Compilers

Interpreter for Mini-PL programming language

Petri Kallio, 012843064

April 1, 2018

---

1 How to build and use the interpreter

The project is made into a MonoDevelop solution. It contains three MonoDevelop projects:

1. `MiniPLInterpreter` contains the source code of all the different components in the compiler and interpreter and is meant to be compiled into a `Compiler.dll` dynamic library file that the other two projects use.

2. `Interpreter` contains the source code of the main program and is meant to be compiled into an `Interpreter.exe` executable file. It needs to be in the same folder with the `Compiler.dll` to be executed.

3. `MiniPLInterpreterTests` contains the unit tests for the compiler and they also need `Compiler.dll` to be run. The tests use Nunit framework.

To run the interpreter (in Windows), make sure the library and the executable are in the same directory and then execute the program by typing

```
Interpreter.exe C:\path\to\the\file.ext
```

where `C:\path\to\the\file.ext` is the path to the source file you want to compile and interpret.

To run the interpreter in Linux, make sure you have Mono installed and type

```
mono Interpreter.exe path/to/the/file.ext
```

where `path/to/the/file.ext` is again the path to the source file you want to compile and interpret.

You can also run it from MonoDevelop IDE.

2 Mini-PL interpreter architecture

In its highest level the architecture is divided in four components: main program, compiler frontend, interpreter and I/O.

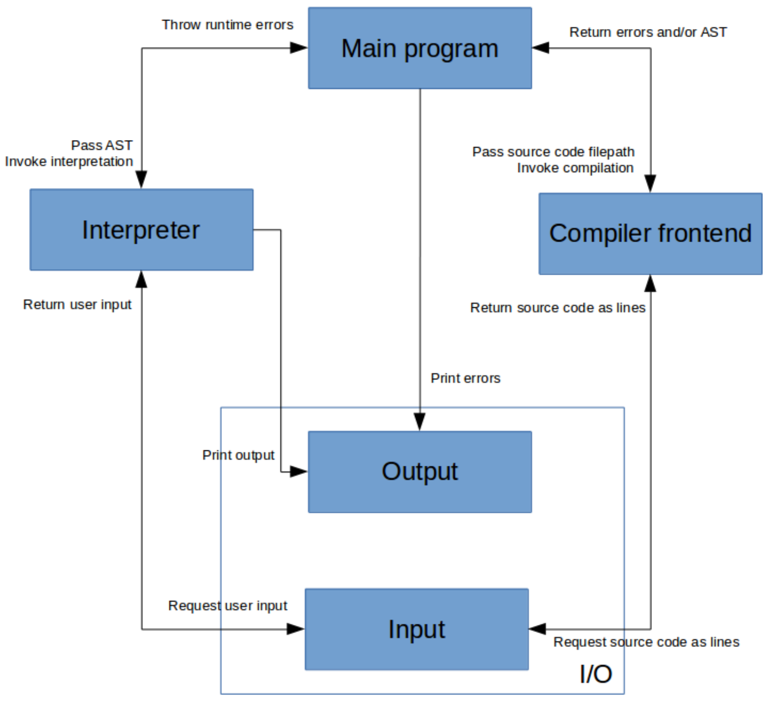

Figure 1: The overall architecture of the interpreter.

The purpose of the main program is of course to control all the needed resources, to inform the user about errors during compilation and interpretation, as well as to make the decision of whether to interpret the program contained in the source code given as argument or not.

The purpose of the I/O is to hide the source file read operations from the compiler frontend and to provide input and output functionality for the interpreter and the main program.

The purpose of the frontend is to analyze the source code and then provide any errors found during the analysis. The compilation, if executed without errors, provides the main program with an abstract syntax tree (AST) of the program.

The purpose of the interpreter is to interpret the program provided as AST by the main program. In case of any runtime error occurs, it is passed to the main program and the interpretation is halted.

Compiler frontend

The compiler frontend's architecture is divided into three components in its highest level: scanner, parser and semantic analyzer. These three provide three different interleaved phases of the frontend compilation.

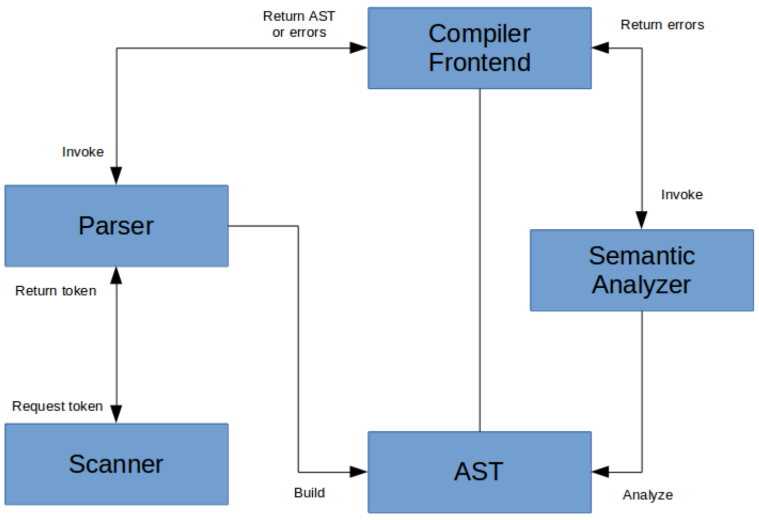

Figure 2: The overall architecture of the compiler frontend.

The compilation is done in two passes. In the first pass, the scanner and the parser cooperate to perform lexical and syntactical analysis of the source code. The pass is made over the original source code. As a product they supply the main program with either a list of lexical and syntactical errors found during the analysis, or a complete AST of the source code. The parser also builds the symbol table that is passed later to the semantic analyzer along with AST. The second pass and the third phase is the static semantical analysis of the AST. This phase is performed only if the first pass was error-free.

The parser has the control during the first pass. It requests one token at a time from the scanner and performs its analysis and builds the AST on its way. The parser is is implemented using recursive-descent parsing.

During semantic analysis the AST is inspected to enforce static semantic rules.

Interpreter

If the compilation finishes without errors, the AST is passed to the interpreter and the program is interpreted. If any dynamic semantical error or other run-time error occurs occurs during the execution of the interpreter, it is passed as an exception to the main program, which in turn displays the error message and terminates.

3 Lexical analysis (scanning phase)

The scanner is implemented as a deterministic finite automaton (DFA) which uses the combined regular expressions of all the valid tokens to make decisions while scanning. In this section the scanner's functionality is explained.

Valid Mini-PL token patterns

The scanner identifies tokens from the character stream by identifying valid token patterns.

The valid token patterns are given as regular expressions. In addition to the common symbols used in regular expressions, here are explanations of the other symbols used in the regular definitions:

`[0-9aqz]`	matches any character in the given range '0' to '9' and characters 'a', 'q' and 'z'
`[^0-9aqz]`	matches any character EXCEPT the range '0' to '9' or characters 'a', 'q' and 'z', including a linebreak
`\n`	a linebreak
`\(`	the character '(', not the regular expression grouping parenthesis
$\texttt{a}^?$	the pattern is optional, meaning it is repeated 0 or 1 times
$\texttt{a}^+$	the pattern is repeated 1 or more times

All the regular expression quantifiers are superscripted, so $\texttt{*}^*$ would mean 0 or more `*`-symbols and $\texttt{?}^?$ would mean 0 or 1 `?`-symbols. The regular expressions for different tokens:

```
    special symbol ::= < | = | & | ! | ; | \( | \) | * | / | + | - | .. | :=?
           integer ::= (+|-)?0|[1-9][0-9]*
        identifier ::= [a-zA-Z]([a-zA-Z0-9]|_)*
            string ::= "[^\n]*"
  one-line comment ::= //[^\n]*\n
multi-line comment ::= /*([^*]|*[^/])**/
```

By combining the overlap of integers and `+` and `-` operators and the overlap in comments and `/` operator, we get the following regular expressions:

```
integer or addition operator ::= +(0|[1-9][0-9]*)?|-(0|[1-9][0-9]*)?|0|[1-9][0-9]*
         comment or division ::= /(/[^\n]*\n|*([^*]|*[^/])**/)?
```

Altogether we get the following combined regular expression for valid token patterns:

```
mini-pl token ::= +(0|[1-9][0-9]*)? | -(0|[1-9][0-9]*)? | 0|[1-9][0-9]*
                | /(/[^\n]*\n|*([^*]|*[^/])**/)? | [a-zA-Z]([a-zA-Z0-9]|_)*
                | "[^\n]*" | < | = | & | ! | ; | \( | \) | * | .. | :=?
```

Error handling

The error handling in the scanner is implemented in the following manner:

1. If the token is a variable identifier whose value is a keyword, it reports an error.

2. If the token is a string literal and it spans multiple lines, it reports an error.

3. If the token is a string literal and the end of file is reached while scanning, it reports an error.

4. If the token is invalid, it reports an error.

5. In other cases it marks the token as an error token and passes it to the parser to handle.

Other remarks

Some remarks on the functionality of the scanner:

1. Because the scanner is implemented so that the maximal munch rule is obeyed, identifiers and integers must be delimited by non-identifier and non-numeric characters respectively.

2. Keywords are identified as respective tokens by screening identifier tokens; the scanner first scans an identifier and then compares the identifier to a dictionary of keyword to determine whether it is a keyword or a variable identifier.

3. Comments and whitespaces are screened as well; whenever a comment or a whitespace is read from the character stream, it is passed by and not returned as a token.

4. There are two special cases when the scanner needs to know whether it needs to provide the parser with a certain type of token, if possible:

    - When a keyword token must be followed by a variable identifier token and another keyword is read instead, the scanner must understand to report an error of the use of a keyword as an identifier. Yet, to enable the parser to make its own analysis to the statement, it is still passed to the parser as a variable identifier token.
    - When a integer begins with a sign, it mustn't be confused with an addition or a subtraction operator.

    Because of these ambiguities, the parser can pass the token it received last as an argument to the scanner when requiring for a new token. Thus, for example, when the scanner reads a minus-sign, it can check the previous token to make the right decision. If the previous token is a binary operator, it tries to parse an integer and if it's an integer, it parses a binary operation.

5. The tokens supplied by the scanner contain information about

   - the token's type,
   - its corresponding value (if it is interesting, meaning namely an identifier name, string literal, integer literal or boolean literal) and
   - its position in the source code (row and column).

   The tokens and their types are maintained during the whole execution of the program, attached to their corresponding nodes in the AST. This helps creating valuable information to the user when printing errors. Other information is represented using the `TokenType` enumerator as well, such as the evaluation value of an expression.

6. In addition to the original definition of the Mini-PL language, two more keywords have been added to enable the assignment of boolean values through the source code:

   `true` the boolean variable evaluates to true

   `false` the boolean variable evaluates to false

   These keywords, as the others are screened after the scanning of a identifier and their type is set to `BooleanValue`.

4 Syntax analysis (parsing phase)

The parser is implemented as a top-down recursive-descent parser. In this section the parser's functionality is explained.

LL(1)-grammar

Because the parser is implemented as a top-down parser, it needs a LL-grammar to make decisions while parsing. Below is an LL(1)-grammar that defines the Mini-PL programming language accompanied by the expectation sets in curly braces that are needed to choose the right rule while parsing.

⟨program⟩ → ⟨stmts⟩ $$ {`var, for, read, print, assert, ID, $$`}
⟨stmts⟩ → ⟨stmt⟩ **;** ⟨stmts⟩ {`var, for, read, print, assert, ID`}
→ ϵ {`end, $$`}
⟨stmt⟩ → **var** ⟨var-id⟩ **:** ⟨type⟩ ⟨assign⟩ {`var`}
→ ⟨var-id⟩ **:=** ⟨expr⟩ {`ID`}
→ **for** ⟨var-id⟩ **in** ⟨expr⟩ **..** ⟨expr⟩ **do** ⟨stmts⟩ **end for** {`for`}
→ **read** ⟨var-id⟩ {`read`}
→ **print** ⟨expr⟩ {`print`}
→ **assert** (⟨expr⟩) {`assert`}
⟨assign⟩ → := ⟨expr⟩ {`:=`}
→ ϵ {`;`}
⟨expr⟩ → ⟨opnd⟩ ⟨binary-op⟩ {`INTEGER, STRING, BOOLEAN, (, ID`}
→ ⟨unary-op⟩ ⟨opnd⟩ {`!`}
⟨binary-op⟩ → ⟨op⟩ ⟨opnd⟩ {`+, -, *, /, <, =, &`}

$\rightarrow \epsilon$ {`)`, `;`, `..`, `do`}
⟨opnd⟩ $\rightarrow$ *INTEGER* {`INTEGER`}
$\rightarrow$ *STRING* {`STRING`}
$\rightarrow$ *BOOLEAN* {`BOOLEAN`}
$\rightarrow$ ⟨var-id⟩ {`ID`}
$\rightarrow$ (⟨expr⟩) {`(`}
⟨type⟩ $\rightarrow$ **int** {`int`}
$\rightarrow$ **string** {`string`}
$\rightarrow$ **bool** {`bool`}
⟨var-id⟩ $\rightarrow$ *ID* {`ID`}
⟨op⟩ $\rightarrow$ + {`+`}
$\rightarrow$ − {`-`}
$\rightarrow$ * {`*`}
$\rightarrow$ / {`/`}
$\rightarrow$ < {`<`}
$\rightarrow$ = {`=`}
$\rightarrow$ & {`&`}
⟨unary-op⟩ $\rightarrow$ **!** {`!`}

The symbols inside angles are non-terminal symbols and the symbols written in bold (reserved symbols) or capitalized (string, integer and variable identifier literals) are terminal symbols. The ϵ-character depicts an empty token.

Building the AST and the symbol table

As mentioned before, the parser is also responsible for building the AST and the symbol table while parsing. Below is the same LL(1)-grammar as shown before, but this time embedded with action routines to display the building of the AST's nodes and populating the symbol table, as well as the flow of the evaluations from leaves up to branching nodes when executing the program in the AST.

⟨program⟩ $\rightarrow$ ⟨stmts⟩ { `<program>.ptr := <stmts>.ptr` }
⟨stmts⟩$_1$ $\rightarrow$ ⟨stmt⟩ **;** { `<stmts>`$_1$`.ptr := create_statements(<stmt>.ptr)` }
⟨stmts⟩$_2$ { `set_statements_sequitor(<stmts>`$_1$`.ptr, <stmts>`$_2$`.ptr)` }
⟨stmts⟩ $\rightarrow \epsilon$ { `<stmts>.ptr := null` }
⟨stmt⟩ $\rightarrow$ **var** ⟨var-id⟩ { `<stmt>.ptr := <var-id>.ptr` }
: ⟨type⟩ { `set_type(<stmt>.ptr, <type>.ptr)` }
⟨assign⟩ { `set_value(<stmt>.ptr, <assign>.ptr)` }
⟨stmt⟩ $\rightarrow$ ⟨var-id⟩ { `<stmt>.ptr := <var-id>.ptr` }
:= ⟨expr⟩ { `set_value(<stmt>.ptr, <expr>.ptr)` }
⟨stmt⟩ $\rightarrow$ **for** ⟨var-id⟩ { `<stmt>.ptr := make_loop(<var-id>.ptr)` }
in ⟨expr⟩$_1$ { `set_value(<var-id>.ptr, <expr>`$_1$`.ptr)` }
.. ⟨expr⟩$_2$ { `set_upto(<stmt>.ptr, <expr>`$_2$`.ptr)` }

do ⟨stmts⟩ { `set_statements(<stmt>.ptr, <stmts>.ptr)` } **end for**
⟨stmt⟩ → **read** ⟨var-id⟩ { `<stmt>.ptr := read_input(<var-id>.ptr)` }
⟨stmt⟩ → **print** ⟨expr⟩ { `<stmt>.ptr := print_expr(<expr>.ptr)` }
⟨stmt⟩ → **assert** (⟨expr⟩) { `<stmt>.ptr := assert(<expr>.ptr)` }
⟨assign⟩ → := ⟨expr⟩ { `<assign>.ptr := <expr>.ptr` }
⟨assign⟩ → ϵ { `<assign>.ptr := make_default_value()` }
⟨expr⟩ → ⟨opnd⟩ { `<binary-op>.st := <opnd>.ptr` }
⟨binary-op⟩ { `<expr>.ptr := <binary-op>.ptr` }
⟨expr⟩ → **!** ⟨opnd⟩ { `<expr>.ptr := make_un_op("!", <opnd>.ptr)` }
⟨binary-op⟩ → + ⟨opnd⟩ { `<binary-op>.ptr := make_bin_op("+", <binary-op>.st, <opnd>.ptr)` }
⟨binary-op⟩ → − ⟨opnd⟩ { `<binary-op>.ptr := make_bin_op("-", <binary-op>.st, <opnd>.ptr)` }
⟨binary-op⟩ → * ⟨opnd⟩ { `<binary-op>.ptr := make_bin_op("*", <binary-op>.st, <opnd>.ptr)` }
⟨binary-op⟩ → / ⟨opnd⟩ { `<binary-op>.ptr := make_bin_op("/", <binary-op>.st, <opnd>.ptr)` }
⟨binary-op⟩ → < ⟨opnd⟩ { `<binary-op>.ptr := make_bin_op("<", <binary-op>.st, <opnd>.ptr)` }
⟨binary-op⟩ → = ⟨opnd⟩ { `<binary-op>.ptr := make_bin_op("=", <binary-op>.st, <opnd>.ptr)` }
⟨binary-op⟩ → & ⟨opnd⟩ { `<binary-op>.ptr := make_bin_op("&", <binary-op>.st, <opnd>.ptr)` }
⟨binary-op⟩ → ϵ { `<binary-op>.ptr := <binary-op>.st` }
⟨opnd⟩ → *INTEGER* { `<opnd>.ptr := make_int_leaf(INTEGER.ptr)` }
⟨opnd⟩ → *STRING* { `<opnd>.ptr := make_str_leaf(STRING.ptr)` }
⟨opnd⟩ → *BOOLEAN* { `<opnd>.ptr := make_str_leaf(BOOLEAN.ptr)` }
⟨opnd⟩ → ⟨var-id⟩ { `<opnd>.ptr := <var-id>.ptr` }
⟨opnd⟩ → (⟨expr⟩) { `<opnd>.ptr := <expr>.ptr` }
⟨type⟩ → **int** { `<type>.ptr := make_type("integer")` }
⟨type⟩ → **string** { `<type>.ptr := make_type("string")` }
⟨type⟩ → **bool** { `<type>.ptr := make_type("boolean")` }
⟨var-id⟩ → *ID* { `<var-id>.ptr := make_ID_leaf(ID.ptr)` }

For each node, the `node.ptr` means the flow of a value from child nodes to parent nodes and `node.st` the flow of values from parent nodes to child nodes.

Default values

During the declaration of a variable, if no assignment is made, the following default values are assigned to the variables:

`var x : int;` → the value of x is set to `0`.

`var y : bool;` → the value of y is set to `false`.

`var z : string;` → the value of z is set to `""` (empty string).

Error handling

The error handling in the parser is implemented so that the parsing method throws an exception when faced with an invalid token. The catching method then disables the the building of the AST and fastforwards to a safe point from which to continue parsing. This means that even one error while parsing will prevent the safe interpretation of the program.

5 Semantic analysis

The semantic analysis phase is implemented by using the Visitor design pattern. All the nodes of the AST are required to implement the `Accept(Visitor visitor)` method defined in the `ISyntaxTreeNode` interface.

The semantic analyzer creates an instance of `StatementCheckVisitor`. Then the AST's root node's `Accept` method is called with the `StatementCheckVisitor` as it's argument. The visitor then checks the whole AST in a depth-first fashion when each node calls the `Accept` methods of their children.

The other visitor required in this phase is the `TypeCheckingVisitor`, which checks the nodes' expressions' evaluations' types. The `StatementCheckVisitor` calls a node's `Accept` method with a `TypeCheckingVisitor` as its argument whenever needed.

Error handling

Whenever a static semantic error is encountered, it is reported.

6 Abstract syntax tree (AST)

The AST is build using the Composite design pattern. An object of class `SyntaxTree` contains a pointer to its root node. All the nodes implement the interface `ISyntaxTreeNode`, either directly or indirectly through another interface.

Common properties to all the nodes are:

`Token` accessor	The token that this node represents in the source
`Accept` method	Accepts an `INodeVisitor`

The types of nodes, their properties and functionality are the following:

RootNode Represents the very root of the program.

`Sequitor`	Contains a pointer to the first `StatementsNode` of the program, if any.

StatementsNode Ties a statement and it follower together.

`statement`	Contains a pointer to the current statement.
`sequitor`	Contains a pointer to the next `StatementsNode` of the program, if any.

DeclarationNode Represents a variable declaration.

`idNode`	Contains a pointer to the `VariableIdNode` this declaration declares.
`assignNode`	Contains a pointer to the `AssignNode` that assigns a value to the variable.

AssignNode Represents an assignment.

`idNode`	Contains a pointer to the `VariableIdNode` this assignment assigns a value to.
`exprNode`	Contains a pointer to the expression node that evaluates to the value that this assignment assigns to the variable.

AssertNode Represents an assertion.

`expressionNode`	Contains a pointer to the expression node whose truth value is evaluated.
`ioPrintNode`	Contains a pointer to the `IOPrintNode` that prints the failed assertion to the user.

BinOpNode Represents an binary operation.

`leftOperand`	Contains a pointer to the expression node that evaluates to the lefthand side of this operation.
`rightOperand`	Contains a pointer to the expression node that evaluates to the righthand side of this operation.
`operation`	The `TokenType` that equals to the binary operation evaluated in this `BinOpNode`.
`evaluationType`	The `TokenType` that equals to the type that this binary operation evaluates to.

BoolValueNode Represents a boolean value.

`value`	The value of this node.

IntValueNode Represents an integer value.

`value`	The value of this node.

StringValueNode Represents a string value.

`value`	The value of this node.

ForLoopNode Represents a for-loop.

`idNode` Contains a pointer to this for-loop's control variable.

`indexAccumulator` Contains a pointer to the `AssignNode` that is executed every time the loop starts.

`max` Contains a pointer to the expression node that evaluates to the maximum value of the control variable.

`statements` Contains a pointer to the `StatementsNode` that are executed during every loop.

`rangeFrom` Contains a pointer to the `AssignNode` that assigns the initial value of the control variable.

IOPrintNode Represents a print statement.

`expression` The expression node that evaluates to the value that this node prints.

IOReadNode Represents a read statement.

`idNode` Contains a pointer to the `VariableIdNode` this read node reads it's value into.

`assignNode` Contains a pointer to the `AssignNode` that this read operation's value is read into to be assigned to the variable.

UnOpNode Represents a unary operation.

`operand` Contains a pointer to the expression node that evaluates to the operand of this operation.

`operation` The `TokenType` that equals to the unary operation evaluated in this `UnOpNode`.

`evaluationType` The `TokenType` that equals to the type that this unary operation evaluates to.

VariableIdNode Represents a variable id.

`id` The identifier that corresponds to the entry of this variable in the symbolic table.

`symbolTable` The symbol table that this variable's value is saved into.

`variableType` The `TokenType` that equals to the type that this variable evaluates to.

Here is an example of an AST as a graph:

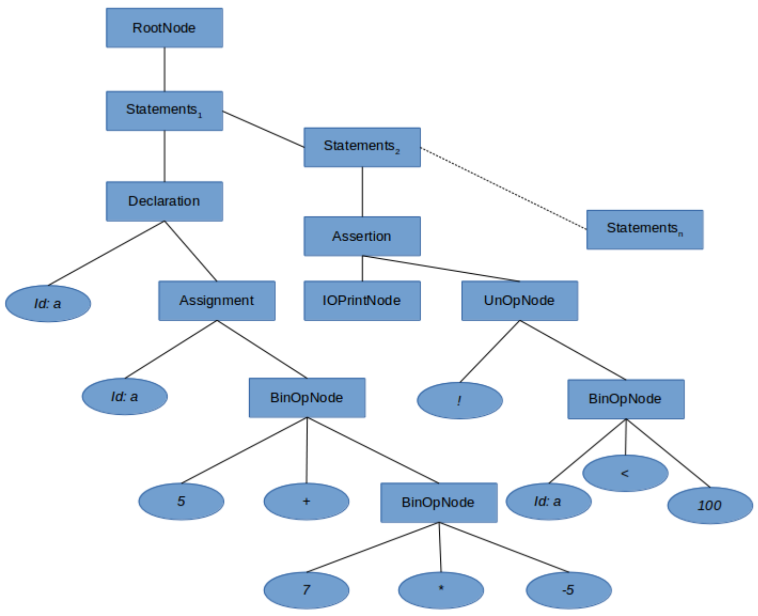

Figure 3: An example of an AST as a graph.

7 Interpretation

The interpretation, like the semantic analysis phase, is implemented using the Visitor design pattern. The AST's root node's `Accept` method is called with an `ExecutionVisitor` as its argument. The execution is then performed in a depth-first fashion to each node's children.

Whenever an expression needs to be evaluated, its `Accept` method is called with an `EvaluationVisitor` as its argument.

Error handling

The visitors used in the interpretation are designed to catch dynamic semantic errors and throw a `RuntimeException` for the main program to catch and report to user. The different types of errors are discussed in the Testing section.

8 Testing

In addition to the case specific contextual testing during the development of each feature, the testing of the Mini-PL Interpreter is implemented by extensive unit testing of different parts of the compiler frontend and interpreter. The unit tests are included within the `MiniPLInterpreter2000` solution as their own project named `MiniPLInterpreterTests`.

All the different test files contain tests both for valid and invalid sources.

Scanner testing

The scanner reports three different types of errors:

1. String literal errors (a string literal spans multiple lines or doesn't end)
2. Token is invalid (the token contains illegal characters)
3. A reserved keyword is used as an identifier

The scanner tests are in the `ScannerTest.cs` file.

Parser testing

The parser inspects the syntactical integrity of the program. This is why the parser has been tested so that each type of statement has been tested as a valid input and by dropping different tokens from the valid input and testing that an error is reported only when the input is malformed.

The other type of error the parser reports is an `IntegerOverflowError`, which happens if a signed integer literal doesn't fit in a 32-bit memory location. This is tested as well by testing the negative and positive boundaries.

The parser tests are in the `ParserTest.cs` file.

Semantic analyzer testing

The semantic analyzer checks that the semantic constrains that can be checked statically are enforced. The semantic analyzer testing therefore includes running test on programs that should pass as well as covering each different possibility of invalid code that should report an error during the semantic analysis. These include type checking in assignments, ensuring that a variable is declared before it is accessed, ensuring that a variable is declared only once, ensuring that a loop's control variable is not assigned a new value inside the loop, and so on.

The semantic analyzer tests are in the `SemanticAnalyzerTest.cs` file.

Interpreter

The interpreter is meant to keep track of the program's variables' state and to ensure that the dynamic semantic checks are made. Also, the interpreter is supposed to do input and output, so this must be tested as well. The tests test that

1. a declared variable has its default value, if not assigned another during the declaration
2. an assignment changes the variable's value in the symbol table
3. after a read operation, if a non-integer is being assigned to an integer variable, an exception is thrown
4. a print statement prints output
5. an assert statement, if the assertion fails, prints output

The semantic analyzer tests are in the `InterpreterTest.cs` file.

9 Shortcomings

1. The error message pointer doesn't point to the right column if tabs are used.

2. Division by zero is not checked until in the runtime. This is because of the way the semantic analyzer was implemented: the types of expressions are evaluated, but their values are not.