
Compiler Theory and Practice

Coursework

Juan Scerri
123456A

April 9, 2024

A coursework submitted in fulfilment of study unit CPS2000.



Contents

| | |
|--------------------------------------|----|
| Contents | i |
| Listings | iv |
| Report | 1 |
| § 1 Lexical Analysis | 1 |
| 1.1 The DFSA | 1 |
| Micro-Syntax & Categories | 2 |
| 1.2 The Director & Builder | 7 |

| | | |
|-----|---|----|
| 1.3 | The Actual Lexer | 9 |
| 1.4 | Hooking up the Lexer to the Runner | 10 |
| § 2 | The Parser | 11 |
| 2.1 | Modified EBNF | 11 |
| | Improved Precedence | 14 |
| | Better Arrays | 14 |
| | Removing Eye-Candy | 17 |
| 2.2 | Parsing & The Abstract Syntax Tree (AST) | 17 |
| 2.3 | The Actual Parser | 18 |
| | Token Buffering | 19 |
| | Token Matching | 20 |
| | Error Handling/Synchronization | 23 |
| | AST Generator Methods | 26 |
| 2.4 | Pretty Printing | 28 |
| | Using Parser in the Runner | 28 |
| § 3 | Semantic Analysis | 31 |
| 3.1 | Phases & Design | 31 |
| 3.2 | The Environment Tree | 33 |
| 3.3 | Actual Semantic Analysis | 41 |
| | The Symbol Type | 41 |
| | Symbol Registration & Search | 42 |
| | Type Checking | 44 |
| | Return Statement Verification | 46 |
| § 4 | Code Generation | 49 |
| 4.1 | Passing on the Environment Tree | 49 |
| | The TypeVisitor Class | 50 |
| 4.2 | Reordering | 51 |
| 4.3 | Function Declarations | 52 |
| 4.4 | Scopes, Variable (or Formal Parameter) Declarations & Variable Accesses | 54 |
| 4.5 | Properly Closing Scopes in Function Declarations | 57 |
| 4.6 | Assignments | 58 |
| 4.7 | Expressions | 59 |
| | Type Specific VM Code | 59 |
| | Function Calls | 60 |
| 4.8 | Branching | 61 |
| § 5 | Arrays | 64 |
| 5.1 | Changes in Lexical Analysis | 64 |
| 5.2 | Changes in Parsing | 65 |
| 5.3 | Changes in Semantic Analysis | 65 |
| 5.4 | Changes in Code Generation | 68 |

| | | |
|-----|---|----|
| § 6 | Testing & Usage | 71 |
| 6.1 | Usage | 71 |
| 6.2 | The Lexer, Parser & Semantic Analyser | 71 |
| | Lexical Analysis | 71 |
| | Parsing | 72 |
| | Semantic Analysis | 73 |
| 6.3 | Code Generation | 74 |
| 6.4 | Video | 75 |
| § 7 | Limitations | 75 |

Listings

| | | |
|----|--|----|
| 1 | DFSA class declaration (lexer/DFSA.hpp). | 1 |
| 2 | Code representation of Figure 7 in the LexerDirector class (lexer/LexerDirector.cpp). | 7 |
| 3 | Registration of the hexadecimal (<i>H</i>) category checker (lexer/LexerDirector.cpp). | 8 |
| 4 | Construction of a Lexer (lexer/LexerBuilder.cpp). | 8 |
| 5 | The updateLocationState() Lexer method (lexer/Lexer.cpp). | 9 |
| 6 | Error handling mechanism in the nextToken() Lexer method (lexer/Lexer.cpp). | 9 |
| 7 | The Runner constructor passing mLexer into the Parser constructor (runner/Runner.cpp). | 10 |
| 8 | Mechanism for internally storing types within the compiler (parl/Core.hpp). | 15 |
| 9 | The Primitive class declaration (parl/Core.hpp). | 15 |
| 10 | An size unbounded type in C++ (parl/Core.hpp). | 16 |
| 11 | The FunctionCall AST node class (parl/AST.hpp). | 17 |
| 12 | The Binary AST node class (parl/AST.hpp). | 18 |
| 13 | The Unary AST node class (parl/AST.hpp). | 18 |
| 14 | The moveWindow() Parser methods (parser/Parser.cpp). | 19 |
| 15 | The nextToken() Parser methods (parser/Parser.cpp). | 20 |
| 16 | The consume() Parser method (parser/Parser.hpp). | 21 |
| 17 | Usage of the consume() method in the formalParam() generator method (parser/Parser.cpp). | 21 |
| 18 | The peek() Parser method (parser/Parser.cpp). | 22 |
| 19 | The abort functionality present in the codebase (parl/Core.hpp). | 22 |
| 20 | The SyncParser exception and the error() method which kick-starts the synchronisation process (parser/Parser.hpp). | 24 |
| 21 | The synchronize() method in the Parser class (parser/Parser.cpp). | 24 |
| 22 | The ifStmt() node generator method in the Parser class (parser/Parser.cpp). | 26 |

| | | |
|----|---|----|
| 23 | The main body of methods in the Parser class (parser/Parser.hpp). | 27 |
| 24 | The parse segment of the run() method in the Runner class (runner/Runner.cpp). | 28 |
| 25 | The debugParsing() method in the Runner class (runner/Runner.cpp). | 29 |
| 26 | A segment of the pure virtual Visitor class (parl/Visitor.hpp). | 30 |
| 27 | The visit(core::PadRead *) method in the PrinterVisitor (parser/PrinterVisitor.cpp). | 31 |
| 28 | The visit(Program *) method in the AnalysisVisitor class, showing the hack used for resolving function symbols before-hand (analysis/AnalysisVisitor.cpp). | 32 |
| 29 | The Environment class with mChildren highlighted (backend/Environment.hpp). | 36 |
| 30 | The pushEnv() method in the EnvStack class (analysis/EnvStack.cpp). | 38 |
| 31 | The popEnv() method in the EnvStack class (analysis/EnvStack.cpp). | 38 |
| 32 | The pushEnv() method in the RefStack class (ir_gen/RefStack.cpp). | 38 |
| 33 | The popEnv() method in the RefStack class (ir_gen/RefStack.cpp). | 39 |
| 34 | VariableSymbol and FunctionSymbol class declarations (backend/Symbol.hpp). | 41 |
| 35 | The visit(FormalParam *) method in the AnalysisVisitor class (analysis/AnalysisVisitor.cpp). | 42 |
| 36 | The visit(Variable *) method in the AnalysisVisitor class (analysis/AnalysisVisitor.cpp). | 44 |
| 37 | A part of the visit(Binary *) method in the AnalysisVisitor class (analysis/AnalysisVisitor.cpp). | 44 |
| 38 | Another part of the visit(Binary *) method in the AnalysisVisitor class (analysis/AnalysisVisitor.cpp). | 45 |
| 39 | Checking whether a return statement is inside a function declaration in the visit(ReturnStmt *) method in the AnalysisVisitor class (analysis/AnalysisVisitor.cpp). | 46 |
| 40 | Checking whether the return expression has the same type as the function return type in the visit(ReturnStmt *) method in the AnalysisVisitor class (analysis/AnalysisVisitor.cpp). | 46 |
| 41 | The visit(IfStmt *) method in the ReturnVisitor class (analysis/ReturnVisitor.cpp). | 48 |
| 42 | The visit(Block *) method in the ReturnVisitor class with the segment pruning statements highlighted (analysis/ReturnVisitor.cpp). | 48 |
| 43 | Getting the global environment from the AnalysisVisitor and passing it on for reordering and code generation (runner/Runner.cpp). | 49 |
| 44 | The visit(FunctionCall *) method in the TypeVisitor class (ir_gen/TypeVisitor.cpp). | 50 |

| | | |
|----|---|----|
| 45 | The <code>reorder()</code> method in the <code>ReorderVisitor</code> class (<code>preprocess/ReorderVisitor.cpp</code>). | 51 |
| 46 | The <code>visit(FunctionDecl *)</code> method in the <code>GenVisitor</code> class (<code>ir_gen/GenVisitor.cpp</code>). | 52 |
| 47 | The <code>visit(VariableDecl *)</code> method in the <code>VarDeclCountVisitor</code> class (<code>ir_gen/VarDeclCountVisitor.cpp</code>). | 53 |
| 48 | The <code>emit_line()</code> method in the <code>GenVisitor</code> class (<code>ir_gen/GenVisitor.cpp</code>). | 54 |
| 49 | The <code>visit(VariableDecl *)</code> method in the <code>GenVisitor</code> class (<code>ir_gen/GenVisitor.cpp</code>). | 54 |
| 50 | A segment of the <code>visit(Program *)</code> method in the <code>GenVisitor</code> class (<code>ir_gen/GenVisitor.cpp</code>). | 57 |
| 51 | The <code>visit(ReturnStmt *)</code> method in the <code>GenVisitor</code> class (<code>ir_gen/GenVisitor.cpp</code>). | 57 |
| 52 | The <code>computeLevel()</code> method in the <code>GenVisitor</code> class (<code>ir_gen/GenVisitor.cpp</code>). | 58 |
| 53 | A segment of the <code>visit(Binary *)</code> method in the <code>GenVisitor</code> class (<code>ir_gen/GenVisitor.cpp</code>). | 59 |
| 54 | The <code>visit(FunctionCall *)</code> method in the <code>GenVisitor</code> class (<code>ir_gen/GenVisitor.cpp</code>). | 60 |
| 55 | The <code>visit(WhileStmt *)</code> method in the <code>GenVisitor</code> class (<code>ir_gen/GenVisitor.cpp</code>). | 61 |
| 56 | The <code>visit(IfStmt *)</code> method in the <code>GenVisitor</code> class (<code>ir_gen/GenVisitor.cpp</code>). | 62 |
| 57 | Changes in the <code>LexerDirector</code> for supporting left '[' and right ']' brackets (<code>lexer/LexerDirector.cpp</code>). | 64 |
| 58 | A part of the <code>visit(Binary *)</code> method in the <code>AnalysisVisitor</code> class (<code>analysis/AnalysisVisitor.cpp</code>). | 65 |
| 59 | The <code>isViableCast()</code> method in the <code>AnalysisVisitor</code> class (<code>analysis/AnalysisVisitor.cpp</code>). | 66 |
| 60 | A part of the <code>visit(Variable *)</code> method in the <code>GenVisitor</code> class, specifically the hack mentioned above (<code>ir_gen/GenVisitor.cpp</code>). . . | 68 |



Report

1 | Lexical Analysis

The lexical analysis phase makes use of three components. A DFSA class, a generic table-driven lexer, and a lexer builder.

1.1 | The DFSA

The DFSA class is an almost-faithful implementation of the formal concept of a DFSA. Listing 1, outlines the behaviour of the DFSA class. Additionally, it contains a number of helper functions which facilitate getting the initial state and checking whether a state or a transition *category* is valid. The `getInitialState()` helper is present, because after construction the DFSA does not guarantee, that the initial state used by the user will be the same.

```
class Dfsa {
public:
    Dfsa(
        size_t noOfStates,
        size_t noOfCategories,
        std::vector<std::vector<int>> const&
            transitionTable,
        int initialState,
        std::unordered_set<int> const& finalStates
    );

    [[nodiscard]] int getInitialState() const;

    [[nodiscard]] bool isValidState(int state) const;
    [[nodiscard]] bool isValidCategory(int category) const;
```

```

[[nodiscard]] bool isFinalState(int state) const;

[[nodiscard]] int getTransition(
    int state,
    std::vector<int> const& categories
) const;

private:
    const size_t mNoOfStates;           // Q
    const size_t mNoOfCategories;       // Sigma
    const std::vector<std::vector<int>>
        mTransitionTable;               // delta
    const int mInitialState;             // q_0
    const std::unordered_set<int> mFinalStates; // F
};

```

Listing 1: DFSA class declaration (lexer/DFSA.hpp).

The only significant difference from a formal DFSA is the `getTransition()` method. In fact, it accepts a vector of transition *categories* instead of a single *category*.

This is because a symbol e.g. 'a' or '9', might be in multiple *categories*. For instance 'a' is considered to be both a letter (*L*) and a hexadecimal (*H*).

Micro-Syntax & Categories

The DFSA for accepting PArL's micro-syntax is built as follows.

Let \mathfrak{U} be the set of all possible characters under the system encoding (e.g. UTF-8).

They will use the following *categories*:

- $L := \{A, \dots, Z, a, \dots, z\}$
- $D := \{0, \dots, 9\}$
- $H := \{A, \dots, F, a, \dots, f\} \cup D$
- $S := \{\alpha \in \mathfrak{U}: \alpha \text{ is whitespace}\} \setminus \{\text{LF}\}$

NOTE

LF refers to line-feed or as it is more commonly known '\n' i.e. new-line.

Together these *categories* form our alphabet Σ :

$$\Sigma := L \cup D \cup S \cup \{., \#, _, (,), [,], \{, \}, *, /, +, -, <, >, =, !, ,, :, ;, \text{LF}\}$$

For improved readability the DFSA, which is usually a single drawing, has been split across multiple drawings. Hence, in each drawing initial state (0) refers to the **same** initial state (a DFSA has one and only one initial state).

Additionally, each final state is annotated with the token type it should produce.

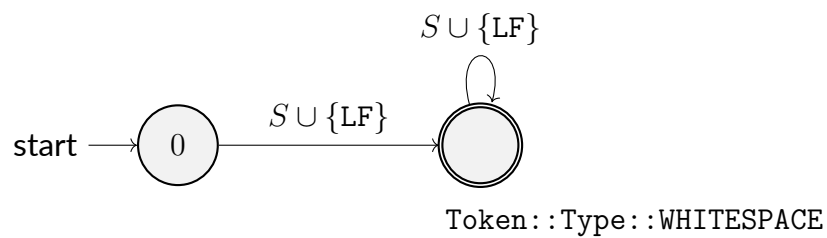


Figure 1: States & transitions for recognising whitespace.

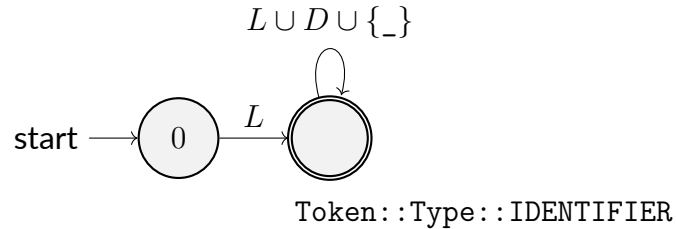


Figure 2: States & transitions for recognising identifiers/keywords.

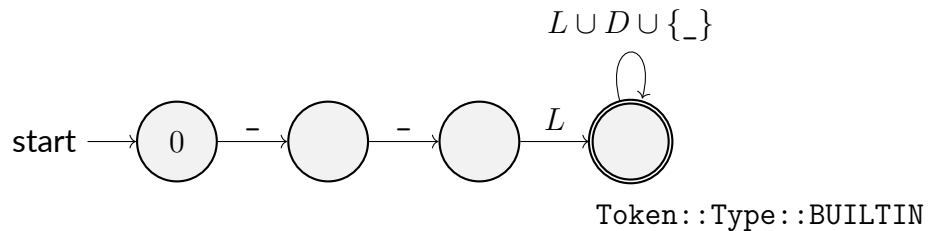


Figure 3: States & transitions for recognising builtins.

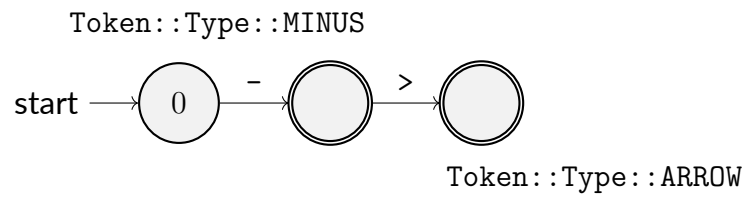


Figure 4: States & transitions for recognising 'minus' and 'arrow' (->).

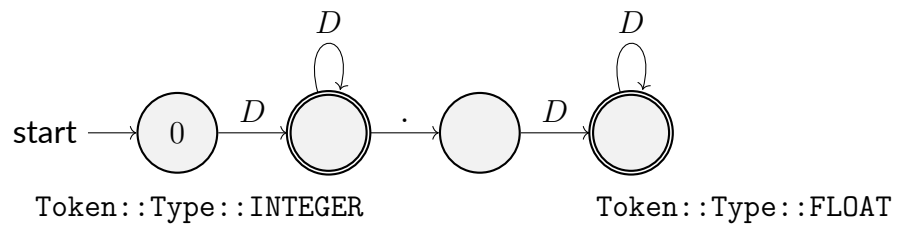


Figure 5: States & transitions for recognising integers and floats.

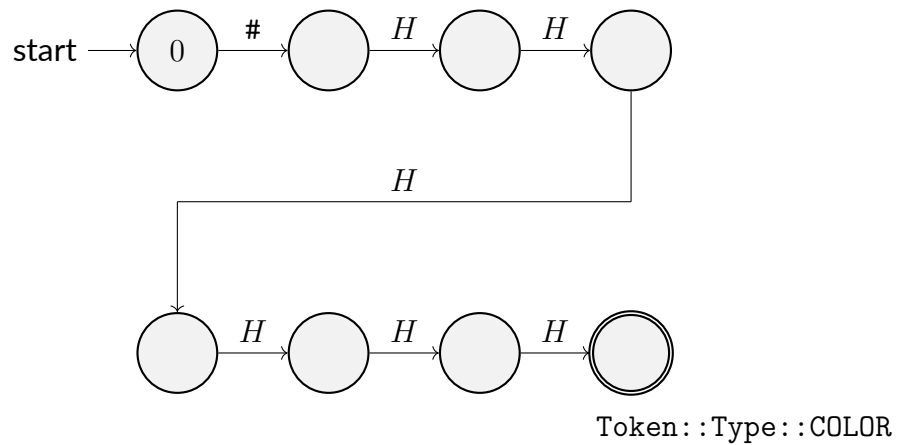


Figure 6: States & transitions for recognising colours.

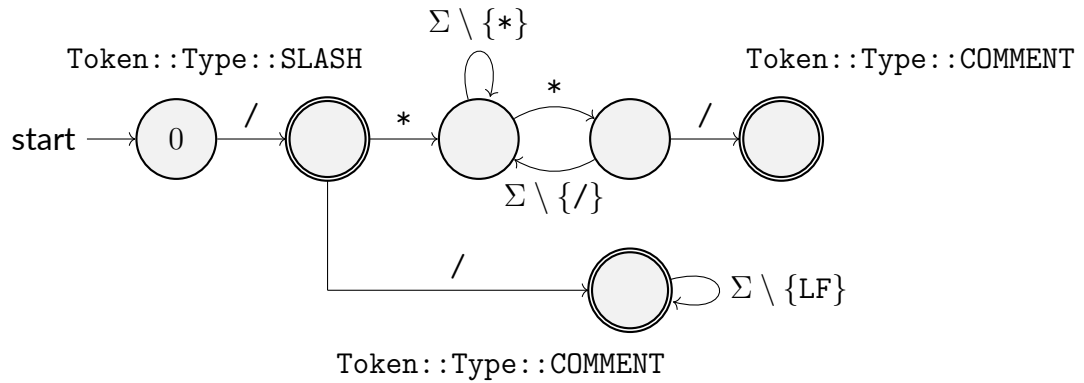


Figure 7: States & transitions for recognising 'division' and comments.

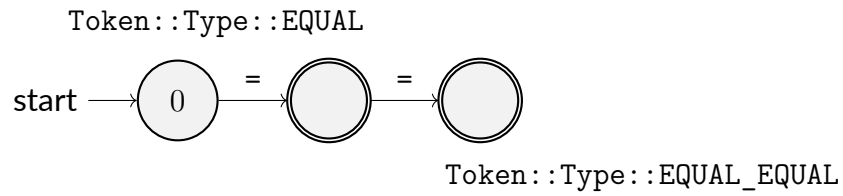


Figure 8: States & transitions for recognising 'assign' and 'is equal to'.

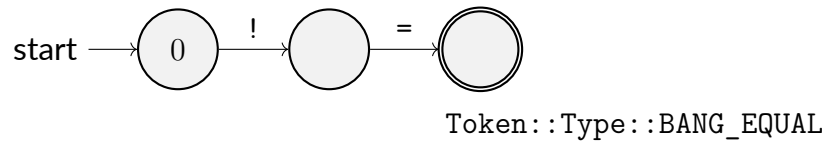


Figure 9: States & transitions for recognising 'not equal to'.

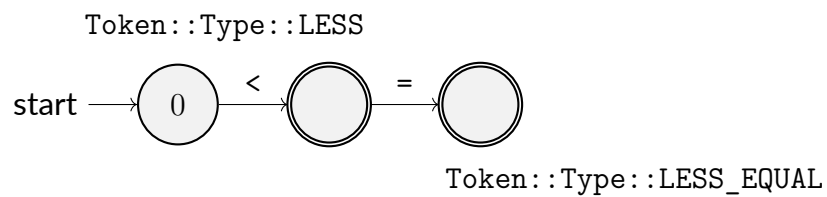


Figure 10: States & transitions for recognising 'less than' and 'less than or equal to'.

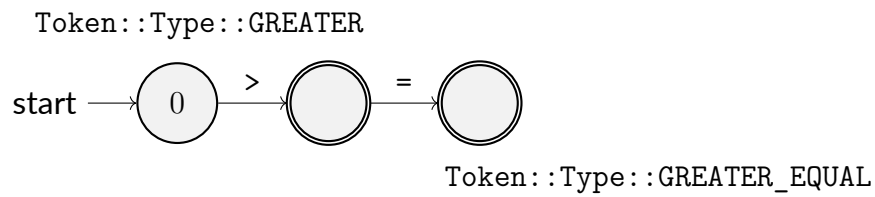


Figure 11: States & transitions for recognising 'greater than' and 'greater than or equal to'.

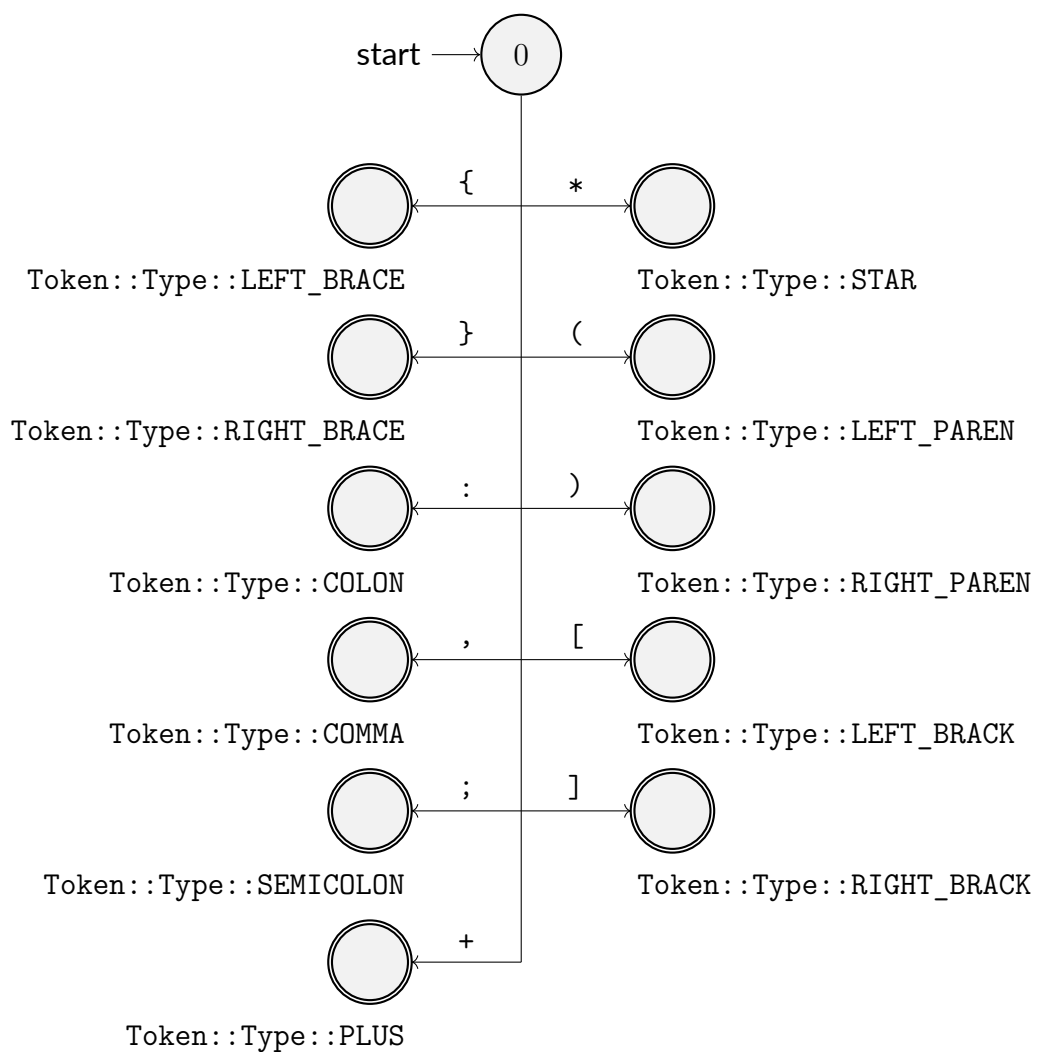


Figure 12: States & transitions for recognising other single letter tokens.

1.2 | The Director & Builder

Each drawing in 1.1 is directly represented in code within the `LexerDirector` using methods provided by the `LexerBuilder`.

```
// "/", "//", "/* ... */"
builder.addTransition(0, SLASH, 34)
    .setStateAsFinal(34, Token::Type::SLASH)
    .addTransition(34, SLASH, 35)
    .addComplementaryTransition(35, LINEFEED, 35)
    .setStateAsFinal(35, Token::Type::COMMENT)
    .addTransition(34, STAR, 36)
    .addComplementaryTransition(36, STAR, 36)
    .addTransition(36, STAR, 37)
    .addComplementaryTransition(37, SLASH, 36)
    .addTransition(37, SLASH, 38)
    .setStateAsFinal(38, Token::Type::COMMENT);
```

Listing 2: Code representation of Figure 7 in the `LexerDirector` class (`lexer/LexerDirector.cpp`).

The `LexerBuilder` keeps track of these transitions using less efficient data structures such as hash maps (`std::unordered_map`) and sets (`std::unordered_set`).

Then the `build()` method processes the user defined transitions and normalises everything into a single transition table for use in a DFSA. Additionally, it also produces two other artefacts. The first is called `categoryIndexToChecker`. It is a hash map from the index of a category to a lambda function which takes a character as input and returns true or false.

The lambdas and category indices are registered by the user (see Listing 3). Additionally, the category indices although they are integers, for readability they should be defined as a (classical) enumeration.

```
.addCategory(  
    HEX,  
    [](char c) -> bool {  
        return ('0' <= c && c <= '9') ||  
               ('A' <= c && c <= 'F') ||  
               ('a' <= c && c <= 'f');  
    }  
)
```

Listing 3: Registration of the hexadecimal (*H*) category checker (lexer/LexerDirector.cpp).

The second artefact produced by the builder is also a hash map whose keys are final state and values token types.

The transition table is then passed into a DFSA, and the DFSA, along with the two other artefacts, is passed into a Lexer.

```
// create dfsa  
Dfsa dfsa(  
    noOfStates,  
    noOfCategories,  
    transitionTable,  
    initialStateIndex,  
    finalStateIndices  
);  
  
// create lexer  
Lexer lexer(  
    std::move(dfsa),  
    std::move(categoryIndexToChecker),  
    std::move(finalStateIndexToTokenType)  
);
```

Listing 4: Construction of a Lexer (lexer/LexerBuilder.cpp).

1.3 | The Actual Lexer

A Lexer's core, as described in class, is the method `simulateDFSA()`.

NOTE

The Lexer class has a number of very important auxiliary methods and behavioural changes. Specifically, the `updateLocationState()`, see Listing 5, is critical for providing adequate error messages both during lexing and later stages. This function is called every time a lexeme is consumed allowing the Lexer to keep track of where it is in the file (in terms of lines and columns).

```
void Lexer::updateLocationState(std::string const& lexeme) {
    for (char ch : lexeme) {
        mCursor++;

        if (ch == '\n') {
            mLine++;

            mColumn = 1;
        } else {
            mColumn++;
        }
    }
}
```

Listing 5: The `updateLocationState()` Lexer method (lexer/Lexer.cpp).

If an invalid/non-accepting state is reached the invalid lexeme is consumed and the user is warned (see Listing 6). After such an encounter the `lexer`, is still in an operational state. Hence, the `nextToken()` method can be used again.

This is critical to provide users of the PArL compiler with a list of as many errors as possible. Such behaviour is desired because having to constantly run the PArL compiler to see the next error, would be a bad user experience.

```
if (state == INVALID_STATE) {
    mHasError = true;

    fmt::println(
        stderr,
        "lexical error at {}:{}:: unexpected "
        "lexeme '{}'",
        mLine,
```

```
        mColumn,
        lexeme
    );
} else {
    try {
        token = createToken(
            lexeme,
            mFinalStateToTokenType.at(state)
        );
    } catch (UndefinedBuiltin& error) {
        mHasError = true;

        fmt::println(
            stderr,
            "lexical error at {}:{}:: {}",
            mLine,
            mColumn,
            error.what()
        );
    }
}
```

Listing 6: Error handling mechanism in the `nextToken()` Lexer method (lexer/Lexer.cpp).

1.4 | Hooking up the Lexer to the Runner

The `Runner` class is the basic structure which connects all the stages of the compiler together.

```
Runner::Runner(bool dfsaDbg, bool lexerDbg, bool parserDbg)
    : mDfsaDbg(dfsaDbg),
      mLexerDbg(lexerDbg),
      mParserDbg(parserDbg),
      mLexer(LexerDirector::buildLexer()),
      mParser(Parser(mLexer)) {
}
```

Listing 7: The `Runner` constructor passing `mLexer` into the `Parser` constructor (runner/Runner.cpp).

Initially, the `Runner` passes a reference to the `Lexer` into the `Parser`. This allows the `Parser` to request tokens on demand improving the overall performance of the compiler. Additionally, this has the benefit of allowing the parsing of larger and multiple files. This is because, the `Parser` is no longer limited by the amount of usable memory, as it never needs to load whole files.

TODO The groundwork for such an optimisation has been laid out however, it is not being used. This is because the mechanism for file reading requires changing.

2 | The Parser

2.1 | Modified EBNF

Some modifications were applied to the original EBNF. These modifications were motivated by the prospects of an improved user experience, a more uniform type system and the reduction of code complexity in later stages.

```

<Letter>      ::= 'A'-'Z' | 'a'-'z'
<Digit>       ::= '0'-'9'
<Hex>         ::= 'A'-'F' | 'a'-'f' | <Digit>
<Identifier>  ::= <Letter> { '_' | <Letter> | <Digit> }
<BooleanLiteral> ::= 'true' | 'false'
<IntegerLiteral> ::= <Digit> { <Digit> }
<FloatLiteral>  ::= <Digit> { <Digit> } '.' <Digit> { <Digit> }
<ColorLiteral>  ::= '#' <Hex> <Hex> <Hex> <Hex> <Hex> <Hex>
<ArrayLiteral>  ::= '[' [<Epxr> { ',' <Epxr> } ]
<PadWidth>     ::= '__width'
<PadHeight>    ::= '__height'
<PadRead>      ::= '__read' <Epxr> ',' <Epxr>
<PadRandomInt> ::= '__random_int' <Epxr>

```



```

<Program>      ::= {<Stmt>}

<Stmt>         ::= <Block>
                  | <VariableDecl> ';'
                  | <FunctionDecl>
                  | <Assignment> ';'
                  | <PrintStmt> ';'
                  | <DelayStmt> ';'
                  | <WriteBoxStmt> ';'
                  | <WriteStmt> ';'
                  | <ClearStmt> ';'
                  | <IfStmt>
                  | <ForStmt>
                  | <WhileStmt>
                  | <ReturnStmt> ';'

<Block>        ::= '{' {<Stmt>} '}'

<VariableDecl> ::= 'let' <Identifier> ':' <Type> '=' <Expr>

<FormalParam>  ::= <Identifier> ':' <Type>

<FunctionDecl> ::= 'fun' <Identifier> '(' [ <FormalParam> {',' <FormalParam>} ] ')' '->'
                  <Type> <Block>

<Assignment>   ::= <Identifier> '[' <Expr> ']' '=' <Expr>

<PrintStmt>     ::= '__print' <Expr>

<DelayStmt>     ::= '__delay' <Expr>

<WriteBoxStmt>  ::= '__write_box' <Expr> ',' <Expr> ',' <Expr> ',' <Expr> ',' <Expr>

<WriteStmt>     ::= '__write' <Expr> ',' <Expr> ',' <Expr>

<ClearStmt>     ::= '__clear' <Expr>

<IfStmt>        ::= 'if' '(' <Expr> ')' <Block> ['else' <Block>]

<ForStmt>       ::= 'for' '(' [ <VariableDecl> ] ';' <Expr> ';' [ <Assignment> ] ')' <Block>

<WhileStmt>     ::= 'while' '(' <Expr> ')' <Block>

<ReturnStmt>    ::= 'return' <Expr>

```

Improved Precedence

The changes to the EBNF which improve programmer usability are the additions of a number other expression stages, such as $\langle \text{LogicOr} \rangle$, $\langle \text{LogicAnd} \rangle$, etc. The main reason for the addition of such rules is to further enforce a more natural operation precedence. For example, a programmer often expects that comparison operators such as $<$ and $>$, bind tighter than `and` or `or`, hence the compiler needs to make sure that comparison operators are executed before logical operators. This can be enforced by the grammar itself hence the changes.

Better Arrays

The way arrays were being implemented in the original grammar was very restrictive. Instead an approach for treating arrays as their own type and literal was taken up.

The $\langle \text{Type} \rangle$ and $\langle \text{Literal} \rangle$ productions were augmented to improve array support. This helped simplify the $\langle \text{Identifier} \rangle$ (in the original EBNF), $\langle \text{VariableDecl} \rangle$ and $\langle \text{FormalParam} \rangle$ productions.

This opens up further support for more complicated types later on. For example, the $\langle \text{Type} \rangle$ production can be further augmented to support more expressive types.

```

 $\langle \text{StructField} \rangle$     ::=  $\langle \text{Identifier} \rangle$  ':'  $\langle \text{Type} \rangle$ 

 $\langle \text{Struct} \rangle$         ::= 'struct'  $\langle \text{Identifier} \rangle$  '{' { $\langle \text{StructField} \rangle$  ';' } '}'

 $\langle \text{TypeDecl} \rangle$      ::=  $\langle \text{Struct} \rangle$ 

 $\langle \text{Base} \rangle$          ::= ('bool' | 'int' | 'float' | 'color')

 $\langle \text{Array} \rangle$         ::=  $\langle \text{Type} \rangle$  '['  $\langle \text{IntegerLiteral} \rangle$  ']'

 $\langle \text{Pointer} \rangle$       ::=  $\langle \text{Type} \rangle$  '*'

 $\langle \text{Type} \rangle$          ::=  $\langle \text{Identifier} \rangle$ 
                       |  $\langle \text{Base} \rangle$ 
                       |  $\langle \text{Array} \rangle$ 
                       |  $\langle \text{Pointer} \rangle$ 

```

NOTE

The `<ForamlParam>` is indeed being repeated in a number of places. However, this is not really a problem. When it comes to specifications, repetition which improves clarity is “good” repetition.

Additionally, some of the ground work for these improvements has already been laid out in the internal type system (see Listing 8 and Listing 9).

```
struct Primitive;

enum class Base {
    BOOL,
    COLOR,
    FLOAT,
    INT,
};

struct Array {
    size_t size;
    box<Primitive> type;
};
```

Listing 8: Mechanism for internally storing types within the compiler (parl/Core.hpp).

```
struct Primitive {
    template <typename T>
    [[nodiscard]] bool is() const {
        return std::holds_alternative<T>(data);
    }

    template <typename T>
    [[nodiscard]] const T &as() const {
        return std::get<T>(data);
    }

    ...

    bool operator!=(Primitive const &other) {
        return !operator==(other);
    }
};
```

```
std::variant<std::monostate, Base, Array> data{};
};
```

Listing 9: The Primitive class declaration (parl/Core.hpp).

The `box` type is a special type of pointer object which has value semantics that is it behaves as though it were the object it contains.

This is critical because self-referential types like `<Array>` are not easily representable within C++ since, something like Listing 10, is not allowed.

```
struct SelfRef;

struct Container {
    Other other;
    SelfRef ref;
};

struct Primitive {
    std::variant<std::monostate, Container> data{};
};
```

Listing 10: An size unbounded type in C++ (parl/Core.hpp).

This is because the C++ compiler is incapable of determining the size of said type at compile-time. Hence, a pointer for said type is required. And the pointer wrapper `box<>` allows it to be copied as though it were a value.

ATTRIB. Jonathan Müller presented the `box<>` type in a discussion regarding the exact same issue, on his [blog](#), foonathan.

These changes to type also require additional changes to how variables are referenced in expressions, hence why `<RefExpr>` was added.

TODO Due to `<RefExpr>` more complicated referencing such as `'object.something'` (struct member referencing) should be possible.

Finally, the `<ArrayLiteral>` has been improved to support expressions instead of just only literals.

Removing Eye-Candy

Adding this system however, would significantly increase the complexity of semantic analysis, if the proposed syntax sugar for arrays is kept.

Because of this the following two conveniences `let a: int[] = [1,1,1];` and `let a: int[3] = [1];` have been dropped, from the language.

This is because such syntax not only complicates the grammar but it also significantly increases the complexity of semantic analysis.

The best way to handle such syntax is to have a de-sugaring & type inference sub-phase before type checking, ensure proper separation of concerns.

2.2 | Parsing & The Abstract Syntax Tree (AST)

The AST is the data structure which is produced by the Parser. The nodes of the AST are *almost* a one-to-one representation of the productions in the grammar (see Listing 11).

```
struct FunctionCall : public Reference {
    explicit FunctionCall(std::string, std::vector<std::
        unique_ptr<Expr>>);

    void accept(Visitor*) override;

    const std::string identifier;
    std::vector<std::unique_ptr<Expr>> params;
};
```

Listing 11: The FunctionCall AST node class (parl/AST.hpp).

The only significant difference in these nodes is the `position` field. This gets populated by the parser using the location of a token in the original source file. This is again critical for adequate error messaging in later stages.

```
struct Binary : public Expr {
    explicit Binary(std::unique_ptr<Expr>, Operation, std::
        unique_ptr<Expr>);

    void accept(Visitor*) override;

    std::unique_ptr<Expr> left;
    const Operation op;
    std::unique_ptr<Expr> right;
};
```

Listing 12: The Binary AST node class (parl/AST.hpp).

```
struct Unary : public Expr {
    explicit Unary(Operation, std::unique_ptr<Expr>);

    void accept(Visitor*) override;

    const Operation op;
    std::unique_ptr<Expr> expr;
};
```

Listing 13: The Unary AST node class (parl/AST.hpp).

Apart from position field the only other difference is the usage of a Binary and Unary node (see Listing 12 and Listing 13), instead of a node for each type of the expression types $\langle \text{LogicOr} \rangle$, etc. grammar rules discussed in 2.1. This is because those productions enforce precedence, and precedence, within an AST is not controlled by the nodes but by the structure of the AST itself.

Additionally, a number of the nodes override the `accept()` method specified by the pure virtual class `Node`. This is the basis for the visitor pattern which apart from the parser is the backbone of the later stages.

2.3 | The Actual Parser

The Parser can be split into these four main sections.

- AST Generation;

- Token Buffering;
- Token Matching;
- and, Error Handling/Recovery.

Token Buffering

The Parser of course requires access to the tokens generated from the source file. However, sometimes the Parser might require more than token to decide. This is quite easy to implement if the Parser has available to it, at initialisation, all the tokens.

However, as described in 1.4, This is not the case. The Parser requests token on demand from `Lexer`. This of course means that the machinery for handling tokens is a bit more complicated as it needs to cater for lookahead.

To solve this issue a window-based approach was adopted. The Parser has a moving buffer/window called `mTokenBuffer`, whose size is specified at compile-time using a C-style macro `#define LOOKAHEAD (2)`. The core methods for this aspect of the parser are `moveWindow()` and `nextToken()` (see Listing 14 and Listing 15).

```
void Parser::moveWindow() {
    mPreviousToken = mTokenBuffer[0];

    for (int i = 1; i < LOOKAHEAD; i++) {
        mTokenBuffer[i - 1] = mTokenBuffer[i];
    }

    mTokenBuffer[LOOKAHEAD - 1] = nextToken();
}
```

Listing 14: The `moveWindow()` Parser methods (parser/Parser.cpp).

```
Token Parser::nextToken() {
    std::optional<Token> token;

    do {
        token = mLexer.nextToken();
    } while (!token.has_value() ||
             token->getType() == Token::Type::WHITESPACE ||
             token->getType() == Token::Type::COMMENT);

    return *token;
}
```

Listing 15: The nextToken() Parser methods (parser/Parser.cpp).

NOTE

Within the nextToken() method whitespace and comments are being explicitly ignored.

TODO

Comments can be integrated into the AST. This would allow printing or formatting visitors to properly format code whilst still preserving any comments.

Token Matching

The previous methods all facilitate the more important token matching methods which are:

- peek();
- advance();
- previous();
- isAtEnd();
- peekMatch();
- match();
- and, consume().

Arguably, the most important of these methods is consume(). It takes in a token type and an error message, which it uses to warn the user if the specified token type is not matched (see Listing 16).

```
template <typename... T>
void consume(
    Token::Type type,
    fmt::format_string<T...> fmt,
    T&&... args
) {
    if (check(type)) {
        advance();
    } else {
        error(fmt, args...);
    }
}
```

Listing 16: The `consume()` Parser method (parser/Parser.hpp).

The main reason for templating such a method is to provide an easy interface for formattable strings using `fmtlib`. The main feature this library provides is the ability to specify placeholders in the string itself using `{}`. Usage of this functionality is demonstrated in the AST generator methods (see Listing 17).

```
std::unique_ptr<core::FormalParam> Parser::formalParam() {
    consume(
        Token::Type::IDENTIFIER,
        "expected identifier token "
        "instead received {}",
        peek().toString()
    );
    ...
}
```

Listing 17: Usage of the `consume()` method in the `formalParam()` generator method (parser/Parser.cpp).

The `peekMatch()` method is a simple method which returns true if at least one of the provided token types match. An example demonstrating the usage of `peekMatch()` is the `<Comparison>` production, since there are four token types which match. The `match()` method is a simple extension of `peekMatch()` which consumes the token if it matches.

The methods `advance()`, `previous()` and `isAtEnd()` are quite self explanatory. `advance()` moves the buffer window one step forward, `previous()` returns the

last consumed token, and `isAtEnd()` checks whether or not the Parser has reached an 'End of File' token.

`peek()` is also very simple it allows the Parser to see the token without consuming it. However, since the Parser might need to lookahead `peek()` supports an offset. Because of this calls to `peek()` have to be checked to ensure valid access (see Listing 18). This is done using the `abort_if()` function which prints an error message and aborts the program. This is very similar to a `static_assert` however, it is performed at runtime (see Listing 19).

```
Token Parser::peek(int lookahead) {
    core::abort_if(
        !(0 <= lookahead && lookahead < LOOKAHEAD),
        "exceeded lookahead of {}",
        LOOKAHEAD
    );

    return mTokenBuffer[lookahead];
}
```

Listing 18: The `peek()` Parser method (parser/Parser.cpp).

```
#ifdef NDEBUG
template <typename... T>
inline void abort(fmt::format_string<T...>, T &&...) {
}

template <typename... T>
inline void
abort_if(bool, fmt::format_string<T...>, T &&...) {
}
#else
template <typename... T>
inline void
abort(fmt::format_string<T...> fmt, T &&...args) {
    fmt::println(stderr, fmt, args...);

    std::abort();
}

template <typename... T>
inline void abort_if(
```

```

    bool cond,
    fmt::format_string<T...> fmt,
    T &&...args
) {
    if (cond) {
        abort(fmt, args...);
    }
}
#endif

```

Listing 19: The abort functionality present in the codebase (parl/Core.hpp).

NOTE

Since the usage of `abort()` and `abort_if()` is internal to the code, it only makes sense for these functions to be enabled during debug builds only. Hence, the implementations are enclosed between an `#ifdef`, `#else`, `#endif` macro. When `NDEBUG`, which stand for no-debug, is defined the function bodies are hollowed out allowing the C++ compiler to optimise them out.

Error Handling/Synchronization

Error handling and synchronization is a critical part of the Parser. With regards to developer productivity, having meaningful errors is extremely valuable. But apart from that being able to see all the errors in a file is also crucial. This means that a developer will waste less time re-running the compiler to find all the errors present in the source code.

ATTRIB.

This process is referred to as **Synchronization** by the author of *Crafting Interpreters*, **Robert Nystrom**.

The method for synchronisation in the PARL compiler was inspired by the above credited author and his usage of Exceptions as an unrolling primitive. Although unorthodox, it is a very simple way to implement synchronisation.

```
class SyncParser : public std::exception {};  
  
...  
  
template <typename... T>  
void error(fmt::format_string<T...> fmt, T&&... args) {  
    mHasError = true;  
  
    Token violatingToken = peek();  
  
    fmt::println(  
        stderr,  
        "parsing error at {}:{}:: {}",  
        violatingToken.getPosition().row(),  
        violatingToken.getPosition().col(),  
        fmt::format(fmt, args...)  
    );  
  
    throw SyncParser{};  
}
```

Listing 20: The SyncParser exception and the error() method which kick-starts the synchronisation process (parser/Parser.hpp).

However, there is of course a major downside to this. Where synchronisation happens will affect other error messages which are reported down stream. This of course has the possibility of producing false positives. However, in this case error handling is only best-effort and therefore the possibility of false positives is accepted. The basic process of synchronising is consuming as many tokens as possible until the Parser reaches a token which it believes to be a good restarting point (see Listing 21).

```
void Parser::synchronize() {  
    while (!isAtEnd()) {  
        Token peekToken = peek();  
  
        switch (peekToken.getType()) {  
            case Token::Type::SEMICOLON:  
                advance();  
  
            return;  
        }  
    }  
}
```

```
    case Token::Type::FOR:
        /* fall through */
    case Token::Type::FUN:
        /* fall through */
    case Token::Type::IF:
        /* fall through */
    case Token::Type::LET:
        /* fall through */
    case Token::Type::RETURN:
        /* fall through */
    case Token::Type::WHILE:
        /* fall through */
    case Token::Type::LEFT_BRACE:
        /* fall through */
        return;

    case Token::Type::BUILTIN: {
        auto builtinType =
            *peekToken.asOpt<core::Builtin>();

        switch (builtinType) {
            case core::Builtin::PRINT:
                /* fall through */
            case core::Builtin::DELAY:
                /* fall through */
            case core::Builtin::WRITE:
                /* fall through */
            case core::Builtin::CLEAR:
                /* fall through */
            case core::Builtin::WRITE_BOX:
                return;
            default;; // Do nothing
        }
    }
    default;; // Do nothing
}

    advance();
}
}
```

Listing 21: The `synchronize()` method in the `Parser` class (`parser/Parser.cpp`).

Finally, the most natural choice for capturing SyncParser is in the AST generator methods responsible for generating statements, those being `program()` and `block()`.

AST Generator Methods

Finally, the bulk of the parser is actually the generator methods which build the AST. These methods are not that complicated and they follow the specified grammar faithfully.

See, Listing 22 for an example of such a method and see Listing 23 for a full list of generator methods.

```
std::unique_ptr<core::IfStmt> Parser::ifStmt() {
    consume(
        Token::Type::IF,
        "expected 'if' at start of if statement"
    );

    Token token = previous();

    consume(
        Token::Type::LEFT_PAREN,
        "expected '(' after 'if'"
    );

    std::unique_ptr<core::Expr> cond = expr();

    consume(
        Token::Type::RIGHT_PAREN,
        "expected ')' after expression"
    );

    std::unique_ptr<core::Block> thenBlock = block();

    std::unique_ptr<core::Block> elseBlock{};

    if (match({Token::Type::ELSE})) {
        elseBlock = block();
    }

    return make_with_pos<core::IfStmt>(
        token.getPosition(),
```



```
        std::move(cond),
        std::move(thenBlock),
        std::move(elseBlock)
    );
}
```

Listing 22: The `ifStmt()` node generator method in the Parser class (parser/Parser.cpp).

```
std::unique_ptr<core::Type> type();

std::unique_ptr<core::Program> program();
std::unique_ptr<core::Stmt> statement();
std::unique_ptr<core::Block> block();
std::unique_ptr<core::VariableDecl> variableDecl();
std::unique_ptr<core::Assignment> assignment();
std::unique_ptr<core::PrintStmt> printStatement();
std::unique_ptr<core::DelayStmt> delayStatement();
std::unique_ptr<core::WriteBoxStmt> writeBoxStatement();
std::unique_ptr<core::WriteStmt> writeStatement();
std::unique_ptr<core::ClearStmt> clearStatement();
std::unique_ptr<core::IfStmt> ifStmt();
std::unique_ptr<core::ForStmt> forStmt();
std::unique_ptr<core::WhileStmt> whileStmt();
std::unique_ptr<core::ReturnStmt> returnStmt();
std::unique_ptr<core::FunctionDecl> functionDecl();
std::unique_ptr<core::FormalParam> formalParam();

std::unique_ptr<core::PadWidth> padWidth();
std::unique_ptr<core::PadHeight> padHeight();
std::unique_ptr<core::PadRead> padRead();
std::unique_ptr<core::PadRandomInt> padRandomInt();
std::unique_ptr<core::BooleanLiteral> booleanLiteral();
std::unique_ptr<core::ColorLiteral> colorLiteral();
std::unique_ptr<core::FloatLiteral> floatLiteral();
std::unique_ptr<core::IntegerLiteral> integerLiteral();
std::unique_ptr<core::ArrayLiteral> arrayLiteral();
std::unique_ptr<core::SubExpr> subExpr();
std::unique_ptr<core::Variable> variable();
std::unique_ptr<core::ArrayAccess> arrayAccess();
std::unique_ptr<core::FunctionCall> functionCall();
```

```
std::unique_ptr<core::Expr> expr();  
std::unique_ptr<core::Expr> logicOr();  
std::unique_ptr<core::Expr> logicAnd();  
std::unique_ptr<core::Expr> equality();  
std::unique_ptr<core::Expr> comparison();  
std::unique_ptr<core::Expr> term();  
std::unique_ptr<core::Expr> factor();  
std::unique_ptr<core::Expr> unary();  
std::unique_ptr<core::Expr> primary();
```

Listing 23: The main body of methods in the Parser class (parser/Parser.hpp).

2.4 | Pretty Printing

Using Parser in the Runner

The Parser is used in the Runner and assuming that the parser does not encounter any errors and the `mParserDbg` flag is set it can be used to print the AST using a `PrinterVisitor`.

```
mParser.parse(source);  
  
if (mLexer.hasError() || mParser.hasError()) {  
    return;  
}  
  
std::unique_ptr<core::Program> ast = mParser.getAst();  
  
if (mParserDbg) {  
    debugParsing(ast.get());  
}
```

Listing 24: The parse segment of the `run()` method in the Runner class (runner/Runner.cpp).

Calling the produced PArL binary with the `-p` flag will set the `mParserDbg`, see [Figure 13](#) and [Figure 14](#).

```
> cat testing/test9.parl
fun AverageOfTwo(x: int, y: int) -> float {
  let t0: int = x + y;
  if (t0 > 10) {
    return 10;
  }
  let t1: float = t0 / 2 as float;
  return t1;
}
```

Figure 13: cat of the test9.parl.

```
./run.sh -p testing/test9.parl
Parser Debug Print
Program =>
  Func Decl AverageOfTwo =>
    Formal Param x =>
      int
    Formal Param y =>
      int
    float
    {
      let t0 :
        int
        Binary Operation + =>
          Variable x
          Variable y
      If =>
        Binary Operation > =>
          Variable t0
          int 10
        {
          Return =>
            int 10
        }
      let t1 :
        float
        Binary Operation / =>
          Variable t0
          int 2
        as
          float
      Return =>
        Variable t1
    }
  }
semantic error at 4:9:: incorrect return type in function AverageOfTwo
```

Figure 14: The AST generated by the syntactically correct program in testing/test9.parl.

```
void Runner::debugParsing(core::Program* program) {
  fmt::println("Parser Debug Print");
}
```

```
PrinterVisitor printer;

program->accept(&printer);
}
```

Listing 25: The debugParsing() method in the Runner class (runner/Runner.cpp).

The PrinterVisitor used in Listing 25 is a specialization of the pure virtual Visitor class in parl/Visitor.hpp.

```
...

struct ClearStmt;
struct Block;
struct FormalParam;
struct FunctionDecl;
struct IfStmt;
struct ForStmt;
struct WhileStmt;
struct ReturnStmt;
struct Program;

class Visitor {
public:
    virtual void visit(Type*) = 0;
    virtual void visit(Expr*) = 0;
    virtual void visit(PadWidth*) = 0;
    virtual void visit(PadHeight*) = 0;
    virtual void visit(PadRead*) = 0;
    virtual void visit(PadRandomInt*) = 0;
    virtual void visit(BooleanLiteral*) = 0;
    virtual void visit(IntegerLiteral*) = 0;
    virtual void visit(FloatLiteral*) = 0;
    ...
}
```

Listing 26: A segment of the pure virtual Visitor class (parl/Visitor.hpp).

Due to the way C++ handles symbols, the classes which the visitor can visit must be forward declared manually (see Listing 26). If no such forward declaration is

made, the compiler will complain about the `Visitor` and the `AST` classes being cyclically dependent.

Additionally, any inheriting visitor such as the `PrinterVisitor` can hold state. In fact, this is where the true power of visitors arises. Being able to hold state means that complex computations can be carried out on the `AST`. For example the `PrinterVisitor` although simple makes use of a single variable `mTabCount` (see Listing 27), which it uses to affect how much indentation should be used in printing, allowing us to visualise the `AST`.

```
void PrinterVisitor::visit(core::PadRead *expr) {
    print_with_tabs("__read =>");
    mTabCount++;
    expr->x->accept(this);
    expr->y->accept(this);
    mTabCount--;

    expr->core::Expr::accept(this);
}
```

Listing 27: The `visit(core::PadRead *)` method in the `PrinterVisitor` (`parser/PrinterVisitor.cpp`).

3 | Semantic Analysis

3.1 | Phases & Design

As described in 2.1 the Semantic Analysis phase should be split into a number of sub-phases specifically: symbol resolution, de-sugaring/type inference and type checking.

However, these steps can be combined together into a single phase. There are benefits and downsides to both approaches. The main benefit of using the first approach is a reduction in algorithmic complexity. The logic can be separated into different phases with ease and information from one phase can be propagated to another. The downside of this approach is that it actually adds more code for maintenance, since each individual sub-phase will probably need to be implemented as its own visitor. The other down side is error management. If an error occurs in a particular phase due to separation, a mechanism for propagation needs to be devised which again further increases complexity. Of course

by the very nature of this argument the monolithic approach does not suffer for the issues that the sub-phases approach has. However, it significantly increases complexity since all sub-phases are being done in a single phase. The other more glaring issue is the fact that it is much more difficult to resolve symbols before-hand, in a nice way.

```
void AnalysisVisitor::visit(core::Program *prog) {
    // HACK: this is piece of code to support
    // mutually exclusive recursion such as in test45.parl
    for (auto &stmt : prog->stmts) {
        if (isFunction.check(stmt.get())) {
            try {
                registerFunction(
                    dynamic_cast<core::FunctionDecl *>(
                        stmt.get()
                    )
                );
            } catch (SyncAnalysis &) {
                // noop
            }
        }
    }

    for (auto &stmt : prog->stmts) {
        try {
            stmt->accept(this);
        } catch (SyncAnalysis &) {
            // noop
        }
    }
}
```

Listing 28: The `visit(Program *)` method in the `AnalysisVisitor` class, showing the hack used for resolving function symbols before-hand (`analysis/AnalysisVisitor.cpp`).

You would want to do so to allow for the location agnostic function declarations.

Now, since the current compiler's semantic analysis is done in a single phase, it suffers from some of the issues described above, specifically an unneat solution to the function declaration problem (see Listing 28).

3.2 | The Environment Tree

Some terminology is required to properly describe the number of structures which will be used in this section. The following terms are critical and need to be differentiated properly:

- `SymbolTable`;
- `SymbolTableStack`;
- `Environment`;
- `EnvStack`;
- `and`, `RefStack`.

During semantic analysis there is a need for specific data structures which facilitate scoping rules, type checking, etc.

The most basic data structure which achieves this, is a single `SymbolTable`. A symbol table in its simplest form, is a wrapper around a hash map whose keys are identifiers and values are `Symbols` (or any structure which is capable of storing variable and function signatures).

This approach is quite limiting since it restricts user of the language to a single global scope. Therefore, this structure is not a sufficient solution for most toy-languages let alone production-ready languages.

A better solution is using a stack of symbol tables. This allows symbol tables to shadow each other allowing for the reuse of identifiers. Additionally, this opens up the possibility of implementing modules at the language-level. The likelihood that names will be reused across different modules is quite high and being able to scope modules so they do not interfere with global scope and each other is necessary for any sufficiently large project.

```

[fun XGreaterThenY [X:int, y:int] → bool {
  let ans : bool = true;
  if (X <= y) { ans = true; }
  return ans;
}]

```

Note: '[' denote a push and ']' denote a pop.

Figure 15: A PArL function with annotated scopes.

The basic premise of using a symbol table stack is described in the algorithm below and Figure 16. A new symbol table, also referred to as a scope, is pushed onto a stack only for specific types of AST nodes. The node is then processed, where “processing” often times means considering all the sub-children of the node, and then that is, processing finishes, the scope is popped. In the context of a purely stack based implementation this implies that, scopes are temporary that is, they are lost after being popped.

Algorithm: Basic depiction of SymbolTableStack usage.

Data: N the AST node, S the symbol table stack

```

begin
  if  $N$  opens a scope then
    | PushScope( $S$ );
  end
  ProcessNode( $N$ );
  if  $N$  opens a scope then
    | PopScope( $S$ );
  end
end

```

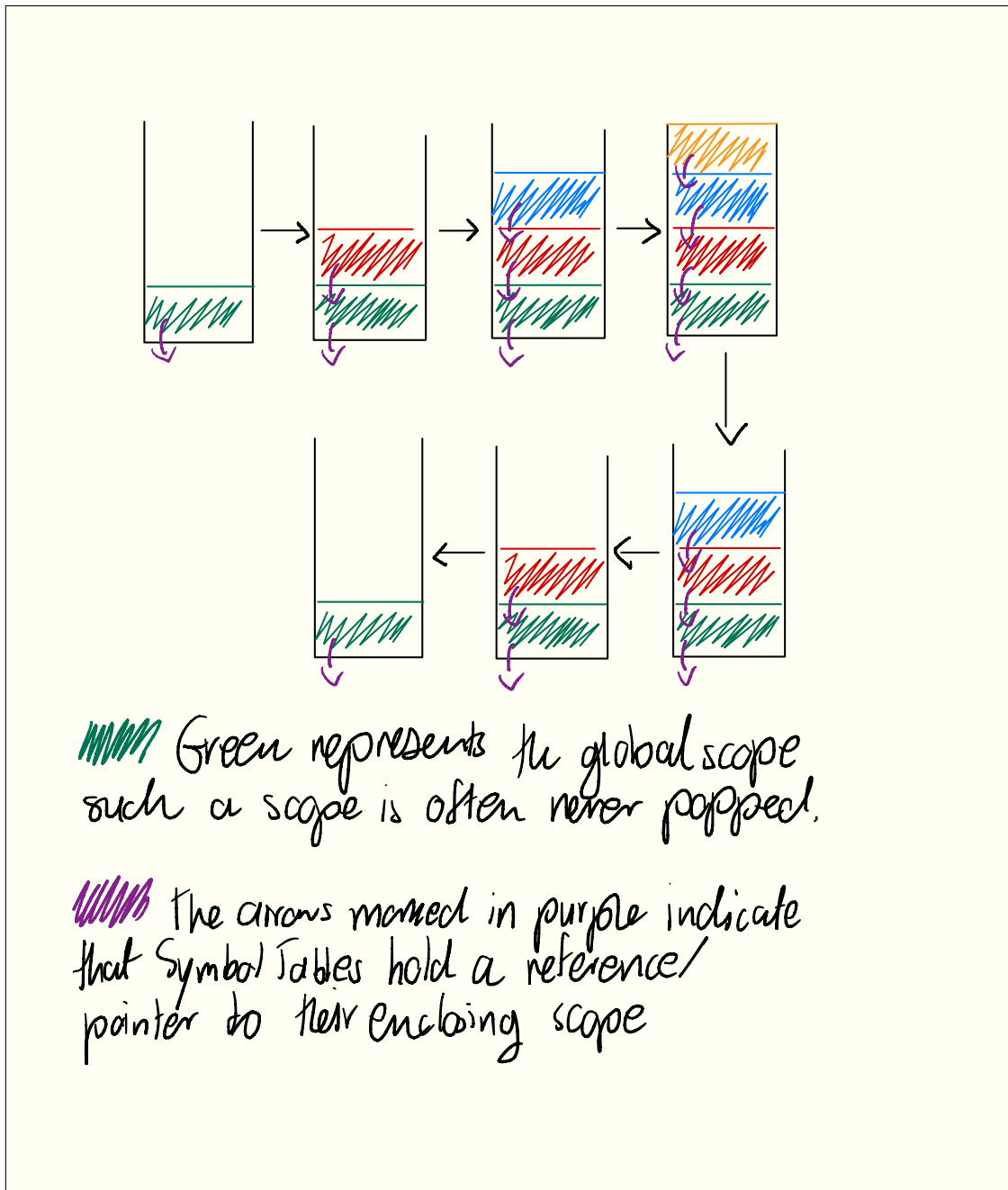


Figure 16: Behaviour of the SymbolTableStack when considering the code in Figure 15.

The main advantage of using this approach is that the worst case memory usage

is $O(DI)$, where D is the length of a longest chain of open scopes and I is the size of largest number of variable declarations within a scope.

However, due to the temporary nature of this approach it does not lend itself well to a multi-phase approach. This is because at each phase SymbolTables have to be regenerated increasing the complexity of each individual phase.

Instead a different approach was used. This approach uses an [Environment Tree](#) (which has also been inspired by [Crafting Interpreters](#)).

The notion of an Environment is essentially, the same as a SymbolTable. The only significant change is the addition of vector of unique pointers to other Environments (see [Listing 29](#)).

```
class Environment {
public:
    enum class Type {
        GLOBAL,
        IF,
        ELSE,
        FOR,
        WHILE,
        FUNCTION,
        BLOCK
    };

    void addSymbol(
        std::string const& identifier,
        Symbol const& Symbol
    );
    [[nodiscard]] std::optional<Symbol> findSymbol(
        std::string const& identifier
    ) const;
    Symbol& getSymbolAsRef(std::string const& identifier);

    [[nodiscard]] Environment* getEnclosing() const;
    void setEnclosing(Environment* enclosing);

    [[nodiscard]] Type getType() const;
    void setType(Type type);

    [[nodiscard]] std::optional<std::string> getName(
    ) const;
```

```
void setName(const std::string& name);

std::vector<std::unique_ptr<Environment>>& children();

[[nodiscard]] bool isGlobal() const;

[[nodiscard]] size_t getIdx() const;
void incIdx();
void incIdx(size_t inc);

[[nodiscard]] size_t getSize() const;
void setSize(size_t size);

private:
    std::unordered_map<std::string, Symbol> mMap{};
    Type mType{Type::GLOBAL};
    std::optional<std::string> mName{};
    Environment* mEnclosing{nullptr};
    std::vector<std::unique_ptr<Environment>> mChildren{};
    size_t mSize{0};
    size_t mIdx{0};
};
```

Listing 29: The Environment class with mChildren highlighted (backend/Environment.hpp).

The additional fields present in the Environment class are there to cater for the needs of each of the stages.

Specifically, mType and mName are used during semantic analysis and mSize and mIdx are used during code generation.

Additionally, it is preferable if the interface with which the phases interact with the Environment tree is identical to that of a SymbolTableStack that is, the same push() and pop() methods are used.

Now to facilitate this another two classes need to be defined EnvStack and RefStack. The main difference between EnvStack and RefStack is that the purpose of EnvStack is to construct the Environment tree whilst RefStack traverses the environment tree. Due to this EnvStack is strictly speaking only required for symbol resolution. However, in the current implementation since symbol resolution is combined with type checking, the Environment tree is generated at the same time, as it is being type checked.

Additionally, in both an `EnvStack` and a `RefStack`, the creation and traversal of the Environment tree is managed via `push()` and `pop()` methods.

In an `EnvStack` `push()` works by creating a new Environment, setting the enclosing Environment to the current Environment and changing the current Environment to the new Environment (see Listing 30). `pop()` makes use of the `mEnclosing` field in the current Environment. It just sets the current Environment to the enclosing Environment, essentially moving up an Environment (see Listing 31).

```
EnvStack& EnvStack::pushEnv() {
    auto& ref = mCurrent->children().emplace_back(
        std::make_unique<Environment>()
    );

    ref->setEnclosing(mCurrent);

    mCurrent = ref.get();

    return *this;
}
```

Listing 30: The `pushEnv()` method in the `EnvStack` class (analysis/EnvStack.cpp).

```
EnvStack& EnvStack::popEnv() {
    mCurrent = mCurrent->getEnclosing();

    return *this;
}
```

Listing 31: The `popEnv()` method in the `EnvStack` class (analysis/EnvStack.cpp).

The implementations in `RefStack` are slightly different. This is because the compiler has to perform a step-wise left-first depth-first traversal. The best way to do this is to make use of a stack and the procedures in Listing 32 and Listing 33 (see Figure 17, for the initial segments of a traversal).

```
RefStack& RefStack::pushEnv() {
    size_t index = mStack.top();

    mStack.top()++;
}
```

```
mStack.push(0);  
  
mCurrent = mCurrent->children()[index].get();  
  
return *this;  
}
```

Listing 32: The `pushEnv()` method in the `RefStack` class (`ir_gen/RefStack.cpp`)

```
RefStack& RefStack::popEnv() {  
    mStack.pop();  
  
    mCurrent = mCurrent->getEnclosing();  
  
    return *this;  
}
```

Listing 33: The `popEnv()` method in the `RefStack` class (`ir_gen/RefStack.cpp`)

! The correctness of the traversal entirely depends on the usage of the `push()` and `pop()` methods. If used incorrectly the traversal might be incorrect or the program might crash. Of course, this is only an implementation detail that is, if used correctly by the compiler developer then, no such issue should occur.

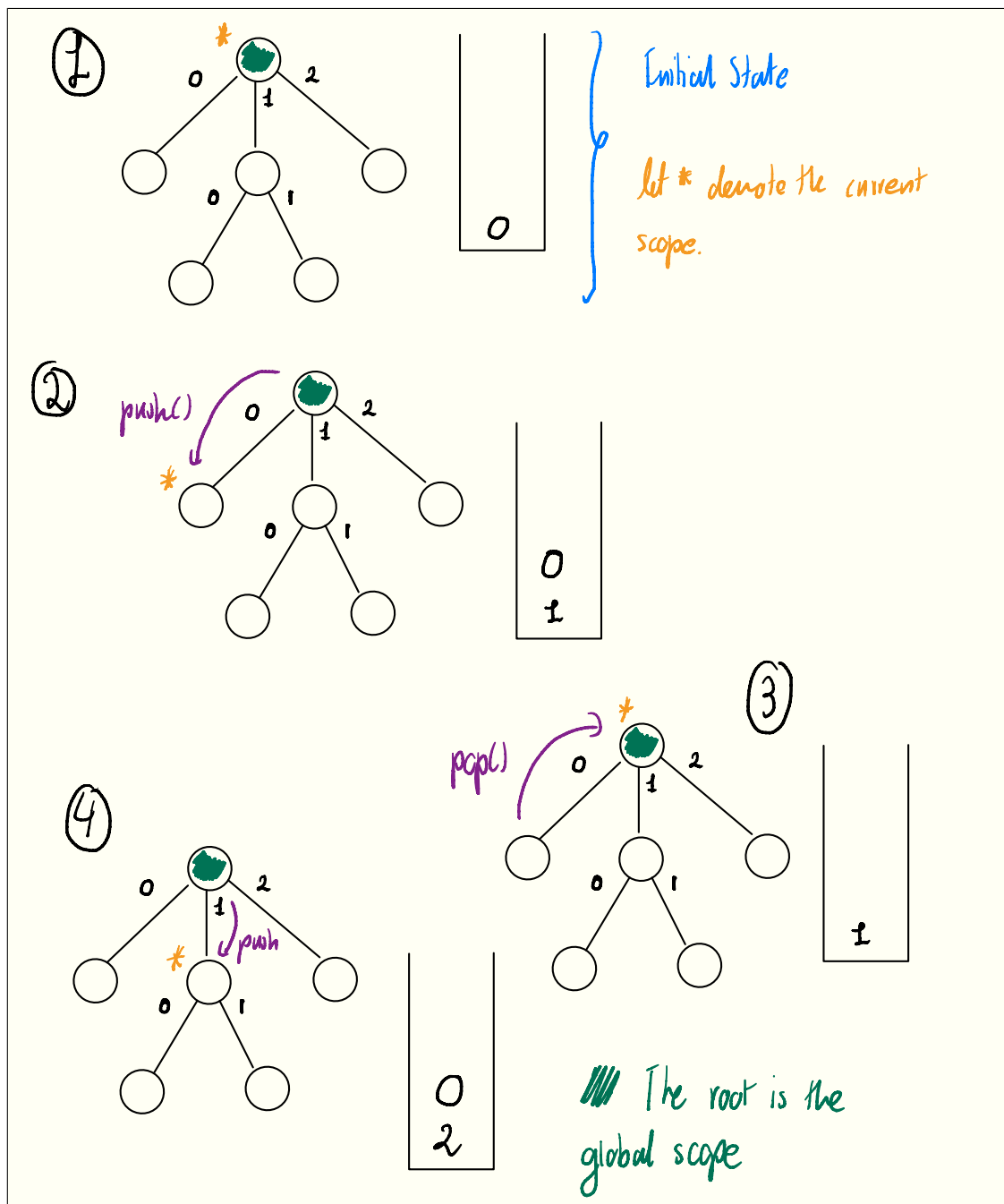


Figure 17: The initial steps of a traversal performed by RefStack.

In the current implementation of PArL three types of nodes control scopes, these are: Blocks, ForStmts and FunctionDecls. There is also a slight particularity in

that, `ForStmts` and `FunctionDecls` as per the grammar actually, cause two scopes to open. This has an effect on the shadowing rules of the language. In particular it presents the following question: Should the language allow for-loop variable declarations and function parameters to be shadowed in the body of the respective statement?

For simplicities sake in PArL a second scope is being opened and hence, immediate shadowing is allowed. However there is no particular consensus on this issue, for example in C++ shadowing of function parameters or for-loop variable declarations is not allowed since they create a single scope, but Rust allows shadowing for the same variable in the same scope without any problems.

3.3 | Actual Semantic Analysis

Having the necessary data structures in place the discussion will now revolve around Semantic Analysis that is, the process of making sure the meaning of the program is correct.

There are four main areas of interest which shall be discussed: the **symbol type**, **symbol registration/search**, **type checking** and **return statement verification**.

The Symbol Type

Within PArL functions are special because they are not treated as values. This means that the `Symbol` type is larger than the `Primitive` type referenced in 2.1. It has to support both variables and functions, hence the `Symbol` type is truly the union of two different types: `VariableSymbol` and `FunctionSymbol`.

```
struct VariableSymbol {
    size_t idx{0};
    core::Primitive type;

    ...
};

struct FunctionSymbol {
    size_t labelPos{0};
    size_t arity{0};
    std::vector<core::Primitive> paramTypes;
    core::Primitive returnType;

    ...
};
```

```
};
```

Listing 34: VariableSymbol and FunctionSymbol class declarations (backend/Symbol.hpp).

The additional fields present (see Listing 34), within said Symbols helps reduce complexity further down the pipeline, especially in code generation.

NOTE Only primitives are comparable. Being able to compare the Symbols that is, adding said logic in the Symbol class violates separation of behaviour and state.

Symbol Registration & Search

A Symbol in PArL is an attribute associated with an identifier (a string). One of the main jobs of semantic analysis is to solidify these associations from the AST into each currently Environment. In PArL the only nodes which are meant to cause a registration are the Variable and Function Declaration nodes.

The following two considerations are the only important considerations for registration: an identifier is never registered twice with in the same scope and function identifiers are globally unique that is, they cannot be shadowed.

```
Environment *enclosingEnv = env->getEnclosing();

for (;;) {
    if (enclosingEnv == nullptr)
        break;

    std::optional<Symbol> identifierSignature =
        enclosingEnv->findSymbol(param->identifier);

    if (identifierSignature.has_value() &&
        identifierSignature->is<FunctionSymbol>()) {
        error(
            param->position,
            "redeclaration of {}(...) as a "
            "parameter",
            param->identifier
        );
    }
}
```



```

        enclosingEnv = enclosingEnv->getEnclosing();
    }

    env->addSymbol(param->identifier, {type});
}

void AnalysisVisitor::registerFunction(
    core::FunctionDecl *stmt
) {
    core::abort_if(
        !mEnvStack.isCurrentEnvGlobal(),
        "registerFunction can only be called in "
        "visit(Program *)"
    );

    std::vector<core::Primitive> paramTypes{
        stmt->params.size()
    };
}

```

Listing 35: The `visit(FormalParam *)` method in the `AnalysisVisitor` class (`analysis/AnalysisVisitor.cpp`).

Searching happens when an identifier in a none-declaration node is met. The procedure for searching basically requires querying the current environment for the symbol associated with the identifier. If nothing is found climb to the enclosing scope and repeat the process. If the root node or *terminating node* is reached and no associated symbol is found that means that the identifier is unbound.

NOTE

The specification of a *terminating node* (a `Environment*`) is important. This is because it allows the analyser to restrict access to outer scopes when necessary. The main example of this is function declarations. In PArL functions are (for the most) part pure, that is they do not produce side-effect at a language level. To enforce this, functions are not allowed to access any state outside of their own scope. The only exception to this fact is the usage of built-ins like `__write` which directly effect the simulator.

This is where one of the extra fields mentioned in 3.2, specifically `mType` is used because it allows scopes to be of different types and this is necessary to find function scopes. For example in Listing 36, `findEnclosingEnv()` is being used to find the closest enclosing function scope and if no such scope is found an empty optional is returned. Then searching terminates on the stopping scope which is either global or an enclosing function scope.

```
void AnalysisVisitor::visit(core::Variable *expr) {
    std::optional<Environment *> stoppingEnv =
        findEnclosingEnv(Environment::Type::FUNCTION);

    std::optional<Symbol> symbol{findSymbol(
        expr->identifier,
        stoppingEnv.has_value() ? *stoppingEnv
                                : mEnvStack.getGlobal()
    )};

    if (!symbol.has_value()) {
        error(
            expr->position,
            "{} is undefined",
            expr->identifier
        );
    }

    ...
}
```

Listing 36: The `visit(Variable *)` method in the `AnalysisVisitor` class (`analysis/AnalysisVisitor.cpp`).

Type Checking

This is the most tedious aspect of semantic analysis. Each individual case where the types affect the behaviour of program or are simply not allowed have to be considered. For this the `AnalysisVisitor` uses the field `mReturn` which is essentially used as the return of a visit, see Listing 37.

```
void AnalysisVisitor::visit(core::Binary *expr) {
    expr->left->accept(this);
    auto leftType{mReturn};

    expr->right->accept(this);
    auto rightType{mReturn};

    ...
}
```

Listing 37: A part of the `visit(Binary *)` method in the `AnalysisVisitor` class (`analysis/AnalysisVisitor.cpp`).

In terms of design a strict approach was chosen, that is no implicit casting takes place. Operations are not capable of operating with two distinct types, for example `5 / 2.0` is not allowed. Instead operations which should support multiple types have been overloaded (later on in code generation). For example, `__print 5 / 2` outputs `2`, `__print 5 / 2.0` is a type error and `__print 5.0 / 2.0` outputs `2.5`.

```
...

case core::Operation::ADD:
case core::Operation::SUB:
    if (leftType.is<core::Array>()) {
        error(
            expr->position,
            "operator {} is not defined on array "
            "types",
            core::operationToString(expr->op)
        );
    }

    if (leftType != rightType) {
        error(
            expr->position,
            "operator {} expects both operands to "
            "be of same type",
            core::operationToString(expr->op)
        );
    }

    mReturn = leftType;
    break;

...
```

Listing 38: Another part of the `visit(Binary *)` method in the `AnalysisVisitor` class (`analysis/AnalysisVisitor.cpp`).

Of course the remainder of the `AnalysisVisitor` is essentially, type checking of the form present in Listing 38.

Return Statement Verification

```
...

std::optional<Environment *> optEnv =
    findEnclosingEnv(Environment::Type::FUNCTION);

if (!optEnv.has_value()) {
    error(
        stmt->position,
        "return statement must be within a "
        "function block"
    );
}

...
```

Listing 39: Checking whether a return statement is inside a function declaration in the `visit(ReturnStmt *)` method in the `AnalysisVisitor` class (`analysis/AnalysisVisitor.cpp`).

The only non-trivial check is that of a return statement. First of all, return statements must be enclosed within functions, otherwise it is a semantic error (see Listing 39).

```
...

Environment *env = *optEnv;

std::string enclosingFunc = env->getName().value();

auto funcSymbol = env->getEnclosing()
    ->findSymbol(enclosingFunc)
    ->as<FunctionSymbol>();

if (exprType != funcSymbol.returnType) {
    error(
        stmt->position,
```

```

        "incorrect return type in function {}",
        enclosingFunc
    );
}

...

```

Listing 40: Checking whether the return expression has the same type as the function return type in the `visit(ReturnStmt *)` method in the `AnalysisVisitor` class (`analysis/AnalysisVisitor.cpp`).

Additionally, they must have the same return type as the enclosing function (see Listing 40), and finally and most importantly, **a function must return in all possible branches**. Additionally, the implementation **should be capable of recognising unreachable areas of code**.

- ! Not stopping when an unreachable code segment is met could result in an invalid assessment of whether a function returns from all branches or not.

During the first iteration of the compiler, the mechanism for checking the return type was embedded directly into the `AnaylsisVisitor`. However, there is a slight problem with this approach. If an unreachable segment of code is met to avoid the issue mentioned above the remainder of the unreachable code cannot be type checked. Of course this is a problem since such behaviour, would mean that the compiler can ignore semantically incorrect code and still produce VM instructions.

To solve this issue a separate visitor, `ReturnVisitor`, was created to handle checking that functions return from all possible branches. This allows for return statement verification to happen after type checking, circumventing any of the above discussed issues.

The mechanism which `ReturnVisitor` uses to establish whether a function returns or not is using a class member `mBranchReturns`.

This member is a boolean and it is explicitly set to true only by a return statement. Function declarations reset the `mBranchReturns` to false and other statements which do not exhibit branching do not modify the value of `mBranchReturns`. This means that the only statements which effect `mBranchReturns` are if-else statements, for-loops and while-loops.

For-loops and while-loops reset `mBranchReturns` to false after their block of statements. This is because entry into the body of the loop (both while and for) is conditional that is, a program might never enter into the body of the loop. Therefore,

it is not enough to return from within the body of the loop.

```
void ReturnVisitor::visit(core::IfStmt *stmt) {
    stmt->thenBlock->accept(this);
    bool thenBranch = mBranchReturns;

    bool elseBranch = false;

    if (stmt->elseBlock) {
        stmt->elseBlock->accept(this);
        elseBranch = mBranchReturns;
    }

    mBranchReturns = thenBranch && elseBranch;
}
```

Listing 41: The `visit(IfStmt *)` method in the `ReturnVisitor` class (analysis/ReturnVisitor.cpp).

If-else statements are different because if the ‘then’ block returns and the ‘else’ block returns then, the whole if-else statement returns, otherwise the if-else statement does not return. This is essentially a logical and between the results of the visitor from visiting the ‘then’ and ‘else’ blocks (see Listing 41).

```
void ReturnVisitor::visit(core::Block *block) {
    size_t i = 0;

    for (; i < block->stmts.size(); i++) {
        block->stmts[i]->accept(this);

        if (mBranchReturns) {
            i++;

            break;
        }
    }

    if (i < block->stmts.size()) {
        core::Position start = block->stmts[i]->position;

        warning(
            start,
```

```
        "statements starting from line {} upto line {} "  
        "are unreachable",  
        start.row(),  
        block->position.row()  
    );  
}  
  
for (; i < block->stmts.size(); i++) { block->stmts.pop_back(); }  
}
```

Listing 42: The `visit(Block *)` method in the `ReturnVisitor` class with the segment pruning statements highlighted (analysis/ReturnVisitor.cpp).

Finally, in the block statements if one of the statements inside manages to successfully return, the remaining statements will never be reached and hence the user can be notified and the statements can be dropped (see Listing 42).

4 | Code Generation

Having completed semantic analysis, the compiler can proceed to code generation.

4.1 | Passing on the Environment Tree

Before, proceeding onto to the details of code generation it is important to note that here is where all the upfront work that was put into using `Environment` trees, pays off. This is because there is no need to re-establish the types of any of the variables, at any scope.

```
std::unique_ptr<Environment> environment =  
    mAnalyser.getEnvironment();  
  
ReorderVisitor reorder{};  
  
reorder.reorderAst(ast.get());  
  
if (mParserDbg) {  
    debugParsing(ast.get());  
}
```

```
reorder.reorderEnvironment(environment.get());

GenVisitor gen{environment.get()};
```

Listing 43: Getting the global environment from the AnalysisVisitor and passing it on for reordering and code generation (runner/Runner.cpp).

The TypeVisitor Class

Having said that there is still the need to recompute the types of expressions since the type of a composite expression e.g. $2 + 3$ cannot be stored in an Environment. To handle this another visitor, this time called TypeVisitor was created.

TODO In the event that type checking/generation is completely delegated to the TypeVisitor, future version of compiler can drop usage of Environment trees in favour of re-computing types as requested. Additionally, such a change improves memory usage in exchange for time as described in 3.2.

```
void TypeVisitor::visit(core::FunctionCall *expr) {
    std::optional<Symbol> symbol{
        findSymbol(expr->identifier, mRefStack.getGlobal())
    };

    mReturn = symbol->as<FunctionSymbol>().returnType;

    expr->core::Expr::accept(this);
}
```

Listing 44: The visit(FunctionCall *) method in the TypeVisitor class (ir_gen/TypeVisitor.cpp).

Currently, a TypeVisitor is given a RefStack for Environment access and its main capability is re-calculating the type of **already** type-checked expressions (see Listing 44).

! Usage of a TypeVisitor on a non-type-checked expression will at best crash and at worst still work, leading to undefined behaviour.

4.2 | Reordering

Due to the fact that the PArDis VM Simulator expects all functions to be defined before the `.main` segment, which in the case of PArL is everything not in a function declaration, a solution needs to be devised to ensure that this order requirement is satisfied.

A solution to this problem can be achieved by reordering the AST and the Environments. Since function declarations can only exist in global scope, moving them such that they are the first statements, which are encountered, is a sufficient modification to ensure code generation as required by the VM.

TODO Currently the compiler has support for named scopes (see backend/Environmentmnt.cpp, lines 25 and 34). With a specially designed naming scheme to uniquely identify each scope, support for nested functions can be enabled by mangling function names with scope names and lifting function declarations into global scope.

This approach required two visitors, named `IsFunctionVisitor` and `ReorderVisitor`, and a function, `reorderEnvironment()` (the choice to embed it into the `ReorderVisitor` was arbitrary).

The `IsFunctionVisitor` is quite self-explanatory, it is a very simple visitor with an exposed method `check()`, which returns true only if a node is a function declaration.

```
void ReorderVisitor::reorder(
    std::vector<std::unique_ptr<core::Stmt>> &stmts
) {
    for (auto &stmt : stmts) {
        if (isFunction.check(stmt.get())) {
            mFuncQueue.push_back(std::move(stmt));
        } else {
            mStmtQueue.push_back(std::move(stmt));
        }
    }

    stmts.clear();

    for (auto &stmt : mFuncQueue) {
        stmts.push_back(std::move(stmt));
    }
}
```

```

    for (auto &stmt : mStmtQueue) {
        stmts.push_back(std::move(stmt));
    }

    reset();
}

```

Listing 45: The `reorder()` method in the `ReorderVisitor` class (preprocess/ReorderVisitor.cpp).

The `ReorderVisitor` implements a method `reorder()`, which is called only for program and block nodes. The `reorder()` makes use of two queues. The first queue keeps track of function declarations and the second queue keeps track of everything else. The `reorder()` method traverse the provided statement vector and enqueues statements in there respective queues. Then the elements of the first queue are dequeued back into the vector followed by the elements of second queue. Due to the usage of queues this essentially moves all function declarations (in their original order) as the first elements of the statements vector. The visitor is then accepted by all the reordered statements.

NOTE

The visitor is accepted by the reordered statements, to cater for the case when function declarations are also present in other scopes not just the global scope. However, this is currently not the case (see analysis/AnalysisVisitor.cpp, line 955).

4.3 | Function Declarations

Due to the reordering described in 4.2, the first nodes to be emitted are function declarations.

```

void GenVisitor::visit(core::FunctionDecl *stmt) {
    Environment *nextEnv = mRefStack.peekNextEnv();

    size_t aritySize{0};

    for (auto &stmt : stmt->params) {
        aritySize +=
            mDeclCounter.count(stmt.get(), nextEnv);
    }

    emit_line(".{", stmt->identifier);
}

```

```
mRefStack.pushEnv(aritySize);

for (auto &param : stmt->params) {
    param->accept(this);
}

stmt->block->accept(this);

mRefStack.popEnv();
}
```

Listing 46: The `visit(FunctionDecl *)` method in the `GenVisitor` class (`ir_gen/GenVisitor.cpp`).

Again for code generation two more visitors apart from the `TypeVisitor` were used. These are the `GenVisitor` and the `VarDeclCountVisitor`.

```
void VarDeclCountVisitor::visit(core::VariableDecl *stmt) {
    std::optional<Symbol> symbol =
        mEnv->findSymbol(stmt->identifier);

    auto &variable = symbol->asRef<VariableSymbol>();

    mCount += variable.type.is<core::Array>()
        ? variable.type.as<core::Array>().size
        : 1;
}
```

Listing 47: The `visit(VariableDecl *)` method in the `VarDeclCountVisitor` class (`ir_gen/VarDeclCountVisitor.cpp`)

The `GenVisitor` as its name suggest is the main visitor which produces the VM instructions. The `VarDeclCountVisitor` is an additional visitor which is used to calculate the size of the memory required when opening a frame on the VM.

NOTE

Within the VM memory is implemented in the form a JavaScript array. This means that there are no restrictions on how big a single location is, for this reason there is no true standard unit of size. However, it is implicitly assumed that the base types: `bool`, `int`, `float` and `color` all occupy single a unit. Hence, when calculating the size of for example `int[4]` the `VarDeclCountVisitor` returns 4 (see Listing 47).

```
template <typename... T>
void emit_line(fmt::format_string<T...> fmt, T &&...args) {
    mCode.push_back(fmt::format(fmt, args...));
}
```

Listing 48: The `emit_line()` method in the `GenVisitor` class (`ir_gen/GenVisitor.cpp`).

On the other hand, the main mechanism with which the `GenVisitor` emits VM code is `emit_line()`. Again similar to `error()` and `warning()` (from other visitors), `emit_line()` wraps an `fmtlib` function (see Listing 46 and Listing 48).

In the context of function declarations the only significant usage of `emit_line()` is to output the identifier of the function.

4.4 | Scopes, Variable (or Formal Parameter) Declarations & Variable Accesses

```
void GenVisitor::visit(core::VariableDecl *stmt) {
    stmt->expr->accept(this);

    Environment *env = mRefStack.currentEnv();

    Environment *stoppingEnv = env;

    while (stoppingEnv->getType() !=
           Environment::Type::GLOBAL &&
           stoppingEnv->getType() !=
           Environment::Type::FUNCTION) {
        stoppingEnv = stoppingEnv->getEnclosing();
    }

    std::optional<Symbol> left{};
```

```

for (;;) {
    left = env->findSymbol(stmt->identifier);

    if (left.has_value() || env == stoppingEnv)
        break;

    env = env->getEnclosing();
}

core::abort_if(
    !left.has_value(),
    "symbol is undefined"
);

size_t idx = env->getIdx();

auto symbol = left->as<VariableSymbol>();

if (symbol.type.is<core::Base>()) {
    env->incIdx();
} else if (symbol.type.is<core::Array>()) {
    env->incIdx(symbol.type.as<core::Array>().size);
} else {
    core::abort("unknown type");
}

// NOTE: make sure this is actually a reference.
env->getSymbolAsRef(stmt->identifier)
    .asRef<VariableSymbol>()
    .idx = idx;

if (symbol.type.is<core::Array>()) {
    emit_line(
        "push {}",
        symbol.type.as<core::Array>().size
    );
}
emit_line("push {}", idx);
emit_line("push 0");
if (symbol.type.is<core::Array>()) {
    emit_line("sta");
} else {

```

```

        emit_line("st");
    }
}

```

Listing 49: The `visit(VariableDecl *)` method in the `GenVisitor` class (`ir_gen/GenVisitor.cpp`).

Next, specific attention has to be given to variable declarations and formal parameters. Referencing of variables or formal parameters within the VM happens with the instruction `push [i:l]` (for behaviour see Figure 18). The key take-away from how this instruction works is the fact that frames (which are actually just memory) are accessed via an index (in the instruction `i`).

Hence, during code generation any reference to a variable or formal parameter needs to happen through an index. This is why, when defining a new variable or formal parameter, the internal index of the current environment is copied into the variable or formal parameter's associated symbol. This facilitates referencing a variable or formal parameter in the VM's instructions (see Listing 49 and `ir_gen/GenVisitor.cpp`, line 105).

Additionally, the index of the environment is incremented to the next free position i.e. the next position a new variable or formal parameter can be stored within the frame.

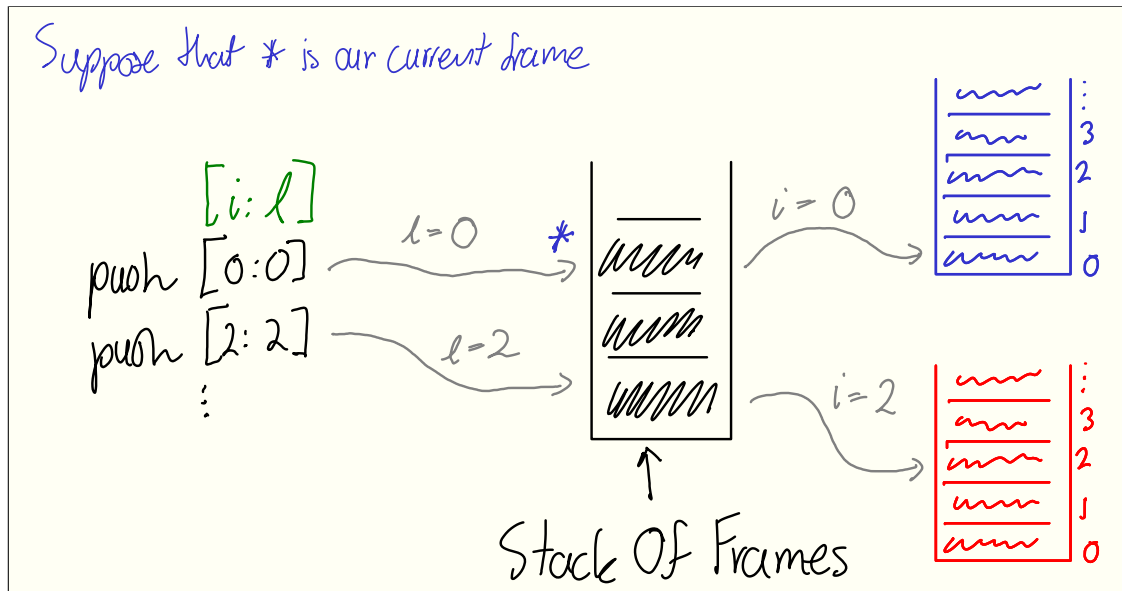


Figure 18: A pictorial example of how the `push [i:l]` operates.

4.5 | Properly Closing Scopes in Function Declarations

A very important consideration when it comes to function declarations is closing of scopes.

This is a problem of interest because functions should be capable of returning anywhere within the body of a function even if they are nested within multiple scopes.

```
...

emit_line("push {}", count);
emit_line("oframe");

mFrameDepth++;

...
```

Listing 50: A segment of the `visit(Program *)` method in the `GenVisitor` class (`ir_gen/GenVisitor.cpp`).

```
void GenVisitor::visit(core::ReturnStmt *stmt) {
    stmt->expr->accept(this);

    for (size_t i = 0; i < mFrameDepth; i++) {
        emit_line("cframe");
    }

    emit_line("ret");
}
```

Listing 51: The `visit(ReturnStmt *)` method in the `GenVisitor` class (`ir_gen/GenVisitor.cpp`).

To solve this issue, a variable `mFrameDepth` keeps track of the number of frames open with the `oframe` instruction, see Listing 50. When a return statement node is reached the close frame instruction, `cframe`, is emitted as many times as `mFrameDepth` (see Listing 51). This ensures that extra open frames are closed before returning, allowing the VM to continue executing the program normally.

4.6 | Assignments

Similar, to when variables are first declared the store (st) instruction is used for mutating the value in a specified location. However, unlike variable declarations which always define a level of 0, assignments/mutations can occur on variables defined in outer scopes. Hence, the level of variable access needs to be calculated before being emitting st and its necessary operands.

```
size_t GenVisitor::computeLevel(Environment *stoppingEnv) {
    size_t level = 0;

    Environment *env = mRefStack.currentEnv();

    while (stoppingEnv != env) {
        switch (env->getType()) {
            case Environment::Type::FOR:
            case Environment::Type::BLOCK:
                level++;
                break;
            case Environment::Type::FUNCTION:
                core::abort("unreachable");
                break;
            default:
                // noop
                break;
        }

        env = env->getEnclosing();
    }

    return level;
}
```

Listing 52: The `computeLevel()` method in the `GenVisitor` class (`ir_gen/GenVisitor.cpp`).

The basic idea for calculating the level of a variable is as follows, copy the pointer to the `Environment` which holds the variable then traverse from the current `Environment` until the `Environment` with the exact pointer as the `Environment` the variable was found in is reached, all the while keeping track of the number of scopes jumped through to reach said environment.

NOTE

Not, all `Environments` contribute an open frame (`oframe`), hence only certain types of scopes should be counted as contributing to the number of levels. In the case of PArL with its current restrictions, the only such scopes are for-scopes and block-scopes (see Listing 52).

4.7 | Expressions

Type Specific VM Code

```
...
case core::Operation::DIV: {
    core::Primitive type = mType.getType(
        expr->right.get(),
        mRefStack.getGlobal(),
        mRefStack.currentEnv()
    );
    if (type == core::Primitive{core::Base::INT}) {
        expr->right->accept(this);
        expr->right->accept(this);
        expr->left->accept(this);
        emit_line("mod");
        expr->left->accept(this);
        emit_line("sub");
        emit_line("div");
    } else {
        expr->right->accept(this);
        expr->left->accept(this);
        emit_line("div");
    }
} break;
...
```

Listing 53: A segment of the `visit(Binary *)` method in the `GenVisitor` class (`ir_gen/GenVisitor.cpp`).

When it comes to expressions, due to the fact that operators within the PArL language are overloaded some attention to detail is required. The majority of the usage of `TypeVisitor` is within expression nodes, specifically, in places where behaviour is dependent on the operand's types. For example in the case of division, the type of one of the operands is computed (recall that semantic analysis guarantees consistency of types, in the case of binary operations both the left and right

operands have the same type), then different instructions are emitted depending on the returned type (see Listing 53).

Function Calls

```

void GenVisitor::visit(core::FunctionCall *expr) {
    for (auto itr = expr->params.rbegin();
         itr != expr->params.rend();
         itr++) {
        (*itr)->accept(this);
    }

    size_t size{0};

    for (auto &param : expr->params) {
        core::Primitive paramType = mType.getType(
            param.get(),
            mRefStack.getGlobal(),
            mRefStack.currentEnv()
        );

        size += paramType.is<core::Array>()
            ? paramType.as<core::Array>().size
            : 1;
    }

    emit_line("push {}", size);
    emit_line("push .{}", expr->identifier);
    emit_line("call");
}

```

Listing 54: The `visit(FunctionCall *)` method in the `GenVisitor` class (`ir_gen/GenVisitor.cpp`).

Functions calls are performed using the `call` VM instruction. The most important aspect of function calls, similar to scopes, is the calculation of the total size of the parameters. This essentially determines how many operands will be popped off the operand stack and moved into the implicitly created function frame. Additionally, it is important for all the parameters of a function to be evaluated in reverse order due to the fact that popping from the operand stack into the function frame reverse the order of the parameters (see Listing 54).

4.8 | Branching

Branching is somewhat challenging since at the jumping point the compiler is unaware of how many instructions it might need to jump. There are two possible solutions for this problem. The first is to create a visitor which is capable of calculating the number of instruction which a node expands to. The second makes use of a placeholder instruction that is then changed after the rest of the associated instructions are generated.

Previous iterations of the PArL compiler used the first approach however, the first approach is quite unmaintainable since changes in other areas of the code generation would require changes in the visitor which computes the number of instructions. Hence, the compiler was changed to adopt the second approach.

PArL has only three node types which branch, if-else statements, for-loops and while-loops.

```
void GenVisitor::visit(core::WhileStmt *stmt) {
    mRefStack.pushEnv();

    size_t condOffset = PC();

    stmt->cond->accept(this);

    emit_line("not");

    size_t patchOffset = PC();

    emit_line("push #PC+{ }");
    emit_line("cjmp");

    stmt->block->accept(this);

    emit_line("push #PC-{ }", PC() - condOffset);
    emit_line("jmp");

    mCode[patchOffset] =
        fmt::format(mCode[patchOffset], PC() - patchOffset);

    mRefStack.popEnv();
}
```

Listing 55: The `visit(WhileStmt *)` method in the `GenVisitor` class (`ir_gen/GenVisitor.cpp`).

Both for-loops and while-loops branch in the same manner. Patching the relevant instructions for branching in for- and while-loops works as follows (additionally, see Listing 55):

1. The location of the instruction which starts the condition is stored (in this case in `condOffset`);
2. the location of the `'push #PC+{'}` instruction used for `cjmp` is stored (in this case in `patchOffset`);
3. then the block associated with the for- or while-loop accepts the `GenVisitor`, emitting further instructions;
4. the offset required to jump back to the start of the condition instructions is calculated and used with a `jmp` instruction;
5. and finally, the `'push #PC+{'}` instruction at `patchOffset` is patched with the appropriate offset to jump clear all of the for- or while- loop instructions.

```
void GenVisitor::visit(core::IfStmt *stmt) {
    if (stmt->elseBlock) {
        mRefStack.pushEnv();

        stmt->cond->accept(this);

        emit_line("not");

        size_t ifPatchOffset = PC();

        emit_line("push #PC+{[]}");
        emit_line("cjmp");

        stmt->thenBlock->accept(this);

        mRefStack.popEnv();
    }
```

```

    mRefStack.pushEnv();

    size_t elsePatchOffset = PC();

    emit_line("push #PC+{}}");
    emit_line("jmp");

    mCode[ifPatchOffset] = fmt::format(
        mCode[ifPatchOffset],
        PC() - ifPatchOffset
    );

    stmt->elseBlock->accept(this);

    mCode[elsePatchOffset] = fmt::format(
        mCode[elsePatchOffset],
        PC() - elsePatchOffset
    );

    mRefStack.popEnv();
} else {
    mRefStack.pushEnv();

    stmt->cond->accept(this);

    emit_line("not");

    size_t patchOffset = PC();

    emit_line("push #PC+{}}");
    emit_line("cjmp");

    stmt->thenBlock->accept(this);

    mCode[patchOffset] = fmt::format(
        mCode[patchOffset],
        PC() - patchOffset
    );

    mRefStack.popEnv();
}
}

```

Listing 56: The `visit(IfStmt *)` method in the `GenVisitor` class (`ir_gen/GenVisitor.cpp`).

The other form of branching, if-else statements, is a bit more involved. This is because there are two cases which need consideration: when only an `if` is present and when both an `if` and an `else` are present. Nevertheless, the basic principal is the same, offsets into the code are used and specific lines are patched after (see Listing 56).

5 | Arrays

Adding support for arrays in PArL had driven a number of changes across the whole of the compiler. Some of the changes will be explained in the following sections whilst others were already described in earlier sections due to the fundamental changes they caused.

5.1 | Changes in Lexical Analysis

```
...

.addCategory(
    LEFT_BRACKET,
    [](char c) -> bool {
        return c == '[';
    }
)
.addCategory(
    RIGHT_BRACKET,
    [](char c) -> bool {
        return c == ']';
    }
)

...

.addTransition(0, LEFT_BRACKET, 20)
.addTransition(0, RIGHT_BRACKET, 21)
```

```
...

.setStateAsFinal(20, Token::Type::LEFT_BRACK)
.setStateAsFinal(21, Token::Type::RIGHT_BRACK)

...
```

Listing 57: Changes in the `LexerDirector` for supporting left ('[') and right (']') brackets (`lexer/LexerDirector.cpp`).

Within lexical analysis a very lax approach to arrays was adopted that is, there is no enforcement of the more complicated `<Identifier>` syntax proposed in the original EBNF of the language. The only additions have been support for consuming left ('[') and right (']') brackets.

5.2 | Changes in Parsing

As described in 2.1, the grammar of the language was significantly reordered to improve the upgrade-ability of the language down the line. Additionally, the majority of the changes to the parser were also discussed in 2.1.

TODO The addition of de-sugaring, type inference and more types is facilitated by the changes proposed in the EBNF. These additions would allow for more complicated language mechanisms and hence an improved developer experience.

5.3 | Changes in Semantic Analysis

Again due to the changes in the way types are managed (again see 2.1), the following notable changes were required: extra validation, restriction on type casting, figuring out the type of array access and finally, generating the type for array literals.

```
...
case core::Operation::ADD:
case core::Operation::SUB:
    if (leftType.is<core::Array>()) {
        error(
            expr->position,
            "operator {} is not defined on array "
            "types",
```

```

        core::operationToString(expr->op)
    );
}

if (leftType != rightType) {
    error(
        expr->position,
        "operator {} expects both operands to "
        "be of same type",
        core::operationToString(expr->op)
    );
}

mReturn = leftType;
break;
...

```

Listing 58: A part of the `visit(Binary *)` method in the `AnalysisVisitor` class (`analysis/AnalysisVisitor.cpp`).

Of course, validation is necessary to ensure that arrays do not interact with each other, for example `[1,2,3] + [1,2,3]` is not semantically correct in PArL.

```

void AnalysisVisitor::isViableCast(
    core::Primitive &from,
    core::Primitive &to

) {
    const core::Primitive *lPtr = &from;
    const core::Primitive *rPtr = &to;

    while (lPtr != nullptr && rPtr != nullptr) {
        if (lPtr->is<core::Base>() &&
            rPtr->is<core::Array>()) {
            error(
                mPosition,
                "primitive cannot be cast to an array"
            );
        }

        if (lPtr->is<core::Array>() &&
            rPtr->is<core::Base>()) {

```



```
        error(
            mPosition,
            "array cannot be cast to a primitive"
        );
    }

    if (lPtr->is<core::Array>() &&
        rPtr->is<core::Array>()) {
        size_t lSize = lPtr->as<core::Array>().size;
        size_t rSize = rPtr->as<core::Array>().size;

        lPtr = lPtr->as<core::Array>().type.get();
        rPtr = rPtr->as<core::Array>().type.get();

        if (lSize != rSize) {
            error(
                mPosition,
                "array of size {} cannot be cast to an "
                "array of size {}",
                lSize,
                rSize
            );
        }

        continue;
    }

    return;
}

core::abort("unreachable");
}
```

Listing 59: The `isViableCast()` method in the `AnalysisVisitor` class (`analysis/AnalysisVisitor.cpp`).

Necessary checks to ensure that casting is viable, needed to be considered. This approach was taken over allowing all forms of casting because casting from one type to another where the sizes of the types are different would result in substantial complexity later on in code generation.

Consider the following example:

```
let x: int[5] = [1,2,3,4,5];
let y: int = (x as color[10])[6];
```

Not only is there ambiguity as to what the upper elements of the type cast should be but additional padding is necessary to ensure no out of bounds accesses during VM execution.

Finally, the process of computing the type of an array access and an array literal is quite easy, see `analysis/AnalysisVisitor.cpp` lines 252 and 170 respectively.

5.4 | Changes in Code Generation

One of the more important changes for code generation is the calculation of frame sizes. This has already been described in detail even in the context of arrays in the note in 4.3. Of course where appropriate `sta`, `pusha`, `printa` and `push +[i:1]` which are meant for storing arrays, pushing arrays onto the operand stack, printing arrays and pushing an single element of an array onto the operand stack respectively, are being used accordingly. See `ir_gen/GenVisitor.cpp`, lines 456, 111, 478 and 166, for examples of each.

A decision to opt out of using `reta` was made due to the inconsistent behaviour of arrays in the VM. Instead a hack for reversing the order manually was devised.

```
...
if (symbol.type.is<core::Array>()) {
    size_t arraySize =
        symbol.type.as<core::Array>().size;
    emit_line("push {}", arraySize);
    emit_line("pusha [{}:{}] ", symbol.idx, level);
    emit_line("push {} // START HACK", arraySize);
    emit_line("oframe");
    emit_line("push {}", arraySize);
    emit_line("push 0");
    emit_line("push 0");
    emit_line("sta");
    emit_line("push {}", arraySize);
    emit_line("pusha [0:0]");
    emit_line("cframe // END HACK");

    return;
```

```
}  
...
```

Listing 60: A part of the `visit(Variable *)` method in the `GenVisitor` class, specifically the hack mentioned above (`ir_gen/GenVisitor.cpp`).

The problem of arrays is the following:

Consider the following code: `__print [1,2,3];`

```
.main  
push 0  
oframe  
push 3  
push 2  
push 1  
push 3  
printa  
cframe  
halt
```

It generates the code above and what is important to notice is the last element in the array is the first to be pushed onto the operand stack ensuring that printing is correct.

However, now consider the following piece of code: `let x: int[3] = [1,2,3];`
`__print x;`

```

.main
push 3
oframe
push 3
push 2
push 1
push 3
push 0
push 0
sta
push 3
pusha [0:0]
push 3
printa
cframe
halt

```

The naïve implementation generates the code on the left. Running this on the VM one would expect the exact same output as the first. However, this is not the case the order of the elements is reversed.



Figure 19: The result of running the code generated by `let x: int[3] = [1,2,3]; __print x;` in the VM.

This happens because reading from a frame using `pusha` actually reverse the order of the elements on the operand stack. To circumvent this issue a second tempo-

rary frame is created where the elements are stored in the frame and read again reversing the order one-more time into the correct order (see Listing 60).

Due to this hack which works in all situations `reta` is not being used as it would flip the order again.

6 | Testing & Usage

6.1 | Usage

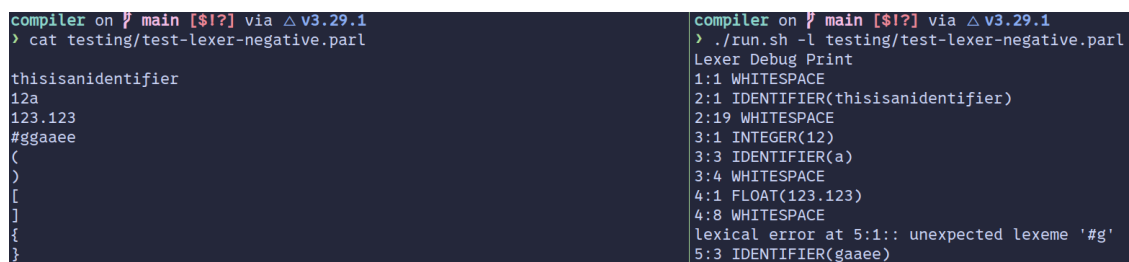
The current produced binary accepts the following flags `-d`, `-l`, `-p`. `-d` is currently disabled, `-l` prints each of the lexemes generated by the source and `-p` outputs the AST both before and after preprocessing.

6.2 | The Lexer, Parser & Semantic Analyser

As discussed in previous sections all the phases which are capable of producing errors or warnings are either doing so manually using `fmt::println(stderr, ...)` or using some customised methods like `error(...)` or `warning(...)`. Additionally, as described in previous sections, the Lexer, Parser and analyser should not fail upon the first inconsistency within the program, instead they should proceed to a state in which they continue processing normally.

The main aim of testing will be to make sure that both correct and incorrect input produce the expected behaviour.

Lexical Analysis



```

compiler on / main [!?] via v3.29.1
> cat testing/test-lexer-negative.parl

thisisanidentifier
12a
123.123
#ggaaee
(
[
]
{
}

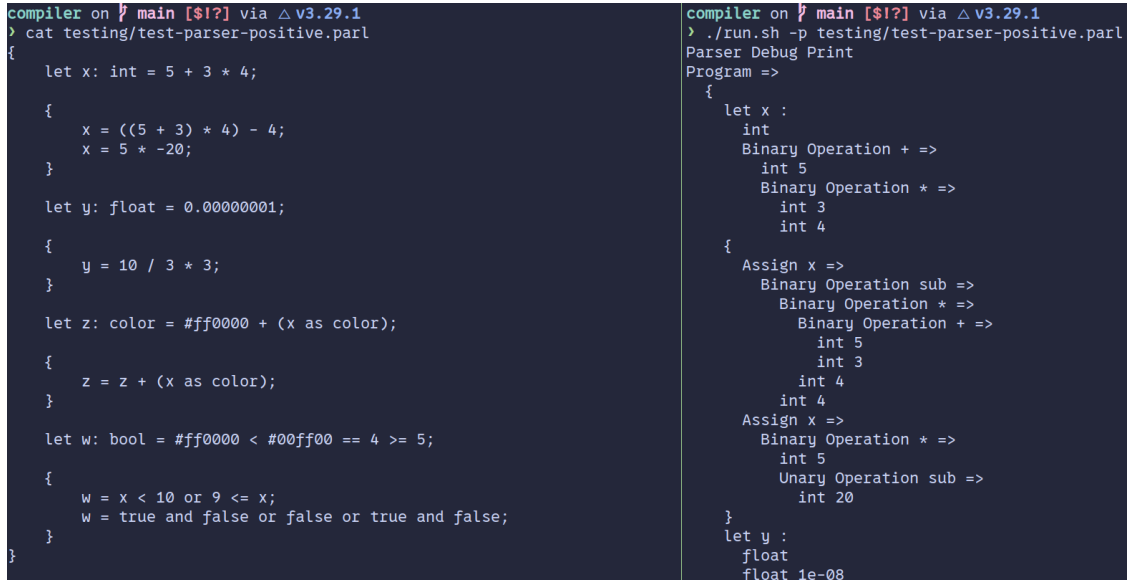
compiler on / main [!?] via v3.29.1
> ./run.sh -l testing/test-lexer-negative.parl
Lexer Debug Print
1:1 WHITESPACE
2:1 IDENTIFIER(thisisanidentifier)
2:19 WHITESPACE
3:1 INTEGER(12)
3:3 IDENTIFIER(a)
3:4 WHITESPACE
4:1 FLOAT(123.123)
4:8 WHITESPACE
lexical error at 5:1:: unexpected lexeme '#g'
5:3 IDENTIFIER(gaaee)

```

Figure 20: Running the negative lexer test (testing/test-lexer-negative.parl).

For lexical analysis two major tests were used, testing/test-lexer-negative.parl and testing/test-lexer-positive.parl. Additional tests include: testing/test1.parl upto testing/test9.parl.

Parsing



```

compiler on P main [$!?] via Δ v3.29.1
> cat testing/test-parser-positive.parl
{
  let x: int = 5 + 3 * 4;

  {
    x = ((5 + 3) * 4) - 4;
    x = 5 * -20;
  }

  let y: float = 0.00000001;

  {
    y = 10 / 3 * 3;
  }

  let z: color = #ff0000 + (x as color);

  {
    z = z + (x as color);
  }

  let w: bool = #ff0000 < #00ff00 == 4 >= 5;

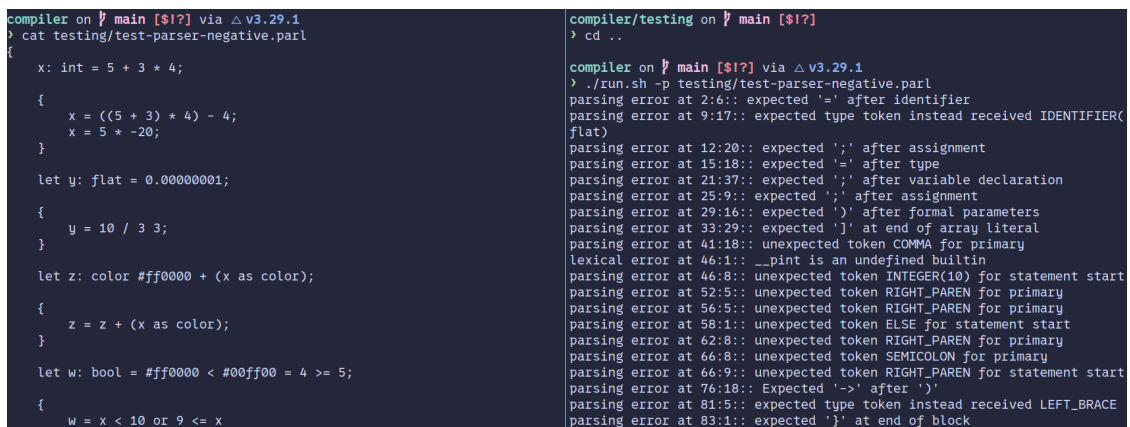
  {
    w = x < 10 or 9 <= x;
    w = true and false or false or true and false;
  }
}

compiler on P main [$!?] via Δ v3.29.1
> ./run.sh -p testing/test-parser-positive.parl
Parser Debug Print
Program =>
{
  let x :
    int
    Binary Operation + =>
      int 5
      Binary Operation * =>
        int 3
        int 4
    {
      Assign x =>
        Binary Operation sub =>
          Binary Operation * =>
            Binary Operation + =>
              int 5
              int 3
            int 4
          int 4
        Assign x =>
          Binary Operation * =>
            int 5
            Unary Operation sub =>
              int 20
    }
  let y :
    float
    float 1e-08
}

```

Figure 21: Running the positive Parser test (testing/test-parser-positive.parl).

Again for parsing two major tests where developed testing/test-parser-negative.parl and testing/test-parser-positive.parl.



```

compiler on P main [$!?] via Δ v3.29.1
> cat testing/test-parser-negative.parl
{
  x: int = 5 + 3 * 4;

  {
    x = ((5 + 3) * 4) - 4;
    x = 5 * -20;
  }

  let y: flat = 0.000000001;

  {
    y = 10 / 3 3;
  }

  let z: color #ff0000 + (x as color);

  {
    z = z + (x as color);
  }

  let w: bool = #ff0000 < #00ff00 = 4 >= 5;

  {
    w = x < 10 or 9 <= x
  }
}

compiler/testing on P main [$!?]
> cd ..
compiler on P main [$!?] via Δ v3.29.1
> ./run.sh -p testing/test-parser-negative.parl
parsing error at 2:6:: expected '=' after identifier
parsing error at 9:17:: expected type token instead received IDENTIFIER(
flat)
parsing error at 12:20:: expected ';' after assignment
parsing error at 15:18:: expected '=' after type
parsing error at 21:37:: expected ';' after variable declaration
parsing error at 25:9:: expected ';' after assignment
parsing error at 29:16:: expected ')' after formal parameters
parsing error at 33:29:: expected ']' at end of array literal
parsing error at 41:18:: unexpected token COMMA for primary
lexical error at 46:1:: __pint is an undefined builtin
parsing error at 46:8:: unexpected token INTEGER(10) for statement start
parsing error at 52:5:: unexpected token RIGHT_PAREN for primary
parsing error at 56:5:: unexpected token RIGHT_PAREN for primary
parsing error at 58:1:: unexpected token ELSE for statement start
parsing error at 62:8:: unexpected token RIGHT_PAREN for primary
parsing error at 66:8:: unexpected token SEMICOLON for primary
parsing error at 66:9:: unexpected token RIGHT_PAREN for statement start
parsing error at 76:18:: Expected '->' after ')'
parsing error at 81:5:: expected type token instead received LEFT_BRACE
parsing error at 83:1:: expected '}' at end of block

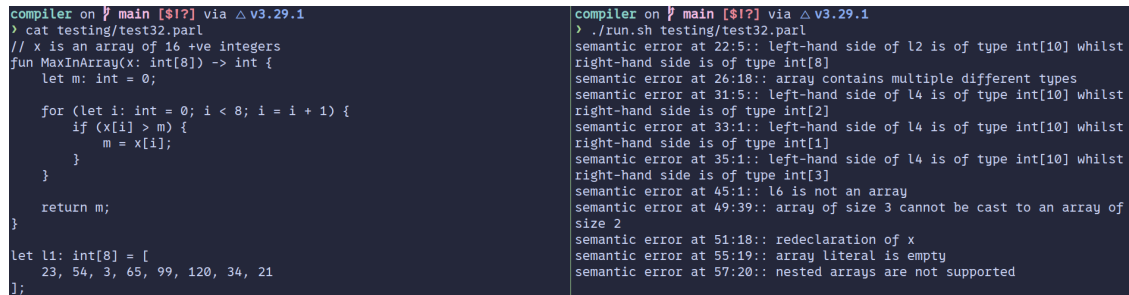
```

Figure 22: Running the negative Parser test (testing/test-parser-negative.parl).

In particular the positive test covers all of the happy paths of the Parser. However, there are way more possibilities where the Parser flags the input as not conforming to the grammar. Hence, not all possible code paths are being tested

with the negative test. Additional, files used for testing during development include: testing/test10.parl, testing/test15.parl and testing/test16.parl.

Semantic Analysis



```

compiler on / main [ $? ] via Δ v3.29.1
> cat testing/test32.parl
// x is an array of 16 +ve integers
fun MaxInArray(x: int[8]) -> int {
  let m: int = 0;

  for (let i: int = 0; i < 8; i = i + 1) {
    if (x[i] > m) {
      m = x[i];
    }
  }

  return m;
}

let l1: int[8] = [
  23, 54, 3, 65, 99, 120, 34, 21
];

```

```

compiler on / main [ $? ] via Δ v3.29.1
> ./run.sh testing/test32.parl
semantic error at 22:5:: left-hand side of l2 is of type int[10] whilst
right-hand side is of type int[8]
semantic error at 26:18:: array contains multiple different types
semantic error at 31:5:: left-hand side of l4 is of type int[10] whilst
right-hand side is of type int[2]
semantic error at 33:1:: left-hand side of l4 is of type int[10] whilst
right-hand side is of type int[1]
semantic error at 35:1:: left-hand side of l4 is of type int[10] whilst
right-hand side is of type int[3]
semantic error at 45:1:: l6 is not an array
semantic error at 49:39:: array of size 3 cannot be cast to an array of
size 2
semantic error at 51:18:: redeclaration of x
semantic error at 55:19:: array literal is empty
semantic error at 57:20:: nested arrays are not supported

```

Figure 23: Running test 32 (testing/test32.parl).

Semantic analysis is again a leap in complexity. And hence, the order of possible errors and issues is also greater. Because of this completely testing all possible scenarios is very difficult. Instead the tests presented and mentioned are those written during development to test certain aspects of the semantic analyser as it was being developed.

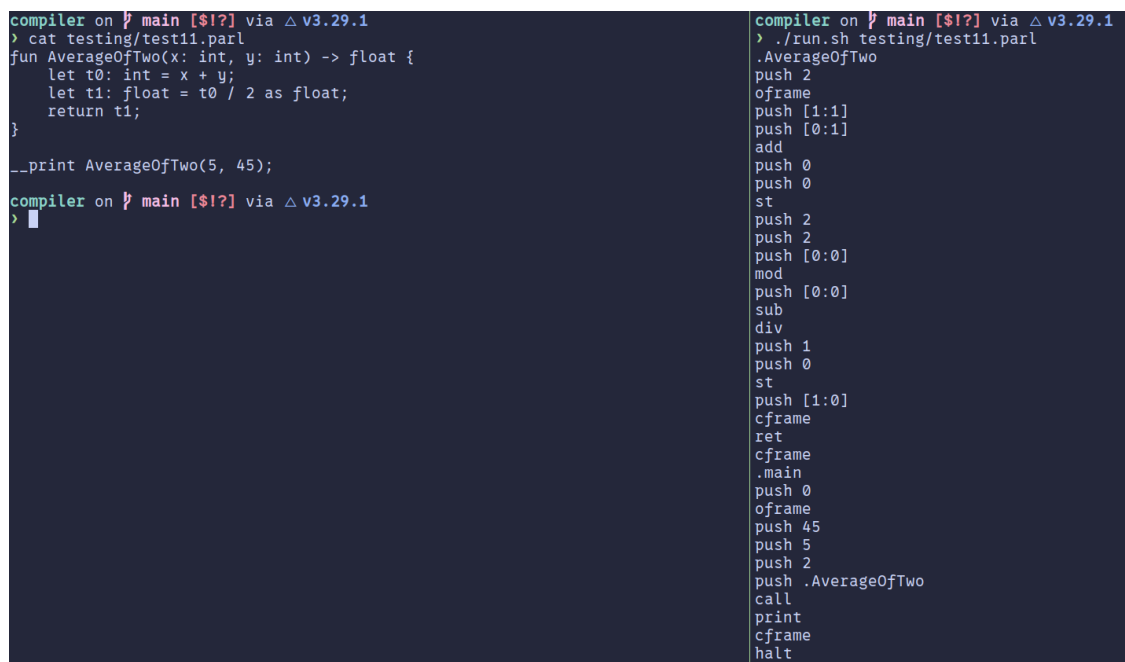
The tests mentioned above are:

- testing/test13.parl;
- testing/test14.parl;
- testing/test17.parl;
- testing/test18.parl (parameter types);
- testing/test19.parl (parameter types);
- testing/test21.parl (casting);
- testing/test23.parl (no main functions);
- testing/test24.parl (for nested functions);
- testing/test25.parl;
- testing/test28.parl (for unreachable code);

- testing/test31.parl (for arrays);
- testing/test32.parl;
- testing/test33.parl;
- testing/test41.parl (for unreachable code).
- testing/test41.parl (for unreachable code).
- testing/test45.parl (recursive call chain).

Some additional, tests have been run during the recording at, <https://drive.google.com/file/d/11HW-MxJbIX9eFRX1wTw9koo4Rzl0bTND/view?usp=sharing>.

6.3 | Code Generation



```

compiler on  main [$!?] via v3.29.1
> cat testing/test11.parl
fun AverageOfTwo(x: int, y: int) -> float {
  let t0: int = x + y;
  let t1: float = t0 / 2 as float;
  return t1;
}
__print AverageOfTwo(5, 45);
compiler on  main [$!?] via v3.29.1
>

```

```

compiler on  main [$!?] via v3.29.1
> ./run.sh testing/test11.parl
.AverageOfTwo
push 2
oframe
push [1:1]
push [0:1]
add
push 0
push 0
st
push 2
push 2
push [0:0]
mod
push [0:0]
sub
div
push 1
push 0
st
push [1:0]
cframe
ret
cframe
.main
push 0
oframe
push 45
push 5
push 2
push .AverageOfTwo
call
print
cframe
halt

```

Figure 24: Running test 11 (testing/test11.parl)

Code generation was tested using a number of semantically valid programs. The following is a list of such programs:

- testing/test11.parl;

- testing/test20.parl;
- testing/test26.parl;
- testing/test27.parl;
- testing/test29.parl;
- testing/test30.parl;
- testing/test34.parl;
- testing/test37.parl;
- and, testing/test45.parl.

Again, some additional examples are run in the video at <https://drive.google.com/file/d/11HW-MxJbIX9eFRX1wTw9koo4Rzl0bTND/view?usp=sharing>.

6.4 | Video

The video demonstrates the PArL compiler with a selection, of a few hand-picked examples, in particular: test30.parl, test37.parl, test41.parl and test45.parl.

The video can be viewed at the following URL: <https://drive.google.com/file/d/11HW-MxJbIX9eFRX1wTw9koo4Rzl0bTND/view?usp=sharing>.

7 | Limitations

The only known limitations of the compiler in relation to the specification is the lack of sytanx sugar for the array syntax as described in 2.1.