Documentation of MiniPL Interpreter

Siiri Kuoppala

March 17, 2023

Contents

1 Introduction

This project is an interpreter for a small toy language called MiniPL. The interpreter was implemented in C++. It performs all the main phases of interpretation: scanning, parsing, semantic analysis (including type checking), and execution. The project is built with CMake and tested with GoogleTest.

Dependencies: Building the project requires CMake, which can be installed on Ubuntu with:

```
sudo apt install cmake
```

GoogleTest should be included in the folder `libraries` to enable testing.

Usage: Build the project as shown below.

```
# create build directory
mkdir build
cd build
cmake ..

# build and run source target
make MiniPL_interpreter_run
bin/MiniPL_interpreter_run <filename>

# build and run test target
make MiniPL_interpreter_tst
bin/MiniPL_interpreter_tst
```

The `samples` directory contains some example programs that can be run with the interpreter. Some of them are malformed and produce errors to demonstrate the error recovery strategy of the interpreter. Here are some examples of running the sample files from the build directory:

```
project/build$ bin/MiniPL_interpreter_run ../samples/1.mpl
16

project/build$ bin/MiniPL_interpreter_run ../samples/2.mpl
Found semantic errors. Ending process.
Semantic error: Variable ntimes used without declaration
at 10:8-10:13 in variable (10:8-10:13)
10 |if x = ntimes do
   |       ^-----

project/build$ bin/MiniPL_interpreter_run ../samples/3.mpl
Give a number 4
The result is: 24
x
```

2 Architecture

The codebase has the following main components:

- `MiniPL`: The overall interpreter. Has methods `runFile(filename)` and `run(program)` for interpreting a given program.
- `Scanner`: Scans the input program and produces tokens
- `Parser`: Parses the tokens and produces an Abstract Syntax Tree (AST).
- `SemanticAnalyzer`: Finds semantic errors, excluding type errors.
- `TypeChecker`: Finds type errors.
- `Interpreter`: Executes the program.
- `ErrorHandler`: Keeps track of errors. Prints them with `printErrors()`.

The class `MiniPL` is the heart of the interpreter. When interpreting a program, it initializes and runs all the other components in order. It can be constructed with references to specific input and output streams as parameters, but the default values are standard input and output. The `main` function in `main.cpp` just takes a filename as a command line parameter and calls `MiniPL.runFile(filename)`.

Most of the data is stored in structs of the following types, which only depend on each other:

- `Position`: A position in the program code (row and column).
- `Span`: The start and end positions of a token, node or other object.
- `Token`: One token. Contains a `Span` and a `TokenValue`.
- AST-nodes: Many types of nodes (see Section 5).
- Error: Different error types all inherit from `ErrorBase`.

The main components (other than `MiniPL`) do not directly communicate with each other, but they all depend on some of these structs. The `Scanner` does not depend on the AST-nodes, and the other components (apart from `Parser`) can not access tokens. Each component has their own error type that they can create. `ErrorHandler` can access the error base class `ErrorBase` but not the specific kinds of errors.

The code heavily utilizes C++ variants to store values with multiple possible types. For example, a `TokenValue` is a variant that may contain any type of token, and `StatementNode` and `ExprNode` are variants that may contain different types of AstNodes. The evaluated values of expressions are also stored in variants, which are of type `ExprValue`. Using variants is useful because they make it possible to store and handle different types of objects in the same way, and still recover the underlying object whenever needed.

3 Scanning

The `Scanner` is an ad-hoc scanner with one-character look-ahead. It has one public method: `getToken()`, which scans the next token in the program. It iterates through the program using a small helper-class `ProgramIterator`, which has methods `currentChar()`, `peekChar()`, and `move()` for iterating. The iterator additionally keeps track of the current program `Position`, which the scanner uses to add a `Span` to the tokens.

The scanner always returns a viable token. After reaching the end of the program, it just keeps returning an `Eof` token with the same span. When encountering an error, it skips the wrong character and returns the current token or scans the next token, depending on where the error happened. For information about error messages, see Section 6.

3.1 Token patterns

There are four five of tokens: identifiers, Keywords, literals (integers or strings), operators, and delimiters. Below is a regular definition of the possible tokens.

```
VarIdent → [A-Za-z][A-Za-z0-9_]*

Literal → [0-9]+|"([^\n"]|\\(n|t|\n|.))*"

Operator → +|-|*|/|<|=|\&|!

Delimiter → :=|;|:|..|(|)|$

TokenValue → VarIdent|Literal|Operator|Delimiter
```

The `Keyword` tokens form an exception, as they are not scanned based on a regular definition. Whenever an identifier is scanned, the scanner checks whether it is actually a keyword, and returns a `Keyword` if necessary. The possible keywords are "var", "for", "end", "in", "do", "read", "print", "int", "string", "bool", "assert", "if", and "else".

The names above correspond to the type names in the code. In the code, identifiers are stored as strings, and literals in a struct of type `Literal`, containing a value which is an integer/string variant, while values of type `Operator`, `Delimiter`, and `Keyword` are stored as enum classes. The values are stored in the `std::variant TokenValue`, which may hold any of the above types.

In addition to the tokens, two types of comments are allowed in the language. Single-line comments have the format `//.*`, and end at the end of the line. Multi-line comments have the format `/*...*/`, and can be nested. Nested comments do not allow a regular definition, so they form yet another exception. The scanner skips all comments and whitespace between tokens.

4 Parsing

The parser is a recursive-descent parser with one-token look-ahead. It iterates through the tokens with a small helper class `TokenIterator`, which provides the methods `currentToken()` and `nextToken()` (which moves to the next token and returns that).

4.1 LL(1) grammar

The parser implements the LL(1) grammar shown below. The terminals are <literal>, <op>, <ident>, <unary_op>, the end-of-file symbol $$, punctuation and the bolded Keywords.

<prog> → <stmts> $$
<stmts> → <stmt>; <stmts> | ε
<stmt> → <decl> | <assign> | <for> | <read> | <print> | <if>
<decl> → **var** <ident> : <type> <delc_assign>
<decl_assign> → := <expr> | ε
<assign> → <ident> := <expr>
<for> → **for** <ident> **in** <expr>..<expr> **do** <stmts> **end for**
<read> → **read** <ident>
<print> → **print** <expr>
<if> → **if** <expr> **do** <stmts> <else> **end if**
<else> → **else** <stmts> | ε
<expr> → <expr(0)>
<expr(i)> → <expr(i+1)> <expr_tail(i)> | <unary_op(i)> <epxr(i)>
<expr_tail(i)> → <op(i)> <expr($i+1$)> <expr_tail(i)> | ε
<expr(max $\_i+1$)> → <literal> | <ident> | **(** <expr> **)**

The grammar is modified from the specification to allow for expression with more than two operands. Operator precedence is represented with the precedence index i in variables <expr(i)> , <expr_tail(i)>, <unary_op(i)> and <op(i)>. The index goes from 0 (lowest) to max_i (highest), and the 'operands' of expression with level i are expressions of level $i + 1$. There are 6 different precedence classes (copied from C++):

5: logical not (!)

4: div and mul (/, $*$)

3: add, subtract (+, $-$)

2: less than (<)

1: equal (=)

0: logical and (&)

The structure of the `Parser` differs slightly from this grammar because the parsing of the variables <decl> and <decl_assign>, as well as <expr> and <expr_tail> are merged into single methods `declaration()` and `expression()`. While parsing, the parser constructs an Abstract Syntax Tree, whose structure is described in the next section.

5 Abstract Syntax Trees

All AST-nodes inherit the `AstNodeBase` class, which contains a `Span` for error messages. AST-nodes are split into statements and expressions, which are mostly handled differently. They are also stored in different variants.

The type `StatementNode` is a `std::variant` which may contain an AST-node of one of the following types:

- `DeclAstNode`: a declaration.

  `VarIdent varId`

  `Type type`

  `optional<ExprAstNode> expr`

- `AssignAstNode`: assignment

  `VarIdent varId`

  `ExprAstNode expr`

- `ForAstNode`: for-statement

  `VarIdent varId`: loop variable

  `ExprAstNode startExpr, endExpr`: start and end of the range

  `StatementsAstNode stmts`: the statements inside the loop

- `IfAstNode`: if-statement

  `ExprAstNode expr`

  `StatementsAstNode ifStatements`

  `StatementsAstNode elseStatements`

- `ReadAstNode`: read-statement

  `VarIdent varId`

- `PrintAstNode`: print-statement

  `ExprAstNode expr`

- `StatementsAstNode`: a list of statements

  `vector<StatementNode> stmts`: list of statements

The variant `ExprNode` holds nodes of the types:

- `LiteralNode`: contains a single literal.
- `VarNode`: contains a variable identifier.
- `UnaryOp`: contains a unary `Operator` and a pointer to an `ExprNode`.
- `BinaryOp`: contains a binary `Operator` and pointers to two expressions.

5.1 Traversal

The components `SemanticAnalyzer`, `TypeChecker` and `Interpreter` all traverse through the AST in the same way. They implement `visit` functions for all types of AST nodes and recursively visit the children. Instead of using the usual visitor design pattern, they use `std::visit` to visit the actual objects inside the variants `StatementNode` and `ExprNode`. This way the AST nodes do not need to know anything about the visitors, and do not need any methods to accept visitors.

The traversing components do not use inheritance because their `visit`-functions to expression nodes need different types of return values and so can not inherit from the same base class. This is not a large problem in the current use case, because each component has some special behavior for most node types. However, using inheritance would probably be better because it would make implementing changes easier and less likely to cause bugs. One would just have to circumvent the return-value problem in some way.

5.2 Semantic analysis and behavior

The specification states that MiniPL uses a single global scope for all names. This makes it unclear what should happen when a variable is declared inside an if- or for-statement, especially since variables can only be declared once. For clarity, I have decided that variables can not be declared in inner 'scopes', i.e. inside `if` or `for`.

The semantic analysis is split into two components: `SemanticAnalyzer`, and `TypeChecker`. The semantic analyzer checks that all variables are declared exactly once, before they are used, that variables are not declared inside if- or for-statements, and that the loop variable of a for-statement can not be assigned inside the loop. The `TypeChecker` checks that all expressions and variables have the correct types. If the analysis produces no errors, the code can be executed by the `Interpreter`.

The `Interpreter` visits the AST and executes the program based on the information in the nodes. The variable identifiers are stored in an `std::map`. The read statement is implemented using `std::cin`, with the fail bits on. If an incorrect value is given as input, `cin` throws an exception which is caught by the interpreter and converted into a normal MiniPL runtime error. Integers are stored as the usual 32-bit `int`, and integer overflow is undefined behavior (same as C++, no additional checks). In a division operation the interpreter checks that the divisor is not zero and causes a runtime error if necessary.

The language specification mentions that print and read statements can handle integers and strings, but I decided to allow boolean values in the print and read statements as well. I feel that this is more convenient for a programmer who wants to use e.g. debug prints in their program. This also means the there are no type checks for read and print. In the input and output, booleans are always printed as '0' if false and '1' if true. Likewise, when reading a boolean from input, the input has to be either '0' or '1'. Inputting any other value produces a runtime error.

6 Error handling

All components are given a reference to the `ErrorHandler`, which they use to raise errors when something unexpected happens. The `ErrorHandler` prints out error messages and keeps track of whether errors have been encountered. The components then do something component-specific to recover and continue processing the input program. If the `ErrorHandler` has errors after the scanning and parsing passes, the interpreter ends the process. Otherwise, it continues with semantic analysis and, if no errors are encountered, finally runs the program.

6.1 Error types

Errors are stored as structs, that all inherit from the base struct `ErrorBase`. The base struct has members `context`, `contextScope`, and `scope`, and a public method `description()`, which uses virtual functions of the derived classes to construct an error message. The derived classes and the errors they can represent are listed below:

- `ScanningError`: unexpected character, newline or Eof
- `ParsingError`: unexpected token
- `SemanticError`: one of the following
  - variable not declared
  - redeclaration
  - declaration in inner scope
  - assignment to a constant variable
- `TypeError`: wrong type
- `RuntimeError`: division by zero or IO failure (could not read input)

```
---------------------------------------------------------------
Lexical error: Unexpected newline at 13:23-13:23
Found in string literal (13:9-13:23)
13 |  print "This is y: p;
   |                       ^

Syntax error: Unexpected keyword 'print' at 14:3-14:7
Found in print-statement (13:3-14:7)
13 |  print "This is y: p;
14 |  print y;
   |  ^----
---------------------------------------------------------------
Runtime error: Division by zero at 3:16-3:18
Found in expression (3:16-3:18)
3 |var x : int := 1/0;
  |              ^--
---------------------------------------------------------------
```

6.2 Error recovery

The `Scanner` always returns a token when asked, whether or not it runs into errors while processing. When a problem is encountered and the current token can not be scanned, the scanner skips it and scans the next token (after raising an error to the handler). In some cases, e.g. when encountering a newline while scanning a string, the `Scanner` can return the token it was scanning while the error happened. When the end of the program has been reached, the Scanner returns an end-of-file token.

The `Parser` has an exception-based recovery approach. When the `match`-function can not match the current token, an error is raised and an exception thrown. The exception is caught in the `statements` function, and tokens are skipped until the current token is a semicolon or one of the Keywords **var**, **for**, **read**, **print**, or **if**. The parser uses these to find the start of a new statements and resumes parsing from that point.

The `TypeChecker` can return a `Broken` type from e.g. a malformed expression or an undeclared variable. Type checks for broken types are automatically skipped to avoid cascading errors. Thus, a long expression with a type error in inner parenthesis only produces one error. In case of multiple declarations of the same variable, the first declaration is assumed to be correct.

When `Interpreter` produces a runtime error, it raises an error to the `ErrorHandler` and throws an exception. The exception is caught by `MiniPL` which asks the `ErrorHandler` to print the error message and stops running the program.

7 Testing

The GoogleTest framework was used to write automated tests for the project. All tests and sample programs were written by hand, based on whatever things I thought were important. When encountering a bug, I would fix the bug and write a test for it.

The scanner is well tested with unit tests covering all different token types, while other components have only a few limited unit tests. The whole system is tested with integration tests that check whether running the test programs produces the expected behavior. Additionally, there is one test that runs all programs in the `samples` directory and (hopefully) checks that the interpreter doesn't crash.

There are no automatic tests for the error handling and recovery functionalities. These were tested by hand, by running the incorrect sample programs and looking at the output.