Compiler Theory and Practice

Coursework

Juan Scerri

123456A

April 2, 2024

A coursework submitted in fulfilment of study unit CPS2000.

Contents

Listings

Report

1 | Lexer

1.1 | Design & Implementation

The lexer was split into three-main components. A DFSA class, a generic table-driven lexer, and a lexer builder.

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. 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. These helpers specifically, `getInitialState()` is present since after building the DFSA there is no guarantee 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 is the `getTransition()` functions. In fact, it accepts a vector of transition categories instead of a single category.

This is because a symbol e.g. ‘a’, ‘9’ etc, might be valid for multiple categories. For instance ‘a‘ is considered to be both a letter and a number in hexadecimal.

The DFSA for accepting the micro-syntax `PArL` is built as follows.

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

The will use the following categories:

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

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:

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

Now, the following drawing describe the transitions of the DFSA. For improved readability the DFSA has been split across mulitple 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.

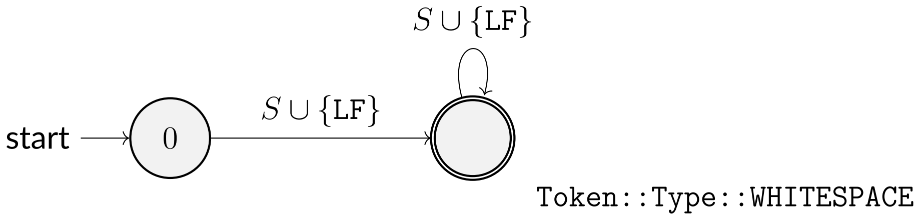


Figure 1: States & transitions for recognising whitespace

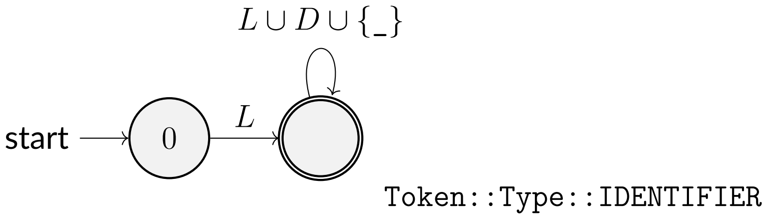


Figure 2: States & transitions for recognising identifiers/keywords

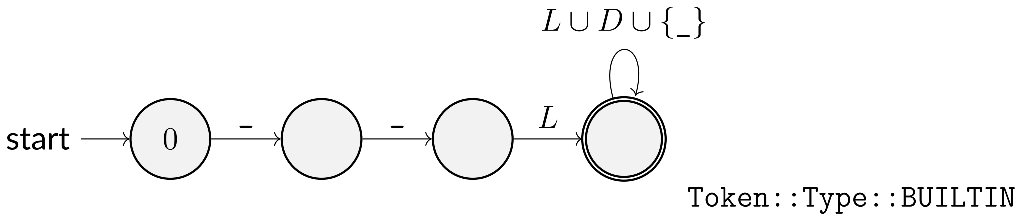


Figure 3: States & transitions for recognising builtins

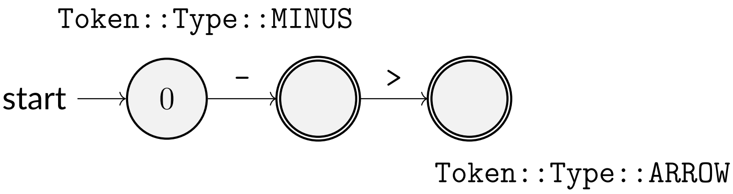


Figure 4: States & transitions for recognising minus and arrow (->)

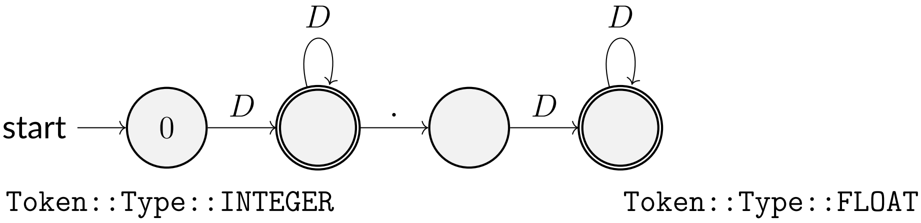


Figure 5: States & transitions for recognising integers and floats

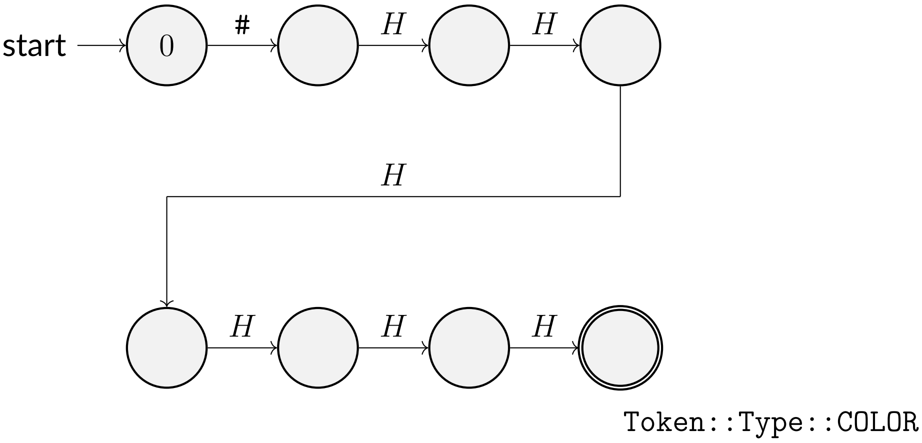


Figure 6: States & transitions for recognising colours

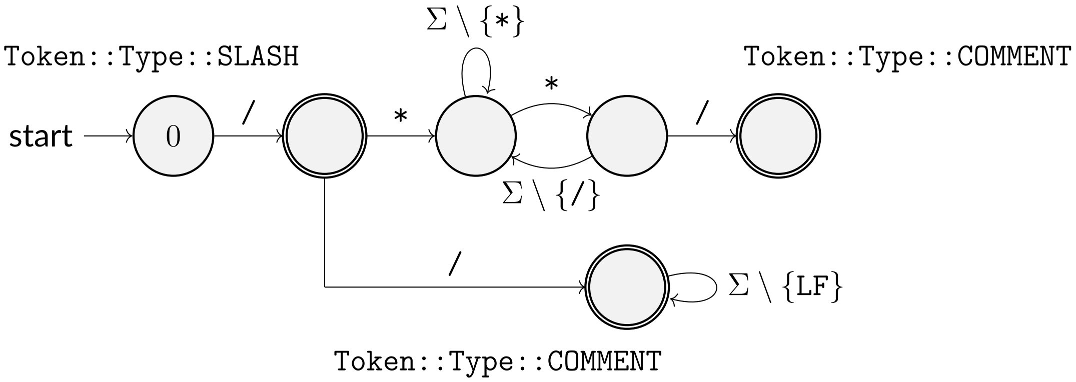


Figure 7: States & transitions for recognising slashes and comments

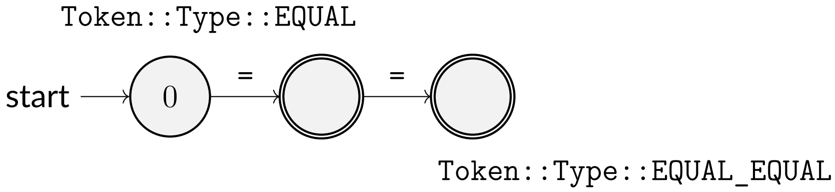


Figure 8: States & transitions for assign and is equal to

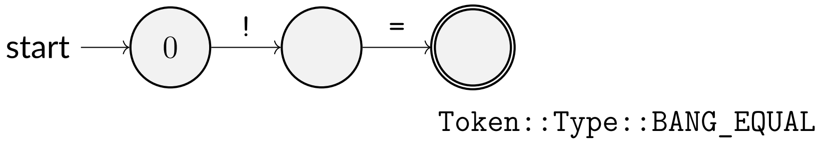


Figure 9: States & transitions for not equal to

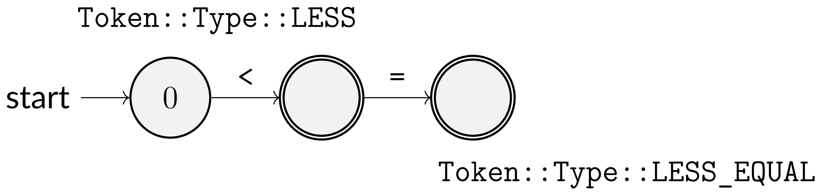


Figure 10: States & transitions for less than and less than or equal to

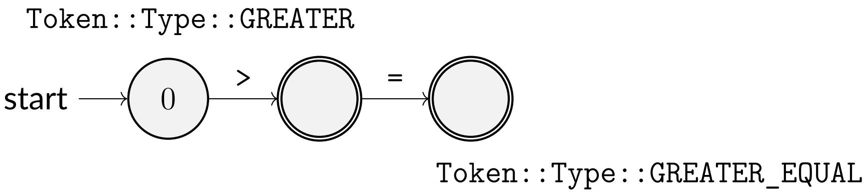


Figure 11: States & transitions for greater than and greater than or equal to

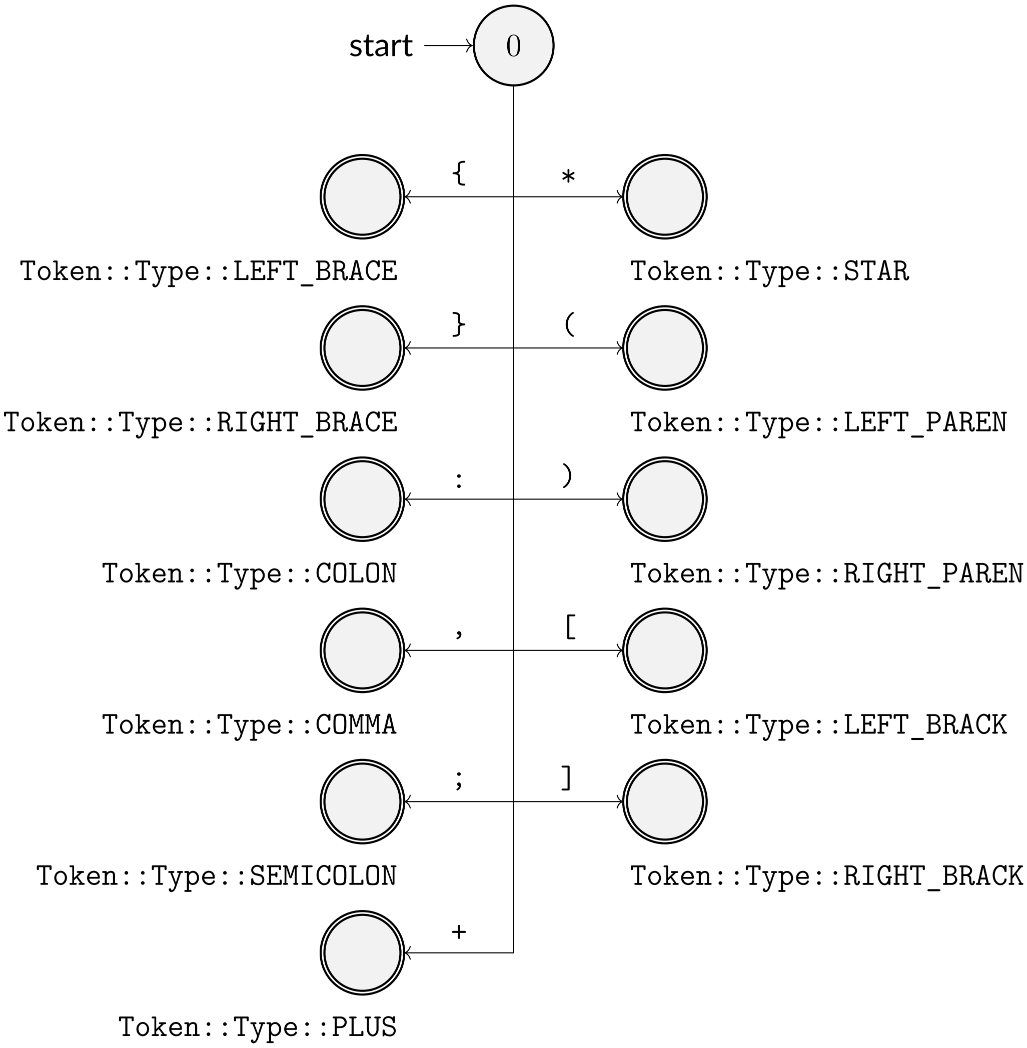


Figure 12: States & transitions for single letter tokens

The Builder & Director

Each sequence of states present 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 specification of the comments in the `LexerDirector` (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 the category indices are also registered by the user. See Listing 3 for a registration example. Additionally, the category indices although they are integers for readability they are defined as an 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 category checker (lexer/LexerDirector.cpp)

The second artefact produced by the builder is also a hash map from final states to their associated token type.

The transition table is then passed onto the DFSA. And the DFSA, and the two artefacts are passed onto the Lexer class.

```
    // create dfsa
    Dfsa dfsa(
        noOfStates,
        noOfCategories,
        transitionTable,
        initialStateIndex,
        finalStateIndices
    );

    // create lexer
    Lexer lexer(
        std::move(dfsa),
        std::move(categoryIndexToChecker),
        std::move(finalStateIndexToTokenType)
    );
```

Listing 4: Constructions of the Lexer (lexer/LexerBuilder.cpp)

The Actual Lexer

The lexer's core is as was described during the lectures and the core/main method is `simulateDFSA()`.

It also 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 the current stage and for later stages. This function is called every time a lexeme is consumed allowing the lexer to keep track of where in the file it is, 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)

Additionally, if an invalid / non-accepting state is reached the invalid lexeme is consumed and the user is warned, see Listing 6. After this the lexer, is left in a still operational state. Hence, `nextToken()` can be used again.

This is critical to provide users of the `PArL` compiler with a list of as many errors as possible, since it would be a bad experience to have to constantly run the `PArL` compiler to see the next error.

```
    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)

Hooking up the Lexer to the Runner

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

In this case the Runner passes in a reference to the lexer into the parser, this allows the parser to request tokens and they are computed on demand improving overall performance. Additionally, this has the benefit of allowing the parsing of larger and multiple files since, the parser is no longer limited by the amount of usable memory, since it does not need to load the whole file.

However, in this case no such optimisation is present.

```
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 passes `mLexer` into the Parser constructor (runner/Runner.cpp)

2 | The Parser

2.1 | Modified EBNF

Some modifications were applied to the original EBNF. Some of the modifications were either motivated by improved user experience, a more uniform mechanism and others to reduce complexity further down the pipeline.

```
⟨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⟩

⟨Literal⟩         ::= ⟨BooleanLiteral⟩
                   |  ⟨IntegerLiteral⟩
                   |  ⟨FloatLiteral⟩
                   |  ⟨ColorLiteral⟩
                   |  ⟨ArrayLiteral⟩
                   |  ⟨PadWidth⟩
                   |  ⟨PadHeight⟩
                   |  ⟨PadRead⟩
                   |  ⟨PadRandomInt⟩

⟨Type⟩            ::= (‘bool’ | ‘int’ | ‘float’ | ‘color’) [ ‘[’ ⟨IntegerLiteral⟩ ‘]’ ]
```

```
⟨SubEpxr⟩         ::= ‘(’ ⟨Epxr⟩ ‘)’

⟨Variable⟩        ::= ⟨Identifier⟩

⟨ArrayAccess⟩     ::= ⟨Identifier⟩ ‘[’ ⟨Epxr⟩ ‘]’

⟨FunctionCall⟩    ::= ⟨Identifier⟩ ‘(’ [⟨Epxr⟩ {‘,’ ⟨Epxr⟩}] ‘)’

⟨Epxr⟩            ::= ⟨LogicOr⟩ [‘as’ ⟨Type⟩]

⟨LogicOr⟩         ::= ⟨LogicAnd⟩ {‘or’ ⟨LogicAnd⟩}

⟨LogicAnd⟩        ::= ⟨Equality⟩ {‘and’ ⟨Equality⟩}

⟨Equality⟩        ::= ⟨Comparison⟩ {(‘==’ | ‘!=’) ⟨Comparison⟩}

⟨Comparison⟩      ::= ⟨Term⟩ {(‘<’ | ‘<=’ | ‘>’ | ‘>=’) ⟨Term⟩}

⟨Term⟩            ::= ⟨Factor⟩ {(‘+’ | ‘–’) ⟨Factor⟩}

⟨Factor⟩          ::= ⟨Unary⟩ {(‘*’ | ‘/’) ⟨Unary⟩}

⟨Unary⟩           ::= (‘–’ | ‘not’) ⟨Unary⟩ | ⟨Primary⟩

⟨RefExpr⟩         ::= ⟨Variable⟩
                   |  ⟨ArrayAccess⟩
                   |  ⟨FunctionCall⟩

⟨Primary⟩         ::= ⟨Literal⟩
                   |  ⟨SubExpr⟩
                   |  ⟨RefExpr⟩

⟨Program⟩         ::= {⟨Stmt⟩}

⟨Stmt⟩            ::= ⟨Block⟩
                   |  ⟨VaribaleDecl⟩ ‘;’
                   |  ⟨FunctionDecl⟩
                   |  ⟨Assignment⟩ ‘;’
                   |  ⟨PrintStmt⟩ ‘;’
                   |  ⟨DelayStmt⟩ ‘;’
                   |  ⟨WriteBoxStmt⟩ ‘;’
                   |  ⟨WriteStmt⟩ ‘;’
                   |  ⟨ClearStmt⟩ ‘;’
                   |  ⟨IfStmt⟩
```

```
                  |  ⟨ForStmt⟩
                  |  ⟨WhileStmt⟩
                  |  ⟨ReturnStmt⟩ ‘;’

⟨Block⟩          ::=  ‘{’ {⟨Stmt⟩} ‘}’

⟨VariableDecl⟩   ::=  ‘let’ ⟨Identifier⟩ ‘:’ ⟨Type⟩ ‘=’ ⟨Epxr⟩

⟨FormalParam⟩    ::=  ⟨Identifier⟩ ‘:’ ⟨Type⟩

⟨FunctionDecl⟩   ::=  ‘fun’ ⟨Identifier⟩ ‘(’ [ ⟨ForamlParam⟩ {‘,’ ⟨FormalParam⟩}] ‘)’
                      ‘->’ ⟨Type⟩ ⟨Block⟩

⟨Assignment⟩     ::=  ⟨Identifier⟩ [‘[’ ⟨Epxr⟩ ‘]’] ‘=’ ⟨Epxr⟩

⟨PrintStmt⟩      ::=  ‘__print’ ⟨Epxr⟩

⟨DelayStmt⟩      ::=  ‘__delay’ ⟨Epxr⟩

⟨WriteBoxStmt⟩   ::=  ‘__write_box’ ⟨Epxr⟩‘,’ ⟨Epxr⟩‘,’⟨Epxr⟩‘,’ ⟨Epxr⟩‘,’⟨Epxr⟩

⟨WriteStmt⟩      ::=  ‘__write’ ⟨Epxr⟩‘,’ ⟨Epxr⟩‘,’⟨Epxr⟩

⟨ClearStmt⟩      ::=  ‘__clear’ ⟨Epxr⟩

⟨IfStmt⟩         ::=  ‘if’ ‘(’ ⟨Expr⟩ ‘)’ ⟨Block⟩ [‘else’ ⟨Block⟩]

⟨ForStmt⟩        ::=  ‘for’‘(’ [⟨VariableDecl⟩] ‘;’ ⟨Expr⟩ ‘;’ [⟨Assignment⟩] ‘)’ ⟨Block⟩

⟨WhileStmt⟩      ::=  ‘while’ ‘(’ ⟨Expr⟩ ‘)’ ⟨Block⟩

⟨ReturnStmt⟩     ::=  ‘return’ ⟨Expr⟩
```

Improved Precedence

So, the minor changes which improve programmer usability are the additions of a number of other expression stages, such as ⟨LogicOr⟩, ⟨LogicAnd⟩, 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, and 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 ⟨Type⟩ and ⟨Literal⟩ productions were augmented to improve array support. This helped simplify the ⟨Identifier⟩ (in the original EBNF) , ⟨VariableDecl⟩ and ⟨FormalParam⟩ productions. Additionally, this opens up further support for more complicated types later on.

For example, the ⟨Type⟩ can be further augmented to support more types.

```
⟨StructField⟩   ::= ⟨Identifier⟩ ‘:’ ⟨Type⟩

⟨Struct⟩        ::= ‘struct’ ⟨Identifier⟩ ‘{’ {⟨StructField⟩ ‘;’} ‘}’

⟨TypeDecl⟩      ::= ⟨Struct⟩

⟨Base⟩          ::= (‘bool’ | ‘int’ | ‘float’ | ‘color’)

⟨Array⟩         ::= ⟨Type⟩ ‘[’ ⟨IntegerLiteral⟩ ‘]’

⟨Pointer⟩       ::= ⟨Type⟩ ‘*’

⟨Type⟩          ::= ⟨Identifier⟩
                 |  ⟨Base⟩
                 |  ⟨Array⟩
                 |  ⟨Pointer⟩
```

Note that we are indeed repeating ⟨ForamlParam⟩ but this is not really a problem. When it comes to specification 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 structure (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 the ⟨Array⟩ as described above are not easily representable within C++ since, something like Listing 10, is not allowed.

```
struct SelfRef;
```

```cpp
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 tough it were a value.

The `box<>` type is attributed to Jonathan Müller were he discuss the exact issue described above on his blog, `foonathan`.

These changes to type also require additional, changes to how variables are referenced in expressions hence why ⟨RefExpr⟩ was added. It also ensure that in the future more complicated referencing such as `object.something` (`struct` member referencing) is possible.

Finally, the ⟨ArrayLiteral⟩ has been improved to support expressions instead of just only literals.

Removing Eye-Candy

Adding this system however, significantly increases the complexity of semantic analysis.

Because of this the slight syntax sugar of being able to have the following two conveniences `let a: int[] = [1,1,1];` and `let a: int[3] = [1];` has been dropped.

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 this is to have a de-sugaring & type inference sub-phase before type checking, within the semantic analysis phase.

2.2 | Parsing & The Abstract Syntax Tree

The AST is the data structure which is produced by the parser. The nodes of the AST are in fact almost a one-to-one representation of the productions in the grammar, for example see Listing 11.

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

    void accept(Visitor*) override;

    const Position position;
    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 in the `Position` field. This get populated by the parser using the location of a token in the original source file. This is again critical for adequate error messaging later in the semantic analysis phase.

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

    void accept(Visitor*) override;

    const Position position;
    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(const Position&, Operation, std::unique_ptr<
   Expr>);
```

```
    void accept(Visitor*) override;

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

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

Apart from these the only other real difference is the use of a `Binary` and `Unary` node see Listing 12 and Listing 13, instead of a node for each type of the grammar rules discussed above in 2.1. This is because the reason for said rules was precedence and precedence within the AST is actually described by the structure of the tree itself not the node types.

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 form the parser is the backbone of the remaining phases.

2.3 | The Actual Parser

The parser is split into four main sections.

- AST Generation
- Token Buffering
- Token Matching
- Error Handling/Recovery

Token Buffering

The parser requires access to the tokens for proper functioning. Additionally, sometimes the parser requires more than one token as the case deciding on whether an identifier is just a variable or a function call. This is quite easy to implement if the parser has available to it at initialisation all the tokens.

However, as described in 1.1. This is not the case the parser requests token on demand from lexer which it has a reference. This of course means that the machinery for handling tokens is a bit more complicated due to requiring lookahead.

To solve this issue a window-based approach was adopted. The parser has a moving buffer/window called mTokenBuffer and 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.

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

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

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

Token Parser::nextToken() {
    Token token;

    do {
        token = mLexer.nextToken().value();
    } while (token.getType() == Token::Type::WHITESPACE ||
             token.getType() == Token::Type::COMMENT);

    return token;
}
```

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

Additionally, note that within nextToken() the whitespace and comments are being explicitly ignored.

This might lead to further minor improvement. Specifically, comments can be integrated into the AST. This basically allows printing visitors or, formatting visitors properly format the code whilst still preserving any comments created by programmers.

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 improtant of these methods is `consume()`, it takes in a token type and a error message if the specified token type is not matched, see Listing 15.

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

Listing 15: The `consume()` Parser methods (parser/Parser.hpp)

The main reason for the templating of 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 16.

```
consume(
    Token::Type::IDENTIFIER,
    "expected identifier token "
    "instead received {}",
    peek().toString()
);
```

Listing 16: The 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. The main usage for this is cases where more than different token is valid option for example such is the case for ⟨Comparison⟩ production. `match()` 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. The only issue is that all calls to `peek()` have to be checked to ensure that valid access, see Listing 17. This is done using `abort_if()` function which prints an error message an aborts the program. This is very similar to a `static_assert` however, it is has to be performed at runtime, see Listing 18.

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

    return mTokenBuffer[lookahead];
}
```

Listing 17: 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 18: The abort functionality present in the codebase (parl/Core.hpp)

Note that 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 implementation 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 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. This processes is referred to as Synchronization by the author of Crafting Interpreters, Robert Nystrom. The method for synchronization in the `PArL` compiler was inspired by the above credited author and his usage of Exceptions as an unrolling primitive although unorthodox is very simple to implement and has in fact been used in the parser and also later stages of the compiler with success.

```
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 19: The `ParseSync` exception and the `error()` method which kickstarts the synchronization process (parser/Parser.hpp)

However, there is of course a major downside to this where synchronization 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 accepted. The basic process of synchronizing is consuming as many tokens as possible until the parser reaches a token which it believes to be a good restarting point, see Listing 20.

```
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 */
                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 20: The `synchronize()` method in the Parser class (parser/Parser.cpp)

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

AST Generator Methods

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

See, Listing 21 for an example of such a method and see Listing 22 for a full list of the 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 std::make_unique<core::IfStmt>(
        token.getPosition(),
        std::move(cond),
        std::move(thenBlock),
        std::move(elseBlock)
    );
}
```

Listing 21: 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 22: The main body of methods in the Parser class (parser/Parser.hpp)

2.4 | Pretty? Printing

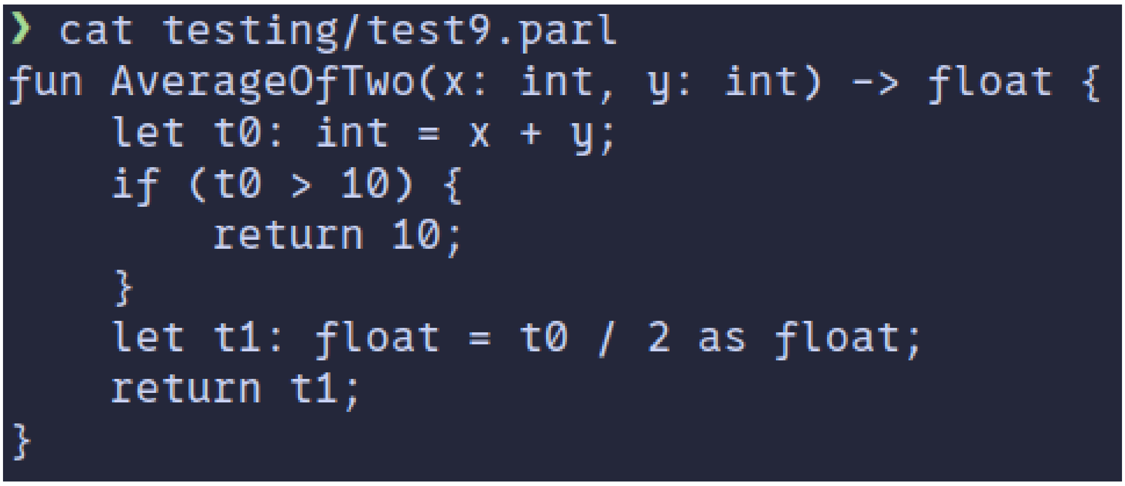
Using Parser in the Runner

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());
    }
```

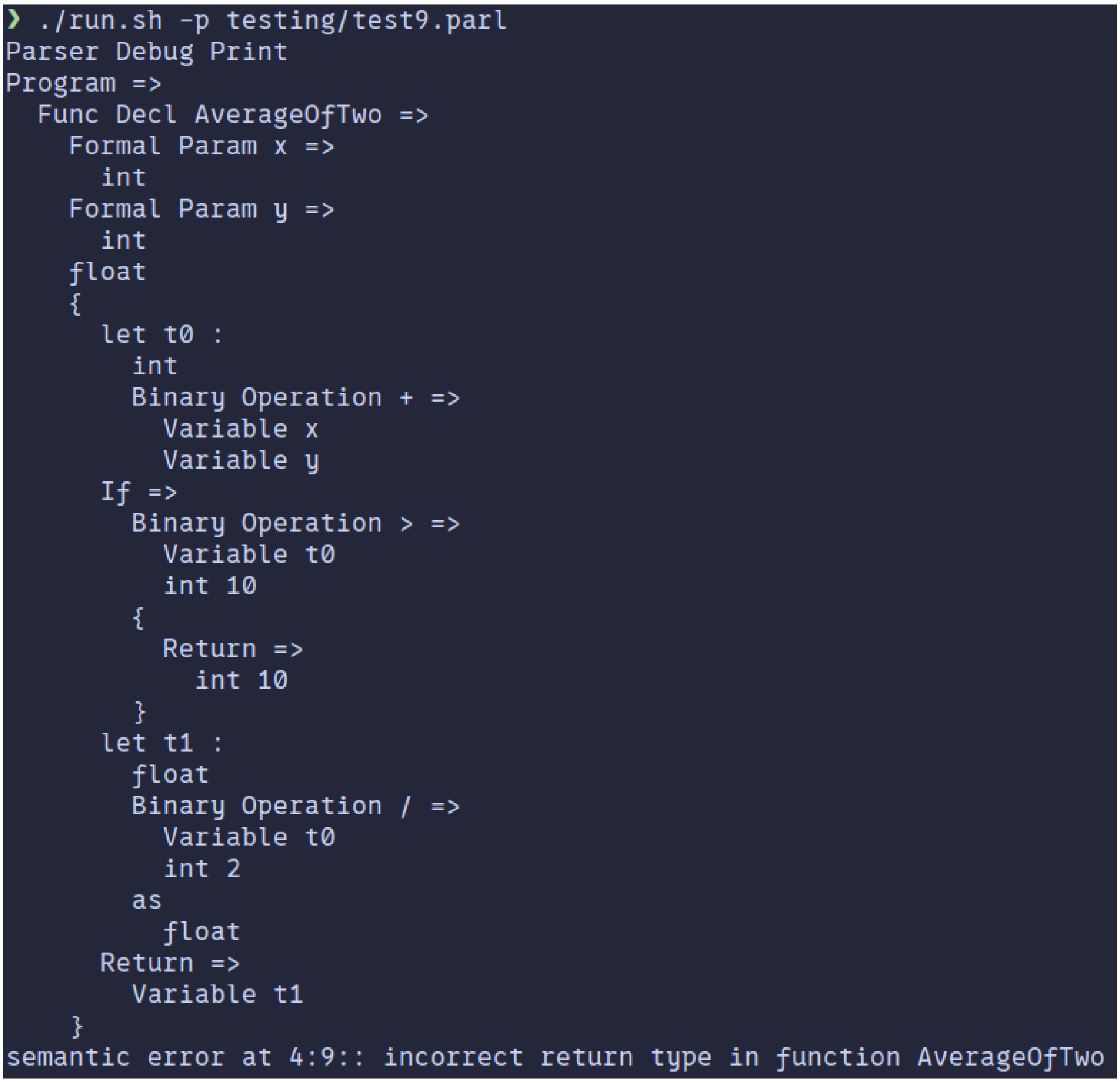
Listing 23: The parse segment of the `run()` method in the Runner class (runner/Runner.cpp)

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 test9.parl

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

    PrinterVisitor printer;

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

Listing 24: The `debugParsing()` method in the Runner class (runner/Runner.cpp)

The `PrinterVisitor` used in Listing 24 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 25: 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 25. 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 26, 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 26: 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 step phase. There a 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 since said phase is completely disjoint from the phases after it 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.

You would want to do so to allow for the location of function declarations in code to not effect resolution, that is, a function can be called before it is referenced.

Unfortunately, due to the fact that compiler development was an organic process and not too much time was spend on deliberation the semantic analysis phase became monolithic, and hence it suffers from the issues described above. Of course, this means location agnostic function declaration are currently *not* supported by the compiler.

3.2 | The Environment Tree

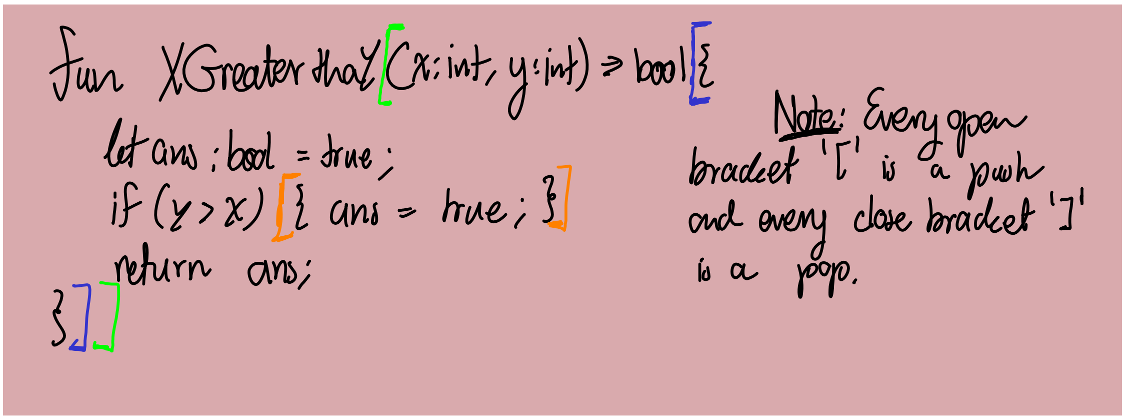
Some terminology is required to properly describe the number of structures which will be used in this section. The following terms are critical and need to be differentiated properly:

- `SymbolTable`
- `SymbolTableStack`
- `Environment`
- `EnvStack`
- `RefStack`

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

The most basic form of this is a single `SymbolTable`. A symbol table in its simplest form is a wrapper around a hash map. Of course this is quite a limiting structure. In the context of a full program usage of such a structure for semantic analysis will restrict the users of the language to be careful regarding their naming as everything in the language would be in global scope.

Therefore, this is not even a sufficient solution for most toy-languages let alone full-blown production ready languages. The solution to this problem is the use of a stack of symbol tables. This allows symbol tables to shadow each other allowing for the reuse of names. Additionally, this further opens up the possibility of implementing language level modules. The likelyhood 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 allows multiple modules and the main executable to co-exist.



```
fun XGreaterThanY[(x:int, y:int) -> bool[{
    let ans : bool = true;
    if (y > x) [{ ans = true; }]
    return ans;
}]]
```

Note: Every open bracket '[' is a push and every close bracket ']' is a pop.

Figure 15: A PArL function with annotated scopes

The basic premise of using a symbol table stack is described in

Algorithm 1: Basic description of `SymbolTableStack` usage

```
Data: $N$ the AST node, $S$ the SymbolTableStack
begin
    $V \longleftarrow U$;
    $S \longleftarrow \emptyset$;
    for $x \in X$ do
        $NbSuccInS(x) \longleftarrow 0$;
        $NbPredInMin(x) \longleftarrow 0$;
        $NbPredNotInMin(x) \longleftarrow |ImPred(x)|$;
    end
    for $x \in X$ do
        if $NbPredInMin(x) = 0$ and $NbPredNotInMin(x) = 0$ then
            $AppendToMin(x)$
        end
    end
1   while $S \neq \emptyset$ do
REM     remove $x$ from the list of $T$ of maximal index;
2       while $|S \cap ImSucc(x)| \neq |S|$ do
            for $y \in S - ImSucc(x)$ do
                { remove from $V$ all the arcs $zy$ : };
                for $z \in ImPred(y) \cap Min$ do
                    remove the arc $zy$ from $V$;
                    $NbSuccInS(z) \longleftarrow NbSuccInS(z) - 1$;
                    move $z$ in $T$ to the list preceding its present list;
                    {i.e. If $z \in T[k]$, move $z$ from $T[k]$ to $T[k-1]$};
                end
                $NbPredInMin(y) \longleftarrow 0$;
                $NbPredNotInMin(y) \longleftarrow 0$;
                $S \longleftarrow S - \{y\}$;
                $AppendToMin(y)$;
            end
        end
        $RemoveFromMin(x)$;
    end
end
```

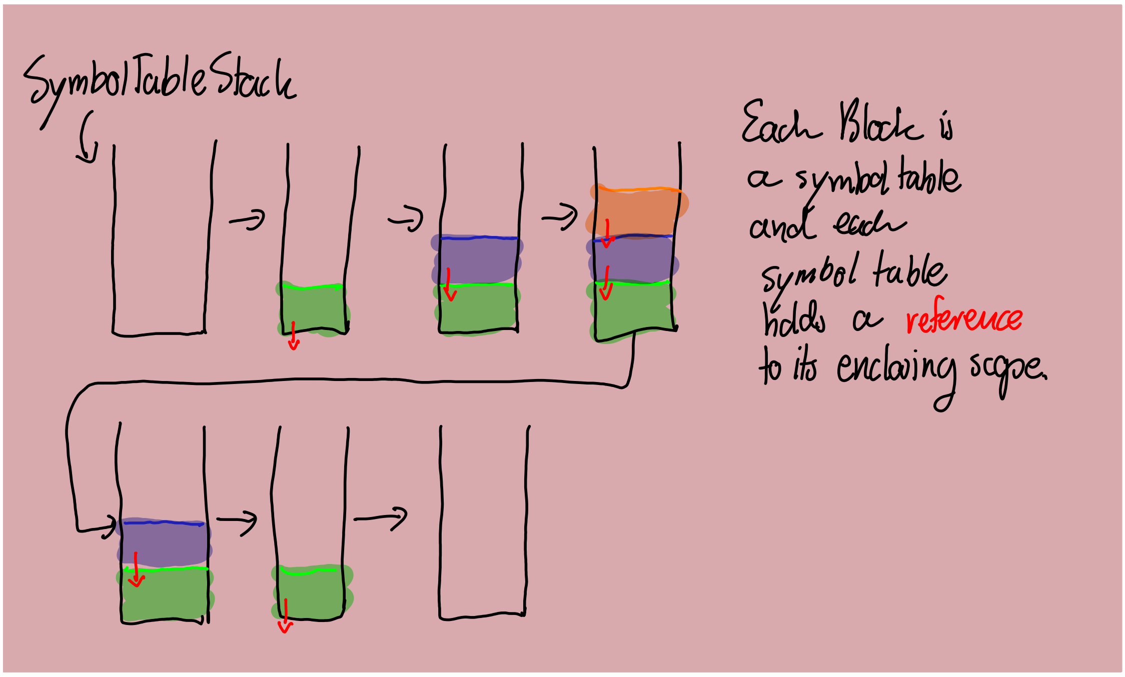

SymbolTableStack
Each Block is a symbol table and each symbol table holds a reference to its enclosing scope.

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

Arguably the most important part of the later stages of a compilation is the Environment tree.

The Environment tree is a data structure

- Describe Symbol Table.
- Describe each element of the symbol table.
- Describe the tree approach.
- attribute the tree approach to Robert Nystrom as well.
- describe the main advantage/disadvantage of using a environment approach.
- describe the main advantage/disadvantage of using a symbol stack approach.

4 | Attributions

- Sandro Spina for the brilliant description of table-driven lexers

- Robert Nystrom and his great book Crafting Intepreters for a great outline for parsing and error recovery/management for languages which support exceptions