Minipascal

0.1

Viktor Horsmanheimo - 014930373 - Compilers

Generated on Mon Mar 13 2023 01:42:05 for Minipascal by Doxygen 1.9.6

Mon Mar 13 2023 01:42:05

Chapter 1

Minipascal

1.1 Building

The project requires the build system `meson` and `ninja`, compilation is trivial; if you want to specify which compiler to use set the environmental variable `CC` before running `meson`. You may install meson with your favorite package manager, it should be available on all major distributions. Non-linux platforms that are unix like may work but are not supported.

```
meson setup build && cd build && ninja
```

1.2 Running

```
./minipascal foo.mpl
```

1.3 Brief introduction to the minipascal interpreter

The project interpreter is quite small consisting of ∼2000 lines, and each phase is split into it's respective files, parsing can be found in parser.{h,cpp} while the lexer is in lexer.{h,cpp} etc.

The parser has a quite a bit of boilerplate code due to the nature of making one in C++. In hindsight choosing a functional language such as ocaml would probably have been a beter idea. The different nodes for the AST can be found in expr.h, recursive descent is used to build the tree.

1.4 Lexing

Lexing is implemented in a naive way, simply a bunch of if statements grouped together when encountering a token. Supports most control characters; octal, hexadecimal and unicode code points are not currently supported.

1.5 Parsing

Parsing consists of a top-down parser (LL(1)), it does not create a parse tree, instead creates an AST directly. You may view the different nodes relationships in the AST with UML diagrams in relevant chapters below. A brief explanation is also shown in Required info.

See also

Expr

1.6 Semantical analysis

Currently the only semantical analysis done is initialization of variables and type checking. Since minipascal is globally scoped, no scope checking is done.

1.7 Error handling

Error handling is implemented using the crash and burn approach in the parser. It will exit if it encounters a fatal error like missing tokens such as do or end. I did not have enough time to implement proper error handling. The analyser will analyse the entire file though, it will report all errors it finds.

The lexer currently is incapable of detecting non-terminated strings, it also implements the crash and burn approach, if it encounters a symbol that is not expected, it will exit.

Future work would consist of adding an error type which has ways to handle such errors. For example skipping tokens until a valid one is encountered and getting a new statement from there.

1.8 Testing

I was not proficient in automatic testing from earlier and I did not have time to learn how to set everything up so all testing has been done manually. Future work would consist of likely adding this next.

I've used a file which contained all the different constructs of the language. It can be found below:

```
var x: int := 10/5;
var y: int;

print y;
y := 123;
print y;
var mybool: bool := !0;
print "foo" + " bar";
print x;
print mybool;

if !(0) do // foo bar
    print "!(0 is true)";
    if x do
        print "truthy!\n";
    end if;
end if;

for x in x..(1 + 20 / 4) do
    print x;
end if;
```

1.9 Hours

Rough estimate of hours, I'm very bad at tracking time, I easily forget how much time I've actually spent.

Date	Hours
Sun Mar 12	6
Sat Mar 11	5
Fri Mar 10	6
Thu Mar 9	6
Wed Mar 8	6
Thu Mar 2	8
Wed Mar 1	6
Tue Feb 28	6
Mon Feb 27	6
Wed Feb 22	5
Tue Feb 21	6
Sun Feb 19	6
Thu Feb 2	6
Tue Jan 31	8
Mon Jan 30	4
Sun Jan 29	5
Wed Jan 18	5
Tue Jan 17	6
Total	106

1.10 Required info

1. A dot is considered any character in this context. An unescaped '$*$' is zero or more, an unescaped '+' is one or more, a pipe '|' is or.

```
token = [A-z]+[A-z0-9]*
digit = [0-9]+
string = ".*"
symbols = [!\+-\*<=&/] | :=
language_tokens = (token | digit | symbols | string)
```

2. `e` is considered as epsilon in this context.

```
program     -> stmt_list $$
stmt_list   -> stmt stmt_list | e
stmt        -> var id : type := expr;
            | id := expr;
            | print expr
            | read id
            | for id in expr..expr do stmt_list end for
            | if expr do stmt_list do stmt_list if_tail

if_tail     -> else stmt_list | end

type        -> int | string | bool

expr        -> term term_tail
term_tail   -> add_op term term_tail | e
term        -> factor factor_tail
factor_tail -> mult_op factor factor_tail | e
factor      -> ( expr ) | id | digit
add_op      -> + | -
mult_op     -> * | /
```

3. See children of Expr, TL;DR, tree looks like this:

```
    StatementList
Expr        StatementList
       Expr        StatementList
```

`Expr` can be any statement (a little confusing naming convention, it was named such at first and now it's a bit too much work to rename it), a binary operation, print etc. StatementList is just a tree structure to contain all of the different statements.

4. See Error handling

5. See Hours

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 5

Class Documentation

5.1 Analyser Class Reference

`#include <analysis.h>`

Public Member Functions

- bool analyse (StatementList *ast)

5.1.1 Detailed Description

Semantical analyser

5.1.2 Member Function Documentation

5.1.2.1 analyse()

```
bool Analyser::analyse (
            StatementList * ast )
```

Analyses the AST generated by the parser.

Returns

true or error, false on success.

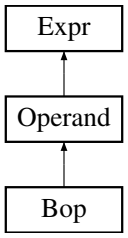
The documentation for this class was generated from the following files:

- analysis.h
- analysis.cpp

5.2 Bop Class Reference

`#include <expr.h>`

Inheritance diagram for Bop:



Public Member Functions

- Bop (std::unique_ptr< Token > tok, std::unique_ptr< Operand > left, std::unique_ptr< Operand > right)
- bool analyse () const override
- virtual Literal * get_value () override
- bool truthy () override
- virtual int get_type () override
- virtual void interpet (void) override
- virtual void visit (void) const override
- virtual Literal operator+ (Operand &) override
- virtual Literal operator- (Operand &) override
- virtual Literal operator* (Operand &) override
- virtual Literal operator/ (Operand &) override
- virtual Literal operator&& (Operand &) override
- virtual Literal operator== (Operand &) override
- virtual Literal operator< (Operand &) override
- virtual Literal operator! () override

Public Member Functions inherited from Operand

- **Operand** (std::unique_ptr< Token > t)
- **Operand** (const Token &t)
- **Operand** (const Token &&t)
- virtual Literal * get_value ()=0
- virtual bool truthy ()=0
- virtual int get_type ()=0
- virtual Literal operator+ (Operand &l)=0
- virtual Literal operator- (Operand &l)=0
- virtual Literal operator* (Operand &l)=0
- virtual Literal operator/ (Operand &l)=0
- virtual Literal operator&& (Operand &l)=0
- virtual Literal operator== (Operand &l)=0
- virtual Literal operator< (Operand &l)=0
- virtual Literal operator! ()=0

Public Member Functions inherited from Expr

- **Expr** (std::unique_ptr< Token > t)
- **Expr** (const Token &t)
- virtual void interpet (void)=0
- virtual void visit (void) const =0
- virtual bool analyse () const =0

Protected Member Functions

- bool **is_boolean** ()

Additional Inherited Members

Protected Attributes inherited from Expr

- std::unique_ptr< Token > **token**

5.2.1 Detailed Description

A binary operation, 1 + 1 for example

5.2.2 Constructor & Destructor Documentation

5.2.2.1 Bop()

```
Bop::Bop (
            std::unique_ptr< Token > tok,
            std::unique_ptr< Operand > left,
            std::unique_ptr< Operand > right )  [inline]
```

Parameters

tok	- which type of operation it is.
left	- left side of the operation.
right	- the right side of the operation.

5.2.3 Member Function Documentation

5.2.3.1 analyse()

```
bool Bop::analyse ( ) const  [inline], [override], [virtual]
```

Does analysis on the current statement.

Implements Expr.

5.2.3.2 get_type()

```
virtual int Bop::get_type ( )  [inline], [override], [virtual]
```

Gets the type of a variable/literal/operation.

Implements Operand.

5.2.3.3 get_value()

```
virtual Literal * Bop::get_value ( )  [inline], [override], [virtual]
```

Evaluates the binary expression, if called multiple times it will use a cached value.

Implements Operand.

5.2.3.4 interpet()

```
virtual void Bop::interpet (
            void  )  [inline], [override], [virtual]
```

Implements Expr.

5.2.3.5 operator"!()

```
virtual Literal Bop::operator!  ( )  [inline], [override], [virtual]
```

Does unary on a literal, extended from the miniPL spec to be defined for both integers and booleans.

Returns

Implements Operand.

5.2.3.6 operator&&()

```
virtual Literal Bop::operator&& (
            Operand & l )  [inline], [override], [virtual]
```

Does logical AND on a literal, only defined for booleans.

Returns

The result placed into a Literal

Implements Operand.

5.2.3.7 operator∗()

```
virtual Literal Bop::operator* (
            Operand & l )  [inline], [override], [virtual]
```

Does multiplication on a literal, only defined for integers.

Returns

The result placed into a Literal

Implements Operand.

5.2.3.8 operator+()

```
virtual Literal Bop::operator+ (
            Operand & l )  [inline], [override], [virtual]
```

Does addition on a literal, defined for integers and strings.

Returns

The result placed into a Literal

Implements Operand.

5.2.3.9 operator-()

```
virtual Literal Bop::operator- (
            Operand & l )  [inline], [override], [virtual]
```

Does subtraction on an Operand, only defined for integers.

Returns

The result placed into a Literal

Implements Operand.

5.2.3.10 operator/()

```
virtual Literal Bop::operator/ (
            Operand & l )  [inline], [override], [virtual]
```

Does division on a literal, only defined for integers.

Returns

The result placed into a Literal

Implements Operand.

5.2.3.11 operator<()

```
virtual Literal Bop::operator< (
            Operand & l )  [inline], [override], [virtual]
```

Does logical LESS THAN on a literal, defined for any any type.

Returns

The result placed into a Literal

Implements Operand.

5.2.3.12 operator==()

```
virtual Literal Bop::operator== (
            Operand & l )  [inline], [override], [virtual]
```

Does logical EQUALS on a literal.

Returns

The result placed into a Literal

Implements Operand.

5.2.3.13 truthy()

```
bool Bop::truthy ( )  [inline], [override], [virtual]
```

Evaluates the expression and checks if it's truthy, all non-zero values and non-empty strings are considered truthy.

Implements Operand.

5.2.3.14 visit()

```
virtual void Bop::visit (
            void  ) const  [inline], [override], [virtual]
```

Print the AST prettily

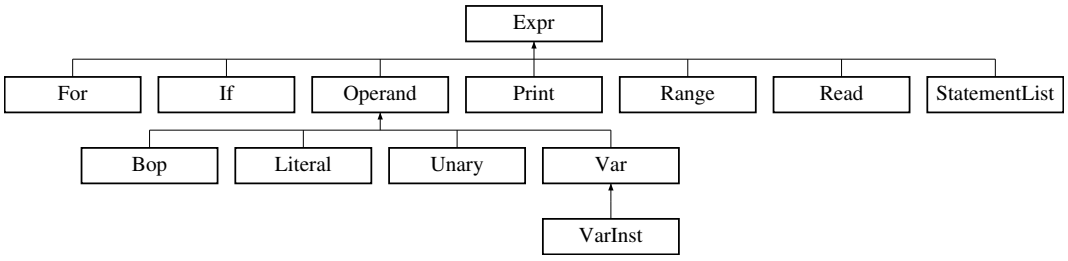
Implements Expr.

The documentation for this class was generated from the following file:

- expr.h

5.3 Expr Class Reference

Inheritance diagram for Expr:



Public Member Functions

- **Expr** (std::unique_ptr< Token > t)
- **Expr** (const Token &t)
- virtual void interpet (void)=0
- virtual void visit (void) const =0
- virtual bool analyse () const =0

Protected Attributes

- std::unique_ptr< Token > **token**

5.3.1 Member Function Documentation

5.3.1.1 analyse()

```
virtual bool Expr::analyse ( ) const  [pure virtual]
```

Does analysis on the current statement.

Implemented in StatementList, Literal, Var, VarInst, Bop, If, Range, For, Print, Read, and Unary.

5.3.1.2 interpet()

```
virtual void Expr::interpet (
            void  )  [pure virtual]
```

Implemented in Range.

5.3.1.3 visit()

```
virtual void Expr::visit (
            void  ) const  [pure virtual]
```

Print the AST prettily

Implemented in If, Range, StatementList, Literal, Var, VarInst, Bop, For, Print, Read, and Unary.

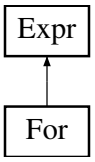
The documentation for this class was generated from the following file:

- expr.h

5.4 For Class Reference

`#include <expr.h>`

Inheritance diagram for For:



Public Member Functions

- **For** (std::unique_ptr< Token > variable, std::unique_ptr< Range > range, std::unique_ptr< StatementList > statement_list)
- bool analyse () const override
- void interpet () override
- void visit (void) const override

Public Member Functions inherited from Expr

- **Expr** (std::unique_ptr< Token > t)
- **Expr** (const Token &t)
- virtual void interpet (void)=0
- virtual void visit (void) const =0
- virtual bool analyse () const =0

Additional Inherited Members

Protected Attributes inherited from Expr

- std::unique_ptr< Token > **token**

5.4.1 Detailed Description

For loop node in the AST.

5.4.2 Member Function Documentation

5.4.2.1 analyse()

```
bool For::analyse ( ) const  [inline], [override], [virtual]
```

Does analysis on the current statement.

Implements Expr.

5.4.2.2 interpet()

```
void For::interpet (
            void  )  [inline], [override], [virtual]
```

Implements Expr.

5.4.2.3 visit()

```
void For::visit (
            void  ) const  [inline], [override], [virtual]
```

Print the AST prettily

Implements Expr.

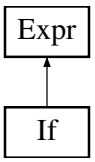
The documentation for this class was generated from the following file:

- expr.h

5.5 If Class Reference

```
#include <expr.h>
```

Inheritance diagram for If:



Public Member Functions

- If (std::unique_ptr< Operand > condition, std::unique_ptr< StatementList > truthy, std::unique_ptr< StatementList > falsy)
- bool analyse () const override
- void interpet () override
- void visit () const override

Public Member Functions inherited from Expr

- **Expr** (std::unique_ptr< Token > t)
- **Expr** (const Token &t)
- virtual void interpet (void)=0
- virtual void visit (void) const =0
- virtual bool analyse () const =0

Additional Inherited Members

Protected Attributes inherited from Expr

- std::unique_ptr< Token > **token**

5.5.1 Detailed Description

If statement node

5.5.2 Constructor & Destructor Documentation

5.5.2.1 If()

```
If::If (
            std::unique_ptr< Operand > condition,
            std::unique_ptr< StatementList > truthy,
            std::unique_ptr< StatementList > falsy )  [inline]
```

Parameters

condition	- Expression to be checked if truthy, if true it will execute the `truthy` tree, else the `falsy`.
truthy	- tree to evaluate if the condition is true.
falsy	- tree to evaluate if the condition is false, if any.

5.5.3 Member Function Documentation

5.5.3.1 analyse()

```
bool If::analyse ( ) const  [inline], [override], [virtual]
```

Does analysis on the current statement.

Implements Expr.

5.5.3.2 interpet()

```
void If::interpet (
            void  )  [inline], [override], [virtual]
```

Implements Expr.

5.5.3.3 visit()

```
void If::visit ( ) const  [inline], [override], [virtual]
```

Print the AST prettily

Implements Expr.

The documentation for this class was generated from the following file:

- expr.h

5.6 Interpreter Class Reference

Public Member Functions

- **Interpreter** (std::string_view filename)
- void **run** ()

The documentation for this class was generated from the following files:

- interpreter.h
- interpreter.cpp

5.7 Lexer Class Reference

Public Member Functions

- **Lexer** (std::string_view filename)
- std::unique_ptr< Token > get_token (bool consume=true)
- std::unique_ptr< Token > peek_token (void)
- bool is_reserved (std::string_view lexeme)

5.7.1 Member Function Documentation

5.7.1.1 get_token()

```
std::unique_ptr< Token > Lexer::get_token (
            bool consume = true )
```

Gets the next token

Parameters

consume	- whether to consume the returned token or not.

Returns

The next token from the file.

5.7.1.2 is_reserved()

```
bool Lexer::is_reserved (
            std::string_view lexeme )  [inline]
```

Checks whether a lexeme is reserved.

Parameters

lexeme	- Lexeme to validate.

5.7.1.3 peek_token()

```
std::unique_ptr< Token > Lexer::peek_token (
            void  )
```

Skips all whitespace and returns the next token, it does not consume the token.

Returns

The next token from the file.

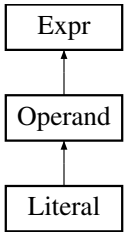
The documentation for this class was generated from the following files:

- lexer.h
- lexer.cpp

5.8 Literal Class Reference

```
#include <expr.h>
```

Inheritance diagram for Literal:



Public Member Functions

- **Literal** (std::variant< int, std::string, bool > value)
- **Literal** (const Literal &l)
- **Literal** (const Literal &&l)
- **Literal** (std::unique_ptr< Token > &tok)
- bool analyse () const override
- virtual void interpet (void) override
- virtual void visit (void) const override
- virtual Literal * get_value () override
- bool truthy () override
- virtual int get_type () override
- Literal operator+ (Operand &l) override
- Literal **operator=** (Literal &l)
- Literal **operator=** (std::variant< int, std::string, bool > value)
- Literal operator- (Operand &l) override
- Literal operator* (Operand &l) override
- Literal operator/ (Operand &l) override
- Literal operator&& (Operand &l) override
- Literal operator== (Operand &l) override
- Literal operator< (Operand &l) override
- Literal operator! () override

Public Member Functions inherited from Operand

- **Operand** (std::unique_ptr< Token > t)
- **Operand** (const Token &t)
- **Operand** (const Token &&t)
- virtual Literal * get_value ()=0
- virtual bool truthy ()=0
- virtual int get_type ()=0
- virtual Literal operator+ (Operand &l)=0
- virtual Literal operator- (Operand &l)=0
- virtual Literal operator* (Operand &l)=0
- virtual Literal operator/ (Operand &l)=0
- virtual Literal operator&& (Operand &l)=0
- virtual Literal operator== (Operand &l)=0
- virtual Literal operator< (Operand &l)=0
- virtual Literal operator! ()=0

Public Member Functions inherited from Expr

- **Expr** (std::unique_ptr< Token > t)
- **Expr** (const Token &t)
- virtual void interpet (void)=0
- virtual void visit (void) const =0
- virtual bool analyse () const =0

Public Attributes

- std::variant< int, std::string, bool > **value**

Additional Inherited Members

Protected Attributes inherited from Expr

- std::unique_ptr< Token > **token**

5.8.1 Detailed Description

A literal, a digit or string, since booleans are not part of the spec, they can not exist as a literal. Internally the symbol table contains Literals, so they're possible to store in a Literal but can't be created without an expression in the language.

5.8.2 Member Function Documentation

5.8.2.1 analyse()

```
bool Literal::analyse ( ) const  [inline], [override], [virtual]
```

Does analysis on the current statement.

Implements Expr.

5.8.2.2 get_type()

```
virtual int Literal::get_type ( )  [inline], [override], [virtual]
```

Gets the type of a variable/literal/operation.

Implements Operand.

5.8.2.3 get_value()

```
virtual Literal * Literal::get_value ( )  [inline], [override], [virtual]
```

Gets the value of a variable/literal/operation.

Implements Operand.

5.8.2.4 interpet()

```
virtual void Literal::interpet (
            void  )  [inline], [override], [virtual]
```

Implements Expr.

5.8.2.5 operator"!()

```
Literal Literal::operator!  ( )  [inline], [override], [virtual]
```

Does unary on a literal, extended from the miniPL spec to be defined for both integers and booleans.

Returns

Implements Operand.

5.8.2.6 operator&&()

```
Literal Literal::operator&& (
            Operand & l )  [inline], [override], [virtual]
```

Does logical AND on a literal, only defined for booleans.

Returns

The result placed into a Literal

Implements Operand.

5.8.2.7 operator∗()

```
Literal Literal::operator* (
            Operand & l )  [inline], [override], [virtual]
```

Does multiplication on a literal, only defined for integers.

Returns

The result placed into a Literal

Implements Operand.

5.8.2.8 operator+()

```
Literal Literal::operator+ (
            Operand & l )  [inline], [override], [virtual]
```

Does addition on a literal, defined for integers and strings.

Returns

The result placed into a Literal

Implements Operand.

5.8.2.9 operator-()

```
Literal Literal::operator- (
            Operand & l )  [inline], [override], [virtual]
```

Does subtraction on an Operand, only defined for integers.

Returns

The result placed into a Literal

Implements Operand.

5.8.2.10 operator/()

```
Literal Literal::operator/ (
            Operand & l )  [inline], [override], [virtual]
```

Does division on a literal, only defined for integers.

Returns

The result placed into a Literal

Implements Operand.

5.8.2.11 operator<()

```
Literal Literal::operator< (
            Operand & l )  [inline], [override], [virtual]
```

Does logical LESS THAN on a literal, defined for any any type.

Returns

The result placed into a Literal

Implements Operand.

5.8.2.12 operator==()

```
Literal Literal::operator== (
            Operand & l )  [inline], [override], [virtual]
```

Does logical EQUALS on a literal.

Returns

The result placed into a Literal

Implements Operand.

5.8.2.13 truthy()

```
bool Literal::truthy ( )  [inline], [override], [virtual]
```

Evaluates the expression and checks if it's truthy, all non-zero values and non-empty strings are considered truthy.

Implements Operand.

5.8.2.14 visit()

```
virtual void Literal::visit (
            void  ) const  [inline], [override], [virtual]
```

Print the AST prettily

Implements Expr.

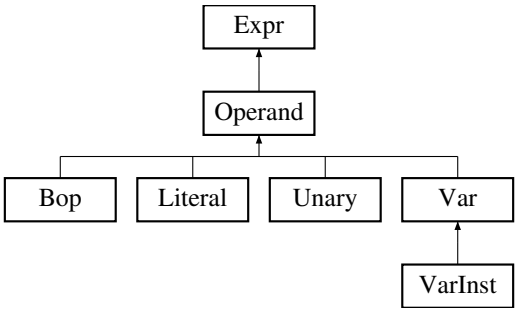
The documentation for this class was generated from the following file:

- expr.h

5.9 Operand Class Reference

`#include <expr.h>`

Inheritance diagram for Operand:



Public Member Functions

- **Operand** (std::unique_ptr< Token > t)
- **Operand** (const Token &t)
- **Operand** (const Token &&t)
- virtual Literal * get_value ()=0
- virtual bool truthy ()=0
- virtual int get_type ()=0
- virtual Literal operator+ (Operand &l)=0
- virtual Literal operator- (Operand &l)=0
- virtual Literal operator* (Operand &l)=0
- virtual Literal operator/ (Operand &l)=0
- virtual Literal operator&& (Operand &l)=0
- virtual Literal operator== (Operand &l)=0
- virtual Literal operator< (Operand &l)=0
- virtual Literal operator! ()=0

Public Member Functions inherited from Expr

- **Expr** (std::unique_ptr< Token > t)
- **Expr** (const Token &t)
- virtual void interpet (void)=0
- virtual void visit (void) const =0
- virtual bool analyse () const =0

Additional Inherited Members

Protected Attributes inherited from Expr

- std::unique_ptr< Token > **token**

5.9.1 Detailed Description

Abstract class that handles is base for all derived nodes that can do arithmetic.

5.9.2 Member Function Documentation

5.9.2.1 get_type()

```
virtual int Operand::get_type ( )  [pure virtual]
```

Gets the type of a variable/literal/operation.

Implemented in Literal, Var, VarInst, Bop, and Unary.

5.9.2.2 get_value()

```
virtual Literal * Operand::get_value ( )  [pure virtual]
```

Gets the value of a variable/literal/operation.

Implemented in Literal, Var, Bop, and Unary.

5.9.2.3 operator"!()

```
virtual Literal Operand::operator!  ( )  [pure virtual]
```

Does unary on a literal, extended from the miniPL spec to be defined for both integers and booleans.

Returns

Implemented in Literal, Var, VarInst, Bop, and Unary.

5.9.2.4 operator&&()

```
virtual Literal Operand::operator&& (
            Operand & l )  [pure virtual]
```

Does logical AND on a literal, only defined for booleans.

Returns

The result placed into a Literal

Implemented in VarInst, Bop, Unary, Literal, and Var.

5.9.2.5 operator∗()

```
virtual Literal Operand::operator* (
            Operand & l )  [pure virtual]
```

Does multiplication on a literal, only defined for integers.

Returns

The result placed into a Literal

Implemented in VarInst, Bop, Unary, Literal, and Var.

5.9.2.6 operator+()

```
virtual Literal Operand::operator+ (
            Operand & l )  [pure virtual]
```

Does addition on a literal, defined for integers and strings.

Returns

The result placed into a Literal

Implemented in VarInst, Bop, Unary, Literal, and Var.

5.9.2.7 operator-()

```
virtual Literal Operand::operator- (
            Operand & l )  [pure virtual]
```

Does subtraction on an Operand, only defined for integers.

Returns

The result placed into a Literal

Implemented in VarInst, Bop, Unary, Literal, and Var.

5.9.2.8 operator/()

```
virtual Literal Operand::operator/ (
            Operand & l )  [pure virtual]
```

Does division on a literal, only defined for integers.

Returns

The result placed into a Literal

Implemented in VarInst, Bop, Unary, Literal, and Var.

5.9.2.9 operator<()

```
virtual Literal Operand::operator< (
            Operand & l )  [pure virtual]
```

Does logical LESS THAN on a literal, defined for any any type.

Returns

The result placed into a Literal

Implemented in VarInst, Bop, Unary, Literal, and Var.

5.9.2.10 operator==()

```
virtual Literal Operand::operator== (
            Operand & l )  [pure virtual]
```

Does logical EQUALS on a literal.

Returns

The result placed into a Literal

Implemented in VarInst, Bop, Unary, Literal, and Var.

5.9.2.11 truthy()

```
virtual bool Operand::truthy ( )  [pure virtual]
```

Evaluates the expression and checks if it's truthy, all non-zero values and non-empty strings are considered truthy.

Implemented in Literal, Var, Bop, and Unary.

The documentation for this class was generated from the following file:

- expr.h

5.10 Parser Class Reference

Public Member Functions

- **Parser** (std::string_view filename)
- std::unique_ptr< StatementList > **parse_file** ()

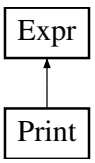
The documentation for this class was generated from the following files:

- parser.h
- parser.cpp

5.11 Print Class Reference

```
#include <expr.h>
```

Inheritance diagram for Print:



Public Member Functions

- **Print** (std::unique_ptr< Operand > expr)
- void interpet (void) override
- void visit (void) const override
- bool analyse () const override

Public Member Functions inherited from Expr

- **Expr** (std::unique_ptr< Token > t)
- **Expr** (const Token &t)
- virtual void interpet (void)=0
- virtual void visit (void) const =0
- virtual bool analyse () const =0

Additional Inherited Members

Protected Attributes inherited from Expr

- std::unique_ptr< Token > **token**

5.11.1 Detailed Description

Print node in the AST.

5.11.2 Member Function Documentation

5.11.2.1 analyse()

```
bool Print::analyse ( ) const  [inline], [override], [virtual]
```

Does analysis on the current statement.

Implements Expr.

5.11.2.2 interpet()

```
void Print::interpet (
            void  )  [inline], [override], [virtual]
```

Implements Expr.

5.11.2.3 visit()

```
void Print::visit (
            void  ) const  [inline], [override], [virtual]
```

Print the AST prettily

Implements Expr.

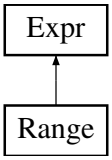
The documentation for this class was generated from the following file:

- expr.h

5.12 Range Class Reference

```
#include <expr.h>
```

Inheritance diagram for Range:



Public Member Functions

- **Range** (std::unique_ptr< Operand > start, std::unique_ptr< Operand > end)
- int get_next ()
- bool is_done ()
- bool analyse () const override
- void interpet () override
- void visit () const override

Public Member Functions inherited from Expr

- **Expr** (std::unique_ptr< Token > t)
- **Expr** (const Token &t)
- virtual void interpet (void)=0
- virtual void visit (void) const =0
- virtual bool analyse () const =0

Additional Inherited Members

Protected Attributes inherited from Expr

- std::unique_ptr< Token > **token**

5.12.1 Detailed Description

Range node for the AST.

5.12.2 Member Function Documentation

5.12.2.1 analyse()

```
bool Range::analyse ( ) const  [inline], [override], [virtual]
```

Does analysis on the current statement.

Implements Expr.

5.12.2.2 get_next()

```
int Range::get_next ( )  [inline]
```

Gets the next number for the range, needs to be manually checked that it does not go out of range.

5.12.2.3 interpet()

```
void Range::interpet (
            void  )  [inline], [override], [virtual]
```

Must be called before loop is executed, initializes val for the loop, cannot be done in analyse, since it's a constant function and in the constructor the symbol table is not initialized yet.

Implements Expr.

5.12.2.4 is_done()

```
bool Range::is_done ( )  [inline]
```

Checks if all numbers from the range is consumed.

5.12.2.5 visit()

```
void Range::visit ( ) const  [inline], [override], [virtual]
```

Print the AST prettily

Implements Expr.

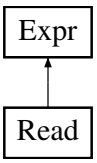
The documentation for this class was generated from the following file:

- expr.h

5.13 Read Class Reference

```
#include <expr.h>
```

Inheritance diagram for Read:



Public Member Functions

- **Read** (std::unique_ptr< Var > op)
- void interpet (void) override
- void visit (void) const override
- bool analyse () const override

Public Member Functions inherited from Expr

- **Expr** (std::unique_ptr< Token > t)
- **Expr** (const Token &t)
- virtual void interpet (void)=0
- virtual void visit (void) const =0
- virtual bool analyse () const =0

Additional Inherited Members

Protected Attributes inherited from Expr

- std::unique_ptr< Token > **token**

5.13.1 Detailed Description

Read node for standard input in the AST.

5.13.2 Member Function Documentation

5.13.2.1 analyse()

```
bool Read::analyse ( ) const  [inline], [override], [virtual]
```

Does analysis on the current statement.

Implements Expr.

5.13.2.2 interpet()

```
void Read::interpet (
            void  )  [inline], [override], [virtual]
```

Implements Expr.

5.13.2.3 visit()

```
void Read::visit (
            void  ) const  [inline], [override], [virtual]
```

Print the AST prettily

Implements Expr.

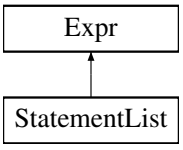
The documentation for this class was generated from the following file:

- expr.h

5.14 StatementList Class Reference

```
#include <expr.h>
```

Inheritance diagram for StatementList:



Public Member Functions

- **StatementList** (std::unique_ptr< Expr > stmt)
- bool analyse () const override
- void interpet (void) override
- void visit (void) const override
- void **add_child** (std::unique_ptr< Expr > stmt)
- StatementList * **get_next** () const
- Expr * **get_statement** () const

Public Member Functions inherited from Expr

- **Expr** (std::unique_ptr< Token > t)
- **Expr** (const Token &t)
- virtual void interpet (void)=0
- virtual void visit (void) const =0
- virtual bool analyse () const =0

Additional Inherited Members

Protected Attributes inherited from Expr

- std::unique_ptr< Token > **token**

5.14.1 Detailed Description

The core of the AST, it will have the pointer to the current statement and to the next.

Example: X := 1; := X 1

5.14.2 Member Function Documentation

5.14.2.1 analyse()

```
bool StatementList::analyse ( ) const  [inline], [override], [virtual]
```

Does analysis on the current statement.

Implements Expr.

5.14.2.2 interpet()

```
void StatementList::interpet (
            void  )  [inline], [override], [virtual]
```

Implements Expr.

5.14.2.3 visit()

```
void StatementList::visit (
            void  ) const  [inline], [override], [virtual]
```

Print the AST prettily

Implements Expr.

The documentation for this class was generated from the following file:

- expr.h

5.15 SymbolTable Class Reference

Public Member Functions

- bool add_symbol (std::string_view symbol, std::unique_ptr< Literal > value)
- bool add_symbol (std::string_view symbol, int type)
- Literal * **get_symbol** (std::string_view symbol)
- bool exists (std::string_view symbol)
- bool **set_value** (std::string_view symbol, Literal *literal)
- bool **set_value** (std::string_view symbol, std::variant< int, std::string, bool > value)

5.15.1 Member Function Documentation

5.15.1.1 add_symbol() [1/2]

```
bool SymbolTable::add_symbol (
            std::string_view symbol,
            int type )
```

Adds a symbol to the symbol table.

Returns

true succeeding to add to table, false if not.

5.15.1.2 add_symbol() [2/2]

```
bool SymbolTable::add_symbol (
            std::string_view symbol,
            std::unique_ptr< Literal > value )
```

Adds a symbol to the symbol table.

Returns

true succeeding to add to table, false if not.

5.15.1.3 exists()

```
bool SymbolTable::exists (
            std::string_view symbol )
```

Checks whether a symbol is present in the symbol table.

The documentation for this class was generated from the following files:

- symbols.h
- symbols.cpp

5.16 Token Struct Reference

Public Member Functions

- **Token** (std::string token, std::size_t line, enum token_type type)
- **Token** (const Token &tok)
- **Token** (const Token &&tok)

Public Attributes

- std::string **token**
- std::size_t **line**
- enum token_type **type**

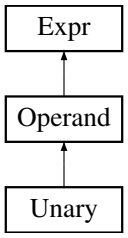
The documentation for this struct was generated from the following file:

- token.h

5.17 Unary Class Reference

`#include <expr.h>`

Inheritance diagram for Unary:



Public Member Functions

- **Unary** (std::unique_ptr< Operand > op)
- virtual Literal * get_value () override
- bool truthy () override
- virtual int get_type () override
- virtual void interpet (void) override
- virtual void visit (void) const override
- virtual bool analyse () const override
- Literal operator+ (Operand &) override
- Literal operator- (Operand &) override
- Literal operator* (Operand &) override
- Literal operator/ (Operand &) override
- Literal operator&& (Operand &) override
- Literal operator== (Operand &) override
- Literal operator< (Operand &) override
- Literal operator! () override

Public Member Functions inherited from Operand

- **Operand** (std::unique_ptr< Token > t)
- **Operand** (const Token &t)
- **Operand** (const Token &&t)
- virtual Literal * get_value ()=0
- virtual bool truthy ()=0
- virtual int get_type ()=0
- virtual Literal operator+ (Operand &l)=0
- virtual Literal operator- (Operand &l)=0
- virtual Literal operator* (Operand &l)=0
- virtual Literal operator/ (Operand &l)=0
- virtual Literal operator&& (Operand &l)=0
- virtual Literal operator== (Operand &l)=0
- virtual Literal operator< (Operand &l)=0
- virtual Literal operator! ()=0

Public Member Functions inherited from Expr

- **Expr** (std::unique_ptr< Token > t)
- **Expr** (const Token &t)
- virtual void interpet (void)=0
- virtual void visit (void) const =0
- virtual bool analyse () const =0

Additional Inherited Members

Protected Attributes inherited from Expr

- std::unique_ptr< Token > **token**

5.17.1 Detailed Description

Unary operation in the AST, needs to be handled differently from a normal BOP.

5.17.2 Member Function Documentation

5.17.2.1 analyse()

```
virtual bool Unary::analyse ( ) const  [inline], [override], [virtual]
```

Does analysis on the current statement.

Implements Expr.

5.17.2.2 get_type()

```
virtual int Unary::get_type ( )  [inline], [override], [virtual]
```

Gets the type of a variable/literal/operation.

Implements Operand.

5.17.2.3 get_value()

```
virtual Literal * Unary::get_value ( )  [inline], [override], [virtual]
```

Gets the value of a variable/literal/operation.

Implements Operand.

5.17.2.4 interpet()

```
virtual void Unary::interpet (
            void  )  [inline], [override], [virtual]
```

Implements Expr.

5.17.2.5 operator"!()

```
Literal Unary::operator!  ( )  [inline], [override], [virtual]
```

Does unary on a literal, extended from the miniPL spec to be defined for both integers and booleans.

Returns

Implements Operand.

5.17.2.6 operator&&()

```
Literal Unary::operator&& (
            Operand & l )  [inline], [override], [virtual]
```

Does logical AND on a literal, only defined for booleans.

Returns

The result placed into a Literal

Implements Operand.

5.17.2.7 operator∗()

```
Literal Unary::operator* (
            Operand & l )  [inline], [override], [virtual]
```

Does multiplication on a literal, only defined for integers.

Returns

The result placed into a Literal

Implements Operand.

5.17.2.8 operator+()

```
Literal Unary::operator+ (
            Operand & l )  [inline], [override], [virtual]
```

Does addition on a literal, defined for integers and strings.

Returns

The result placed into a Literal

Implements Operand.

5.17.2.9 operator-()

```
Literal Unary::operator- (
            Operand & l )  [inline], [override], [virtual]
```

Does subtraction on an Operand, only defined for integers.

Returns

The result placed into a Literal

Implements Operand.

5.17.2.10 operator/()

```
Literal Unary::operator/ (
            Operand & l )  [inline], [override], [virtual]
```

Does division on a literal, only defined for integers.

Returns

The result placed into a Literal

Implements Operand.

5.17.2.11 operator<()

```
Literal Unary::operator< (
            Operand & l )  [inline], [override], [virtual]
```

Does logical LESS THAN on a literal, defined for any any type.

Returns

The result placed into a Literal

Implements Operand.

5.17.2.12 operator==()

```
Literal Unary::operator== (
            Operand & l )  [inline], [override], [virtual]
```

Does logical EQUALS on a literal.

Returns

The result placed into a Literal

Implements Operand.

5.17.2.13 truthy()

```
bool Unary::truthy ( )  [inline], [override], [virtual]
```

Evaluates the expression and checks if it's truthy, all non-zero values and non-empty strings are considered truthy.

Implements Operand.

5.17.2.14 visit()

```
virtual void Unary::visit (
            void  ) const  [inline], [override], [virtual]
```

Print the AST prettily

Implements Expr.

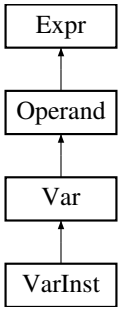
The documentation for this class was generated from the following file:

- expr.h

5.18 Var Class Reference

`#include <expr.h>`

Inheritance diagram for Var:



Public Member Functions

- **Var** (std::unique_ptr< Token > tok)
- **Var** (const Var &var)
- bool analyse () const override
- virtual void interpet (void) override
- virtual void visit (void) const override
- Literal * get_value () override
- bool truthy () override
- int get_type () override
- void **set_value** (Literal *literal)
- void **set_value** (std::variant< int, std::string, bool > value)
- Literal operator+ (Operand &l) override
- Literal operator- (Operand &l) override
- Literal operator* (Operand &l) override
- Literal operator/ (Operand &l) override
- Literal operator&& (Operand &l) override
- Literal operator== (Operand &l) override
- Literal operator! () override
- Literal operator< (Operand &l) override

Public Member Functions inherited from Operand

- **Operand** (std::unique_ptr< Token > t)
- **Operand** (const Token &t)
- **Operand** (const Token &&t)
- virtual Literal * get_value ()=0
- virtual bool truthy ()=0
- virtual int get_type ()=0
- virtual Literal operator+ (Operand &l)=0
- virtual Literal operator- (Operand &l)=0
- virtual Literal operator* (Operand &l)=0
- virtual Literal operator/ (Operand &l)=0
- virtual Literal operator&& (Operand &l)=0
- virtual Literal operator== (Operand &l)=0
- virtual Literal operator< (Operand &l)=0
- virtual Literal operator! ()=0

Public Member Functions inherited from Expr

- **Expr** (std::unique_ptr< Token > t)
- **Expr** (const Token &t)
- virtual void interpet (void)=0
- virtual void visit (void) const =0
- virtual bool analyse () const =0

Additional Inherited Members

Protected Attributes inherited from Expr

- std::unique_ptr< Token > **token**

5.18.1 Detailed Description

A reference to a variable.

5.18.2 Member Function Documentation

5.18.2.1 analyse()

```
bool Var::analyse ( ) const  [inline], [override], [virtual]
```

Does analysis on the current statement.

Implements Expr.

Reimplemented in VarInst.

5.18.2.2 get_type()

```
int Var::get_type ( )  [inline], [override], [virtual]
```

Gets the type of a variable/literal/operation.

Implements Operand.

Reimplemented in VarInst.

5.18.2.3 get_value()

```
Literal * Var::get_value ( )  [inline], [override], [virtual]
```

Gets the value of a variable/literal/operation.

Implements Operand.

5.18.2.4 interpet()

```
virtual void Var::interpet (
            void  )  [inline], [override], [virtual]
```

Implements Expr.

5.18.2.5 operator"!()

```
Literal Var::operator!  ( )  [inline], [override], [virtual]
```

Does unary on a literal, extended from the miniPL spec to be defined for both integers and booleans.

Returns

Implements Operand.

Reimplemented in VarInst.

5.18.2.6 operator&&()

```
Literal Var::operator&& (
            Operand & l )  [inline], [override], [virtual]
```

Does logical AND on a literal, only defined for booleans.

Returns

The result placed into a Literal

Implements Operand.

Reimplemented in VarInst.

5.18.2.7 operator*()

```
Literal Var::operator* (
            Operand & l )  [inline], [override], [virtual]
```

Does multiplication on a literal, only defined for integers.

Returns

The result placed into a Literal

Implements Operand.

Reimplemented in VarInst.

5.18.2.8 operator+()

```
Literal Var::operator+ (
            Operand & l )  [inline], [override], [virtual]
```

Does addition on a literal, defined for integers and strings.

Returns

The result placed into a Literal

Implements Operand.

Reimplemented in VarInst.

5.18.2.9 operator-()

```
Literal Var::operator- (
            Operand & l )  [inline], [override], [virtual]
```

Does subtraction on an Operand, only defined for integers.

Returns

The result placed into a Literal

Implements Operand.

Reimplemented in VarInst.

5.18.2.10 operator/()

```
Literal Var::operator/ (
            Operand & l )  [inline], [override], [virtual]
```

Does division on a literal, only defined for integers.

Returns

The result placed into a Literal

Implements Operand.

Reimplemented in VarInst.

5.18.2.11 operator<()

```
Literal Var::operator< (
            Operand & l )  [inline], [override], [virtual]
```

Does logical LESS THAN on a literal, defined for any any type.

Returns

The result placed into a Literal

Implements Operand.

Reimplemented in VarInst.

5.18.2.12 operator==()

```
Literal Var::operator== (
            Operand & l )  [inline], [override], [virtual]
```

Does logical EQUALS on a literal.

Returns

The result placed into a Literal

Implements Operand.

Reimplemented in VarInst.

5.18.2.13 truthy()

```
bool Var::truthy ( )  [inline], [override], [virtual]
```

Evaluates the expression and checks if it's truthy, all non-zero values and non-empty strings are considered truthy.

Implements Operand.

5.18.2.14 visit()

```
virtual void Var::visit (
            void  ) const  [inline], [override], [virtual]
```

Print the AST prettily

Implements Expr.

Reimplemented in VarInst.

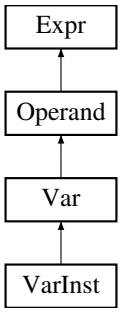
The documentation for this class was generated from the following file:

- expr.h

5.19 VarInst Class Reference

```
#include <expr.h>
```

Inheritance diagram for VarInst:



Public Member Functions

- **VarInst** (std::unique_ptr< Token > tok, enum token_type type)
- **VarInst** (std::unique_ptr< Token > tok, std::unique_ptr< Token > type)
- int get_type () override
- bool analyse () const override
- virtual void interpet () override
- virtual void visit (void) const override
- Literal operator+ (Operand &) override
- Literal operator- (Operand &) override
- Literal operator∗ (Operand &) override
- Literal operator/ (Operand &) override
- Literal operator&& (Operand &) override
- Literal operator== (Operand &) override
- Literal operator< (Operand &) override
- Literal operator! () override

Public Member Functions inherited from Var

- **Var** (std::unique_ptr< Token > tok)
- **Var** (const Var &var)
- bool analyse () const override
- virtual void interpet (void) override
- virtual void visit (void) const override
- Literal * get_value () override
- bool truthy () override
- int get_type () override
- void **set_value** (Literal *literal)
- void **set_value** (std::variant< int, std::string, bool > value)
- Literal operator+ (Operand &l) override
- Literal operator- (Operand &l) override
- Literal operator* (Operand &l) override
- Literal operator/ (Operand &l) override
- Literal operator&& (Operand &l) override
- Literal operator== (Operand &l) override
- Literal operator! () override
- Literal operator< (Operand &l) override

Public Member Functions inherited from Operand

- **Operand** (std::unique_ptr< Token > t)
- **Operand** (const Token &t)
- **Operand** (const Token &&t)
- virtual Literal * get_value ()=0
- virtual bool truthy ()=0
- virtual int get_type ()=0
- virtual Literal operator+ (Operand &l)=0
- virtual Literal operator- (Operand &l)=0
- virtual Literal operator* (Operand &l)=0
- virtual Literal operator/ (Operand &l)=0
- virtual Literal operator&& (Operand &l)=0
- virtual Literal operator== (Operand &l)=0
- virtual Literal operator< (Operand &l)=0
- virtual Literal operator! ()=0

Public Member Functions inherited from Expr

- **Expr** (std::unique_ptr< Token > t)
- **Expr** (const Token &t)
- virtual void interpet (void)=0
- virtual void visit (void) const =0
- virtual bool analyse () const =0

Additional Inherited Members

Protected Attributes inherited from Expr

- std::unique_ptr< Token > **token**

5.19.1 Detailed Description

Initialization of a variable.

5.19.2 Member Function Documentation

5.19.2.1 analyse()

```
bool VarInst::analyse ( ) const  [inline], [override], [virtual]
```

Does analysis on the current statement.

Reimplemented from Var.

5.19.2.2 get_type()

```
int VarInst::get_type ( )  [inline], [override], [virtual]
```

Gets the type of a variable/literal/operation.

Reimplemented from Var.

5.19.2.3 interpet()

```
virtual void VarInst::interpet (
            void  )  [inline], [override], [virtual]
```

Reimplemented from Var.

5.19.2.4 operator"!()

```
Literal VarInst::operator!  ( )  [inline], [override], [virtual]
```

Does unary on a literal, extended from the miniPL spec to be defined for both integers and booleans.

Returns

Reimplemented from Var.

5.19.2.5 operator&&()

```
Literal VarInst::operator&& (
            Operand & l )  [inline], [override], [virtual]
```

Does logical AND on a literal, only defined for booleans.

Returns

The result placed into a Literal

Reimplemented from Var.

5.19.2.6 operator∗()

```
Literal VarInst::operator* (
            Operand & l )  [inline], [override], [virtual]
```

Does multiplication on a literal, only defined for integers.

Returns

The result placed into a Literal

Reimplemented from Var.

5.19.2.7 operator+()

```
Literal VarInst::operator+ (
            Operand & l )  [inline], [override], [virtual]
```

Does addition on a literal, defined for integers and strings.

Returns

The result placed into a Literal

Reimplemented from Var.

5.19.2.8 operator-()

```
Literal VarInst::operator- (
            Operand & l )  [inline], [override], [virtual]
```

Does subtraction on an Operand, only defined for integers.

Returns

The result placed into a Literal

Reimplemented from Var.

5.19.2.9 operator/()

```
Literal VarInst::operator/ (
            Operand & l )  [inline], [override], [virtual]
```

Does division on a literal, only defined for integers.

Returns

The result placed into a Literal

Reimplemented from Var.

5.19.2.10 operator<()

```
Literal VarInst::operator< (
            Operand & l )  [inline], [override], [virtual]
```

Does logical LESS THAN on a literal, defined for any any type.

Returns

The result placed into a Literal

Reimplemented from Var.

5.19.2.11 operator==()

```
Literal VarInst::operator== (
            Operand & l )  [inline], [override], [virtual]
```

Does logical EQUALS on a literal.

Returns

The result placed into a Literal

Reimplemented from Var.

5.19.2.12 visit()

```
virtual void VarInst::visit (
            void  ) const  [inline], [override], [virtual]
```

Print the AST prettily

Reimplemented from Var.

The documentation for this class was generated from the following file:

- expr.h

Chapter 6

File Documentation

6.1 analysis.h

```
00001 #ifndef ANALYSIS_H
00002 #define ANALYSIS_H
00003
00004 #include "parser.h"
00005
00009 class Analyser {
00010     public:
00011         Analyser() = default;
00016         bool analyse(StatementList *ast);
00017 };
00018
00019 #endif  // ANALYSIS_H
```

6.2 expr.h

```
00001
00004 #ifndef EXPR_H
00005 #define EXPR_H
00006 #include <memory>
00007 #include <string>
00008 #include <variant>
00009 #include <iostream>
00010
00011 #include "token.h"
00012 #include "symbols.h"
00013 #include "lexer.h"
00014
00015 class Literal;
00016
00017
00018 class Expr {
00019 protected:
00020     std::unique_ptr<Token> token;
00021 public:
00022     Expr(std::unique_ptr<Token> t) : token{std::move(t)} {}
00023     Expr() = default;
00024     Expr(const Token &t) : token(std::make_unique<Token>(t)) {}
00025     virtual ~Expr() = default;
00026
00027     virtual void interpet(void) = 0;
00031     virtual void visit(void) const = 0;
00032
00036     virtual bool analyse() const = 0;
00037 };
00038
00043 class Operand : public Expr {
00044     public:
00045
00046         Operand(std::unique_ptr<Token> t) : Expr(std::move(t)) {}
00047         Operand(const Token &t) : Expr(t) {}
00048         Operand(const Token &&t) : Expr(t) {}
00049         Operand() = default;
00050
00054         virtual Literal *get_value() = 0;
```

```
00055
00060         virtual bool truthy() = 0;
00061
00065         virtual int get_type() = 0;
00066
00072         virtual Literal operator+(Operand &l) = 0;
00073
00079         virtual Literal operator-(Operand &l) = 0;
00080
00086         virtual Literal operator*(Operand &l) = 0;
00087
00093         virtual Literal operator/(Operand &l) = 0;
00094
00100         virtual Literal operator&&(Operand &l) = 0;
00101
00107         virtual Literal operator==(Operand &l) = 0;
00108
00114         virtual Literal operator<(Operand &l)= 0;
00115
00122         virtual Literal operator!() = 0;
00123 };
00124
00133 class StatementList : public Expr {
00134     std::unique_ptr<Expr> statement;
00135     std::unique_ptr<StatementList> next;
00136
00137 public:
00138     StatementList(std::unique_ptr<Expr> stmt)
00139         : statement{std::move(stmt)} {}
00140
00141     bool analyse() const override {
00142         bool error = statement->analyse();
00143
00144         if (next != nullptr) {
00145             error |= next->analyse();
00146         }
00147         return error;
00148     }
00149
00150     void interpet(void) override {
00151         statement->interpet();
00152         if (next != nullptr) {
00153             next->interpet();
00154         }
00155     }
00156
00157     void visit(void) const override {
00158         statement->visit();
00159         if (next != nullptr) {
00160             next->visit();
00161         }
00162     }
00163
00164     void add_child(std::unique_ptr<Expr> stmt) {
00165         if (next == nullptr) {
00166             next = std::make_unique<StatementList>(std::move(stmt));
00167         } else {
00168             next->add_child(std::move(stmt));
00169         }
00170     }
00171
00172     StatementList *get_next() const {
00173         return next.get();
00174     }
00175
00176     Expr *get_statement() const {
00177         return statement.get();
00178     }
00179 };
00180
00187 class Literal : public Operand {
00188
00189 public:
00190     std::variant<int, std::string, bool> value;
00191     Literal(std::variant<int, std::string, bool> value) : Operand(), value{value} { }
00192     Literal(const Literal &l)  : Operand(), value{l.value} {}
00193     Literal(const Literal &&l) : Operand(), value{l.value} {}
00194
00195     Literal(std::unique_ptr<Token> &tok) : Operand(std::move(tok)) {
00196         switch (token->type) {
00197             case token_type::DIGIT:
00198                 value = std::atoi(token->token.c_str());
00199                 break;
00200             case token_type::STRING:
00201                 value = token->token;
00202                 break;
00203             default:
```

```
00204                 std::cout « "Error line " « token->line «
00205                     ": invalid token";
00206                 break;
00207         }
00208     }
00209
00210     bool analyse() const override { return false; }
00211     virtual void interpet(void) override {}
00212
00213     virtual void visit(void) const override  {
00214         std::visit([](const auto &x) { std::cout « x « " "; }, value);
00215     }
00216
00217     virtual Literal *get_value() override {
00218         return this;
00219     }
00220
00221     bool truthy() override {
00222         if (std::holds_alternative<int>(this->value)) {
00223             return !!std::get<int>(this->value);
00224         } else if(std::holds_alternative<bool>(this->value)) {
00225             return !!std::get<bool>(this->value);
00226         } else {
00227             return std::get<std::string>(this->value) != "";
00228         }
00229     }
00230
00231     virtual int get_type() override {
00232         return value.index();
00233     }
00234
00235     Literal operator+(Operand &l) override {
00236         if (std::holds_alternative<int>(this->value) &&
00237                 std::holds_alternative<int>(l.get_value()->value)) {
00238             return Literal{std::get<int>(this->value) + std::get<int>(l.get_value()->value)};
00239         } else if (std::holds_alternative<std::string>(this->value) &&
00240                 std::holds_alternative<std::string>(l.get_value()->value)) {
00241
00242             return Literal{std::get<std::string>(this->value) +
    std::get<std::string>(l.get_value()->value)};
00243         }
00244         std::cout « "Error invalid types in + operation";
00245         std::exit(1);
00246     }
00247
00248     Literal operator=(Literal &l) {
00249         if (this->value.index() == l.value.index()) {
00250             this->value = l.value;
00251             return *this;
00252         }
00253         std::cout « "Conflicting types in assignment";
00254         std::exit(1);
00255     }
00256
00257     Literal operator=(std::variant<int, std::string, bool> value) {
00258         if (this->value.index() == value.index()) {
00259             this->value = value;
00260             return *this;
00261         }
00262         std::cout « "Conflicting types in assignment";
00263         std::exit(1);
00264     }
00265
00266     Literal operator-(Operand &l) override {
00267         if (std::holds_alternative<int>(this->value) &&
    std::holds_alternative<int>(l.get_value()->value)) {
00268             return Literal{std::get<int>(this->value) - std::get<int>(l.get_value()->value)};
00269         }
00270
00271         std::cout « "Error invalid types in - operation";
00272         std::exit(1);
00273     }
00274
00275     Literal operator*(Operand &l) override {
00276         if (std::holds_alternative<int>(this->value) &&
    std::holds_alternative<int>(l.get_value()->value)) {
00277             return Literal{std::get<int>(this->value) * std::get<int>(l.get_value()->value)};
00278         }
00279
00280         std::cout « "Error invalid types in - operation";
00281         std::exit(1);
00282     }
00283
00284     Literal operator/(Operand &l) override {
00285         if (std::holds_alternative<int>(this->value) &&
    std::holds_alternative<int>(l.get_value()->value)) {
00286             return Literal{std::get<int>(this->value) / std::get<int>(l.get_value()->value)};
```

```
00287         }
00288
00289         std::cout « "Error invalid types in / operation";
00290         std::exit(1);
00291     }
00292
00293     Literal operator&&(Operand &l) override {
00294         if (std::holds_alternative<bool>(this->value) &&
      std::holds_alternative<bool>(l.get_value()->value)) {
00295             return Literal{std::get<bool>(this->value) && std::get<bool>(l.get_value()->value)};
00296         }
00297
00298         std::cout « "Error invalid types in & operation";
00299         std::exit(1);
00300     }
00301
00302     Literal operator==(Operand &l) override {
00303         if (this->value.index() == l.get_value()->value.index()) {
00304             switch(this->value.index()) {
00305                 case 0: // int
00306                     return Literal{std::get<int>(this->value) == std::get<int>(l.get_value()->value)};
00307                 case 1: // std::string
00308                     return Literal{std::get<std::string>(this->value) ==
      std::get<std::string>(l.get_value()->value)};
00309                 case 2:
00310                     return Literal{std::get<bool>(this->value) ==
      std::get<bool>(l.get_value()->value)};
00311             }
00312         }
00313
00314         std::cout « "Error invalid types in = operation";
00315         std::exit(1);
00316     }
00317
00318     Literal operator<(Operand &l) override {
00319         if (this->value.index() == l.get_value()->value.index()) {
00320             switch(this->value.index()) {
00321                 case 0: // int
00322                     return Literal{std::get<int>(this->value) < std::get<int>(l.get_value()->value)};
00323                 case 1: // std::string
00324                     return Literal{std::get<std::string>(this->value) <
      std::get<std::string>(l.get_value()->value)};
00325                 case 2:
00326                     return Literal{std::get<bool>(this->value) <
      std::get<bool>(l.get_value()->value)};
00327             }
00328         }
00329
00330         std::cout « "Error invalid types in < operation";
00331         std::exit(1);
00332     }
00333
00334     Literal operator!() override {
00335         if (std::holds_alternative<bool>(this->value)) {
00336             return Literal{!std::get<bool>(this->value)};
00337         } else if(std::holds_alternative<int>(this->value)) {
00338             return Literal{!std::get<int>(this->value)};
00339         }
00340         std::cout « "Error invalid types in ! operation";
00341         std::exit(1);
00342
00343     }
00344 };
00345
00349 class Var : public Operand {
00350
00351     public:
00352         Var(std::unique_ptr<Token> tok) :
00353             Operand(std::move(tok)) {}
00354         Var(const Var &var) :
00355             Operand(*var.token) {}
00356
00357         bool analyse() const override {
00358             if (!symbol_table.exists(token->token)) {
00359                 std::cout « "Error no variable named " « token->token « " line: " « token->line «
      "\n";
00360                 return true;
00361             }
00362             return false;
00363         }
00364
00365         virtual void interpet(void) override {}
00366         virtual void visit(void) const override  {
00367             std::cout « token->token « " ";
00368         }
00369
00370         Literal *get_value() override {
```

```
00371             return symbol_table.get_symbol(token->token);
00372         }
00373
00374         bool truthy() override {
00375             return symbol_table.get_symbol(token->token)->truthy();
00376         }
00377
00378         int get_type() override {
00379             return symbol_table.get_symbol(token->token)->get_type();
00380         }
00381
00382         void set_value(Literal *literal) {
00383             symbol_table.set_value(token->token, literal);
00384         }
00385
00386         void set_value(std::variant<int, std::string, bool> value) {
00387             symbol_table.set_value(token->token, value);
00388         }
00389
00390
00391         Literal operator+(Operand &l) override {
00392             Literal * ptr = symbol_table.get_symbol(token->token);
00393             return *ptr + *l.get_value();
00394         }
00395
00396         Literal operator-(Operand &l) override {
00397             Literal * ptr = symbol_table.get_symbol(token->token);
00398             return *ptr - *l.get_value();
00399         }
00400
00401         Literal operator*(Operand &l) override {
00402             Literal * ptr = symbol_table.get_symbol(token->token);
00403             return *ptr * *l.get_value();
00404         }
00405
00406         Literal operator/(Operand &l) override {
00407             Literal * ptr = symbol_table.get_symbol(token->token);
00408             return *ptr * *l.get_value();
00409         }
00410
00411         Literal operator&&(Operand &l) override {
00412             Literal * ptr = symbol_table.get_symbol(token->token);
00413             return *ptr && *l.get_value();
00414         }
00415
00416         Literal operator==(Operand &l) override {
00417             Literal * ptr = symbol_table.get_symbol(token->token);
00418             return *ptr == *l.get_value();
00419         }
00420
00421         Literal operator!() override {
00422             Literal * ptr = symbol_table.get_symbol(token->token);
00423             return !*ptr;
00424         }
00425
00426         Literal operator<(Operand &l) override {
00427             Literal * ptr = symbol_table.get_symbol(token->token);
00428             return *ptr < *l.get_value();
00429         }
00430 };
00431
00435 class VarInst : public Var {
00436     int type;
00437
00438     public:
00439         VarInst(std::unique_ptr<Token> tok, enum token_type type) :
00440             Var(std::move(tok)), type{type - token_type::INT} {}
00441
00442         VarInst(std::unique_ptr<Token> tok, std::unique_ptr<Token> type) :
00443             Var(std::move(tok)), type{type->type - token_type::INT} {}
00444
00445         int get_type() override {
00446             return symbol_table.get_symbol(token->token)->get_type();
00447         }
00448
00449         bool analyse() const override {
00450             bool succeeded = symbol_table.add_symbol(token->token, type);
00451
00452             if(!succeeded) {
00453                 std::cout « "Error token variable " « token->token
00454                     « " already initialized";
00455                 return true;
00456             }
00457             return false;
00458         }
00459
00460         // handled by Bop
```

```
00461         virtual void interpet() override {}
00462         virtual void visit(void) const override {
00463             std::cout « token->token « " ";
00464         }
00465 
00466         Literal operator+(Operand &) override {
00467             std::cout « "VarInst invalid operation\n";
00468             std::exit(1);
00469         }
00470 
00471         Literal operator-(Operand &) override {
00472             std::cout « "VarInst invalid operation\n";
00473             std::exit(1);
00474         }
00475 
00476         Literal operator*(Operand &) override {
00477             std::cout « "VarInst invalid operation\n";
00478             std::exit(1);
00479         }
00480 
00481         Literal operator/(Operand &) override {
00482             std::cout « "VarInst invalid operation\n";
00483             std::exit(1);
00484         }
00485 
00486         Literal operator&&(Operand &) override {
00487             std::cout « "VarInst invalid operation\n";
00488             std::exit(1);
00489         }
00490 
00491         Literal operator==(Operand &) override {
00492             std::cout « "VarInst invalid operation\n";
00493             std::exit(1);
00494         }
00495 
00496         Literal operator<(Operand &) override {
00497             std::cout « "VarInst invalid operation\n";
00498             std::exit(1);
00499         }
00500 
00501         Literal operator!() override {
00502             std::cout « "VarInst invalid operation\n";
00503             std::exit(1);
00504         }
00505 };
00506 
00507 
00511 class Bop : public Operand {
00512     std::unique_ptr<Operand> left;
00513     std::unique_ptr<Operand> right;
00514     std::unique_ptr<Literal> evaluated = nullptr;
00515 
00516     public:
00522         Bop(std::unique_ptr<Token> tok, std::unique_ptr<Operand> left, std::unique_ptr<Operand> right)
00523             : Operand(std::move(tok)), left{std::move(left)}, right{std::move(right)} {
00524                 if (left == nullptr || right == nullptr) {
00525                     std::cout « "Missing operand " « token->line;
00526                     std::exit(1);
00527                 }
00528             }
00529 
00530 
00531         bool analyse() const override {
00532             bool has_error = left->analyse();
00533             has_error |= right->analyse();
00534 
00535             if(has_error) {
00536                 std::cout « "Semantical error in bop\n";
00537             }
00538 
00539 
00540             return has_error;
00541         }
00542 
00547         virtual Literal *get_value() override {
00548             if (evaluated) {
00549                 // use cached value
00550                 return evaluated.get();
00551             }
00552 
00553             switch(token->type) {
00554                 case token_type::ADDITION:
00555                     evaluated = std::make_unique<Literal>(*left + *right->get_value());
00556                     break;
00557 
00558                 case token_type::SUBTRACTION:
00559                     evaluated = std::make_unique<Literal>(*left - *right->get_value());
```

```
00560                     break;
00561
00562                 case token_type::MULTIPLICATION:
00563                     evaluated = std::make_unique<Literal>(*left * *right->get_value());
00564                     break;
00565
00566                 case token_type::DIVISION:
00567                     evaluated = std::make_unique<Literal>(*left / *right->get_value());
00568                     break;
00569
00570                 case token_type::LT:
00571                     evaluated = std::make_unique<Literal>(*left < *right->get_value());
00572                     break;
00573
00574                 case token_type::EQ:
00575                     evaluated = std::make_unique<Literal>(*left == *right->get_value());
00576                     break;
00577
00578                 case token_type::AND:
00579                     evaluated = std::make_unique<Literal>(*left && *right->get_value());
00580                     break;
00581
00582                 case token_type::ASSIGN:
00583                     {
00584                         auto var = dynamic_cast<Var&>(*left);
00585                         var.set_value(right->get_value());
00586                     }
00587                     break;
00588                 default:
00589                     std::cout << "Invalid operation (" << token->token << ")";
00590                     break;
00591             }
00592             return evaluated.get();
00593         }
00594
00595         bool truthy() override {
00596             return get_value()->truthy();
00597         }
00598
00599         virtual int get_type() override {
00600             int r = right->get_type();
00601             int l = left->get_type();
00602
00603             if (l != r) {
00604                 std::cout << "Error: incompatible types line " << token->line << "\n";
00605                 std::exit(1);
00606             } else {
00607                 if (token->type == token_type::EQ || token->type == token_type::LT){
00608                     return 1;
00609                 }
00610             }
00611             return l;
00612         }
00613
00614         virtual void interpet(void) override {
00615             this->get_value();
00616         }
00617
00618         virtual void visit(void) const override  {
00619             std::cout << token->token << " ( ";
00620             left->visit();
00621             if (right)
00622                 right->visit();
00623             std::cout << ") ";
00624         }
00625
00626         virtual Literal operator+(Operand &) override {
00627             std::cout << "Invalid operation\n";
00628             std::exit(1);
00629         }
00630         virtual Literal operator-(Operand &) override {
00631             std::cout << "Invalid operation\n";
00632             std::exit(1);
00633         }
00634         virtual Literal operator*(Operand &) override {
00635             std::cout << "Invalid operation\n";
00636             std::exit(1);
00637         }
00638         virtual Literal operator/(Operand &) override {
00639             std::cout << "Invalid operation\n";
00640             std::exit(1);
00641         }
00642         virtual Literal operator&&(Operand &) override {
00643             std::cout << "Invalid operation\n";
00644             std::exit(1);
00645         }
00646         virtual Literal operator==(Operand &) override {
```

```
00647             std::cout « "Invalid operation\n";
00648             std::exit(1);
00649         }
00650         virtual Literal operator<(Operand &) override {
00651             std::cout « "Invalid operation\n";
00652             std::exit(1);
00653         }
00654         virtual Literal operator!() override {
00655             std::cout « "Invalid operation\n";
00656             std::exit(1);
00657         }
00658     protected:
00659         bool is_boolean() {
00660             return token->type == token_type::LT || token->type == token_type::EQ;
00661         }
00662 };
00663
00664
00665
00669 class If : public Expr {
00670     std::unique_ptr<Operand> condition;
00671     std::unique_ptr<StatementList> truthy;
00672     std::unique_ptr<StatementList> falsy;
00673
00674     public:
00683         If(std::unique_ptr<Operand> condition, std::unique_ptr<StatementList> truthy,
00684                 std::unique_ptr<StatementList> falsy) :
00685             Expr(), condition{std::move(condition)}, truthy{std::move(truthy)},
00686             falsy{std::move(falsy)} {}
00687
00688         bool analyse() const override {
00689             bool error = condition->analyse() || truthy->analyse();
00690             if (falsy != nullptr) {
00691                 error |= falsy->analyse();
00692             }
00693             return error;
00694         }
00695
00696         void interpet() override {
00697             if(condition->truthy()) {
00698                 truthy->interpet();
00699             } else {
00700                 if (falsy != nullptr) {
00701                     falsy->interpet();
00702                 }
00703             }
00704         }
00705
00706         void visit() const override {
00707             std::cout « "( IF ";
00708             condition->visit();
00709             std::cout « "(";
00710             truthy->visit();
00711             std::cout « ")";
00712             if (falsy) {
00713                 std::cout « " ELSE (";
00714                 falsy->visit();
00715                 std::cout « ")";
00716             }
00717             std::cout « ")";
00718         }
00719 };
00720
00724 class Range : public Expr {
00725     std::unique_ptr<Operand> start;
00726     std::unique_ptr<Operand> end;
00727     int current;
00728
00729     public:
00730         Range(std::unique_ptr<Operand> start, std::unique_ptr<Operand> end) :
00731             Expr(), start{std::move(start)}, end{std::move(end)}, current{0} { }
00732
00737         int get_next() {
00738             return current++;
00739         }
00740
00744         bool is_done() {
00745             return !(current <= (std::get<int>(end->get_value()->value)));
00746         }
00747
00748         bool analyse() const override {
00749            bool error = start->get_type() || end->get_type();
00750            if (error) {
00751                std::cout « "Error range contains non integers\n";
00752                return true;
00753            }
00754            return false;
```

```
00755         }
00756
00762         void interpet() override {
00763             auto val = std::get_if<int>(&this->start->get_value()->value);
00764             if (val) {
00765                 current = *val;
00766             }
00767         }
00768         void visit() const override {
00769             start->visit();
00770             std::cout « "..";
00771             end->visit();
00772         }
00773 };
00774
00778 class For : public Expr {
00779     std::unique_ptr<Var> var;
00780     std::unique_ptr<Range> range;
00781     std::unique_ptr<StatementList> loop;
00782
00783     public:
00784         For(std::unique_ptr<Token> variable, std::unique_ptr<Range> range,
00785                 std::unique_ptr<StatementList> statement_list) : Expr(),
    var{std::make_unique<Var>(std::move(variable))},
00786                     range{std::move(range)}, loop{std::move(statement_list)} {}
00787
00788         bool analyse() const override {
00789             return var->analyse() || loop->analyse() || range->analyse();
00790         }
00791
00792         void interpet() override {
00793             range->interpet();
00794             while(!range->is_done()) {
00795                 var->set_value(range->get_next());
00796                 loop->interpet();
00797             }
00798         }
00799
00800         void visit(void) const override {
00801             std::cout « "(FOR ";
00802             var->visit();
00803             range->visit();
00804             std::cout « " ( ";
00805             loop->visit();
00806             std::cout « ")";
00807         }
00808 };
00809
00813 class Print : public Expr {
00814     std::unique_ptr<Operand> expr;
00815
00816 public:
00817     Print(std::unique_ptr<Operand> expr) : Expr(), expr{std::move(expr)} {}
00818
00819     void interpet(void) override {
00820         std::visit([](const auto &x) { std::cout « x « "\n"; }, expr->get_value()->value);
00821     }
00822     void visit(void) const override {
00823         std::cout « "( PRINT ";
00824         expr->visit();
00825         std::cout « ") ";
00826     }
00827     bool analyse() const override { return expr->analyse(); }
00828 };
00829
00833 class Read : public Expr {
00834     std::unique_ptr<Var> var;
00835
00836 public:
00837     Read(std::unique_ptr<Var> op) : Expr(), var{std::move(op)} {}
00838
00839     void interpet(void) override {
00840         switch(var->get_type()) {
00841             case 0:
00842                 {
00843                     int val;
00844                     std::cin » val;
00845                     var->set_value(val);
00846                 }
00847                 break;
00848             case 1:
00849                 {
00850                     std::string str;
00851                     std::cin » str;
00852                     var->set_value(str);
00853                 }
00854                 break;
```

```
00855
00856             case 2:
00857                 {
00858                     bool b;
00859                     std::cin » b;
00860                     var->set_value(b);
00861                 }
00862                 break;
00863         }
00864         if (!std::cin) {
00865             std::cout « "Error: input was of the wrong type, exitting...\n";
00866             std::exit(1);
00867         }
00868     }
00869
00870     void visit(void) const override {
00871         std::cout « "( READ ";
00872         var->visit();
00873         std::cout « ") ";
00874     }
00875
00876     bool analyse() const override {
00877         return var->analyse();
00878     }
00879 };
00880
00885 class Unary : public Operand {
00886     std::unique_ptr<Operand> op;
00887     std::unique_ptr<Literal> evaluated;
00888
00889     public:
00890         Unary(std::unique_ptr<Operand> op) : Operand(), op {std::move(op)} {}
00891
00892         virtual Literal *get_value() override {
00893             auto val = op->get_value();
00894             evaluated = std::make_unique<Literal>(!*val);
00895             return evaluated.get();
00896         }
00897
00898         bool truthy() override {
00899             return !op->truthy();
00900         }
00901
00902         virtual int get_type() override {
00903            return 2;
00904         }
00905
00906         virtual void interpet(void) override {
00907
00908         }
00909
00910         virtual void visit(void) const override {
00911             std::cout « "( ! ";
00912             op->visit();
00913             std::cout « ") ";
00914         }
00915
00916         virtual bool analyse() const override {
00917             return op->analyse();
00918         }
00919
00920         Literal operator+(Operand &) override {
00921             std::cout « "Invalid operation\n";
00922             std::exit(1);
00923         }
00924
00925         Literal operator-(Operand &) override {
00926             std::cout « "Invalid operation\n";
00927             std::exit(1);
00928         }
00929
00930         Literal operator*(Operand &) override {
00931             std::cout « "Invalid operation\n";
00932             std::exit(1);
00933         }
00934
00935         Literal operator/(Operand &) override {
00936             std::cout « "Invalid operation\n";
00937             std::exit(1);
00938         }
00939
00940         Literal operator&&(Operand &) override {
00941             std::cout « "Invalid operation\n";
00942             std::exit(1);
00943         }
00944
00945         Literal operator==(Operand &) override {
```

```
00946             std::cout « "Invalid operation\n";
00947             std::exit(1);
00948         }
00949
00950         Literal operator<(Operand &) override {
00951             std::cout « "Invalid operation\n";
00952             std::exit(1);
00953         }
00954
00955         Literal operator!() override {
00956             return !*this->op;
00957         }
00958 };
00959 #endif
```

6.3 interpreter.h

```
00001 #ifndef INTERPRETER_H
00002 #define INTERPRETER_H
00003
00004 #include "parser.h"
00005 class Interpreter {
00006     Parser parser;
00007
00008 public:
00009     Interpreter(std::string_view filename) : parser{Parser(filename)} {}
00010     void run();
00011 };
00012 #endif // INTERPRETER_H
```

6.4 lexer.h

```
00001 #ifndef LEXER_H
00002 #define LEXER_H
00003
00004 #include <iosfwd>
00005 #include <vector>
00006 #include <memory>
00007 #include <string>
00008 #include <string_view>
00009 #include <unordered_map>
00010
00011 #include "token.h"
00012
00013
00014 class Lexer {
00015     std::string content;
00016     std::size_t length;
00017     std::size_t index;
00018     std::size_t line;
00019     char current_char;
00020     Token previous;
00021
00026     const std::unordered_map<std::string, enum token_type> reserved;
00027
00028     enum comment_type {
00029         NONE,
00030         CPP_COMMENT,
00031         C_COMMENT,
00032         C_COMMENT_END
00033     };
00034
00035 public:
00036     Lexer(std::string_view filename) :
00037         content(read_file(filename)), length(content.length()), index(0ULL),
00038         line(0ULL), current_char(EOF), previous{"", 0, NO_SYMBOLS}, reserved{
00039                 {"!", NOT},  {"+", ADDITION}, {"-", SUBTRACTION},
00040                 {"&", AND}, {"*", MULTIPLICATION}, {"/", DIVISION}, {"<", LT}, {"=", EQ},
00041                 {":=", ASSIGN}, {"\"", STRING}, {"(", LPARENTHESES},
00042                 {")", RPARENTHESES},
00043                 {";", SEMICOLON}, {"..", RANGE},
00044
00045                 {"var", VAR},
00046                 {"for", FOR},
00047                 {"end", END},
00048                 {"in", IN},
00049                 {"do", DO},
00050                 {"read", READ},
00051                 {"print", PRINT},
```

```
00052                 {"int", INT},
00053                 {"string", STRING_TYPE},
00054                 {"bool", BOOL},
00055                 {"assert", ASSERT},
00056                 {"if", IF},
00057                 {"else", ELSE}
00058             }
00059     {}
00060
00068     std::unique_ptr<Token> get_token(bool consume = true);
00069
00076     std::unique_ptr<Token> peek_token(void);
00077
00083     bool is_reserved(std::string_view lexeme) {
00084         return reserved.find(lexeme.data()) != reserved.end();
00085     }
00086
00087 private:
00088     using identifier = int (*)(int);
00089
00097     std::string read_file(std::string_view filename);
00098
00108     std::string parse(identifier f);
00109
00113     Lexer::comment_type is_comment(void);
00114
00118     void skip_comment(Lexer::comment_type);
00119
00123     void skip_wspace(void);
00124
00128     std::string get_string(void);
00135     char interpret_escape(char c);
00136     std::string parse_octal();
00137     std::string parse_hex();
00138
00144     char get_char(void);
00145
00149     char peek_char(void);
00150
00156     enum token_type get_token_type(std::string current);
00157 };
00158
00159 #endif /* LEXER_H */
```

6.5 parser.h

```
00001 #ifndef PARSER_H
00002 #define PARSER_H
00003 #include "token.h"
00004 #include "lexer.h"
00005 #include "expr.h"
00006 #include "symbols.h"
00007 #include <memory>
00008 #include <string_view>
00009 #include <iostream>
00010
00011 class Parser {
00012     Lexer lexer;
00013 public:
00014     Parser(std::string_view filename) : lexer(filename) {}
00015     std::unique_ptr<StatementList> parse_file();
00016
00017 private:
00018     // grammar translators
00019
00020     /*
00021      * Parses a statement
00022      */
00023     std::unique_ptr<Expr> statement();
00024
00025     std::unique_ptr<StatementList> statement_list(bool is_block);
00026
00030     std::unique_ptr<Expr> var();
00031
00032     std::unique_ptr<Read> read_statement() {
00033         auto tok = match(token_type::IDENTIFIER);
00034
00035         std::unique_ptr<Read> r = std::make_unique<Read>(
00036                 std::make_unique<Var>(std::move(tok)));
00037         match(token_type::SEMICOLON);
00038         return r;
00039     }
00040
```

```
00041     std::unique_ptr<Print> print_statement() {
00042         std::unique_ptr<Print> r = std::make_unique<Print>(expression());
00043         match(token_type::SEMICOLON);
00044         return r;
00045     }
00046 
00047     std::unique_ptr<For> for_loop() {
00048         std::unique_ptr<Token> identifier = match(token_type::IDENTIFIER);
00049         match(token_type::IN);
00050 
00051         auto start = expression();
00052         match(token_type::RANGE);
00053         auto end = expression();
00054         std::unique_ptr<Range> range =
00055             std::make_unique<Range>(std::move(start), std::move(end));
00056 
00057         match(token_type::DO);
00058         std::unique_ptr<StatementList> body = statement_list(true);
00059         match(token_type::END);
00060         match(token_type::FOR);
00061         match(token_type::SEMICOLON);
00062         return std::make_unique<For>(std::move(identifier), std::move(range), std::move(body));;
00063     }
00064 
00065     std::unique_ptr<If> if_stmt() {
00066         std::unique_ptr<Operand> condition = expression();
00067         match(token_type::DO);
00068         std::unique_ptr<StatementList> list = statement_list(true);
00069         std::unique_ptr<StatementList> else_stmt = nullptr;
00070 
00071         auto token = lexer.peek_token();
00072         switch(token->type) {
00073             case token_type::ELSE:
00074                 lexer.get_token();
00075                 else_stmt = statement_list(true);
00076 
00077                 // fallthrough
00078             case token_type::END:
00079                 match(token_type::END); // expect an end if coming from else
00080                 match(token_type::IF);
00081                 match(token_type::SEMICOLON);
00082                 break;
00083 
00084             default:
00085                 break;
00086         }
00087 
00088         auto if_ = std::make_unique<If>(std::move(condition),
00089                 std::move(list), std::move(else_stmt));
00090         return if_;
00091     }
00092 
00096     std::unique_ptr<Operand> terminal();
00097 
00104     std::unique_ptr<Operand> term_tail(std::unique_ptr<Operand> expr);
00105 
00109     std::unique_ptr<Operand> factor();
00110 
00118     std::unique_ptr<Operand> factor_tail(std::unique_ptr<Operand> ident);
00119 
00123     std::unique_ptr<Operand> expression();
00124 
00131     std::unique_ptr<Token> match(token_type expected) {
00132         auto token = lexer.peek_token();
00133         if (token->type == expected) {
00134             return lexer.get_token();
00135         }
00136         std::cout « "Parse error: expected " « type_to_str(expected) «
00137             " got " « type_to_str(token->type) « "(" « token->token « ")\n";
00138         std::exit(1);
00139     }
00140 };
00141 #endif
```

6.6 symbols.h

```
00001 #ifndef SYMBOLS_H
00002 #define SYMBOLS_H
00003 
00004 #include <string>
00005 #include <unordered_map>
00006 #include <variant>
00007 #include <memory>
```

```
00008
00009 class Literal;
00010
00011 class SymbolTable {
00012     std::unordered_map<std::string, std::unique_ptr<Literal» symbols;
00013
00014 public:
00015     SymbolTable() : symbols{} {}
00016
00022     bool add_symbol(std::string_view symbol, std::unique_ptr<Literal> value);
00023
00029     bool add_symbol(std::string_view symbol, int type);
00030
00031     Literal *get_symbol(std::string_view symbol);
00032
00036     bool exists(std::string_view symbol);
00037
00038     bool set_value(std::string_view symbol, Literal *literal);
00039     bool set_value(std::string_view symbol, std::variant<int, std::string, bool> value);
00040 };
00041
00042 extern SymbolTable symbol_table;
00043
00044 #endif /* SYMBOLS_H */
```

6.7 token.h

```
00001 #ifndef TOKEN_H
00002 #define TOKEN_H
00003 #include <string>
00004
00005
00006
00007 enum token_type {
00008     IDENTIFIER,
00009
00010     // operators
00011     ADDITION,
00012     SUBTRACTION,
00013     MULTIPLICATION,
00014     DIVISION,
00015     LT,
00016     NOT,
00017     EQ,
00018     AND,
00019
00020     // symbols
00021     LPARENTHESES,
00022     RPARENTHESES,
00023     TYPE_DELIM,
00024     SEMICOLON,
00025     ASSIGN,
00026     RANGE,
00027
00028     // literals
00029     DIGIT,
00030     STRING,
00031
00032     // Keywords
00033     VAR,
00034     FOR,
00035     END,
00036     IN,
00037     DO,
00038     READ,
00039     PRINT,
00040     INT,
00041     STRING_TYPE,
00042     BOOL,
00043     ASSERT,
00044     IF,
00045     ELSE,
00046
00047     // errors (essientially)
00048     UNKNOWN,
00049     NO_SYMBOLS
00050 };
00051
00052 extern std::string type_to_str(token_type type);
00053
00054 struct Token {
00055     std::string token;
00056     std::size_t line;
```

```
00057     enum token_type type;
00058
00059     Token(std::string token, std::size_t line, enum token_type type) :
00060         token{std::move(token)}, line{line}, type{type} {}
00061     Token(const Token &tok) :
00062         token{tok.token}, line{tok.line}, type{tok.type} {}
00063     Token(const Token &&tok) :
00064         token{tok.token}, line{tok.line}, type{tok.type} {}
00065 };
00066
00067 #endif // TOKEN_H
```

Index