

MINISTRY OF EDUCATION OF REPUBLIC OF MOLDOVA  
TECHNICAL UNIVERSITY OF MOLDOVA  
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS  
SOFTWARE ENGINEERING DEPARTMENT

## COMPUTER PROGRAMMING

INDIVIDUAL WORK

---

# Domain-Specific Language Calculator

---

*Author:*

Alexandru RUDOI

std. gr. FAF-231

*Verified:*

Alexandru FURDUI

Chişinău 2023

# Contents

<b>Introduction</b>	<b>2</b>
Overview	3
PyScript Grammar in Extended BNF	4
<b>1 The Lexer</b>	<b>7</b>
1.1 Design	7
1.2 Encoding the DFA as a Table	10
1.3 Implementation in C++	14
1.3.1 The Token Class	14
1.3.2 The Tokenisation Algorithm	14
1.3.3 The Lexer Class	16
1.4 Testing	17
<b>2 The Parser</b>	<b>20</b>
2.1 Design	21
2.2 Implementation in C++	22
2.3 Testing	23
<b>3 Representing the Syntax Tree in XML</b>	<b>25</b>
3.1 Design	25
3.2 Implementation in C++	26
3.3 Testing the Parser using XML	26
<b>4 Semantic Analysis</b>	<b>30</b>
4.1 Design	30
4.2 Implementation in C++	32
4.3 Testing	33
<b>5 The Interpreter</b>	<b>35</b>
5.1 Design and C++ Implementation	35
5.2 Testing	36
<b>6 The PyScript Read-Eval-Print-Loop</b>	<b>40</b>
6.1 Design	40
6.2 Implementation in C++	41
6.3 Testing	41

## The Task

In this task, you will create your own simple calculator Domain-Specific Language (DSL), similar to the example provided during our class.

## Introduction

The goal of this assignment was to implement a lexer, parser, interpreter and Read-Eval-Print-Loop environment for the programming language PyScript.

PyScript is an expression-oriented strongly-typed programming language. It supports C-style comments, that is, `//...` for single-line comments and `/*...*/` for multiline comments, is case-sensitive, and every function is expected to return a value. There are four datatypes in the language: `int` for integer values, `real` for floating point numbers, `bool` for boolean values and `string` for string literals. The language detects when integer literals are assigned to real variables and performs an automatic typecast, except for the case of function parameters. For example, the function call `f(1)` to `f(real)` is invalid, so we must write `f(1.0)` or `f(1.)` for the desired result. This strictness in turn allows for multiple functions of the same name to be defined within the same scope, provided they have different signature. PyScript supports variable shadowing.

The following is a syntactically and semantically correct program written in PyScript.

```
1  def square(x : real) : real { return x*x; }
2
3  def lt_hundred(x : real) : bool { return x < 100; }
4
5  def repeat_string(s : string, n : int) : string {
6      var s_rep : string = "";
7      while (n > 0) {
8          set s_rep = s_rep + s;
9          set n = n - 1;
10     }
11     return s_rep;
12 }
13
14 def fac(n : int) : int {
15     if (n == 0) { return 1; }
16     else { return n * fac(n-1); }
17 }
18
19 print lt_hundred(fac(5) - square(5.)); // true, since 95. < 100
20 print repeat_string("Hello", 10);      // Prints 10 "Hello"s
21 print square(fac(5)+0.);               /* 14400, the '+0.'
22                                     converts the integer argument to a real */
```

## Overview

The process of compilation can be viewed as a four-stage procedure: lexical analysis (a.k.a lexing or tokenisation), then syntax analysis (a.k.a parsing), followed by semantic analysis, and finally execution. Accordingly, the report is organised into the following tasks (chapters):

### TASK I: The Lexer.

In this task, a lexer is designed and built in C++. The lexer's job is to take the program as a single string of characters, and return a set of substrings called *tokens*, each of which is a basic lexical unit of the PyScript language.

### TASK II: The Parser.

Here we design and build a parser in C++. The parser's job is to use the tokens provided by the lexer to ensure that the program conforms to the formal grammar of the PyScript language (given in the next section). This is achieved by building a *syntax tree* for the input program. Whilst building the syntax tree, the parser will be able to report any syntax errors which occur in the program.

### TASK III: Representing the Syntax Tree in XML.

Here we verify that the parser is doing its job correctly. Using the visitor pattern from OOP, we 'visit' each node in the syntax tree and produce a properly indented XML file where each tag represents a node in tree. For example, the valid program:

```
var x : int = 1 + 2;
```

should produce the XML:

```
<program>
  <decl>
    <id type = "int">x</id>
    <bin op = "+">
      <int>1</int>
      <int>2</int>
    </bin>
  </decl>
</program>
```

### TASK IV: Semantic Analysis.

This task is similar to the previous, in that the visitor design pattern is used to visit each node in the abstract syntax tree. Instead of generating an XML file however, we perform certain semantic checks at each node; such as type-checking the operands in binary expressions, ensuring that functions return a value, handling undefined references, and so on.

### TASK V: The Interpreter.

Again the visitor design pattern is used here, however this time it is used to evaluate and execute the statements and expressions in the program. No invalid types/undefined references need to be handled here, since they were already dealt with in the previous task. Only runtime errors may occur (such as division by zero).

**TASK VI: Read-Eval-Print-Loop (REPL).**

This task involved the implementation of a LISP/Python-like interactive terminal environment, in which expressions and programs can be directly input and used.

## PyScript Grammar in Extended BNF

The following production rules in extended BNF fully describe the grammar of the PyScript programming language.

```
 $\langle \text{program} \rangle \quad ::= \{ \langle \text{statement} \rangle \}$   
  
 $\langle \text{block} \rangle \quad ::= \text{'{' } \{ \langle \text{statement} \rangle \} \text{'}'}$   
  
 $\langle \text{statement} \rangle \quad ::= \langle \text{variable-decl} \rangle \text{' ;'}$   
 $\quad \quad \quad | \langle \text{assignment} \rangle \text{' ;'}$   
 $\quad \quad \quad | \langle \text{print-statement} \rangle \text{' ;'}$   
 $\quad \quad \quad | \langle \text{if-statement} \rangle$   
 $\quad \quad \quad | \langle \text{while-statement} \rangle$   
 $\quad \quad \quad | \langle \text{return-statement} \rangle \text{' ;'}$   
 $\quad \quad \quad | \langle \text{function-decl} \rangle$   
 $\quad \quad \quad | \langle \text{block} \rangle$   
  
 $\langle \text{function-decl} \rangle \quad ::= \text{'def' } \langle \text{identifier} \rangle \text{' (' } [\langle \text{formal-params} \rangle] \text{' )' ':' } \langle \text{type} \rangle \langle \text{block} \rangle$   
  
 $\langle \text{formal-params} \rangle \quad ::= \langle \text{formal-param} \rangle \{ \text{' , ' } \langle \text{formal-param} \rangle \}$   
  
 $\langle \text{formal-param} \rangle \quad ::= \langle \text{identifier} \rangle \text{' :' } \langle \text{type} \rangle$   
  
 $\langle \text{while-statement} \rangle \quad ::= \text{'while' ' (' } \langle \text{expression} \rangle \text{' )' } \langle \text{block} \rangle$   
  
 $\langle \text{if-statement} \rangle \quad ::= \text{'if' ' (' } \langle \text{expression} \rangle \text{' )' } \langle \text{block} \rangle [\text{'else' } \langle \text{block} \rangle]$   
  
 $\langle \text{return-statement} \rangle ::= \text{'return' } \langle \text{expression} \rangle$   
  
 $\langle \text{print-statement} \rangle ::= \text{'print' } \langle \text{expression} \rangle$ 
```

$$\begin{aligned}
\langle \text{variable-decl} \rangle &::= \text{'var'} \langle \text{identifier} \rangle \text{' : ' } \langle \text{type} \rangle \text{' = ' } \langle \text{expression} \rangle \\
\langle \text{assignment} \rangle &::= \text{'set'} \langle \text{identifier} \rangle \text{' = ' } \langle \text{expression} \rangle \\
\langle \text{expression} \rangle &::= \langle \text{simple-expression} \rangle \{ \langle \text{relational-op} \rangle \langle \text{simple-expression} \rangle \} \\
\langle \text{simple-expression} \rangle &::= \langle \text{term} \rangle \{ \langle \text{additive-op} \rangle \langle \text{term} \rangle \} \\
\langle \text{term} \rangle &::= \langle \text{factor} \rangle \{ \langle \text{multiplicative-op} \rangle \langle \text{factor} \rangle \} \\
\langle \text{factor} \rangle &::= \langle \text{literal} \rangle \\
&\quad | \langle \text{identifier} \rangle \\
&\quad | \langle \text{function-call} \rangle \\
&\quad | \langle \text{sub-expression} \rangle \\
&\quad | \langle \text{unary} \rangle \\
\langle \text{unary} \rangle &::= ( \text{' + ' } | \text{' - ' } | \text{' not ' } ) \{ \langle \text{expression} \rangle \} \\
\langle \text{sub-expression} \rangle &::= \text{' ( ' } \langle \text{expression} \rangle \text{' ) ' } \\
\langle \text{function-call} \rangle &::= \langle \text{identifier} \rangle \text{' ( ' } [ \langle \text{actual-params} \rangle ] \text{' ) ' } \\
\langle \text{actual-params} \rangle &::= \langle \text{expression} \rangle \{ \text{' , ' } \langle \text{expression} \rangle \} \\
\langle \text{relational-op} \rangle &::= \text{' < ' } | \text{' > ' } | \text{' == ' } | \text{' != ' } | \text{' <= ' } | \text{' >= ' } \\
\langle \text{additive-op} \rangle &::= \text{' + ' } | \text{' - ' } | \text{' or ' } \\
\langle \text{multiplicative-op} \rangle &::= \text{' * ' } | \text{' / ' } | \text{' and ' } \\
\langle \text{identifier} \rangle &::= ( \text{' _ ' } | \langle \text{letter} \rangle ) \{ \text{' _ ' } | \langle \text{letter} \rangle | \langle \text{digit} \rangle \} \\
\langle \text{literal} \rangle &::= \langle \text{int-literal} \rangle \\
&\quad | \langle \text{real-literal} \rangle \\
&\quad | \langle \text{bool-literal} \rangle \\
&\quad | \langle \text{string-literal} \rangle \\
\langle \text{int-literal} \rangle &::= \langle \text{digit} \rangle \{ \langle \text{digit} \rangle \}
\end{aligned}$$

$\langle \text{real-literal} \rangle ::= \langle \text{digit} \rangle \{ \langle \text{digit} \rangle \} ' . ' \{ \langle \text{digit} \rangle \}$   
 $\quad \mid ' . ' \langle \text{digit} \rangle \{ \langle \text{digit} \rangle \}$

$\langle \text{bool-literal} \rangle ::= \text{'true'} \mid \text{'false'}$

$\langle \text{string-literal} \rangle ::= \text{'\"'} \{ \langle \text{printable} \rangle \} \text{'\"'}$

$\langle \text{printable} \rangle ::= \langle \text{digit} \rangle \mid \langle \text{letter} \rangle \mid \text{'\u00a0'} \mid \text{'!' } \mid \text{'\"'} \mid \text{'\#'} \mid \text{'$'} \mid \text{'\%'} \mid \text{'\&'} \mid \text{'\' } \mid \text{'(' } \mid$   
 $\text{' )' } \mid \text{'*'} \mid \text{'+' } \mid \text{' , ' } \mid \text{' - ' } \mid \text{' . ' } \mid \text{' / ' } \mid \text{' : ' } \mid \text{' ; ' } \mid \text{' < ' } \mid \text{' = ' } \mid \text{' > ' } \mid \text{' ? ' } \mid$   
 $\text{'@' } \mid \text{'[ ' } \mid \text{' \ ' } \mid \text{' ] ' } \mid \text{' ^ ' } \mid \text{' _ ' } \mid \text{' ` ' } \mid \text{' { ' } \mid \text{' | ' } \mid \text{' } ' } \mid \text{' ~ ' }$

$\langle \text{letter} \rangle ::= \text{'A' } \mid \text{'B' } \mid \text{'C' } \mid \text{'D' } \mid \text{'E' } \mid \text{'F' } \mid \text{'G' } \mid \text{'H' } \mid \text{'I' } \mid \text{'J' } \mid \text{'K' } \mid \text{'L' } \mid \text{'M' } \mid$   
 $\text{'N' } \mid \text{'O' } \mid \text{'P' } \mid \text{'Q' } \mid \text{'R' } \mid \text{'S' } \mid \text{'T' } \mid \text{'U' } \mid \text{'V' } \mid \text{'W' } \mid \text{'X' } \mid \text{'Y' } \mid \text{'Z' } \mid$   
 $\text{'a' } \mid \text{'b' } \mid \text{'c' } \mid \text{'d' } \mid \text{'e' } \mid \text{'f' } \mid \text{'g' } \mid \text{'h' } \mid \text{'i' } \mid \text{'j' } \mid \text{'k' } \mid \text{'l' } \mid \text{'m' } \mid$   
 $\text{'n' } \mid \text{'o' } \mid \text{'p' } \mid \text{'q' } \mid \text{'r' } \mid \text{'s' } \mid \text{'t' } \mid \text{'u' } \mid \text{'v' } \mid \text{'w' } \mid \text{'x' } \mid \text{'y' } \mid \text{'z' }$

$\langle \text{digit} \rangle ::= \text{'0' } \mid \text{'1' } \mid \text{'2' } \mid \text{'3' } \mid \text{'4' } \mid \text{'5' } \mid \text{'6' } \mid \text{'7' } \mid \text{'8' } \mid \text{'9' }$

# TASK I

## The Lexer

The first stage of compilation is the procedure of lexical analysis; that is, the process of converting the string of characters with which we are presented (i.e. the program) into a sequence of *tokens*, which are substrings of the program string (called *lexemes*) together with an assigned meaning (which we are calling the *token type*).

For example, lexical analysis of the statement

```
var x : int = 1 + 2;
```

results in the sequence of lexemes {'var', 'x', ':', 'int', '=', '1', '+', '2', ';' }, to each of which we subsequently assign a token type, obtaining the tokens. The tokens are then made accessible to the parser via the `next_token()` function which will then generate an abstract syntax tree of the program.

### 1.1 Design

The goal here was to design a lexer for the PyScript programming language. Before any code was written, the set of tokens accepted by the language was realised as a discrete finite-state automaton (DFA), built up gradually. First, the DFA in [figure 1.1](#) was constructed to recognise digits such as '3', '0' and '456'.

This was then extended to the DFA shown in [figure 1.2](#) to recognise rational numbers

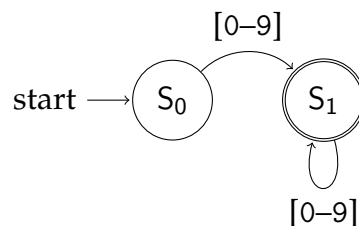


Figure 1.1: DFA recognising sequences of digits



such as '3.14', '0.2', '.75', '3.0' and '3.' corresponding to the type `real`.

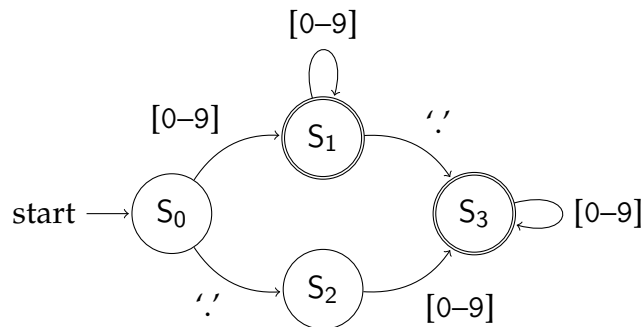


Figure 1.2: DFA recognising real numbers

Next a DFA recognising the operators `+`, `-`, `*`, `=`, `==`, `<`, `>`, `<=`, `>=` and `!=` was constructed, show in [figure 1.3](#). Note that the `+` and `-` operators were given a separate final state to the `*` operator. This is only done to aid in distinguishing between them later when assigning each of them a token type. Also, observe that we have not considered the `/` (division) operator here. This symbol is treated separately due to its use in code comments (`//` and `/* ... */`).

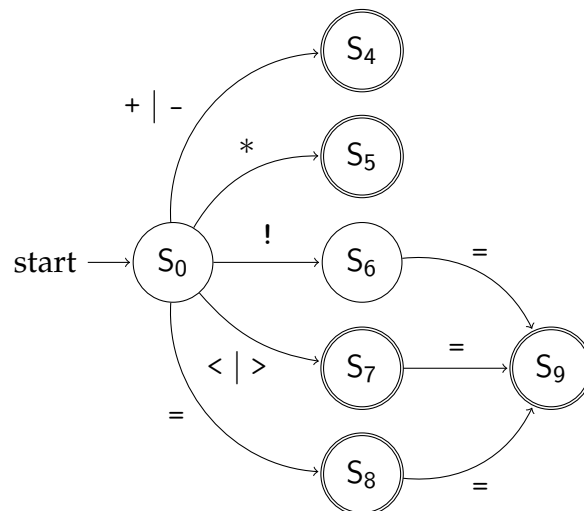


Figure 1.3: DFA recognising the operators `+`, `-`, `*`, `=`, `==`, `<`, `>`, `<=`, `>=`, `!=`

Subsequently, a DFA for *identifiers* was constructed, by which we mean names of variables and functions, as well as keywords such as `var` and `if`. At this stage we do not distinguish between keywords and variable names, we will do so later when assigning token types. Identifiers are made up of underscores (`_`) and letters (`A-Z`, `a-z`). They may also contain but cannot start with digits (`0-9`). The DFA recognising identifiers can be seen in [figure 1.4](#).

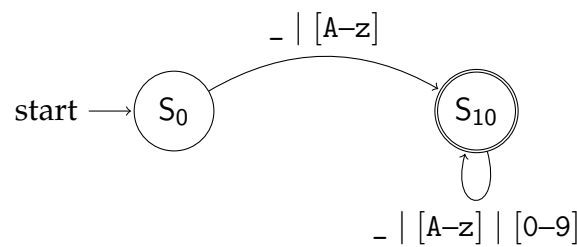


Figure 1.4: DFA recognising identifiers

Now for the division operator and code comments. As is customary in C-like programming languages, single line comments start with two forward slashes (//) and are terminated by a newline character ( $\backslash n$ ). Multiline comments start with the sequence of characters  $/*$  and are terminated by  $*/$ . One needs to be careful when dealing with  $*$  characters since they may or may not indicate that the comment is coming to an end. The DFA which was constructed to recognise single and multiline comments can be seen in [figure 1.5](#).

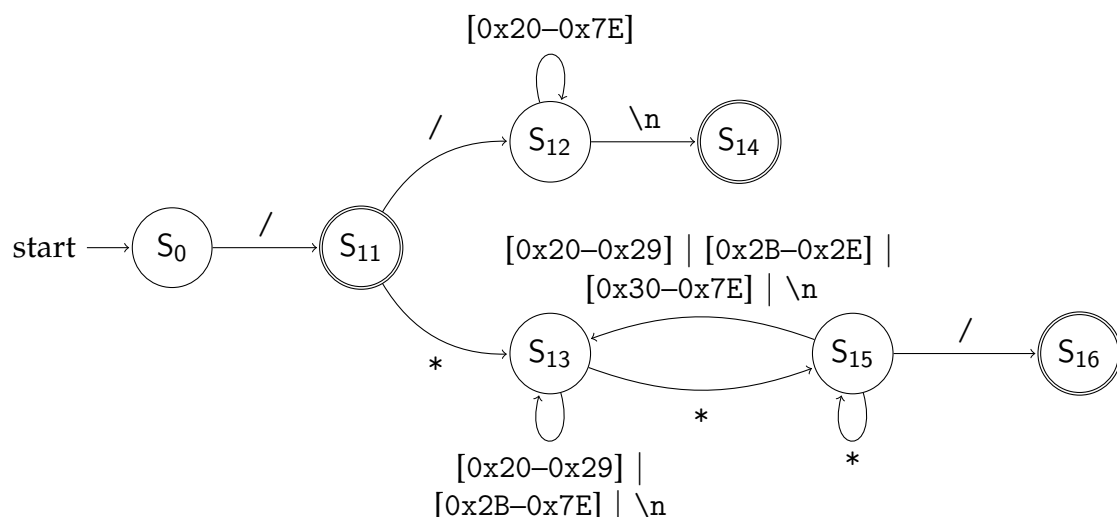


Figure 1.5: DFA recognising single and multiline comments

Next we deal with string literals. Any sequence of characters<sup>1</sup> enclosed between two quotation marks ("...") is treated as a single string token. In order to allow the user to include quotation mark characters (") within a string, any occurrence of the sequence of characters  $\backslash$ " does not terminate the string, but will later be substituted for a " character pertaining to the string. For example, should the user wish to declare the string `"But I don't want to go among mad people," Alice remarked.`, then they would type in PyScript,

```
"\"But I don't want to go among mad people,\" Alice remarked."
```

<sup>1</sup>In the <printable> range of ASCII characters 0x20–0x7E.

A DFA recognising strings for our language is given in [figure 1.6](#).

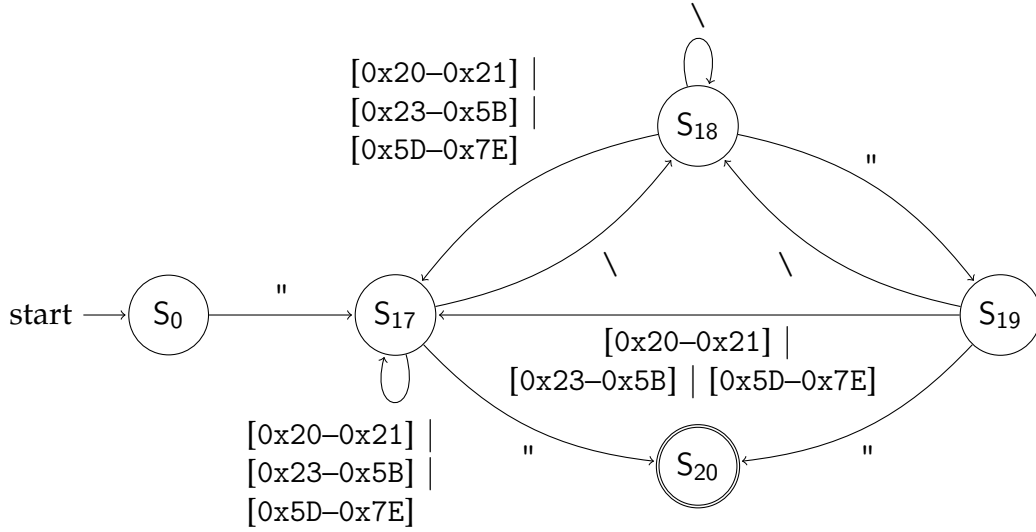


Figure 1.6: DFA recognising string literals

Combining all the DFAs at a common start node  $S_0$ , together with two other nodes whose transitions recognise punctuation symbols ( $\{, \}, (, ), ,, :$  and  $;$ ) and the end-of-file character (EOF), we obtain an automaton which recognises precisely all of the valid lexemes in our language. This can be seen in [figure 1.7](#).

## 1.2 Encoding the DFA as a Table

DFAs are formally encoded as a quintuple  $(S, \Sigma, \delta, s_0, F)$  where  $S$  is a finite set of states,  $\Sigma$  is a finite set of symbols (the alphabet),  $\delta :: S \times \Sigma \rightarrow S$  is a function representing the transitions,  $s_0$  is the starting state and  $F$  is the set of final/accepting states. For the DFA in [figure 1.7](#), we have the set of states  $S = \{S_0, S_1, \dots, S_{22}\}$ , the alphabet  $\Sigma = \{\text{ASCII}(n) : 0x20 \leq n \leq 0x7E\} \cup \{\backslash n, \text{EOF}\}$ , the starting state  $s_0 = S_0$ , and the final states  $F = \{S_i : i \in \{1, 3, 4, 5, 7, 8, 9, 10, 11, 14, 16, 20, 21, 22\}\}$ .

Now the transition function  $\delta(s, \sigma)$  tells us which state  $t \in S$  to go to from the state  $s \in S$  when given the character  $\sigma \in \Sigma$ , effectively encoding the transition  $s \xrightarrow{\sigma} t$ . For example, when we encounter the symbol  $! \in \Sigma$  at  $S_0$ , we go to the state  $S_6$ , meaning that  $\delta(S_0, !) = S_6$ . Similarly we have  $\delta(S_1, .) = S_3$  and  $\delta(S_{10}, k) = S_{10}$ . Note that  $\delta$  is not a total function, meaning that it is not defined for all  $(s, \sigma) \in S \times \Sigma$ . For example,  $\delta(S_4, a)$  has no value. However to simplify the implementation of our DFA, we instead consider the total analogue  $\delta' :: S \times \Sigma \rightarrow S \cup \{S_e\}$  defined by

$$\delta'(s, \sigma) = \begin{cases} \delta(s, \sigma) & \text{if } \delta(s, \sigma) \text{ is defined,} \\ S_e & \text{otherwise,} \end{cases}$$

where  $S_e$  is considered the *error state*. We can then encode  $\delta'$  as a table with  $|\Sigma| = 146$  rows and  $|S| = 23$  columns, placing  $\delta'(S_i, \text{ASCII}(j))$  in the  $i$ th row and  $j$ th column.

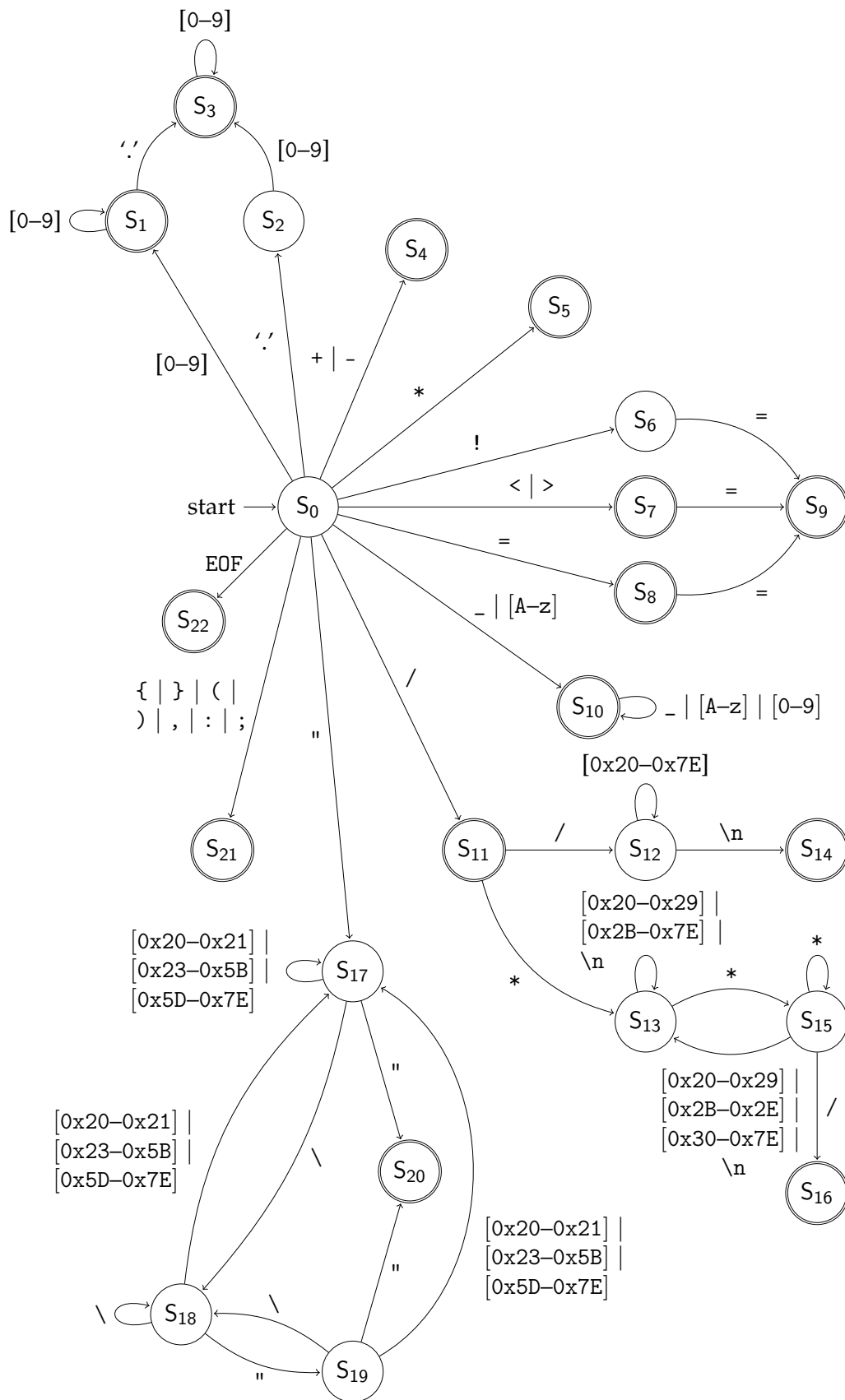


Figure 1.7: DFA recognising the entire PyScript grammar

Since many values of  $\sigma \in \Sigma$  have identical  $\delta'(s, \sigma)$  for all  $s \in S$ , many of the  $|\Sigma| = 146$  rows will be identical. For example: no matter which state  $s$  we start from,  $\delta'(s, a) = \delta'(s, b)$ . Thus to simplify our table-representation of  $\delta'$  even further, we can partition the set  $\Sigma$  into the equivalence classes of the equivalence relation  $\sim \subseteq \Sigma \times \Sigma$ , where for all  $\sigma_1, \sigma_2 \in \Sigma$ ,

$$\sigma_1 \sim \sigma_2 \iff \forall s \in S \cdot \delta'(s, \sigma_1) = \delta'(s, \sigma_2).$$

We will henceforth refer to the equivalence classes of the relation  $\sim$  as the *transition types* of our language, and will refer to them by their transition type names, given in [table 1.1](#).

Equivalence Class	ASCII Range	Transition Type Name
$\{0, 1, 2, \dots, 9\}$	0x30 – 0x39	DIGIT
$\{.\}$	0x2E	PERIOD
$\{+, -\}$	0x2B, 0x2D	ADDITIVE_OP
$\{*\}$	0x2A	ASTERISK
$\{!\}$	0x21	EXCL_MARK
$\{<, >\}$	0x3C, 0x3E	ORDER_REL
$\{=\}$	0x3D	EQUALS
$\{-\}$	0x5F	UNDERSCORE
$\{/ \}$	0x2F	FORWARDSLASH
$\{ \backslash \}$	0x5C	BACKSLASH
$\{ " \}$	0x22	QUOTATION_MARK
$\{ \{, \}, (, ), ,, :, ; \}$	0x28, 0x29, 0x2C, 0x3A, 0x3B, 0x7B, 0x7D	PUNCTUATION
$\{ \backslash n \}$	0x0A	NEWLINE
$\{ EOF \}$	-0x01	EOF
$\{ A, \dots, Z, a, \dots, z \}$	0x41 – 0x5A 0x61 – 0x7A	LETTER
$\{ \_ , \# , \$ , \% , \& , ' ,$ $\? , @ , [ , ] , \^ , \` ,   , \sim \}$	0x20, 0x23 – 0x27, 0x3F, 0x40, 0x5B, 0x5D, 0x5E, 0x60, 0x7C, 0x7E	PRINTABLE
Other symbols	Anything outside the range 0x20 – 0x7E	OTHER

Table 1.1: Different transition types for the DFA

## 1.3 Implementation in C++

### The Token Class

What we desire of our lexer implementation is to produce a set of tokens given an input program as a single `std::string`. Therefore first a Token class was created with the following attributes and functions.

<code>TOKEN</code> type	Stores the token type, the enumerated type <code>TOKEN</code> contains all the distinct token types (see <a href="#">table 1.3</a> ).
<code>std::string</code> value	Stores the lexeme.
<code>int</code> line_number	Stores the line number where the token occurs in the program source, used for error reporting.
<code>Token()</code>	Constructor (default).
<code>Token(int final_state, std::string value, int line number)</code>	Constructor, initialises the attribute value to the argument value, the attribute line_number to the argument line_number and the attribute type to <code>determine_token_type(final_state, value)</code> .
<code>TOKEN determine_token_type(int final_state, std::string &amp;value)</code>	This function determines the token type based on the final state and the value of the token.

The implementation of the constructor is straightforward. The implementation of the function `determine_token_type` uses a switch statement on the value of `final_state`. Indeed, almost all of the token types are uniquely determined by the final state which accepted the lexeme, as can be seen in [table 1.3](#).

### The Tokenisation Algorithm

The idea behind tokenisation is the following: we keep transitioning through the DFA (see [figure 1.7](#)) as far as possible until we can't go further, and then we backtrack to the last final state encountered. For example, let us describe how to tokenise the following program:

```
set str="x\" y";
```

We first encounter an 's'. This takes us from the start node  $S_0$  to  $S_{10}$ , which is a final state. Thus 's' alone is potentially a valid token in the language. Next we have an 'e'. This takes us from  $S_{10}$  to  $S_{10}$  again, a final state, so 'se' is now a potential token. Next we have 't', taking us again from  $S_{10}$  to  $S_{10}$ , so the potential token is 'set'. But now we have a whitespace character, which takes us to  $S_e$ , meaning that we cannot go further. Therefore the first token is 'set'.

Now for the second token, we start again from  $S_0$  continuing where we left off, and (ignoring the whitespace character for now) we similarly end up visiting  $S_{10}$  three times obtaining 'str' until on the fourth iteration, '=' is encountered, taking us from  $S_{10}$  to  $S_e$ . Thus the second token is 'str'. Now for the third token, we start from  $S_0$  and,

Final State	Token Type (in enum TOKEN)	Value
S <sub>1</sub>	TOK_INT	<i>&lt;int-literal&gt;</i>
S <sub>3</sub>	TOK_REAL	<i>&lt;real-literal&gt;</i>
S <sub>4</sub>	TOK_ADDITIVE_OP	+   -
S <sub>5</sub>	TOK_MULTIPLICATIVE_OP	*
S <sub>7</sub>	TOK_RELATIONAL_OP	<   >
S <sub>8</sub>	TOK_EQUALS	=
S <sub>9</sub>	TOK_RELATIONAL_OP	<=   >=   ==
S <sub>10</sub>	TOK_VAR	var
	TOK_SET	set
	TOK_DEF	def
	TOK_RETURN	return
	TOK_IF	if
	TOK_ELSE	else
	TOK_WHILE	while
	TOK_PRINT	print
	TOK_INT_TYPE	int
	TOK_REAL_TYPE	real
	TOK_BOOL_TYPE	bool
	TOK_STRING_TYPE	string
	TOK_BOOL	true   false
	TOK_MULTIPLICATIVE_OP	and
	TOK_ADDITIVE_OP	or
	TOK_NOT	not
	TOK_IDENTIFIER	<i>&lt;identifier&gt;</i>
S <sub>11</sub>	TOK_MULTIPLICATIVE_OP	/
S <sub>14</sub>	TOK_COMMENT	// { <i>&lt;printable&gt;</i> } \n
S <sub>16</sub>	TOK_COMMENT	/* { <i>&lt;printable&gt;</i>   \n } */
S <sub>20</sub>	TOK_STRING	" { <i>&lt;printable&gt;</i> } "
S <sub>21</sub>	TOK_LEFT_CURLY	{
	TOK_RIGHT_CURLY	}
	TOK_LEFT_BRACKET	(
	TOK_RIGHT_BRACKET	)
	TOK_COMMA	,
	TOK_SEMICOLON	;
	TOK_COLON	:
S <sub>22</sub>	TOK_EOF	EOF

Table 1.3: Final states and corresponding token types

consuming the '=' character, we go to  $S_8$  which is a final state. The next character '"' sends us from  $S_8$  to  $S_e$ , so we take '=' as the third token and start again.

For the fourth token we start from the '"' character which takes us to  $S_{17}$ . This state is not final. Consuming the next symbol 'x', we end up on  $S_{17}$  again. Now we encounter the '\' symbol, which takes us to  $S_{18}$ . From  $S_{18}$  we consume the '"' symbol and end up on  $S_{19}$ . From  $S_{19}$ , consuming the whitespace character takes us back to  $S_{17}$ , and then consuming the 'y' we again find ourselves on  $S_{17}$ . Now we encounter another '"' symbol which takes us to  $S_{20}$ , which is the first final state we've encountered since we've started the fourth token. The next character is ';' which sends us to  $S_e$ , so we take "x\" y" as the fourth token. Finally, the remaining tokens are ';' which we obtain from  $S_0$  to  $S_{21}$  and 'EOF' from  $S_0$  to  $S_{22}$ .

The general reasoning behind tokenisation which aided the implementation can be seen in [algorithm 1.1](#).

## The Lexer Class

In the `lexer.h` header file, an enumerated type `TRANSITION_TYPE` was declared in order to allow us to refer to transition types by integer values (`DIGIT = 0`, `PERIOD = 1`, etc). The class `Lexer` was implemented with the following attributes and functions.

<pre>const unsigned int[17][23] transitions</pre>	Stores the transition table (1.2), where the rows encode transition types (according to the enum <code>TRANSITION_TYPE</code> ), and the columns encode the different states. The entry in the $i$ th row and $j$ th column contains the value $k$ where $\delta'(S_j, i) = S_k$ .
<pre>const bool is_final</pre>	Stores in the $i$ th entry whether or not state $S_i$ is final.
<pre>const unsigned int e</pre>	Initialises to 23. Represents the error state (i.e. we have $S_e = S_{23}$ ).
<pre>std::vector&lt;Token&gt; tokens</pre>	Stores the tokens generated by the lexer.
<pre>unsigned int current_token</pre>	Initialises to zero. Stores the index of the current token in the <code>tokens</code> vector.
<pre>int transition_delta(int s, char sigma)</pre>	Implements the transition function $\delta'(s, \sigma)$ . Using a switch statement on <code>sigma</code> , the function determines the appropriate row $i$ of the transitions array and returns <code>transitions[i][s]</code> .
<pre>Token next_token( std::string &amp;program, unsigned int &amp;current_index)</pre>	Implements the reasoning described in <a href="#">algorithm 1.1</a> , returns a <code>Token</code> object.
<pre>Lexer(std::string &amp;program)</pre>	Constructor, executes <code>next_token</code> in a while-loop until <code>TOK_EOF</code> is encountered, populating the vector <code>tokens</code> at each iteration. Does not modify the class attribute <code>current_token</code> .



Token next\_token()

```
int get_line_number(  
std::string &program,  
unsigned int index)
```

This function allows the parser to access the tokens. While the attribute `current_token` is smaller than the size of the tokens vector, this function returns `tokens[current_token]` and increments `current_token`. If the current token is found to be a comment (`TOK_COMMENT`), then the `current_token` attribute is incremented and the next non-comment token is returned instead. Subsequently returns a Token object with type attribute `TOK_ERROR`.

This function traverses the program string from the index 0 to index and keeps track of the number of `'\n'` characters encountered. Used by the Token constructor in the function `next_token(std::string&, unsigned int)` when assigning the attribute `line_number`.

## 1.4 Testing

Having implemented the two classes described in [section 1.3](#), we can test the functionality of the lexer with the following main function and a few example programs.

```
1  int main() {  
2      std::ifstream file;  
3      file.open("program.prog");  
4  
5      std::string line, program;           // Convert file to string  
6      while(std::getline(file, line))  
7          program.append(line + "\n");  
8  
9      lexer::Lexer lexer(program);         // Lexer initialisation  
10     lexer::Token t;  
11  
12     while (t.type != lexer::TOK_EOF) {    // Print all tokens  
13         t = lexer.next_token();  
14         std::cout << t.type + ": " + t.value << std::endl;  
15     }  
16  
17     file.close();  
18     return 0;  
19 }
```

### Test 1

Consider the following PyScript program.

---

**Algorithm 1.1:** next\_token(program, current\_index)

---

```
1 Let current_state =  $S_0$ 
2 Let state_stack be an empty stack of states
3 Let current_symbol be a character variable
4 Let lexeme be an empty string
5 Push  $S_{\text{bad}}$  on state_stack
  // This ignores any whitespace characters or newlines before lexeme
6 while current_index < length(program) & program[current_index]  $\neq$  ' ' or '\n'
  do
7   | Increment current_index
  // If reached EOF
8 if current_index = length(program) then
9   | return TOK_EOF
  // While we can keep going (i.e. we have not hit  $S_e$ )
10 while current_state  $\neq S_e$  do
11   | current_symbol = program[current_index]
12   | Append current_symbol to lexeme
13   | if current_state is final state then
14     | Empty state_stack
15     | // Note the stack keeps track of previous states
16     | Push current_state on state_stack
17     | current_state =  $\delta'$ (current_state, current_symbol)
18     | Increment current_index
  // Rollback loop
19 while current_state is not final and is not  $S_{\text{bad}}$  do
20   | current_state = state_stack.pop()
21   | Drop last character from lexeme
22   | Decrement current_index
23 if current_state is final then
24   | return Token(current_state, lexeme, current line number)
25 else
26   | Error: invalid token
```

---

```
1  var x : int = 1 + 2;
```

Running the main function listed previously produces the following output.

TOK_VAR: var	TOK_EQUALS: =	TOK_SEMICOLON: ;
TOK_IDENTIFIER: x	TOK_INT: 1	TOK_EOF: ✦
TOK_COLON: :	TOK_ADDITIVE_OP: +	
TOK_INT_TYPE: int	TOK_INT: 2	

## Test 2

Now for a more complicated example, consider the following function definition for the Fibonacci number fib(n).

```
1  /*
2   * Fibonacci – this comment will be ignored by the lexer
3   */
4  def fib(n : int) : int {
5      if(n == 0){return 0;}
6      else{
7          if(n == 1){return 1;}
8          else{return fib(n-1) + fib(n-2);}
9      }
10 }
```

TOK_DEF: def	TOK_SEMICOLON: ;	TOK_LEFT_BRACKET: (
TOK_IDENTIFIER: fib	TOK_RIGHT_CURLY: }	TOK_IDENTIFIER: n
TOK_LEFT_BRACKET: (	TOK_ELSE: else	TOK_ADDITIVE_OP: -
TOK_IDENTIFIER: n	TOK_LEFT_CURLY: {	TOK_INT: 1
TOK_COLON: :	TOK_IF: if	TOK_RIGHT_BRACKET: )
TOK_INT_TYPE: int	TOK_LEFT_BRACKET: (	TOK_ADDITIVE_OP: +
TOK_RIGHT_BRACKET: )	TOK_IDENTIFIER: n	TOK_IDENTIFIER: fib
TOK_COLON: :	TOK_RELATIONAL_OP: ==	TOK_LEFT_BRACKET: (
TOK_INT_TYPE: int	TOK_INT: 1	TOK_IDENTIFIER: n
TOK_LEFT_CURLY: {	TOK_RIGHT_BRACKET: )	TOK_ADDITIVE_OP: -
TOK_IF: if	TOK_LEFT_CURLY: {	TOK_INT: 2
TOK_LEFT_BRACKET: (	TOK_RETURN: return	TOK_RIGHT_BRACKET: )
TOK_IDENTIFIER: n	TOK_INT: 1	TOK_SEMICOLON: ;
TOK_RELATIONAL_OP: ==	TOK_SEMICOLON: ;	TOK_RIGHT_CURLY: }
TOK_INT: 0	TOK_RIGHT_CURLY: }	TOK_RIGHT_CURLY: }
TOK_RIGHT_BRACKET: )	TOK_ELSE: else	TOK_RIGHT_CURLY: }
TOK_LEFT_CURLY: {	TOK_LEFT_CURLY: {	TOK_EOF: ✦
TOK_RETURN: return	TOK_RETURN: return	
TOK_INT: 0	TOK_IDENTIFIER: fib	

## TASK II

# The Parser

The second stage of compilation is the the procedure of syntax analysis; that is, the stage at which we ensure that the sequence of lexemes provided by the lexer form a syntactically correct program conformal to the PyScript grammar as it is described in the extended BNF (see [the introduction](#)).

In concrete terms, this is achieved by the construction of a syntax tree of the program, which when built successfully, ensures that the production rules of the formal language have been observed. Should the procedure falter along the way, then this indicates that the program does not adhere to the grammar and is therefore syntactically incorrect.

The process of building a syntax tree is similar to the construction of a sentential derivation tree in linguistics, in that, the valid construction of such a structure solely implies syntactic and not semantic correctness.

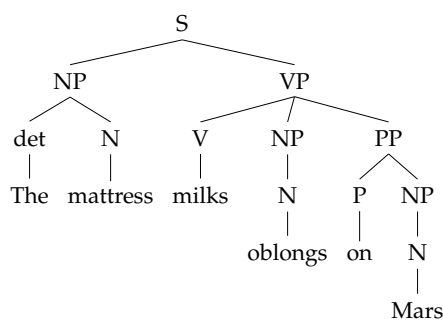


Figure 2.1: A derivation tree for the syntactically correct sentence “The mattress milks oblongs on Mars”

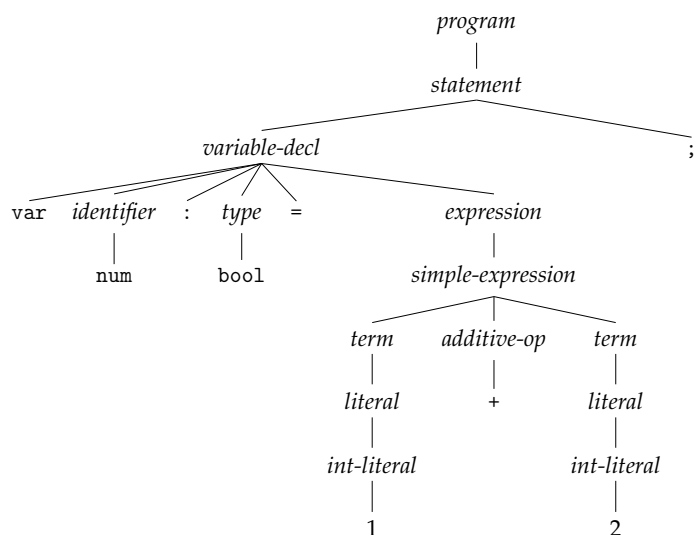


Figure 2.2: A syntax tree for the syntactically correct program “var num : bool = 1 + 2;”

## 2.1 Design

The parser built for PyScript is a top-down recursive-descent parser. The parser is *top-down* in the sense that the syntax tree is to be constructed from the root node to the leaves, and it is a *recursive-descent* parser in the sense that each procedure implemented corresponds to one of the production rules of the grammar.

The parser interacts with the lexer by calling the `next_token()` function until a token of type `TOK_EOF` is encountered. PyScript has an  $LL(1)$  grammar, and therefore the implemented parser uses one lookahead token.

Perhaps the simplest way to illustrate how the parser is designed is with an example. Consider the syntactically correct example given in [figure 2.2](#):

```
var num : bool = 1 + 2;
```

The aim is to recursively construct a hierarchy of node objects (which we will be calling `ASTNodes` in our implementation) resembling what is shown in [figure 2.2](#). The parser always starts by calling the function `parse_program()`, which will return an `ASTProgramNode`. Since by definition, a  $\langle program \rangle$  consists of zero or more  $\langle statement \rangle$  nodes, if a statement is present, the function will call `parse_statement()`, and do so for all the statements in the program (until EOF), storing the resulting nodes in a vector of `ASTStatementNodes`. In the case of the example, the program consists of one statement, so the vector in the returned `ASTProgramNode` will consist of one `ASTStatementNode`.

Now `parse_statement()` will have to determine what kind of statement we have by the first token. In the example, the first token is the keyword 'var', which identifies the statement as a variable declaration statement. Therefore a call to the function `parse_declaration_statement()` is made, returning an `ASTDeclarationNode`. This function will begin to consume tokens and ensure that the statement is syntactically correct, throwing an error at any stage where an unexpected token is encountered. So in the example, after the 'var' keyword, an identifier token must follow. This will be stored as the variable name. After the identifier token, a colon must follow. Next a type, one of 'int', 'real', 'bool', 'string', which will be stored as the variable type. Next an equals token must follow, then an expression, and finally a semicolon to end the statement.

As one can see, parsing is simply a matter of delegation of token consumption to the appropriate functions, and when a function consumes tokens, it simply needs to check that they are as expected. In the example, we glossed over the expression `1+2`. This will be parsed by a separate function called `parse_expression()`, which will return an `ASTExpressionNode`, and will be stored as the variable value. The logic carried out by this function is similar to the example, in that it delegates token consumption. With reference to the [EBNF](#), `parse_expression()` calls `parse_simple_expression()`. This then calls `parse_term()`, which then calls `parse_factor()`, and so on, until a tree is built for the expression in a similar way that it was built for the example statement. Indeed, the resulting tree in terms of `ASTNodes` is shown in [figure 2.3](#).

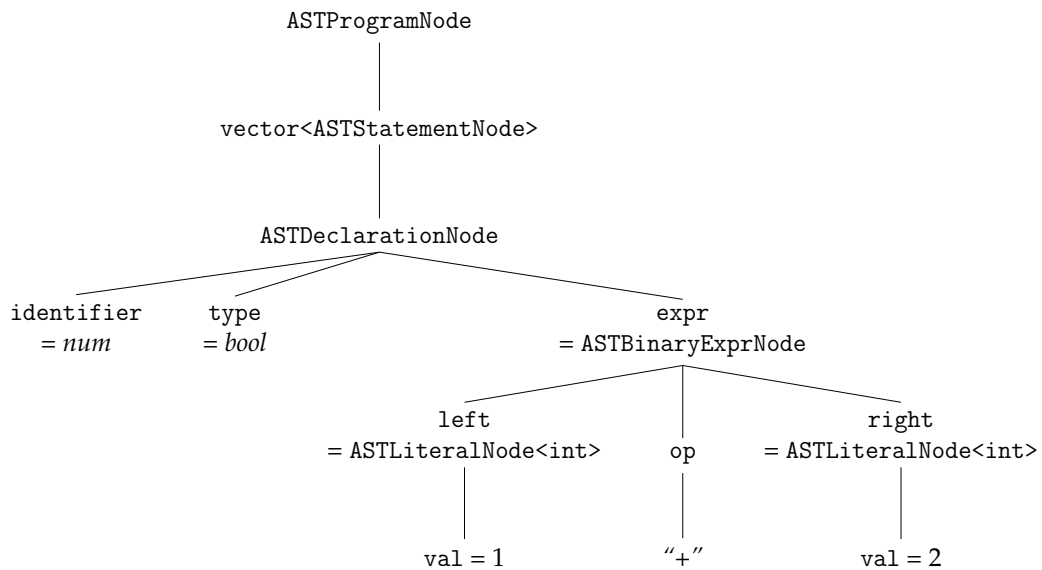


Figure 2.3: Resulting parse tree for the program 'var num : bool = 1 + 2;'

## 2.2 Implementation in C++

Every ASTNode was represented as a class. Since each node is a statement or expression of some kind, the class hierarchy outlined in [figure 2.4](#) was implemented (where classes in grey are abstract). To avoid repeating code for ASTLiteralNodes, a template class was used.

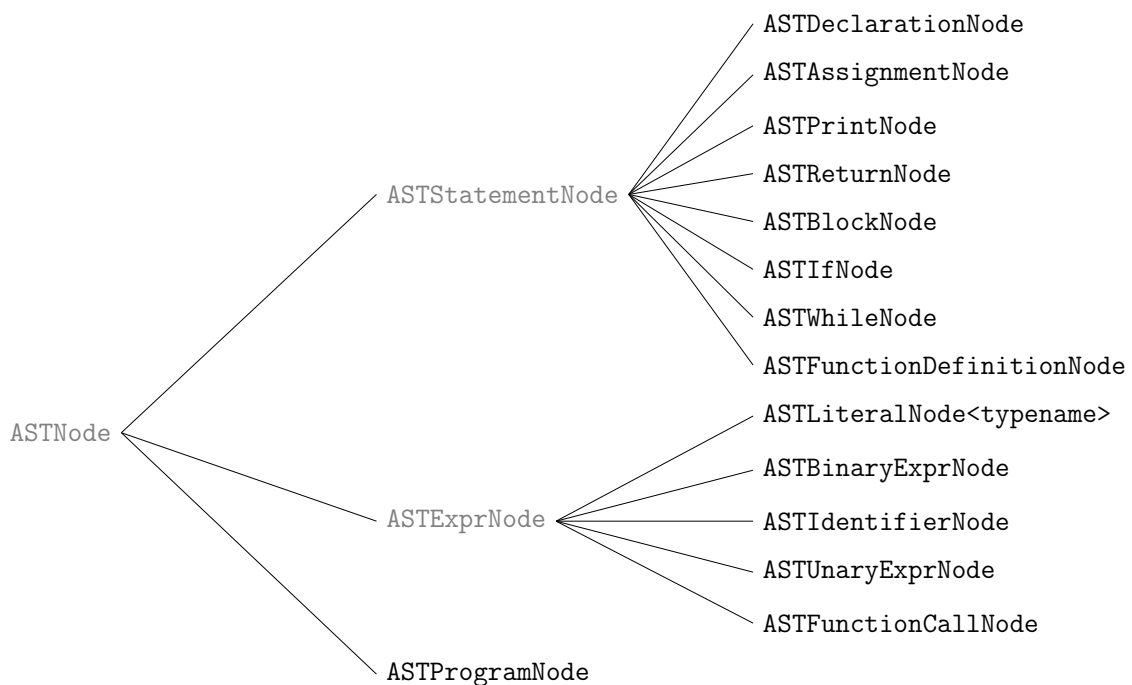


Figure 2.4: Abstract syntax tree class hierarchy

In addition to the fields described in the design stage, each class has a `line_number`

attribute which is obtained from an appropriate token. This makes error reporting more informative for the user.

All the parsing functions described in [section 2.1](#) were implemented in a class called Parser. The constructor for this class takes a pointer to a Lexer object as an argument, and consumes tokens from that Lexer object by calling `next_token()` until the EOF token is encountered. The function `parse_program()` in the Parser class returns an `ASTProgramNode` pointer to the root of the generated syntax tree once parsing is completed.

## 2.3 Testing

The best way to test the parser at this stage is to input incorrect programs and make sure that it ‘breaks’ at the right point. We will make use of the following main function for our tests (adapted from the listing given in [section 1.4](#))

```
1  int main() {
2      std::ifstream file;
3      file.open("program.prog");
4
5      std::string line, program;           // Convert file to string
6      while(std::getline(file, line))
7          program.append(line + "\n");
8
9      lexer::Lexer lexer(program);         // Lexer initialisation
10
11     parser::Parser parser(&lexer);        // Parser initialisation
12     parser.parse_program();               // Program parsing
13
14     file.close();
15     return 0;
16 }
```

### Test 1

Consider the following erroneous program.

```
1  var x real = 2;
```

When executing, this produces the following output.

Expected ‘:’ after x on line 1.

### Test 2

Now for a slightly more complicated example, consider the following simple function definition with a missing semicolon.

```
1  def f(x : int) : int {  
2      return x + 3  
3  }
```

This produced the following output.

Expected ';' after return statement on line 3.

Note that the error is reported to reside on line 3. Although slightly confusing, this is technically true since that is where the offending token was found, namely a premature '}'. To clarify, the error *can* be fixed by adding a semicolon on line 3:

```
1  def f(x : int) : int {  
2      return x + 3  
3      ;}
```

This program is now syntactically correct.

### Test 3

One final example which shows off some more complicated PyScript constructs.

```
1  def f(x : int) : int {  
2      var y : int = 0;  
3  
4      def g(x : int) : bool {  
5          if((x < 1) or (x >= 10)){  
6              set y = 1;  
7              return true;  
8          }  
9          return false;  
10     }  
11  
12     if(g(x){ return x + y; }  
13     else {  
14         set y = 4;  
15         return x + y;  
16     }  
17 }
```

This produced the following error:

Expected ')' after if-condition on line 12.



## TASK III

# Representing the Syntax Tree in XML

The goal of this task is partly to test the parser in a more intelligent way. By employing the Visitor design pattern on all of the ASTNode subclasses, the generated syntax tree can be traversed in a preorder fashion to generate an XML representation.

For example, the program

```
var str : string = "Hello" + "World";
```

whose syntax tree would be similar to the example in [figure 2.3](#), should result in the following XML code.

```
<program>
  <decl>
    <id type = "string">str</id>
    <bin op = "+">
      <string>"Hello"</string>
      <string>"World"</string>
    </bin>
  </decl>
</program>
```

### 3.1 Design

The design pattern requires the creation of a 'visitor' class in which several visit() functions are implemented, each having a different possible ASTNode subclass argument. An accept() method is added to each ASTNode subclass in order to facilitate the continued recursion, traversing the entire tree.

## 3.2 Implementation in C++

An example of a `visit()` function is the following `visit(ASTAssignmentNode*)` for variable assignments such as `'set x = 1 + 2;'`.

```

1  void XMLVisitor::visit(parser::ASTAssignmentNode *assign) {
2
3      // Add initial <assign> tag
4      xmlfile << indentation() << "<assign>" << std::endl;
5
6      // Indent
7      indentation_level++;
8
9      // Add identifier
10     xmlfile << indentation() << "<id>" << assign->identifier << "
11     </id>" << std::endl;
12
13     // Expression tags
14     assign->expr->accept(this);
15
16     // Unindent
17     indentation_level--;
18
19     // Add closing tag
20     xmlfile << indentation() << "</assign>" << std::endl;
21 }

```

To indent the file correctly, a global integer attribute `indentation_level` was kept in the `XMLVisitor` class. The function `indentation()` returns a correct amount of spaces specified by this attribute.

## 3.3 Testing the Parser using XML

Now we have a nice way to test the parser. We can construct a main function similar to that given in [section 2.3](#) to produce the XML code. Let us consider amended versions of the examples given in [section 2.3](#).

### Test 1

The first example in [section 2.3](#) was the following.

```

1  var x : real = 2;

```

This produces the following XML code.

```

<program>
  <decl>
    <id type="real">x</id>
    <int>2</int>

```

```
</decl>
</program>
```

## Test 2

The next example was a simple function definition:

```
1  def f(x : int) : int {
2      return x + 3;
3  }
```

This produced the following XML:

```
<program>
  <func-def type="int">
    <id>f</id>
    <param type="int">x</param>
    <block>
      <return>
        <bin op="+">
          <id>x</id>
          <int>3</int>
        </bin>
      </return>
    </block>
  </func-def>
</program>
```

## Test 3

And now for the more complicated example:

```
1  def f(x : int) : int {
2      var y : int = 0;
3
4      def g(x : int) : bool {
5          if((x < 1) or (x >= 10)){
6              set y = 1;
7              return true;
8          }
9          return false;
10     }
11
12     if(g(x)){
13         return x + y;
14     }
15     else {
16         set y = 4;
```

```
17     return x + y;  
18 }  
19 }
```

which produced the following XML code:

```
<program>  
  <func-def type = "int">  
    <id>f</id>  
    <param type = "int">x</param>  
    <block>  
      <decl>  
        <id type = "int">y</id>  
        <int>0</int>  
      </decl>  
      <func-def type = "bool">  
        <id>g</id>  
        <param type = "int">x</param>  
        <block>  
          <if>  
            <condition>  
              <bin op = "or">  
                <bin op = "&lt;=">  
                  <id>x</id>  
                  <int>1</int>  
                </bin>  
                <bin op = "&gt;=">  
                  <id>x</id>  
                  <int>10</int>  
                </bin>  
              </bin>  
            </condition>  
            <if-block>  
              <block>  
                <assign>  
                  <id>y</id>  
                  <int>1</int>  
                </assign>  
                <return>  
                  <bool>>true</bool>  
                </return>  
              </block>  
            </if-block>  
          </if>  
          <return>  
            <bool>>false</bool>  
          </return>  
        </block>  
      </func-def>  
    </block>  
  </func-def>  
</program>
```

```
    </block>
  </func-def>
  <if>
    <condition>
      <func-call>
        <id>g</id>
        <arg>
          <id>x</id>
        </arg>
      </func-call>
    </condition>
    <if-block>
      <block>
        <return>
          <bin op = "+">
            <id>x</id>
            <id>y</id>
          </bin>
        </return>
      </block>
    </if-block>
    <else-block>
      <block>
        <assign>
          <id>y</id>
          <int>4</int>
        </assign>
        <return>
          <bin op = "+">
            <id>x</id>
            <id>y</id>
          </bin>
        </return>
      </block>
    </else-block>
  </if>
</block>
</func-def>
</program>
```

These verify that the parser implementation is working as desired.

## TASK IV

# Semantic Analysis

This is where the meaning of programs begins to play a role, and syntactically correct programs such as

```
var x : bool = 1 + 2;
```

need to be deemed semantically incorrect. There are also some design decisions which need to be made.

### 4.1 Design

Another visitor class was used to perform semantic analysis. The following design decisions were taken during the implementation of the semantic analyser.

- **Variable shadowing**

Variables can be redeclared in different scopes. For example, the following program will output 1, 2, then 1.

```
var x : int = 1;
{ print x;
  var x : int = 2;
  print x; }
print x;
```

This does not extend to functions definitions however, which are hereditary and cannot be overwritten. For example, this is not allowed:

```
def f(x : int) : int {return x + 1;}
{ def f(x : int) : int {return x + 2;} }
```

This on the other hand is allowed, and will output 4, then 5.

```
{ def f(x : int) : int {return x + 1;}  
  print f(3); }  
{ def f(x : int) : int {return x + 2;}  
  print f(3); }
```

- **Functions can be declared within functions**

This is quite a straightforward functionality to implement, since function declaration statements are statements themselves, therefore should be allowed inside the function definition block.

- **Implicit Typecasting**

Arithmetic combinations of different types are allowed. For example,  $1.2 + 2$  will be computed as  $3.2$  and is not required to be written as  $'1.2 + 2.0'$  or  $'1.2 + 2.'$ . Similarly, assignments/declarations such as

```
set x : real = 3;
```

are also allowed. One has to be careful though, since expressions such as  $0.1 * 2 / 3$  will evaluate to  $0.1 * 0 = 0$ . The order of operations in PyScript is discussed at a later point.

- **Functions are identified by their name and signature**

We can have two functions with the same identifier, so long as their signature is different. For example, the following is allowed, and will output 4, then 6.

```
def f(x : int) : real {  
    return 2. * x;  
}  
  
def f(x : real) : real {  
    return 3 * x;  
}  
  
print f(2);  
print f(2.);
```

Note that to support this, implicit typecasting is *not* carried out for parameters of functions. The same is true for return statements: the appearance of  $'2.'$  on line 2 instead of simply  $'2'$  is to ensure that the returned value is a `real` type.

- **Order of Operations**

The usual order of operations (brackets, division, multiplication, addition, subtraction) is implemented in PyScript, however products after a division sign are all considered to be part of the denominator. For example,  $30 / 2 * 3 + 1 = 30 / 6 + 1 = 5 + 1 = 6$ . Division *still* takes precedence over multiplication however. For example,  $2 * 3 / 4 = 2 * 0 = 0$ .

The implementation makes use of a class `Scope` which keeps track of all live variables and functions in a scope using a symbol table, as well as providing functions such as `already_declared()`, `declare()`, and so on, to make interactions with the symbol tables easy. All the scopes are stored in a vector, so that variables/functions referenced outside the current scope can be searched for in outer scopes by moving backwards through the vector.

The recursive function `returns()` was implemented to determine whether a block surely returns a value or not (a requirement of the language for function definitions). The logic behind the function is given in [algorithm 4.1](#).

---

**Algorithm 4.1:** `returns(ASTStatementNode stmt)`

---

```
1 if stmt is an ASTReturnNode then
2   | return true
3 if stmt is an ASTBlockNode then
4   | foreach block_stmt in the block do
5   |   | if returns(block_stmt) then
6   |   |   | return true
7 if stmt is an ASTIfNode then
8   | if stmt has an else-block then
9   |   | return returns(if-block) and returns(else-block)
10 if stmt is an ASTWhileNode then
11   | return returns(while-block)
12 else
13   | return false
```

---

## 4.2 Implementation in C++

An example of a `visit()` function is the following `visit(ASTAssignmentNode*)` for variable assignments such as `'set x = 1 + 2;'`.

The global variable `current_expression_type` stores the type of the current expression being parsed. This is made global so that it can be used by different `visit()` functions when parsing. Similar attributes include `current_function_parameters` and the stack functions, which keeps track of the type of the current function (to check whether a return statement is valid).

There are some other minor details about the implementation which have been omitted, since they are similar to those already mentioned here, and purely arise due to C++ technicalities.



```

1 void SemanticAnalyser::visit(parser::ASTAssignmentNode *assign) {
2
3     // Determine the inner-most scope in which the value is declared
4     unsigned long i;
5     for (i = scopes.size() - 1;
6         !scopes[i] -> already_declared(assign->identifier);
7         i--)
8         if(i <= 0)
9             throw std::runtime_error(...);
10
11
12     // Get the type of the originally declared variable
13     parser::TYPE type = scopes[i]->type(assign->identifier);
14
15     // Visit the expression to update current type
16     assign->expr->accept(this);
17
18     // allow mismatched type in the case of int to real
19     if (type == REAL && current_expression_type == INT) {}
20
21     // otherwise throw error
22     else if (current_expression_type != type)
23         throw std::runtime_error(...);
24 }

```

## 4.3 Testing

### Test 1

First, let us start with the example we were using in [section 2.1](#), and make sure it fails:

```

1 var num : bool = 1 + 2;

```

Indeed, this produces the output:

```
Found int on line 1 in definition of 'num', expected bool.
```

### Test 2

Next let's try the following:

```

1 while(1+1){ print "Hello World!"; }

```

This produces:

```
Invalid while-condition on line 1, expected boolean expression.
```

### Test 3

Now we test for invalid return types.

```
1  def f(x : int) : bool {  
2    return 1 - 3.5;  
3  }
```

This produces:

Invalid return type on line 2. Expected bool, found real.

#### Test 4

Now let's test the `return()` function that we described in [algorithm 4.1](#).

```
1  def f(x : int) : bool {  
2    if(x > 2){  
3      return true;  
4    } else {  
5      if (x < 0){  
6        return false;  
7      }  
8      else{  
9        print "I don't return here";  
10     }  
11  }  
12 }
```

This produces the error:

Function f defined on line 1 is not guaranteed to return a value.

## TASK V

# The Interpreter

This is the stage at which the statements in the program are executed. The implementation here is almost identical in structure to that of the semantic analyser, except for a few minor tweaks to the structures in use.

### 5.1 Design and C++ Implementation

Again we make use of the visitor pattern. Most `accept()` functions effectively translate the PyScript syntax to C++, for example, the following is the `accept()` function for if-else blocks.

```
1 void visitor::Interpreter::visit(parser::ASTIfNode *ifNode) {
2     // Evaluate if condition
3     ifNode -> condition -> accept(this);
4
5     // Execute appropriate blocks
6     if(current_expression_value.b)
7         ifNode -> if_block -> accept(this);
8     else{
9         if(ifNode -> else_block)
10            ifNode -> else_block -> accept(this);
11    }
12 }
```

Notice the presence of the variable `current_expression_value`. Similarly to the global attributes which were required in the implementation of the parser, we have a similar concept being applied here. In this case, `current_expression_value` is a C++ struct with four attributes: an integer (i), a float (r), a boolean (b) and a string (s). These are updated whenever an expression is parsed and is referred to by various `accept()` functions.

The `visit(ASTFunctionCallNode*)` function was a bit more nuanced. First of all, it was necessary to implement a separate Scope structure which stores functions' blocks and variable values, and other relevant information to the interpreter. When a function

call was encountered, the parameters were stored globally in a vector, and each statement in the block was run with the parameters added as variables to the function's scope.

Very little error handling was done in this stage, since most of the syntax/semantics was checked beforehand, so only runtime errors could occur. Division by zero was handled.

## 5.2 Testing

Now we can finally run some programs.

### Test 1

It would be a crime not to test the *Hello world* program.

```
1  print "Hello World";
```

This produces:

Hello World

as desired.

### Test 2

Next, a function definition followed by some assignments/declarations:

```
1  def f(x : int) : int {
2      if(x > 0){
3          return x + 1;
4      } else {
5          return x - 1;
6      }
7  }
8
9  var x : int = f(2);      // 3
10 var y : real = f(-1);    // -2
11 set y = x + y;          // 1
12 print "Result = ";
13 print y;
```

This produces:

Result = 1

as expected.

### Test 3

Now some expression evaluations:

```
1  /*
2   * Testing evaluation of expressions.
3   */
4  print 1+1.0; // 2
5  print 1+2*3; // 7
6  print 20 / 2 * 5; // 2
7  print (20 / 2) * 5; // 50
8  print 1.2 * 3/4; // 0
9  print 1.2 * 5/3; // 1.2
10 print 2*3+4 < 2+3*4; // 10 < 14, true
11 print (1 < 2) and (4 < 3); // false
12 print "String" + " " + "concatenation" + ".\n";
13 print true and false or (1+1==2); // true
```

This produces:

```
2725001.2truefalseString contatenation.
true
```

as expected.

#### Test 4

Now we testing variable hiding. Consider the following example:

```
1  var s : string = "out";
2  {
3      var s : string = "in";
4      print s;
5  }
6  print s;
```

This produces:

```
inout
```

as expected.

#### Test 5

Let us implement some recursive functions.

```
1  def fac(n : int) : int {
2      if(n==0){ return 1; } else { return n * fac(n-1); }
3  }
4  def choose(n : int, k : int) : int {
5      return fac(n) / fac(k) * fac(n-k);
6  }
7
8  print fac(5);
9  print choose(7, 3);
```

This produces:

12035

(i.e.  $120 = 5!$  and  $35 = \binom{7}{3}$ ) as expected.

### Test 6

A slightly more complex example, a program which approximates the cosine function.

```
1  // Cosine function
2  def cos(x : real) : real {
3      // Positive integer power
4      def pow(x : real, n : int) : real {
5          var y : real = 1;
6          while(n > 0){
7              set y = y*x;
8              set n = n-1;
9          }
10         return y;
11     }
12
13     // Factorial
14     def fac(n : int) : int {
15         if (n == 0) { return 1; } else { return n * fac(n-1); }
16     }
17
18     // Take 8 terms of Maclaurin series for cos(x)
19     var k : int = 0;
20     var cos_x : real = 0;
21     while(k < 8){
22         set cos_x = cos_x + pow(-1., k) * pow(x, 2*k) / fac(2*k);
23         set k = k + 1;
24     }
25     return cos_x;
26 }
27
28 var pi : real = 3.1415926535897932;
29 print cos(pi);
```

This produces:

-1.00703

which is approximately  $-1 = \cos \pi$ , as expected.

### Test 7

Another example, prints the expansion of  $(1 + x)^n$  by making use of `choose()` defined in [test 5](#).

```
1  def binom(n : int) : string {
2      print "(1 + x)^";
3      print n;
```

```
4     print " = 1 + ";
5     print n;
6     print "x + ";
7
8     var out : string = "";
9     var k : int = 2;
10    while(k < n){
11        print choose(n, k);
12        print "x^";
13        print k;
14        set k = k + 1;
15        print " + ";
16    }
17    print "x^";
18    print n;
19
20    return "";
21 }
22
23 print binom(7);
```

This produces:

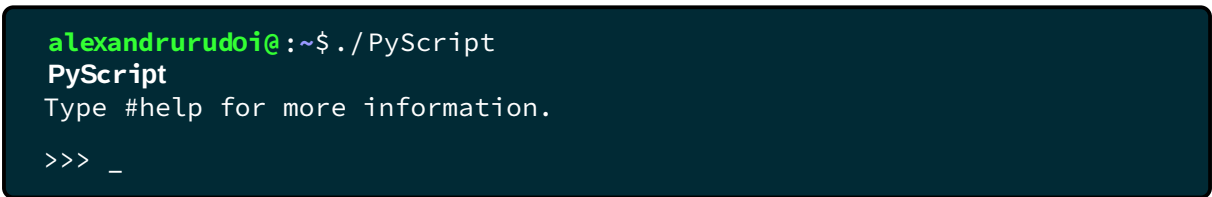
$$(1 + x)^7 = 1 + 7x + 21x^2 + 35x^3 + 35x^4 + 21x^5 + 7x^6 + x^7$$

which is the desired output.

## TASK VI

# The PyScript Read-Eval-Print-Loop

So far we have built separate classes which take care of separate parts of the compilation process, and have only made them interact by the use of bespoke main functions when it came to testing the individual components. Here we will discuss what the main function in the program will actually do. The goal is to produce an interactive REPL not unlike a LISP machine interactive terminal:



```
alexandrurudo@:~$ ./PyScript
PyScript
Type #help for more information.

>>> _
```

Figure 6.1: REPL user interface

## 6.1 Design

There is not much designing to do here, apart from merging the separate components we have built into one. The logic to be carried out is the following.

A global instance of `Scope` and `InterpreterScope` are maintained for semantic analysis and interpreting of global variables and functions. In an infinite loop, we read the user's input. If the input is `#quit`, the REPL is exited. If the input is `#help`, a nice help screen is shown to the user explaining how to use the REPL (see [figure 6.2](#)). If the input is `#load` followed by a directory, the program attempts to execute the program from the specified directory, adding all definitions in the file to the global scopes. If the input is `#st`, the program shows the user a table of variables and functions declared in the global scopes. If the input is `#clear`, the terminal window is cleared. Otherwise, the user input is executed as a program. If execution fails, the error message is caught and the program is interpreted as an expression. If this also fails, then the original error message is printed on screen.



**PyScript**

Type #help for more information.

```
>>> #help
```

Welcome to PyScript 1.0.0!

To use this interactive REPL, just type in regular PyScript commands and hit enter. You can also make use of the following commands:

```
#load file-path    Loads variable and function declarations from a  
                    specified file into memory, e.g.  
                    >>> #load ~/hello_world.prog  
  
#quit              Exits the MiniLang REPL.  
  
#st                Displays the symbol table, a list of currently  
                    declared functions and variables in the global scope.  
  
#clear             Clears the terminal window.  
  
>>> _
```

Figure 6.2: REPL help screen

## Bibliography

1. <https://dev.to/codingwithadam/introduction-to-lexers-parsers-and-interpreters-with-chevrotain-5c7b>
2. <https://martinfowler.com/dsl.html>
3. <https://www.wikiwand.com/en/Domain-specific-language>
4. <https://en.cppreference.com/w/cpp/error/runtime-error>
5. <https://en.cppreference.com/w/cpp/container/map>

Link for the code: <https://github.com/AlexandruRudoi/DSL-Interpreter>