

Minipascal

0.1

Generated on Mon Mar 13 2023 00:37:22 for Minipascal by Doxygen 1.9.6

Mon Mar 13 2023 00:37:22



<b>1 Minipascal</b>	<b>1</b>
1.1 Building	1
1.2 Running	1
1.3 Brief introduction to the minipascal interpreter	1
1.4 Lexing	1
1.5 Parsing	1
1.6 Semantical analysis	2
1.7 Error handling	2
1.8 Testing	2
1.9 Hours	2
1.10 Required info	3
<b>2 Hierarchical Index</b>	<b>5</b>
2.1 Class Hierarchy	5
<b>3 Class Index</b>	<b>7</b>
3.1 Class List	7
<b>4 File Index</b>	<b>9</b>
4.1 File List	9
<b>5 Class Documentation</b>	<b>11</b>
5.1 Analyser Class Reference	11
5.1.1 Detailed Description	11
5.1.2 Member Function Documentation	11
5.1.2.1 analyse()	11
5.2 Bop Class Reference	12
5.2.1 Detailed Description	13
5.2.2 Constructor & Destructor Documentation	13
5.2.2.1 Bop()	13
5.2.3 Member Function Documentation	13
5.2.3.1 analyse()	14
5.2.3.2 get_type()	14
5.2.3.3 get_value()	14
5.2.3.4 interpet()	14
5.2.3.5 operator"!()	14
5.2.3.6 operator"&&()	15
5.2.3.7 operator*()	15
5.2.3.8 operator+()	15
5.2.3.9 operator-()	16
5.2.3.10 operator/()	16
5.2.3.11 operator<()	16
5.2.3.12 operator==()	17
5.2.3.13 truthy()	17

5.2.3.14 visit()	17
5.3 Expr Class Reference	17
5.3.1 Member Function Documentation	18
5.3.1.1 analyse()	18
5.3.1.2 interpet()	18
5.3.1.3 visit()	18
5.4 For Class Reference	19
5.4.1 Detailed Description	19
5.4.2 Member Function Documentation	19
5.4.2.1 analyse()	20
5.4.2.2 interpet()	20
5.4.2.3 visit()	20
5.5 If Class Reference	20
5.5.1 Detailed Description	21
5.5.2 Constructor & Destructor Documentation	21
5.5.2.1 If()	21
5.5.3 Member Function Documentation	21
5.5.3.1 analyse()	21
5.5.3.2 interpet()	22
5.5.3.3 visit()	22
5.6 Interpreter Class Reference	22
5.7 Lexer Class Reference	22
5.7.1 Member Function Documentation	22
5.7.1.1 get_token()	22
5.7.1.2 is_reserved()	23
5.7.1.3 peek_token()	23
5.8 Literal Class Reference	23
5.8.1 Detailed Description	25
5.8.2 Member Function Documentation	25
5.8.2.1 analyse()	25
5.8.2.2 get_type()	25
5.8.2.3 get_value()	25
5.8.2.4 interpet()	26
5.8.2.5 operator"!()	26
5.8.2.6 operator&&()	26
5.8.2.7 operator*()	26
5.8.2.8 operator+()	27
5.8.2.9 operator-()	27
5.8.2.10 operator/()	27
5.8.2.11 operator<()	28
5.8.2.12 operator==()	28
5.8.2.13 truthy()	28

5.8.2.14 visit()	28
5.9 Operand Class Reference	29
5.9.1 Detailed Description	29
5.9.2 Member Function Documentation	30
5.9.2.1 get_type()	30
5.9.2.2 get_value()	30
5.9.2.3 operator"!()	30
5.9.2.4 operator&&()	30
5.9.2.5 operator*()	31
5.9.2.6 operator+()	31
5.9.2.7 operator-()	31
5.9.2.8 operator/()	32
5.9.2.9 operator<()	32
5.9.2.10 operator==()	32
5.9.2.11 truthy()	33
5.10 Parser Class Reference	33
5.11 Print Class Reference	33
5.11.1 Detailed Description	34
5.11.2 Member Function Documentation	34
5.11.2.1 analyse()	34
5.11.2.2 interpet()	34
5.11.2.3 visit()	35
5.12 Range Class Reference	35
5.12.1 Detailed Description	36
5.12.2 Member Function Documentation	36
5.12.2.1 analyse()	36
5.12.2.2 get_next()	36
5.12.2.3 interpet()	36
5.12.2.4 is_done()	36
5.12.2.5 visit()	37
5.13 Read Class Reference	37
5.13.1 Detailed Description	37
5.13.2 Member Function Documentation	38
5.13.2.1 analyse()	38
5.13.2.2 interpet()	38
5.13.2.3 visit()	38
5.14 StatementList Class Reference	38
5.14.1 Detailed Description	39
5.14.2 Member Function Documentation	39
5.14.2.1 analyse()	39
5.14.2.2 interpet()	40
5.14.2.3 visit()	40

5.15 SymbolTable Class Reference	40
5.15.1 Member Function Documentation	40
5.15.1.1 add_symbol() [1/2]	40
5.15.1.2 add_symbol() [2/2]	41
5.15.1.3 exists()	41
5.16 Token Struct Reference	41
5.17 Unary Class Reference	42
5.17.1 Detailed Description	43
5.17.2 Member Function Documentation	43
5.17.2.1 analyse()	43
5.17.2.2 get_type()	43
5.17.2.3 get_value()	43
5.17.2.4 interpet()	44
5.17.2.5 operator"!()	44
5.17.2.6 operator&&()	44
5.17.2.7 operator*()	44
5.17.2.8 operator+()	45
5.17.2.9 operator-()	45
5.17.2.10 operator/()	45
5.17.2.11 operator<()	46
5.17.2.12 operator==()	46
5.17.2.13 truthy()	46
5.17.2.14 visit()	46
5.18 Var Class Reference	47
5.18.1 Detailed Description	48
5.18.2 Member Function Documentation	48
5.18.2.1 analyse()	48
5.18.2.2 get_type()	48
5.18.2.3 get_value()	49
5.18.2.4 interpet()	49
5.18.2.5 operator"!()	49
5.18.2.6 operator&&()	49
5.18.2.7 operator*()	50
5.18.2.8 operator+()	50
5.18.2.9 operator-()	50
5.18.2.10 operator/()	51
5.18.2.11 operator<()	51
5.18.2.12 operator==()	51
5.18.2.13 truthy()	52
5.18.2.14 visit()	52
5.19 VarInst Class Reference	52
5.19.1 Detailed Description	54

5.19.2 Member Function Documentation	54
5.19.2.1 analyse()	54
5.19.2.2 get_type()	54
5.19.2.3 interpet()	54
5.19.2.4 operator"!()	54
5.19.2.5 operator&&()	55
5.19.2.6 operator*()	55
5.19.2.7 operator+()	55
5.19.2.8 operator-()	56
5.19.2.9 operator/()	56
5.19.2.10 operator<()	56
5.19.2.11 operator==(())	57
5.19.2.12 visit()	57
<b>6 File Documentation</b>	<b>59</b>
6.1 analysis.h	59
6.2 expr.h	59
6.3 interpreter.h	69
6.4 lexer.h	69
6.5 parser.h	70
6.6 symbols.h	71
6.7 token.h	72
<b>Index</b>	<b>75</b>





# 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`.

```
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 (LL1 where convenient, peek ahead a bit), it does not create a parse tree, instead creates an AST directly. You may view the structure of the AST in UML diagrams in relevant chapters below.

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 to make it 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. 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

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

Date	Hours
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:

Analyser . . . . .	11
Expr . . . . .	17
For . . . . .	19
If . . . . .	20
Operand . . . . .	29
Bop . . . . .	12
Literal . . . . .	23
Unary . . . . .	42
Var . . . . .	47
VarInst . . . . .	52
Print . . . . .	33
Range . . . . .	35
Read . . . . .	37
StatementList . . . . .	38
Interpreter . . . . .	22
Lexer . . . . .	22
Parser . . . . .	33
SymbolTable . . . . .	40
Token . . . . .	41



## Chapter 3

# Class Index

### 3.1 Class List

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

<a href="#">Analyser</a>	11
<a href="#">Bop</a>	12
<a href="#">Expr</a>	17
<a href="#">For</a>	19
<a href="#">If</a>	20
<a href="#">Interpreter</a>	22
<a href="#">Lexer</a>	22
<a href="#">Literal</a>	23
<a href="#">Operand</a>	29
<a href="#">Parser</a>	33
<a href="#">Print</a>	33
<a href="#">Range</a>	35
<a href="#">Read</a>	37
<a href="#">StatementList</a>	38
<a href="#">SymbolTable</a>	40
<a href="#">Token</a>	41
<a href="#">Unary</a>	42
<a href="#">Var</a>	47
<a href="#">VarInst</a>	52





## Chapter 4

# File Index

### 4.1 File List

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

<a href="#">analysis.h</a>	..	??
<a href="#">expr.h</a>	..	??
<a href="#">interpreter.h</a>	..	??
<a href="#">lexer.h</a>	..	??
<a href="#">parser.h</a>	..	??
<a href="#">symbols.h</a>	..	??
<a href="#">token.h</a>	..	??



## 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.

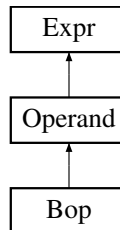
The documentation for this class was generated from the following files:

- 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 [interpret](#) (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 **interpret** (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**

<i>tok</i>	- which type of operation it is.
<i>left</i>	- left side of the operation.
<i>right</i>	- 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

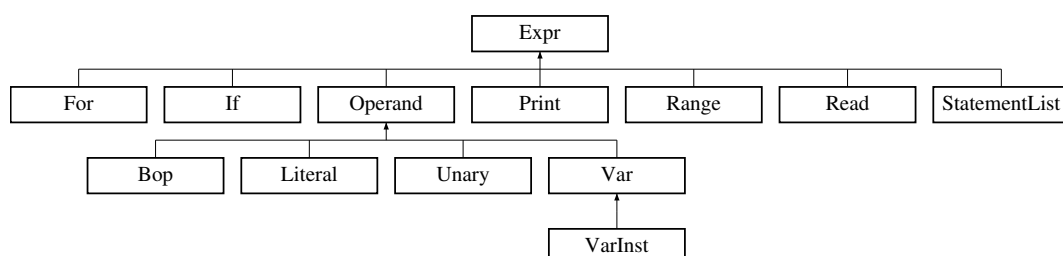
Implements [Expr](#).

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 [interpret](#) (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 interpret()

```
virtual void Expr::interpret (  
    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](#).

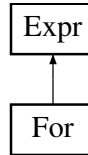
The documentation for this class was generated from the following file:

- [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 [interpret](#) () override
- void [visit](#) (void) const override

### Public Member Functions inherited from [Expr](#)

- **Expr** (std::unique\_ptr< [Token](#) > t)
- **Expr** (const [Token](#) &t)
- virtual void [interpret](#) (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](#).

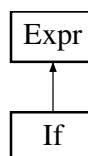
The documentation for this class was generated from the following file:

- [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 **interpret** (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**

<i>condition</i>	- Expression to be checked if truthy, if true it will execute the <code>truthy</code> tree, else the <code>falsy</code> .
<i>truthy</i>	- tree to evaluate if the condition is true.
<i>falsy</i>	- 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 `interpret()`

```
void If::interpret (
    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**

<i>consume</i>	- 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**

<i>lexeme</i>	- 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.

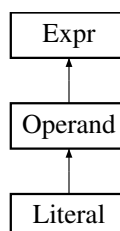
The documentation for this class was generated from the following files:

- 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 [interpret](#) (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 [interpret](#) (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](#).

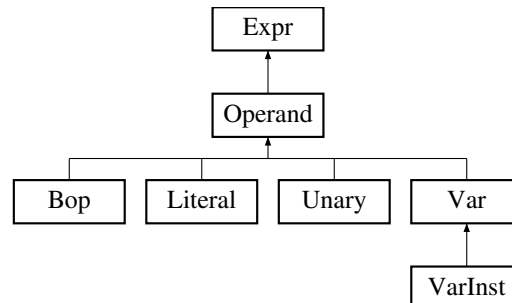
The documentation for this class was generated from the following file:

- [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 [interpret](#) (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 ()`

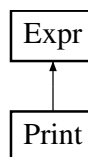
The documentation for this class was generated from the following files:

- `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 [interpret](#) (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 [interpret](#) (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 interpret()

```
void Print::interpret (
    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](#).

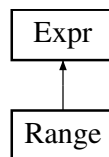
The documentation for this class was generated from the following file:

- [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 [interpret](#) () override
- void [visit](#) () const override

### Public Member Functions inherited from [Expr](#)

- **Expr** (std::unique\_ptr< [Token](#) > t)
- **Expr** (const [Token](#) &t)
- virtual void [interpret](#) (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 [interpret](#) (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 [interpret](#) (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](#).

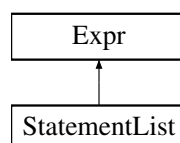
The documentation for this class was generated from the following file:

- [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 [interpret](#) (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 [interpret](#) (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 `interpret()`

```
void StatementList::interpret (
    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**

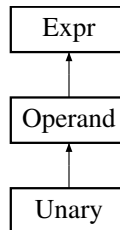
The documentation for this struct was generated from the following file:

- 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 [interpret](#) (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 [interpret](#) (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 `interpert()`

```
virtual void Unary::interpert (
    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](#).

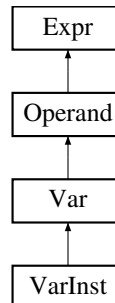
The documentation for this class was generated from the following file:

- `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 [interp](#) (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 [interpret](#) (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 `interpert()`

```
virtual void Var::interpert (
    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](#).

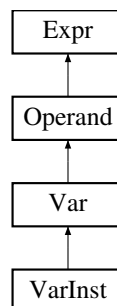
The documentation for this class was generated from the following file:

- `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 [interpret](#) () 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 [interpret](#) (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 [interpret](#) (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) +
00243             std::get<std::string>(l.get_value()->value)};
00244     }
00245     std::cout << "Error invalid types in + operation";
00246     std::exit(1);
00247 }
00248
00249 Literal operator=(Literal &l) {
00250     if (this->value.index() == l.value.index()) {
00251         this->value = l.value;
00252         return *this;
00253     }
00254     std::cout << "Conflicting types in assignment";
00255     std::exit(1);
00256 }
00257
00258 Literal operator=(std::variant<int, std::string, bool> value) {
00259     if (this->value.index() == value.index()) {
00260         this->value = value;
00261         return *this;
00262     }
00263     std::cout << "Conflicting types in assignment";
00264     std::exit(1);
00265 }
00266
00267 Literal operator-(Operand &l) override {
00268     if (std::holds_alternative<int>(this->value) &&
00269         std::holds_alternative<int>(l.get_value()->value)) {
00270         return Literal{std::get<int>(this->value) - std::get<int>(l.get_value()->value)};
00271     }
00272     std::cout << "Error invalid types in - operation";
00273     std::exit(1);
00274 }
00275
00276 Literal operator*(Operand &l) override {
00277     if (std::holds_alternative<int>(this->value) &&
00278         std::holds_alternative<int>(l.get_value()->value)) {
00279         return Literal{std::get<int>(this->value) * std::get<int>(l.get_value()->value)};
00280     }
00281     std::cout << "Error invalid types in - operation";
00282     std::exit(1);
00283 }
00284
00285 Literal operator/(Operand &l) override {
00286     if (std::holds_alternative<int>(this->value) &&
00287         std::holds_alternative<int>(l.get_value()->value)) {
00288         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
00346 class Var : public Operand {
00347 public:
00348     Var(std::unique_ptr<Token> tok) :
00349         Operand(std::move(tok)) {}
00350     Var(const Var &var) :
00351         Operand(*var.token) {}
00352
00353     bool analyse() const override {
00354         if (!symbol_table.exists(token->token)) {
00355             std::cout << "Error no variable named " << token->token << " line: " << token->line <<
"\n";
00356             return true;
00357         }
00358         return false;
00359     }
00360
00361     virtual void interpet(void) override {}
00362     virtual void visit(void) const override {
00363         std::cout << token->token << " ";
00364     }
00365
00366     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
00432 class VarInst : public Var {
00433     int type;
00434
00435     public:
00436     VarInst(std::unique_ptr<Token> tok, enum token_type type) :
00437         Var(std::move(tok)), type{type - token_type::INT} {}
00438
00439     VarInst(std::unique_ptr<Token> tok, std::unique_ptr<Token> type) :
00440         Var(std::move(tok)), type{type->type - token_type::INT} {}
00441
00442     int get_type() override {
00443         return symbol_table.get_symbol(token->token)->get_type();
00444     }
00445
00446     bool analyse() const override {
00447         bool succeeded = symbol_table.add_symbol(token->token, type);
00448
00449         if(!succeeded) {
00450             std::cout << "Error token variable " << token->token
00451                 << " already initialized";
00452             return true;
00453         }
00454         return false;
00455     }
00456
00457     // 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
00525
00526     bool analyse() const override {
00527         bool has_error = left->analyse();
00528         has_error |= right->analyse();
00529
00530         if(has_error) {
00531             std::cout << "Semantical error in bop\n";
00532         }
00533
00534         return has_error;
00535     }
00536
00537     virtual Literal *get_value() override {
00542         if (evaluated) {
00543             // use cached value
00544             return evaluated.get();
00545         }
00546
00547         switch(token->type) {
00548             case token_type::ADDITION:
00549                 evaluated = std::make_unique<Literal>(*left + *right->get_value());
00550                 break;
00551
00552             case token_type::SUBTRACTION:
00553                 evaluated = std::make_unique<Literal>(*left - *right->get_value());
00554                 break;
00555
00556             case token_type::MULTIPLICATION:
00557                 evaluated = std::make_unique<Literal>(*left * *right->get_value());
00558                 break;
00559

```



```

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

```

```

00647         std::exit(1);
00648     }
00649     virtual Literal operator!() override {
00650         std::cout << "Invalid operation\n";
00651         std::exit(1);
00652     }
00653 protected:
00654     bool is_boolean() {
00655         return token->type == token_type::LT || token->type == token_type::EQ;
00656     }
00657 };
00658
00659
00660
00661 class If : public Expr {
00662     std::unique_ptr<Operand> condition;
00663     std::unique_ptr<StatementList> truthy;
00664     std::unique_ptr<StatementList> falsy;
00665
00666 public:
00667     If(std::unique_ptr<Operand> condition, std::unique_ptr<StatementList> truthy,
00668         std::unique_ptr<StatementList> falsy) :
00669         Expr(), condition{std::move(condition)}, truthy{std::move(truthy)},
00670         falsy{std::move(falsy)} {}
00671
00672     bool analyse() const override {
00673         bool error = condition->analyse() || truthy->analyse();
00674         if (falsy != nullptr) {
00675             error |= falsy->analyse();
00676         }
00677         return error;
00678     }
00679
00680     void interpret() override {
00681         if(condition->truthy()) {
00682             truthy->interpret();
00683         } else {
00684             if (falsy != nullptr) {
00685                 falsy->interpret();
00686             }
00687         }
00688     }
00689
00690     void visit() const override {
00691         std::cout << "( IF ";
00692         condition->visit();
00693         std::cout << " ";
00694         truthy->visit();
00695         std::cout << " )";
00696         if (falsy) {
00697             std::cout << " ELSE ";
00698             falsy->visit();
00699             std::cout << " )";
00700         }
00701         std::cout << " ";
00702     }
00703 };
00704
00705
00706 class Range : public Expr {
00707     std::unique_ptr<Operand> start;
00708     std::unique_ptr<Operand> end;
00709     int current;
00710
00711 public:
00712     Range(std::unique_ptr<Operand> start, std::unique_ptr<Operand> end) :
00713         Expr(), start{std::move(start)}, end{std::move(end)}, current{0} {}
00714
00715     int get_next() {
00716         return current++;
00717     }
00718
00719     bool is_done() {
00720         return !(current <= (std::get<int>(end->get_value()->value)));
00721     }
00722
00723     bool analyse() const override {
00724         bool error = start->get_type() || end->get_type();
00725         if (error) {
00726             std::cout << "Error range contains non integers\n";
00727             return true;
00728         }
00729         return false;
00730     }
00731
00732     void interpret() override {
00733         auto val = std::get_if<int>(&this->start->get_value()->value);
00734         if (val) {

```

```

00760         current = *val;
00761     }
00762 }
00763 void visit() const override {
00764     start->visit();
00765     std::cout << "...";
00766     end->visit();
00767 }
00768 };
00769
00773 class For : public Expr {
00774     std::unique_ptr<Var> var;
00775     std::unique_ptr<Range> range;
00776     std::unique_ptr<StatementList> loop;
00777
00778 public:
00779     For(std::unique_ptr<Token> variable, std::unique_ptr<Range> range,
00780         std::unique_ptr<StatementList> statement_list) : Expr(),
00781         var{std::make_unique<Var>(std::move(variable))},
00782         range{std::move(range)}, loop{std::move(statement_list)} {}
00783
00784     bool analyse() const override {
00785         return var->analyse() || loop->analyse() || range->analyse();
00786     }
00787
00788     void interpet() override {
00789         range->interpet();
00790         while(!range->is_done()) {
00791             var->set_value(range->get_next());
00792             loop->interpet();
00793         }
00794     }
00795
00796     void visit(void) const override {
00797         std::cout << "(FOR ";
00798         var->visit();
00799         range->visit();
00800         std::cout << " ( ";
00801         loop->visit();
00802         std::cout << ")";
00803     }
00804 };
00808 class Print : public Expr {
00809     std::unique_ptr<Operand> expr;
00810
00811 public:
00812     Print(std::unique_ptr<Operand> expr) : Expr(), expr{std::move(expr)} {}
00813
00814     void interpet(void) override {
00815         std::visit([](const auto &x) { std::cout << x << "\n"; }, expr->get_value()->value);
00816     }
00817
00818     void visit(void) const override {
00819         std::cout << "( PRINT ";
00820         expr->visit();
00821         std::cout << ") ";
00822     }
00823
00824     bool analyse() const override { return expr->analyse(); }
00825 };
00828 class Read : public Expr {
00829     std::unique_ptr<Var> var;
00830
00831 public:
00832     Read(std::unique_ptr<Var> op) : Expr(), var{std::move(op)} {}
00833
00834     void interpet(void) override {
00835         switch(var->get_type()) {
00836             case 0:
00837                 {
00838                     int val;
00839                     std::cin >> val;
00840                     var->set_value(val);
00841                 }
00842                 break;
00843             case 1:
00844                 {
00845                     std::string str;
00846                     std::cin >> str;
00847                     var->set_value(str);
00848                 }
00849                 break;
00850             case 2:
00851                 {
00852                     bool b;
00853                     std::cin >> b;
00854                 }

```

```

00855         var->set_value(b);
00856     }
00857     break;
00858 }
00859 if (!std::cin) {
00860     std::cout << "Error: input was of the wrong type, exiting...\n";
00861     std::exit(1);
00862 }
00863 }
00864
00865 void visit(void) const override {
00866     std::cout << "( READ ";
00867     var->visit();
00868     std::cout << ") ";
00869 }
00870
00871 bool analyse() const override {
00872     return var->analyse();
00873 }
00874 };
00875
00880 class Unary : public Operand {
00881     std::unique_ptr<Operand> op;
00882     std::unique_ptr<Literal> evaluated;
00883
00884 public:
00885     Unary(std::unique_ptr<Operand> op) : Operand(), op {std::move(op)} {}
00886
00887     virtual Literal *get_value() override {
00888         auto val = op->get_value();
00889         evaluated = std::make_unique<Literal>(!*val);
00890         return evaluated.get();
00891     }
00892
00893     bool truthy() override {
00894         return !op->truthy();
00895     }
00896
00897     virtual int get_type() override {
00898         return 2;
00899     }
00900
00901     virtual void interpret(void) override {
00902     }
00903
00904     virtual void visit(void) const override {
00905         std::cout << "( ! ";
00906         op->visit();
00907         std::cout << ") ";
00908     }
00909
00910     virtual bool analyse() const override {
00911         return op->analyse();
00912     }
00913
00914     Literal operator+(Operand &) override {
00915         std::cout << "Invalid operation\n";
00916         std::exit(1);
00917     }
00918
00919     Literal operator-(Operand &) override {
00920         std::cout << "Invalid operation\n";
00921         std::exit(1);
00922     }
00923
00924     Literal operator*(Operand &) override {
00925         std::cout << "Invalid operation\n";
00926         std::exit(1);
00927     }
00928
00929     Literal operator/(Operand &) override {
00930         std::cout << "Invalid operation\n";
00931         std::exit(1);
00932     }
00933
00934     Literal operator&&(Operand &) override {
00935         std::cout << "Invalid operation\n";
00936         std::exit(1);
00937     }
00938
00939     Literal operator==(Operand &) override {
00940         std::cout << "Invalid operation\n";
00941         std::exit(1);
00942     }
00943
00944     Literal operator<(Operand &) override {
00945

```

```

00946         std::cout << "Invalid operation\n";
00947         std::exit(1);
00948     }
00949
00950     Literal operator!() override {
00951         return !*this->op;
00952     }
00953 };
00954 #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
00022     const std::unordered_map<std::string, enum token_type> reserved;
00023
00024     enum comment_type {
00025         NONE,
00026         CPP_COMMENT,
00027         C_COMMENT,
00028         C_COMMENT_END
00029     };
00030
00031 public:
00032     Lexer(std::string_view filename) :
00033         content(read_file(filename)), length(content.length()), index(0ULL),
00034         line(0ULL), current_char(EOF), previous("", 0, NO_SYMBOLS), reserved{
00035             {"!", NOT}, {"+", ADDITION}, {"-", SUBTRACTION},
00036             {"%", AND}, {"*", MULTIPLICATION}, {"/", DIVISION}, {"<", LT}, {"=", EQ},
00037             {":=", ASSIGN}, {"\""}, STRING}, {"(", LPARENTHESES},
00038             {")", RPARENTHESES},
00039             {";", SEMICOLON}, {"..", RANGE},
00040
00041             {"var", VAR},
00042             {"for", FOR},
00043             {"end", END},
00044             {"in", IN},
00045             {"do", DO},
00046             {"read", READ},
00047             {"print", PRINT},
00048             {"int", INT},
00049             {"string", STRING_TYPE},
00050             {"bool", BOOL},
00051             {"assert", ASSERT},
00052             {"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
00093     std::unique_ptr<Operand> terminal();
00094
00095     std::unique_ptr<Operand> term_tail(std::unique_ptr<Operand> expr);
00096
00097     std::unique_ptr<Operand> factor();
00098
00099     std::unique_ptr<Operand> factor_tail(std::unique_ptr<Operand> ident);
00100
00101     std::unique_ptr<Operand> expression();
00102
00103     std::unique_ptr<Token> match(token_type expected) {
00104         auto token = lexer.peek_token();
00105         if (token->type == expected) {
00106             return lexer.get_token();
00107         }
00108         std::cout << "Parse error: expected " << type_to_str(expected) <<
00109             " got " << type_to_str(token->type) << "(" << token->token << ")\n";
00110         std::exit(1);
00111     }
00112 };
00113 #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     LPARENTHESSES,
00022     RPARENTHESSES,
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 (essentially)
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

- add\_symbol
  - SymbolTable, [40](#)
- analyse
  - Analyser, [11](#)
  - Bop, [13](#)
  - Expr, [18](#)
  - For, [19](#)
  - If, [21](#)
  - Literal, [25](#)
  - Print, [34](#)
  - Range, [36](#)
  - Read, [38](#)
  - StatementList, [39](#)
  - Unary, [43](#)
  - Var, [48](#)
  - VarInst, [54](#)
- Analysers, [11](#)
  - analyse, [11](#)
- Bop, [12](#)
  - analyse, [13](#)
  - Bop, [13](#)
  - get\_type, [14](#)
  - get\_value, [14](#)
  - interpret, [14](#)
  - operator!, [14](#)
  - operator<, [16](#)
  - operator\*, [15](#)
  - operator+, [15](#)
  - operator-, [15](#)
  - operator/, [16](#)
  - operator==, [16](#)
  - operator&&, [14](#)
  - truthy, [17](#)
  - visit, [17](#)
- exists
  - SymbolTable, [41](#)
- Expr, [17](#)
  - analyse, [18](#)
  - interpret, [18](#)
  - visit, [18](#)
- For, [19](#)
  - analyse, [19](#)
  - interpret, [20](#)
  - visit, [20](#)
- get\_next
  - Range, [36](#)
- get\_token
  - Lexer, [22](#)
- get\_type
  - Bop, [14](#)
  - Literal, [25](#)
  - Operand, [30](#)
  - Unary, [43](#)
  - Var, [48](#)
  - VarInst, [54](#)
- get\_value
  - Bop, [14](#)
  - Literal, [25](#)
  - Operand, [30](#)
  - Unary, [43](#)
  - Var, [48](#)
- If, [20](#)
  - analyse, [21](#)
  - If, [21](#)
  - interpret, [21](#)
  - visit, [22](#)
- interpret
  - Bop, [14](#)
  - Expr, [18](#)
  - For, [20](#)
  - If, [21](#)
  - Literal, [25](#)
  - Print, [34](#)
  - Range, [36](#)
  - Read, [38](#)
  - StatementList, [39](#)
  - Unary, [43](#)
  - Var, [49](#)
  - VarInst, [54](#)
- Interpreter, [22](#)
- is\_done
  - Range, [36](#)
- is\_reserved
  - Lexer, [23](#)
- Lexer, [22](#)
  - get\_token, [22](#)
  - is\_reserved, [23](#)
  - peek\_token, [23](#)
- Literal, [23](#)
  - analyse, [25](#)
  - get\_type, [25](#)
  - get\_value, [25](#)
  - interpret, [25](#)
  - operator!, [26](#)

- operator<, 27
- operator\*, 26
- operator+, 26
- operator-, 27
- operator/, 27
- operator==, 28
- operator&&, 26
- truthy, 28
- visit, 28
- Operand, 29
  - get\_type, 30
  - get\_value, 30
  - operator!, 30
  - operator<, 32
  - operator\*, 30
  - operator+, 31
  - operator-, 31
  - operator/, 31
  - operator==, 32
  - operator&&, 30
  - truthy, 32
- operator!
  - Bop, 14
  - Literal, 26
  - Operand, 30
  - Unary, 44
  - Var, 49
  - VarInst, 54
- operator<
  - Bop, 16
  - Literal, 27
  - Operand, 32
  - Unary, 45
  - Var, 51
  - VarInst, 56
- operator\*
  - Bop, 15
  - Literal, 26
  - Operand, 30
  - Unary, 44
  - Var, 49
  - VarInst, 55
- operator+
  - Bop, 15
  - Literal, 26
  - Operand, 31
  - Unary, 44
  - Var, 50
  - VarInst, 55
- operator-
  - Bop, 15
  - Literal, 27
  - Operand, 31
  - Unary, 45
  - Var, 50
  - VarInst, 55
- operator/
  - Bop, 16
  - Literal, 27
  - Operand, 31
  - Unary, 45
  - Var, 50
  - VarInst, 56
- operator==
  - Bop, 16
  - Literal, 28
  - Operand, 32
  - Unary, 46
  - Var, 51
  - VarInst, 56
- operator&&
  - Bop, 14
  - Literal, 26
  - Operand, 30
  - Unary, 44
  - Var, 49
  - VarInst, 54
- Parser, 33
- peek\_token
  - Lexer, 23
- Print, 33
  - analyse, 34
  - interpret, 34
  - visit, 34
- Range, 35
  - analyse, 36
  - get\_next, 36
  - interpret, 36
  - is\_done, 36
  - visit, 36
- Read, 37
  - analyse, 38
  - interpret, 38
  - visit, 38
- StatementList, 38
  - analyse, 39
  - interpret, 39
  - visit, 40
- SymbolTable, 40
  - add\_symbol, 40
  - exists, 41
- Token, 41
- truthy
  - Bop, 17
  - Literal, 28
  - Operand, 32
  - Unary, 46
  - Var, 51
- Unary, 42
  - analyse, 43
  - get\_type, 43
  - get\_value, 43

- interpet, [43](#)
  - operator!, [44](#)
  - operator<, [45](#)
  - operator\*, [44](#)
  - operator+, [44](#)
  - operator-, [45](#)
  - operator/, [45](#)
  - operator==, [46](#)
  - operator&&, [44](#)
  - truthy, [46](#)
  - visit, [46](#)
- Var, [47](#)
- analyse, [48](#)
  - get\_type, [48](#)
  - get\_value, [48](#)
  - interpet, [49](#)
  - operator!, [49](#)
  - operator<, [51](#)
  - operator\*, [49](#)
  - operator+, [50](#)
  - operator-, [50](#)
  - operator/, [50](#)
  - operator==, [51](#)
  - operator&&, [49](#)
  - truthy, [51](#)
  - visit, [52](#)
- VarInst, [52](#)
- analyse, [54](#)
  - get\_type, [54](#)
  - interpet, [54](#)
  - operator!, [54](#)
  - operator<, [56](#)
  - operator\*, [55](#)
  - operator+, [55](#)
  - operator-, [55](#)
  - operator/, [56](#)
  - operator==, [56](#)
  - operator&&, [54](#)
  - visit, [57](#)
- visit
- Bop, [17](#)
  - Expr, [18](#)
  - For, [20](#)
  - If, [22](#)
  - Literal, [28](#)
  - Print, [34](#)
  - Range, [36](#)
  - Read, [38](#)
  - StatementList, [40](#)
  - Unary, [46](#)
  - Var, [52](#)
  - VarInst, [57](#)