo9GgzCLsfxeUpJz_CerqvR_O5vWqcpSti7UB0d2cer_iRaOH7Ex3B-bztGWLIillzG-
ODWg-xeBjGn1t7xkQR_UuMUGmnwJrko917Qr9wPoESmTLm3opM40tj1rxwwy32XwvvlW5H_
6MeWoVm0pLm4U42kdeM39S5pcO2YFBtt520xTjMjHlJbwecglHchstASIAFgdhgctQFtn5I1uePg5bNvUYuU
ItXpQtqlm2eVN4a6aW9x92OBosxWrOOilTLon_xYg5ihFr0Vzuly9HHIOOORx1T_90EBfCmZBpx34mWLjoua
rxr5y5nuqfvCqTCQ_NdVA5P48VdkOEr_i6EhgWgQG8g-HrFZHC3_PF33j-KZXEdhD5VeJk_
S9LIYYniyaiMPEXutBitt--AUjczeOofU5Xi7a7AGo1lhYK9IDp8RQDR2ujBhbPhqHvkivF4Qv4eFdO6WI4h

V-SSIFF-y8s7o-ulNiOWdCS7P4FVPBQmmdVqIGT6nMisZ375ZKCYVLPK4fClzLnljMH3e7EuoizCVeDXxZ
KFLScM7TxF7XfqGsDHj2YPQkcL1531RAtE_QTHWg0QRk7ED4VgHq1rV94bUmX3jMjKVSCAlrvrjkZlvGOA
B5AuFiK5GPDsKA45aswRvHZwR827IZHRdFiNPK7KceAJm8n3hgQQL-5Vj7tz-vxN1q5a5VSmI9sIamTOIw
vnQ7f4BzYUhvkgGHJAfUKwW4CZg6rzHyJt_NCsO802qZmlIUHEd_rNApfQp9ctoJOwaFVG4TCjCvxt31s1
uGkbDWVg9QDbJPeDA3ln1UTU7KDLbahXuZTte555goqoY3Y8XPXJGZatC-GvCB7eB9nOd3wIuqT4pmhhxa
gmkRoB1-JN2JOFwZWdm-YXQPAGJHTKXcuJGv9B2nFHCIZo0XtL2IXPVIxy5P473JY3n2JJy4I9-
Cic3SB-un4vhr59QDDZrq3fIEQ1jk9hEh1eMMUp2Idcbdw61S0ndvDe-QbB35EN5TafGlTCu16Zmu07WGl
1eKez8rVAuGOZN2AblijotWIelsazZePlGsLPrAUR8gB8V8YijZkaQLfAn6mrD2hxkDIsAGwiT4KnzJ_
rtq4n-OmdxdKR-NK6wFhcvnDzxVYfWUZPX-xP_cUpP_W5wa4_59zaQGLVysxz-B_u6.

[3] H. Kähkönen. *Mini-PL interpreters AST UML diagram*. 2023. URL: https://plantuml.
atug.com/svg/lLXDJzmm4BrRuZz4wWMgxbRjiK8hKQ5MID2YGdkrp6QoM6Jjo35jLxJ_
lUCacqmSsxjyg5p0upUUDsyc_c1PYhhEJgwkimUMuO_CcrzAvy3jfe9cL0a3ol9kZdEffENyyAWPhi3ovQ
vizbR3AllGE7uBsDisT2xD7fbnFeBBjxH7FEhhR2mneNg447E3LIc7ugg0fNptYSSsIbo_
6T6GpDAOpvVDAfz689lsNu3aHEgCiSvMKWmxVe1_WOz-1riVccJ4r0wZmZXRNJMqtWzcosoQuT4je5GOU2
N8vFIcOSZ7cnz-BxGnWfRaeu1HmROQLMZoF-irk0usfofLobEVp6ViQXLjneRSSuv_dSAaPOirWsQIdsyW
FF5ufJXeagK98pcUtbCXpIiIvxgVvG8y56E-djE_GQXNHvT9jYZlay3Pav_5IEqOVdWr4Hbc6eddGP7ue1
ToImUKiO6vnztTkRJWzCTunFowNeGgHYLRLIHY7fH_5IgA7IgzGJQbZebNj8bL8qadQ4PA2nnopWhw8vHr
4uXj0CeYsdUFwfpGvZmYxK_GDcgCAf_QWjjxg4bZzukYlLSKWQrzctXeacWRdEwr3ixwEbFGP8jHX62Y7J
HgrkxvpRWbkK24ssSsXTOMkmOHHQYFHAhOHz5jMAaCVtDjqbknr8xGDkrVF6w8dn_WHeywO2R1cO1BhvaC
KXTNe-bzAYyWoSOKkxv_p2XW2CXfyt2WZ3XtLAulNbDN_Oph4OoHqWcTBKzJu4jvanyThdT-
Ni5ZWdzdXnHjjyBFAZp8ql3eLvkuAn6S3kCo32zfvpUTTJTs1AQIgimMPec_M2AH__qNclLTN11LxAko0a
e0Ti7efIiZUlNtF_W40.

[4] H. Kähkönen. *Mini-PL interpreters AST, Symbol Table, and Token UML diagram*.
2023. URL: https://plantuml.atug.com/svg/lLbBR-Cs4BuBo7yGqCjkqQHeZqKGO8jjYm2xRT1aUoK
IXa_RoQA-s9e7vvi7XqDvrTDnRD_TthNFJGiVtjal8Jj6z7vWzccWYWbDF8yGBhWcvCrR-
xXXcgmUUD-1335w8sBmpZyCYccz9uFYGdlI2FnvPK_J3s1MFZ81p_ysJREoy6wKt2Yw_
HqMzp13VAEcccrXeJ47a1351ZK_qc5WeYEpJjwoYSZbF-VOT8DXN5nSsQZzj1YVqFx9Q8UWyWvuVCsPSkj
qMg3WdL_IW0dgmtCunlpy-lsZiZ2MWpflA22MpltVLCFyt75paJ6ciYsqVX_OigJkzxzfx1iwAQIt4QyCz
OGAhyOqmcIRi2UeP7gKWIaeToXdDrsQnWBefULCFnL5QEVnLHLV-nuGdIiKxAucoa0ldgqkATXq6ObIFY_
a2hyaWFYxIiOABLI4kDwOf7WX7r92DnjVg6BFvPW4IU9cp5ebM_ODWC8KTf9QGmN_OynTvk01e49R0nooa
QZS4HSEAQMp_DFEpYuzECqnde2lvmE5gi5X2p9IY5vHFWrLKJIDTIBgKbqBNbKbbuqgdAKOi5XbohXUuux
7DWpwo3PXb2GL7hPAx0TI3dO7Ov607mRBeYr8k67MPUVArpXrOwe3Gj5OSF9-4JryX1oywu5B3JGPB7BAI
Z-aYxq6cQxWT33f23VXVNjvkPNThcTNL_2pVVFnEwbWMtLlMo1yyvwgnrj6VrCCbdasHhtMnONFgJamAHT
CvQ49dlVmL-TEigI9Mk2WV4qQ1Sz62QD0i2Q4ehIEOZjUJakiRDDOI2Q6PJbKoYKYw8e28A97nrxxVtihy
EN-KhvpyP74erYqLOARuolkmboib3tK4eEJW86Y55jfqb5DxUfe2gKfKiIrUFtpJfJbOMSbWQrV-

khYkLZaDTms3hEKI9ePO5fwmpypGKflFf2ghupHKeYUStZuc1Vi2y5gC8S12Y_0dlYeHrWZ446l_
t3CJSJDQbDLBeDzf3-R23CHU5xBGI1u8FfefBBXTf4Bq9pUc6Driv7ZD0RRkQwGs_hElUTfidVVIa7fR65kf
u1F-8ZFDxluNkbcplr4br0e3jYcCd_YD3vxCFNSb-sbo21bChlFaCKksNcKOSGb4NDTSIN3SXYAv2wjdkSad
YzLT6Dj3Y2gFjwZLecQfTMY1iIswdiWab0hct6B52Fa_0kRVdMcQXUodxM3XDA10x6LEIsvkXHPStGuggm9B
nZchsrPKs4vPdGeuLsXGg24ev5orNj6XLnDP3GVCWJUcHqOQHkBRm0HFZdb7Dmcl0bSNzLGKDttTOUAw1a-
d4_-aPilJZrZ33htvc5Pste-FYzb-_rCzlE4s0QfcQGs8qCp8E7JC6q8CtNHsZNvpSVMbEsohW3D3ZMpXg0m
tRBjFLFZMNFMZ2iAvyeFt__O_0S0.

[5] *Python language reference - Lexical Analysis*. Web. Accessed: 2023-07-31. 2021. URL: https://docs.python.org/3.6/reference/lexical_analysis.html.

Appendix A Appendix Programs PlantUML code

In this section, the UML diagram of the whole interpreter program is shown as PlantUML code.

```
class minipl_py <<module>> << (M,orchid) >> #DarkSeaGreen {
    ---
    main()
}
class compiler_py <<module>> << (M,orchid) >> #DarkSeaGreen {
    ---
    compiler()
}

compiler_py --> Parameters : uses
compiler_py --> Scanner : uses
compiler_py --> Parser : uses
compiler_py --> Interpreter : uses
class Visitor {
    errors_found
    __init__()
    if_errors_found()
    visit()
}


class Interpreter {
    parser
    running_after_semantic_analysis
    symbol_table
    __init__()
    in_execution()
    in_semantic_analysis()
    interpret()
    print_output()
    raise_assertion_error()
    raise_error()
    read_input()
    to_int()
    to_mini_pl_type()
    value_is_correct()
    visit_AssertNode()
```

```
    visit_BinaryOperationNode()
    visit_ForLoopNode()
    visit_IdentifierNode()
    visit_IfNode()
    visit_IntegerNode()
    visit_PrintNode()
    visit_ProgramNode()
    visit_ReadNode()
    visit_StringNode()
    visit_UnaryOperationNode()
    visit_VariableAssignNode()
    visit_VariableDeclarationNode()
}

Visitor <|- Interpreter
Interpreter ..> Token : token
Interpreter ..> AST : node
class interpreter_py <<module>> << (M,orchid) >> #DarkSeaGreen {
}

interpreter_py .. Visitor : contains >
interpreter_py .. Interpreter : contains >
class file_handler_py <<module>> << (M,orchid) >> #DarkSeaGreen {
    ---
    file_exists()
    get_file_path()
    read_file_to_string()
}
class ReadAndPrint {
    print()
    read()
}

class read_and_print_py <<module>> << (M,orchid) >> #DarkSeaGreen {
}

read_and_print_py .. ReadAndPrint : contains >
class Parameters {
    print_ast
    print_debug_info
    print_symbol_table
    print_tokens
    __init__()
    print_debug_infos()
```

```
    print_token()
    set_print_ast()
    set_print_debug_info()
    set_print_symbol_table()
    set_print_tokens()
}

class parameters_py <<module>> << (M,orchid) >> #DarkSeaGreen {
}

parameters_py .. Parameters : contains >
class AST {
    id
    node_type
    parent_id
    __init__()
    get_children_nodes()
    get_id()
    pretty_print()
    set_parent_id()
    the_token()
}


class ProgramNode {
    node_type
    statements
    __init__()
    __repr__()
    append_statement()
    the_token()
}

AST <|- ProgramNode
ProgramNode ..> AST : statement

class ReadNode {
    identifier_token
    node_type
    read_keyword_token
    __init__()
    __repr__()
    the_token()
}
```

```
AST <|- ReadNode
ReadNode *--> Token : read_keyword_token
ReadNode *--> Token : identifier_token

class IdentifierNode {
    identifier_token
    node_type
    __init__()
    __repr__()
    the_token()
}

AST <|- IdentifierNode
IdentifierNode *--> Token : identifier_token

class VariableDeclarationNode {
    identifier_token
    node_type
    variable_assignment_expression_root
    variable_type_token
    __init__()
    __repr__()
    the_token()
}

AST <|- VariableDeclarationNode
VariableDeclarationNode *--> Token : identifier_token
VariableDeclarationNode *--> Token : variable_type_token
VariableDeclarationNode *--> AST : variable_assignment_expression_root

class VariableAssignNode {
    expression_root
    identifier_token
    node_type
    __init__()
    __repr__()
    add_expression_node()
    the_token()
}

AST <|- VariableAssignNode
VariableAssignNode *--> Token : identifier_token
VariableAssignNode *--> AST : expression_root
```

```
class PrintNode {
    expression_root
    node_type
    print_keyword_token
    __init__()
    __repr__()
    the_token()
}

AST <|- PrintNode
PrintNode *--> Token : print_keyword_token
PrintNode *--> AST : expression_root

class AssertNode {
    assert_keyword_token
    expression_root
    node_type
    __init__()
    __repr__()
    the_token()
}

AST <|- AssertNode
AssertNode *--> Token : assert_keyword_token
AssertNode *--> AST : expression_root

class ForLoopNode {
    control_variable_token
    end_for_token
    end_keyword_token
    for_keyword_token
    node_type
    range_end_expression_node
    range_start_expression_node
    statements
    __init__()
    __repr__()
    add_statement()
    set_end_keyword_token()
    the_token()
}

AST <|- ForLoopNode
```

```
ForLoopNode *--> Token : control_variable_token
ForLoopNode *--> Token : for_keyword_token
ForLoopNode *--> Token : end_keyword_token
ForLoopNode *--> AST : range_start_expression_node
ForLoopNode *--> AST : range_end_expression_node
ForLoopNode *--> Token : end_for_token
ForLoopNode ..> AST : statement

class IfNode {
   else_statements
   else_token
   end_token
   expression_node
   expression_token
   if_token
   node_type
   statements
   __init__()
   __repr__()
   __repr__()
   add_else_statement()
   add_else_token()
   add_end_token()
   add_expression_node()
   add_statement()
   the_token()
}

AST <|- IfNode
IfNode *--> Token : if_token
IfNode *--> Token : else_token
IfNode *--> Token : end_token
IfNode *--> Token : expression_token
IfNode *--> AST : expression_node
IfNode ..> AST : statement

class BinaryOperationNode {
   left
   node_type
   operation_token
   right
   __init__()
   __repr__()
   add_left_and_right_child()
```

```
    the_token()
}

AST <|- BinaryOperationNode
BinaryOperationNode *--> Token : left
BinaryOperationNode *--> AST : left
BinaryOperationNode *--> Token : right
BinaryOperationNode *--> AST : right
BinaryOperationNode *--> Token : operation_token

class UnaryOperationNode {
    left
    node_type
    operation_token
    __init__()
    __repr__()
    the_token()
}

AST <|- UnaryOperationNode
UnaryOperationNode *--> AST : left
UnaryOperationNode *--> Token : operation_token

class IntegerNode {
    literal_token
    node_type
    value
    __init__()
    __repr__()
    the_token()
}

AST <|- IntegerNode
IntegerNode *--> Token : literal_token

class StringNode {
    literal_token
    node_type
    value
    __init__()
    __repr__()
    the_token()
}
```

```
AST <|- StringNode
StringNode *--> Token : literal_token
class ast_py <<module>> << (M,orchid) >> #DarkSeaGreen {
}

ast_py .. AST : contains >
ast_py .. ProgramNode : contains >
ast_py .. ReadNode : contains >
ast_py .. IdentifierNode : contains >
ast_py .. VariableDeclarationNode : contains >
ast_py .. VariableAssignNode : contains >
ast_py .. PrintNode : contains >
ast_py .. AssertNode : contains >
ast_py .. ForLoopNode : contains >
ast_py .. IfNode : contains >
ast_py .. BinaryOperationNode : contains >
ast_py .. UnaryOperationNode : contains >
ast_py .. IntegerNode : contains >
ast_py .. StringNode : contains >
class NodeType {
    ASSERT
    BINARY_OPERATION
    FOR_LOOP
    IDENTIFIER
    IF
    INTEGER_LITERAL
    PRINT
    PROGRAM
    READ
    ROOT
    STRING_LITERAL
    UNARY_OPERATION
    VARIABLE_ASSIGN
    VARIABLE_DECLARATION
}

Enum <|- NodeType
class node_type_py <<module>> << (M,orchid) >> #DarkSeaGreen {
}

node_type_py .. NodeType : contains >
class OperationType {
    AND
    DIV
```

```
    EQUAL
    MUL
    NOT
    PLUS
    SMALLER
    SUB
}

Enum <|- OperationType
class operation_type_py <<module>> << (M,orchid) >> #DarkSeaGreen {
}

operation_type_py .. OperationType : contains >
class Parser {
    current_data_type
    current_token
    error_in_last_token
    errors_found
    for_loop_depth
    if_block_depth
    program_node
    scanner
    symbol_table
    __init__()
    give_data_type_of_variable()
    is_eof()
    is_literal_or_variable_identifier()
    is_proper_start_of_operand()
    is_proper_start_of_statement()
    match()
    match_eos()
    new_current_token()
    parse_assert()
    parse_expression()
    parse_factor()
    parse_for()
    parse_if()
    parse_print()
    parse_program()
    parse_read()
    parse_statement()
    parse_term()
    parse_variable_assignment()
    parse_variable_declaration()
```

```
    print_error_and_forward_to_next_statement()
    print_wrong_data_type_error()
    undeclared_variable_error()
}

Parser *--> Scanner : scanner
Parser *--> SymbolTable : symbol_table
Parser *--> ProgramNode : program_node
Parser ..> Token : token
class parser_py <<module>> << (M,orchid) >> #DarkSeaGreen {
}

parser_py .. Parser : contains >
class SymbolTableEntry {
    assigned_value_token
    for_loop_node
    identifier
    identifier_token
    is_control_variable
    value
    variable_type
    variable_type_token
    __init__()
    __repr__()
    increment_control_variable()
    is_correct_data_type()
    set_as_control_variable()
    set_value()
    unset_as_control_variable()
}

SymbolTableEntry *--> Token : identifier_token
SymbolTableEntry *--> Token : variable_type_token
SymbolTableEntry *--> AST : for_loop_node
SymbolTableEntry ..> Token : given_identifier_token

class SymbolTable {
    symbol_table
    __init__()
    __repr__()
    add_new_symbol_table_entry()
    exists_in_symbol_table()
    get_value()
    set_new_value_to_variable_in_symbol_table_entry()
```

```
}

SymbolTable ..> Token : identifier_token
SymbolTable ..> Token : variable_type_token
SymbolTable ..> AST : node
class symbol_table_py <<module>> << (M,orchid) >> #DarkSeaGreen {
}

symbol_table_py .. SymbolTableEntry : contains >
symbol_table_py .. SymbolTable : contains >
class Scanner {
    current_raw_data_line
    current_raw_data_line_start_index
    data
    errors_found
    i
    last_error_printed_in_tokens_index
    parameters
    raw_data
    string_literals
    tokens
    __init__()
    append_token()
    get_next_token()
    print_error_if_any()
    scan_next_token()
    skip_multiline_comment()
    skip_oneline_comment()
    skip_spaces_and_tabs()
}

Scanner ..> Token : token
class scanner_py <<module>> << (M,orchid) >> #DarkSeaGreen {
}

scanner_py .. Scanner : contains >
class scanner_helpers_py <<module>> << (M,orchid) >> #DarkSeaGreen {
    digits
    identifier_chars
    letters
    ---
    give_eof_token()
    give_escaped_character()
    give_identifier_or_keyword_token()
```

```
    give_integer_token()
    give_operator_token()
    give_parens_token()
    give_range_of_identifier()
    give_separator_token()
    give_string_literal_token()
    is_EOF()
    is_digit()
    is_end_of_multiline_comment()
    is_eos()
    is_letter()
    is_minus_sign()
    is_newline()
    is_operator()
    is_paren()
    is_quote()
    is_semicolon()
    is_space_or_tab()
    is_start_of_multiline_comment()
    is_start_of_oneline_comment()
    is_valid_identifier_char()
}
class Token {
    end
    error_message
    lexeme
    line_start
    start
    type
    __eq__()
    __init__()
    __repr__()
    create_assignment_token()
    create_colon_token()
    create_eof_token()
    create_eos_token()
    create_error_token()
    create_identifier_token()
    create_integer_literal_token()
    create_keyword_token()
    create_operator_token()
    create_parenthesis_token()
    create_range_separator_token()
    create_string_literal_token()
```

```
    create_variable_type_token()
    is_addition_operator()
    is_assert_token()
    is_assignment_token()
    is_binary_operator()
    is_bool_token()
    is_colon_token()
    is_do_token()
    is_else_token()
    is_end_token()
    is_eof_token()
    is_eos_token()
    is_error_token()
    is_for_token()
    is_identifier_token()
    is_if_token()
    is_in_token()
    is_int_token()
    is_integer_literal()
    is_keyword_token()
    is_left_parenthesis()
    is_multiplication_operator()
    is_operator_token()
    is_parenthesis()
    is_print_token()
    is_range_separator_token()
    is_read_token()
    is_right_parenthesis()
    is_string_literal()
    is_string_token()
    is_unary_operator()
    is_var_token()
    is_variable_type_token()
    lexeme_is_eos()
    lexeme_is_keyword()
    lexeme_is_operator()
    lexeme_is_parenthesis()
    lexeme_is_variable_type()
    pretty_string()
    to_int()
}

Token ..> Type : cls
class token_py <<module>> << (M,orchid) >> #DarkSeaGreen {
```

```
    T
}

token_py .. Token : contains >
token_py --> TypeVar : uses
center footer Mini-PL interpreter
hide empty members

scale 1/2
```

Appendix B Appendix Mini-PL definition document

In this appendix, starting after this page, there is the Mini-PL language definition document referred to in the report.

Syntax and semantics of Mini-PL (16.01.2023)

Mini-PL is a simple programming language designed for pedagogic purposes. The language is purposely small and is not actually meant for any real programming. Mini-PL contains few statements, arithmetic expressions, and some IO primitives. The language uses static typing and has three built-in types representing primitive values: int, string, and bool. The BNF-style syntax of Mini-PL is given below, and the following paragraphs informally describe the semantics of the language.

Mini-PL uses a single global scope for all different kinds of names. All variables must be declared before use, and each identifier may be declared once only. If not explicitly initialized, variables are assigned an appropriate default value.

The Mini-PL read statement can read either an integer value or a single word (string) from the input stream. Both types of items are whitespace-limited (by blanks, newlines, etc). Likewise, the print statement can write out either integers or string values. A Mini-PL program uses default input and output channels defined by its environment.

The arithmetic operator symbols '+', '-', ' * ','/' represent the following functions:

```
‘+’ : (int, int) → int        // integer addition
‘-’ : (int, int) → int        // integer subtraction
‘*’ : (int, int) → int        // integer multiplication
‘/’ : (int, int) → int        // integer division
```

The operator '+' also represents string concatenation (i.e., this one operator symbol is overloaded):

```
‘+’ : (string, string) → string // string concatenation
```

The operators '&' and '!' represent logical operations:

```
‘&’ : (bool, bool) → bool     // logical and
‘!’ : (bool) → bool           // logical not
```

The operators '=' and b '<' are overloaded to represent the comparisons between two values of the same type T (int, string, or bool):

```
‘=’ : (T, T) → bool           // equality comparison
‘<’ : (T, T) → bool           // less than comparison
```

A for statement iterates over the consequent values from a specified integer range. The expressions specifying the beginning and end of the range are evaluated once only, at the beginning of the for statement. The for control variable behaves like a constant inside the loop: it cannot be assigned another value (before exiting the for statement). A control variable needs to be declared before its use in the for statement (in the global scope). Note that loop control variables are not declared inside for statements.

The if statement evaluates the boolean expression between the “**for**” and “**do**” and executes all the statements after “**if**”, if the expression evaluates as true. The optional “**else**” statement will occur if the expression is false.

Context-free grammar for Mini-PL

The syntax definition is given in so-called Extended Backus-Naur form (EBNF). In the following Mini-PL grammar, the notation X* means 0, 1, or more repetitions of the item X. The '|' operator is used to define alternative constructs. Parentheses may be used to group together a sequence of related symbols. Brackets ("[" "]") may be used to enclose optional parts (i.e., zero or one occurrence). Reserved keywords are marked bold (as "var"). Operators, separators, and other single or multiple character tokens are enclosed within quotes (as: " .. "). Note that nested expressions are always fully parenthesized to specify the execution order of operations.

```
<prog>    ::=  <stmts>
<stmts>   ::=  <stmt> “;” ( <stmt> “;” )*
<stmt>    ::=  “var” <var_ident> “:” <type> [ “:=” <expr> ]
               | <var_ident> “:=” <expr>
               | “for” <var_ident> “in” <expr> “..” <expr> “do”
                 <stmts> “end” “for”
               | “read” <var_ident>
               | “print” <expr>
               | “if” <expr> “do” <stmts> [ “else” <stmts> ] “end”
“if”

<expr>    ::= <opnd> <op> <opnd>
               | [ <unary_opnd> ] <opnd>

<opnd>    ::= <int>
               | <string>
               | <var_ident>
               | “(” <expr> “)”

<type>    ::= “int” | “string” | “bool”
<var_ident> ::= <ident>

<reserved_keyword> ::=
               “var” | “for” | “end” | “in” | “do” | “read” |
               “print” | “int” | “string” | “bool” | “assert” |
               “if” | “else”
```

Lexical elements

In the syntax definition the symbol <ident> stands for an identifier (name). An identifier is a sequence of letters, digits, and underscores, starting with a letter. Uppercase letters are distinguished from lowercase.

In the syntax definition the symbol <int> stands for an integer constant (literal). An integer constant is a sequence of decimal digits. The symbol <string> stands for a string literal. String literals follow the C-style convention: any special characters, such as the quote character (") or backslash (\), are represented using escape characters (e.g.: \").

A limited set of operators include (only!) the ones listed below.

```
'+' | '-' | ' * ' | '/' | '<' | '=' | '&' | '!'
```

In the syntax definition the symbol <op> stands for a binary operator symbol. There is one unary operator symbol (<unary_op>): '!', meaning the logical not operation. The operator symbol '&' stands for the logical “and” operation. Note that in Mini-PL, '=' is the equal operator - not assignment. The predefined type names (e.g.,"int") are reserved keywords, so they cannot be used as (arbitrary) identifiers. In a Mini-PL program, a comment may appear between any two tokens. There are two forms of comments: one starts with " /* ", ends with " */ ", can extend over multiple lines, and may be nested. The other comment alternative begins with " // " and goes only to the end of the line.

Sample Programs

```
var X : int := 4 + (6 * 2);
print X;
```

```
var nTimes : int := 0;
print "How many times?";
read nTimes;
var x : int;
for x in 0..nTimes-1 do
     print x;
     print " : Hello, World!\n";
end for;
if x = ntimes do
     print “x is equal to ntimes”;
end if;
```

```
print "Give a number";
var n : int;
read n;
var v : int := 1;
var i : int;
for i in 1..n do
     v := v * i;
end for;
print "The result is: ";
print v;
```