# The PixARLang compiler

**CPS2000 Compiler Theory and Practice**

Mark Mizzi

Last edited: June 2, 2023

## Contents

```
/
├─ src
│  ├─ util.hh .............. Miscellaneous utilities used by several parts of the compiler.
│  ├─ location.hh .......... Definition of Location type used to track source code locations.
│  ├─ lexer.hh ............. Code for the table-driven lexer. Described in Section 2
│  ├─ lexer.cc
│  ├─ parser.hh ............ Code for the parser. Described in Section 3.4
│  ├─ parser.cc
│  ├─ ast.hh ............... Code for the AST representation. Described in Section 3.2
│  ├─ ast.cc
│  ├─ visitor.hh ........... Abstract class definition of a visitor. Described in Section 3.3
│  ├─ xml_visitor.hh ....... Code for XML output of an AST. Described in Section 4
│  ├─ xml_visitor.cc
│  ├─ semantic_visitor.hh .. Code for semantic checking. Described in Section 5
│  ├─ semantic_visitor.cc
│  ├─ codegen.hh ........... Code for generating PixIR bytecode. Described in Section 6
│  ├─ codegen.cc
│  ├─ deadcode.hh .......... Code for dead code elimination. Described in Section 6.3.2
│  ├─ deadcode.cc
│  ├─ peephole.hh .......... Code for peephole optimization. Described in Section 6.3.3
│  ├─ peephole.cc
│  ├─ compiler.hh .......... Definition of a Compiler class which orchestrates compilation.
│  └─ main.cc .............. Main program, contains code for parsing user options.
├─ examples ................ Examples of programs written in the PixAR language.
├─ report .................. LaTeX source code for the report.
└─ CMakeLists.txt .......... CMake file used to build the project.
```

Figure 1: Organization of source code into files.

## 1 Introduction

This report describes the implementation of a compiler for the PixAR langauge targeting the PixAR virtual machine.

### 1.1 Language and tools used

The code for the project is written using C++17, and makes use of several modern features in the language, including structured binding, and the `std::variant` container.

The CMake build system is used to manage and compile the source code.

### 1.2 Code organization

The source code itself can be found in the `src/` subdirectory. There are also some example PixAR programs used to test the compiler in the `examples/` subdirectory.

This report is written in LaTeX and its source can be found in the `report/` subdirectory of the project.

Figure 1 shows a bird's eye view of the source code organization, including helpful links to the relevant parts of the report.

## 1.3 Building the project

In order to build the project, run the following commands from the root directory:

```
cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON -DCMAKE_BUILD_TYPE=Release .
make
```

This report can be compiled by running the following commands:

```
cd report
make
```

## 1.4 Video presentation

A video presentation of the compiler implemented was developed, and can be found on my Google drive.

# 2 Lexing

## 2.1 Types defined for lexing

### 2.1.1 Locations

Both tokens and AST nodes carry locations indicating the source code range which produced them.
A `Location` struct is used for this purpose. This struct contains `sline`, `scol`, `eline` and `ecol` fields, representing the starting and ending line and coloumn in the source code respectively.

`Location` structs also have a `merge()` method, which can be used to combine two `Locations` into a new one which represents the entire range specified by the minimum and maximum bounds of the merged `Locations`.

This method is used during parsing for example, to combine `Locations` of child nodes and tokens into the `Location` of a new AST node.

During parsing or semantic checking, errors are reported by throwing an exception of type `ParserError`, or `SemanticError` respectively. Each of these exception types takes a `Location` argument in its constructor, and uses it to indicate the source code location of the error in its message.

In the case of parsing, the `Location` is sourced from the token being consumed when the parse error occurs. During semantic checking, the `Location` is sourced from the AST node which failed the check.

### 2.1.2 Tokens

Tokens are represented by a `Token` struct which contains three pieces of information:

- A `TokenType` field indicating the kind of token (this is an `enum`).
- A `std::string` containing the text constituting the token,
- and a `Location` field which indicates the location of the token's text in the source code.

## 2.2 Lexing approach

The PixAR compiler uses a hand coded, table driven lexer. Much of the code is encapsulated in a `Lexer` class.

The table driven approach[1] assumes that the microsyntax of each token type $t_i$ can be specified using a regular expression $e_i$. The lexer emulates a particular DFSA (deterministic finite state automaton) which accepts the language specified by the following regular expression:

$$e = e_1|...|e_k$$

A portion of the DFSA emulated by this compiler's lexer is shown in Figure 2.

The transition table of the emulated deterministic finite state automaton is stored in an auxiliary data structure.

When a token needs to be lexed, a "skeleton" function simulates a run of the DFSA using this data structure; the initial state is set to a specified start state, and input is consumed until there is no more, or until the transition table indicates that there is no outgoing transition from the current state with the current character. The "skeleton" function is greedy, as it consumes as much input as possible.

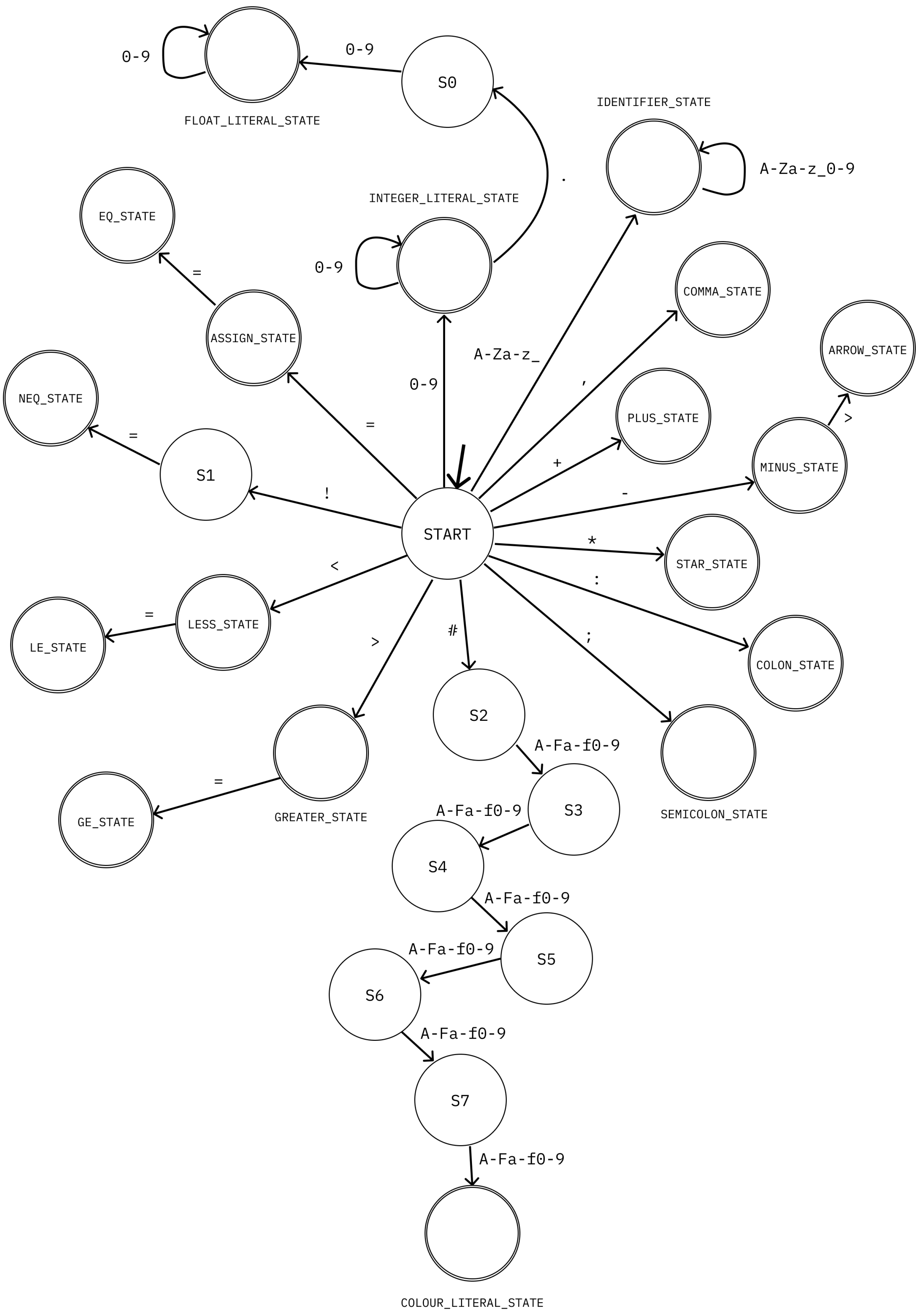


Figure 2: A portion of the partial DFSA on which the PixAR compiler's lexer is based. For simplicity the states for some of the simpler tokens, as well as the handling of whitespace and comments is ommitted from the diagram.

In our implementation, this "skeleton" function is contained in the `nextToken()` method of a `Lexer`. Pseudocode for this method is shown in Algorithm 1.

DFSA states are represented by a `LexerState` `enum`.

**Algorithm 1:** Pseudocode for the skeleton function at the core of the table-driven lexer.

**Data:**

- *input* is the input stream.
- *line* is the line number of the source line being lexed.
- *col* is the column number of the source location being lexed.
- *c* is the last character obtained from the input stream.
- *token* is the result; the token lexed from an initial segment of *input*. In the pseudocode, tokens are represented by a pair, with the first element being the token type, and the second being the value (the input characters which constitute the token).
- *state* is the current state of the lexer.
- *transTable* is the hard coded transition table of the lexer.

```
if isEmpty(input) then
    token ← $\langle END, \epsilon \rangle$;
else
    state ← START;
    while $\neg$isEmpty(input) do
        c ← getNext(input);
        cclass ← characterClass(c);
        if $\langle state, cclass \rangle \notin transTable$ then
            putBack(input, c);
            break;
        end
        state ← transTable[$\langle state, cclass \rangle$];
        token[1] ← token[1] + c;
        if isNewline(c) then
            line ← line + 1;
            col ← 0;
        else
            col ← col + 1;
        end
    end
    if $\neg$isFinalState(state) then
        Deal with lexer error.
    else
        token[0] ← finalStateToTokenType(state);
    end
end
```

Several simplifying decisions were made in implementing the lexer.

Firstly, each valid input character was classified into a character class. This is a standard approach[1].

Character classes are mutually exclusive, and group together characters which result in the same state transitions for each state of the DFSA. For example, the digit characters $0 - 9$ were classified into a single character class. This greatly decreases the size of the table used.

Choosing character classes is not as straightforward as it may seem, and new classes may be needed as more token types are added to the language. For example, although one may think it sufficient to create a single class for the alphabetic characters $[a - zA - Z]$, two classes are needed; $[a - fA - F]$ and $[g - zG - Z]$, as the former characters can appear in hexadecimal literals, whereas the latter cannot.

Secondly, a `std::map` data structure was used to store the transition table. This allowed us to encode the **partial** DFSA shown in Figure 2 directly. Transitions that would lead to an error state are ommitted from this map.

The lexing loop exits when there is no more input to process, or when there is no outgoing transition from the current state with the character class of the current character. In this case, the last state of the lexer may be a final state, indicating successful lexing, or a non-final state, indicating a lexing error.

The difference is determined in the `tokenType()` method, which takes a DFSA state and returns the associated token type if it is a final state, throwing a `std::logic_error` otherwise. Any `std::logic_error` thrown by this method is caught in `nextToken()`, and re-thrown as a `LexerError`.

The use of an array for storing the table would require a total DFSA to be implemented, which necessitates an error state and transitions to it. In addition, the destination states for each transition would have to be referenced using a numerical index rather than an `enum`, making the code harder to debug.

Finally, keywords are not handled directly in the "skeleton" function. Instead keywords are lexed as regular identifiers, and then filtered by a public `getNextToken()` method which wraps the (private "skeleton" function. As a side effect of this design decision, `getNextToken()` must also reject any identifiers which start with an underscore, which is disallowed by the language specification. This cannot be done by the "skeleton" function itself as certain keywords start with an underscore.

`getNextToken()` also has another responsibility; it filters tokens which have a `WHITESPACE_TOK` token type out of the token stream. Such tokens are generated by `nextToken()` when it encounters comments or whitespace. Dealing with whitespace and comments *after* lexing allows us to retain accurate locations in all of the produced `Token`s.

Instead of a `std::string`, the `Lexer` class represents its input stream using a `std::istream&`, which is passed on construction. This means that, for example, a `std::fstream` can be used to feed input from a file to the `Lexer` incrementally, potentially saving memory in cases where a lexing error is thrown halfway through reading the file.

## 2.3 Error handling

Errors produced during lexing are handled by throwing a `LexerError`. This exception type subclasses the generic `CompilationError` type, and its constructor takes two `size_t` parameters which indicate the location of the character that caused the lexing error in the source code. This location is included in `LexerError`'s error message.

There are two situations which can cause errors; these have already been mentioned in passing above.

Firsly, the `nextToken()` function may terminate the lexing loop while the current state is not final. Secondly, an identifier which starts with an underscore character may be lexed, and then rejected by `getNextToken()`.

# 3 Parsing

## 3.1 Modifications made to EBNF grammar

The version of the PixAR language implemented is an extension which supports dynamically sized arrays.

A few modifications had to be made to the EBNF specified in the assignment sheet in order to achieve this.

Array type annotations were added by extending the *Type* nonterminal with a recursive production:

⟨*Type*⟩ ::= ‘`float`’ | ‘`int`’ | ‘`bool`’ | ‘`colour`’ | ‘`[`’ ‘`]`’ ⟨*Type*⟩

Two new types of expression nonterminals were added to the productions of *Factor*:

⟨*Factor*⟩ ::= ... | ⟨*ArrayAccessExpr*⟩ | ⟨*NewArrExpr*⟩

These nonterminals represent expressions used to access and create arrays respectively, and are defined by the following rules:

⟨*ArrayAccessExpr*⟩ ::= ( ⟨*Identifier*⟩ | ⟨*ArrayAccessExpr*⟩ ) ‘`[`’ ⟨*Expr*⟩ ‘`]`’

⟨*NewArrExpr*⟩ ::= ‘`__newarr`’ ⟨*Type*⟩ ‘`,`’ ⟨*Expr*⟩

Array access expressions can be used as lvalues. In order to express this in the grammar, the *Assignment* nonterminal was redefined as

⟨*Assignment*⟩ ::= ⟨*Lvalue*⟩ ‘=’ ⟨*Expr*⟩

with *Lvalue* being a new nonterminal defined as

⟨*Lvalue*⟩ ::= ⟨*ArrayAccessExpr*⟩ | ⟨*Identifier*⟩

## 3.2 Abstract syntax trees

Abstract syntax trees are represented by the inheritance hierarchy shown in Figure 3.

Every node type inherits from an `ASTNode` abstract class. This class contains a `Location` member that stores the position of the source code that produced the node. As mentioned in Section 2.1.1, this allows the compiler to produce very useful error messages which indicate the location of the error to the user.

The `ASTNode` type also contains a pure virtual method `accept()`, that subclasses must implement to support visitors. Another pure virtual method `children()` is implemented by subclasses to return a `std::vector` of pointers to their child nodes.

There are three direct subclasses of `ASTNode`, which are themselves abstract: `StmtNode`, `ExprNode`, `TypeNode`. These three subclasses represent the three fundamental kinds of syntactic structures in SIPLang: statements, expressions, and types.

Concrete subclasses of these three classes use `std::unique_ptr` to reference child nodes.

Type nodes are used extensively in semantic checking (see Section 5.2.1), and in this context, copies of a `TypeNode` may be needed in order to temporarily map an expression AST to a type. For this reason, `TypeNode` has a pure virtual `copy()` method which is implemented by each subclass.

Note that `copy()` must implement deep copying, due to the use of `unique_ptr`. However, copying is rarely needed, and is considered a small cost compared to the convenience of the automatic deallocation offered by `unique_ptr`.

In addition, type nodes also implement the equality operator ==. In order to be useful, the implementation of this operator must be able to compare instances of any two (potentially different) subclasses of `TypeNode`. This requires an unusual pattern.

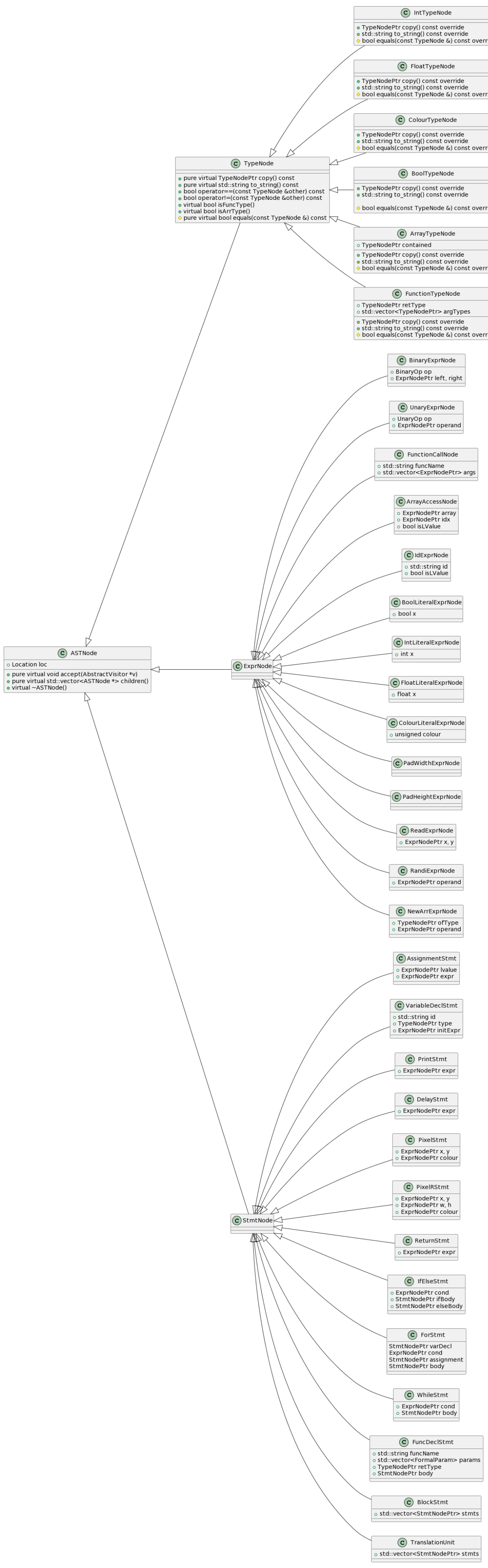


Figure 3: Inheritance hierarchy for AST nodes. Some methods are ommitted for brevity.

The implementation of operator== in TypeNode first compares the typeid of the two nodes passed to it, and returns false if they are not the same. This handles the case where the two TypeNodes are of different types.

If the typeid check succeeds, a pure virtual equals() method is called.

In simple subclasses of TypeNode, such as IntTypeNode or FloatTypeNode, this method unconditionally returns true, as any two instances of such simple nodes are semantically equivalent.

For more complex subclasses, such as ArrayTypeNode, the equals() method statically casts its argument to the type of the subclass. This is type safe provided equals() is only called by operator==, since the latter calls it if and only if the typeid check succeeds. The equals() method can then perform subclass-specific equality checks, (in this case making sure the type of array elements is the same). In order to prevent abuse, the equals() method is declared as protected.

A subclass of TypeNode called FunctionTypeNode is used to represent function types. Nodes of this type are not produced during parsing, but are used to represent the type of function symbols during semantic checking. This allows us to treat function and variable symbols on the same footing in the semantic checker, greatly simplifying the code.

Subclasses of TypeNode also implement isFuncType() and isArrType() methods which can be used to check whether a TypeNode is an ArrayTypeNode or a FunctionTypeNode respectively.

## 3.3 The Visitor Pattern

The visitor pattern is widely used in compilers to cleanly separate the code for the AST representation from the implementation of algorithms that need to traverse it.

The SIPLang compiler employs a fairly pedestrian implementation of the visitor pattern. Visitors subclass an AbstractVisitor class, which has an overloaded pure virtual visit() method for each concrete derived class of ASTNode:

```
class AbstractVisitor {
public:
  virtual void visit(IntTypeNode &node) = 0;
  // ...
  virtual void visit(TranslationUnit &node) = 0;
```

As mentioned above, ASTNode contains a pure virtual method called accept(), which takes a pointer to an AbstractVisitor instance. Each concrete derived class of ASTNode overrides accept() to call the version of the visit() method that accepts its type:

```
  void accept(AbstractVisitor *v) override { v->visit(*this); }
```

In addition to the fundamental visit() methods, AbstractVisitor has concrete utility methods called visitChildren() and rvisitChildren().

These methods both take a pointer to an ASTNode instance, and use its children() method to visit each of its child nodes, by calling the accept() method on each of the child nodes. rvisitChildren traverses the std::vector of children in reverse order[1].

Concrete subclasses of AbstractVisitor are used to generate XML for a parsed AST, for semantic checking, as well as for code generation.

[1]One example of where this proves useful is in code generation, where code for arguments to a function or subexpressions of a binary operator often has to be generated in reverse order of declaration, due to the stack based semantics of the VM. See Listing 5 for an example.

## 3.4 Design considerations for the parser

The compiler uses a recursive descent parser[1] which directly produces the AST representation.

Parse errors are handled by throwing an instance of ParserError. This class inherits from the generic CompilationError class, and reports an error message along with the location in the source file where the error originated.

Logic for the parser is encapsulated in a Parser class, which amongst other things holds a reference to an instance Lexer which is used to obtain a token stream.

Parser also has two methods which are fundamental to parsing; peek() and consume().

peek() takes an integer parameter i, and looks ahead i tokens in the token stream. Since tokens are obtained one by one from the lexer, and there is no way to look-ahead without consuming a token, the parser keeps an internal queue of tokens. If a lookahead smaller than the size of the queue is requested, peek() simply returns the appropriate token from the queue. Otherwise, the method consumes an appropriate number of tokens from the lexer, pushes them onto the queue and returning the last one.

consume() uses the same internal queue. If the latter is non-empty, it pops and returns the token at the front of the queue. Otherwise, it gets the next token from the lexer, and returns it.

Parsing is carred out by a set of methods with the prefix parse. These parsing methods interact with the token stream exclusively through peek() and consume(), allowing for neat seperation of concerns.

Due to the limited expressivity of SIPLang, the largest lookahead required is of 2 tokens. This lookahead of 2 tokens is used to differentiate between function calls (in which case a lookahead of 2 yields a "(" token), array accesses (in which case a lookahead of 2 yields a "[" token), or identity expressions.

In order to aid parsing, a CHECK_TOKEN macro is used to compare the type of a token with an expected type, and throw an appropriate ParserError if these don't match.

In general, duplicate or redundant checks are avoided in the parsing functions. For example, the method parseWhile() which parses a while statement is only invoked by the parseStatement() function when the next lookahead token is a while keyword token. The method therefore starts by consuming a token without checking it, assuming that it is a while keyword token.

```
ast::StmtNodePtr Parser::parseBlock() {
  Location loc = consume().loc; // consume {

  std::vector<ast::StmtNodePtr> stmts;

  while (peek(0).type != lexer::RBRACE_TOK) {
    stmts.push_back(std::move(parseStatement()));
  }

  Location endloc = consume().loc; // consume }.

  return std::make_unique<ast::BlockStmt>(std::move(stmts), loc.merge(endloc));
}
```

Listing 1: Implementation of the parseBlock() method, which parses a *Block* nonterminal, producing a BlockStmt AST node.

In addition, recursion is avoided where possible. For example, where the EBNF of SIPLang specifies that zero or more repetitions of a nonterminal are part of a valid production rule, a loop is used to implement parsing of these potential repetitions. This makes construction of the AST simpler; nodes are simply accumulated into a vector, and then moved directly into the node constructor.

Listing 1 shows the implementation of block parsing, which demonstrates some of the characteristics described above.

Firstly, note that the first token is consumed without checking that it is indeed a “`{`” token. Its location is saved to a variable, as it is needed to construct the location of the node produced.

Secondly, note that statements are parsed using a `while` loop, and collected in a `vector`.

Finally, the “`}`” token is consumed, and a new `BlockStmt` node is constructed using the `StmtNodes` parsed. There is no check to confirm that the last token consumed is a “`}`” token, as this is the terminating condition of the `while` loop.

## 4 XML generation

XML generation is implemented using a subclass of `AbstractVisitor` called `XMLVisitor`.

Generating XML is straightforward but tedious, so two macro functions `XML_ELEM_WITH_CHILDREN()` and `XML_ELEM_WITH_CONTENT()` are used to ease the job. One of the macros generates XML for AST nodes which have children (using, among other things, the `visitChildren()` method), while the other generates XML for leaves of the AST tree, where the content between XML tags is derived directly from the node.

Each of these macros writes to a `std::stringstream` in the `XMLVisitor` class, which is used to store the incrementally generated XML. An `indent` member variable is used to keep track of the current indentation level of the XML being produced.

The generated XML includes source locations for each of the AST nodes.

In addition, special characters are sanitized. For example, the "<" character is converted to "`&gt;`". This allows external programs, such as a browser, to correctly parse the XML produced.

Once an entire AST has been visited, the generated XML can be extracted from the visitor using the `xml()` method, which simply produces a `std::string` from the `std::stringstream` member.

Listing 3 shows the XML generated by the compiler for the example program in Listing 2.

```
__pixelr 0, 0, 3, 4, #ff0000;
```

Listing 2: Example program which renders a red rectangle in the bottom left corner of the VM display.

```
<TranslationUnit loc="[1:0]-[1:29]">
 <PixelRStmt loc="[1:0]-[1:29]">
  <IntLiteralExprNode loc="[1:9]-[1:10]">0</IntLiteralExprNode>
  <IntLiteralExprNode loc="[1:12]-[1:13]">0</IntLiteralExprNode>
  <IntLiteralExprNode loc="[1:15]-[1:16]">3</IntLiteralExprNode>
  <IntLiteralExprNode loc="[1:18]-[1:19]">4</IntLiteralExprNode>
  <ColourLiteralExprNode loc="[1:21]-[1:28]">16711680</ColourLiteralExprNode>
 </PixelRStmt>
</TranslationUnit>
```

Listing 3: XML generated for the program shown in Listing 2.

# 5 Semantic checking

## 5.1 Types defined for semantic checking

### 5.1.1 Errors

Semantic errors encountered while visiting an AST are handled by throwing an instance of `SemanticError`, with a useful message and the location of the error in the source code. The latter is obtained from the AST.

`SemanticError` subclasses `CompilationError`, calling its constructor with a formatted version of the error message passed to it, in similar fashion to `LexerError` and `ParserError`.

### 5.1.2 Symbol tables

A symbol table maps a `StmtNode` in the AST to information about the symbols in the scope it defines. For example, a `ForStmt` node gets mapped to a scope containing information about (at least) its loop index variable. Not every type of `StmtNode` can have a scope associated with it, for example `AssignmentStmt` cannot.

Symbol tables are represented by a `SymbolTable` type. This is an alias for a `std::map` which maps `StmtNode` pointers to a `Scope`.

### 5.1.3 Scopes

A `Scope` is a struct which contains the actual symbol information. This is stored inside a `std::map` field which maps `std::string` (identifiers) to a `SymbolTableEntry` struct.

Aside from symbol information, `Scope`s also contain a pointer to their parent scope (if any). This pointer is used in the `get()` method of a `Scope`, which returns the symbol information for a variable if it is in the current `Scope` instance, or in some `Scope` which can be reached via the parent pointer.

The `get()` is used during semantic checking, chiefly when checking that a variable is defined at a point where it is referenced or assigned to.

`Scope`s also contain a `std::optional<FunctionTypeNode>` field, which contains nothing for most `Scope`s, but contains the function type (signature) for the `Scope` corresponding to a `FuncDeclStmt` node (this is the outermost scope of a function).

The `getFuncType()` method gets the function type (signature) for the function which the scope is defined in. Since only the outermost scope of a function carries its signature, this method may recursively call itself on the parent scope.

### 5.1.4 Symbol table entries

`SymbolTableEntry` instances hold the information for a specific program symbol. In the final compiler implemented, the only field in this struct is a `std::unique_ptr<TypeNode>` field containing the type of the symbol. Types are deep copied from the AST when constructing a `SymbolTableEntry`, using the `copy()` method mentioned above.

Note that the semantic checker treats function names as regular symbols, and uses the same structures discussed above to keep track of information for them. As mentioned in Section 3.2, this is catered to by the `FunctionTypeNode` type, a subclass of `TypeNode` which is not produced by a `Parser`, but is used in `SymbolTableEntry`.

#### 5.1.5 Type checking tables

A `TypeCheckerTable` is a temporary table used by the `SemanticVisitor` class during semantic checking. It is a type alias for a `std::map` which maps `ExprNode` pointers to a `std::unique_ptr`<TypeNode>.

This allows an `ExprNode` to be type checked by looking up the types of its subexpressions in an instance of `TypeCheckerTable`.

### 5.2 The semantic visitor

Semantic checking is implemented using a subclass of `AbstractVisitor` called `SemanticVisitor`.

Information about program symbols is collected by the `SemanticVisitor`, and placed inside an instance of `SymbolTable`. This symbol table is also used by the code generation visitor, and hence a `SemanticVisitor` contains a reference to a `SymbolTable`, which is passed to it during construction (by an instance of the `Compiler` class).

`SemanticVisitors` also contain a `std::stack` of `TypeCheckerTables`. The visit methods for expression nodes first visit the child nodes (sub-expressions), and then type check the current node by looking up the type of the sub-expressions in the `TypeCheckerTable` at the top of the stack.

Finally, provided that the type check succeeds, an entry mapping the current expression node to its type is placed in the `TypeCheckerTable` at the top of the stack, so that it can be used to type-check parent nodes.

`SemanticVisitor` also has a pair of `enterScope()`/`exitScope()` helper functions, which are used in `visit` functions for `StmtNodes` with an associated scope.

`enterScope()` adds a new entry to the `SymbolTable` for the given `StmtNode`, and sets the `currentScope` pointer to point to the newly created `Scope` instance (which has the old value of `currentScope` as its parent). In addition, it pushes a new `TypeCheckerTable` onto the stack.

`exitScope()` sets the `currentScope` pointer to the old value's parent pointer, and pops a `TypeCheckerTable` off the stack, effectively de-allocating it. This is a safe operation, as the types of sub-expressions within a scope cannot affect types of expressions in the parent scope.

This is precisely why a stack of `TypeCheckerTables` is used; the compiler saves memory by getting rid of type information when it is no longer needed.

Symbols can be accessed in a scope that is more deeply nested than the one in which they are defined; given this, one may ask how the stack of `TypeCheckerTables` can be used to get type information about symbols (since a `TypeCheckerTable` has no parent pointer).

The answer lies in the implementation of the `visit()` method for `IdExprNode`. This method indiscrimately fetches type information for a symbol using the `get()` method of `currentScope`, and places it in the `TypeCheckerTable` at the top of the stack, where it is available for type checking of parent expressions.

#### 5.2.1 Type checking

Several checks were implemented in the `SemanticVisitor`'s `visit()` methods.

Firstly, as mentioned above, type checking was implemented for all compound expressions. Table 1 shows valid type combinations for the binary and unary operators in the language. Table 2 shows the valid type combinations for builtin expressions and statements.

[2] Array comparision is based on reference equality.

[3] Denotes the universe of valid PixAR types

[4] where *T* is the type passed as the first argument.

| Operator | Left operand type | Right operand type | Result type |
|---|---|---|---|
| +,-, * | `int` | `int` | `int` |
| +,-, * | `float` | `float` | `float` |
| / | `int` | `int` | `float` |
| / | `float` | `float` | `float` |
| `and`, `or` | `bool` | `bool` | `bool` |
| >, <, >=, <= | `int` | `int` | `bool` |
| >, <, >=, <= | `float` | `float` | `bool` |
| ==, != | `int` | `int` | `bool` |
| ==, != | `float` | `float` | `bool` |
| ==, != | `bool` | `bool` | `bool` |
| ==, != | `colour` | `colour` | `bool` |
| ==, !=[2] | `[]`$T$ | `[]`$T$ | `bool` |

| Operator | Operand type | Result type |
|---|---|---|
| ! | `bool` | `bool` |
| - | `int` | `int` |
| - | `float` | `float` |

Table 1: Valid type combinations for binary and unary operators in the PixAR language. $T$ indicates a type variable, i.e. a variable ranging over all valid PixAR types.

| Builtin | Result type |
|---|---|
| `__width` | `int` |
| `__height` | `int` |
| `__read int int` | `colour` |
| `__randi int` | `int` |
| `__newarr` $\mathcal{T}$[3] `int` | `[]`$T$[4] |
| `__print` $T$ | N/A |
| `__delay int` | N/A |
| `__pixelr int int int int colour` | N/A |
| `__pixel int int colour` | N/A |

Table 2: Valid type combinations for PixAR builtin expressions and statements

The type checking rules used for array expressions are straightforward. Let $T$ be a valid PixAR type. Then

- A `__newarr` expression which is passed element type $T$ as the first argument has type `[]`$T$. Also, the size expression passed to `__newarr` must have type `int`.
- If an expression of type $T$ is assigned to an array location, that array must have type `[]`$T$. In addition, the location must be specified using an expression of type `int`.
- If an array location is read, then the expression specifying the location must have a type `int`. If the array has type `[]`$T$, the resulting array access expression has type $T$.

These rules are summarized in the following inference rules:

$$\frac{T : \mathcal{T}, n : \texttt{int}}{\texttt{\_\_newarr}\ T, n : \texttt{[]}T} \qquad \frac{e_1[e_2] = e_3}{e_2 : \texttt{int} \wedge \left(\exists T \in \mathcal{T} \cdot e_1 : \texttt{[]}T \wedge e_3 : T\right)} \qquad \frac{e_1[e_2], e_1 : \texttt{[]}T}{e_2 : \texttt{int}, e_1[e_2] : T}$$

where $\mathcal{T}$ is the universe of valid PixAR types.

For FunctionCallNodes, the FunctionTypeNode (signature) of the function is fetched from the currentScope pointer using the get() method, and the types of the arguments are checked against it. There is also a check which makes sure that the symbol used as a function actually has the type of a function.

ReturnStmt nodes have their return expression checked in a similar way. In addition the visit() method for this kind of node handles the semantic error where a return statement is used outside of a function definition. This case is detected by checking whether currentScope->getFuncType() returns std::nullopt.

Note that FunctionTypeNodes for each function are constructed and passed to enterScope() (which in turn uses them in constructing the new Scope) in the visit() method invocation for the corresponding FuncDeclStmt node.

The AST nodes dedicated to control flow (IfElseStmt, WhileStmt and ForStmt) are also type checked to ensure that the child expression nodes representing conditions have the right (bool) type.

In order to make type checking easier, a CHECK_TYPE macro function is defined. This macro takes a pointer to an ExprNode and a TypeNode, and checks (by consulting the top of the TypeCheckerTable stack) whether the given node has the expected type. If this is not the case, a SemanticError is thrown with a useful message including both the expected and the received type.

An example of such an error message is shown in Listing 4.

```
Semantic error at [45:28]-[45:44]: Expected type colour, found incompatible type []int.
```

Listing 4: An example message from a semantic error thrown during type checking. This is printed to the standard error stream.

### 5.2.2 Scope checking

When visiting a VariableDeclStmt, the currentScope is checked to see whether it already contains a variable (or function symbol) with the same name as the one being declared. Note that the get() method is not used in this case, as we only want to check the current scope and not its parent scopes; according to our semantics a variable defined in a scope can override a variable of the same name defined in a parent scope.

If the check does not succeed (indicating there is no name clash), the variable and its type information are added to currentScope.

When visiting a FunctionCallNode or an IdExprNode, currentScope->get() is used to determine whether the required variable or function symbol is available. If this is not the case, a SemanticError with a useful message is thrown.

Note that an AssignmentStmt actually represents its LHS using a pointer to an ExprNode, which may be an IdExprNode but also a more complex ArrayAccessNode. For this reason, there are no scoping checks when visiting AssignmentStmt; the necessary checks are performed when the LHS is visited.

# 6 Code generation and Optimizations

## 6.1 Types defined for code generation

Several aspects of a compiled program must be represented during code generation. For this reason, a number of auxiliary structs were defined which support the task. These will now be described.

### 6.1.1 Opcodes and instructions

PixAR VM opcodes are represented using an `enum` called `PixIROpcode`.

Instructions in the generated program are represented using a `PixIRInstruction` struct. This struct contains an opcode indicating the operation performed by the instruction, as well as a `std::variant` which holds any data associated with the instruction.

The variant is only used with `PUSH` instructions, and can hold a `std::monostate` (representing the case where there is no data associated with the instruction), a `std::string`, and a pointer to a `BasicBlock`. The latter is used temporarily during compilation (in order to make computing jump offsets easier), and is later converted into a `std::string`, as described below.

Finally, `PixIRInstruction` has a `to_string()` method which returns a `std::string` representation of the instruction in a form that the VM can parse.

### 6.1.2 Basic blocks and Functions

Basic blocks are runs of instructions where control flow executes sequentially. There are no jumps in the middle of or to the middle of a basic block.

Basic blocks are represented by a `BasicBlock` struct, which contains a parent pointer to the `PixIRFunction` containing the basic block, as well as a `std::list` of `PixIRInstructions` constituting the basic block.

The rationale for using a `std::list`, which is a linked list data structure, over another container (such as `std::vector`) is that such a data structure is more efficient when changing or removing elements at the middle of the container, as done by the peephole optimizer (see Section 6.3.3).

The list of instructions is also iterated over, and elements are added to/removed from its end by parts of the code generator/optimizer. In these respects, however, `std::list` offers roughly the same performance as other container types.

Functions in the generated PixIR code are represented by a `PixIRFunction` struct. This contains a field with the function name, as well as a `std::vector` of `unique_ptrs` to the `BasicBlocks` of the function. Note that a basic block can only belong to a single function in our compiler, justifying the use of `unique_ptr`[5].

An entire PixIR program is represented by a type called `PixIRCode`, which is an alias for a `std::vector` of `unique_ptrs` to `PixIRFunctions`.

### 6.1.3 Frame index maps

During code generation, each variable is allocated an index in some frame. In order to generate `PUSH` instructions correctly, a data structure is needed which allows easy access to a variable's frame depth (relative to the top of the frame stack at that point in the code) and index.

[5]Some compilers, such as the Solidity compiler may share basic blocks between functions in order to reduce code size.

The FrameIndexMap struct is defined for this purpose. In many respects, it resembles the Scope struct from Section 5.1.3; it has a parent pointer to another FrameIndexMap, and a map from variable names to a FrameIndex (an alias for int). A FrameIndexMap is also created for each Scope in the SymbolTable.

However, FrameIndexMap does not contain information about function symbols, as these are not allocated space in frames in the final PixIR program.

FrameIndexMap has a recursive getDepthAndIndex method which takes a variable name and returns the depth of the frame it is in (this depth starts out at 0 and is incremented on each recursive call to getDepthAndIndex) and the index of the slot allocated to it in the frame.

This method is used extensively when generating PUSH instructions of the form PUSH [x:y].

## 6.2 Code generation

Code generation is carried out by a subclass of AbstractVisitor called CodeGenerator.

CodeGenerator uses a number of state variables. These variables will now be listed and their role in code generation will be described:

- A pointer currentScope which points to the Scope for which code is currently being generated.
- A reference to a SymbolTable. This reference is passed to the CodeGenerator on construction, and is meant to be the same table populated by the SemanticVisitor for a run of the compiler.

  The symbol table is mainly used to set the currentScope pointer.
- A PixIRCode field containing the code generated so far by the CodeGenerator.
- A std::stack of BasicBlock pointers. New instructions are always added to the block at the top of this stack (using an addInstr() method) as the visitor visits AST nodes.

  A terminateBlock() method can be used to finalize the current block, popping it off the stack, and pushing a new one. An important subtlety is that the new block inherits the same backpointer to PixIRFunction as the old one, and is added to that function's list of (owned) BasicBlocks. So terminateBlock() is used to finalize and start new blocks within the same function.

  The reason we need a std::stack instead of a single pointer is due to our language allowing nested function definitions. When a function definition is encountered, a new function is added to the PixIRCode field of the CodeGenerator. In addition, a new basic block is added to the new function, and is pushed onto the basic block stack.

  After visiting all children of a FuncDeclStmt, the top of the basic block stack is popped. Because within a single function, we only modify the block stack using terminateBlock(), which never changes the overall stack height, this action results in the new stack top being a basic block within the previous function, and code generation can continue correctly where it left off before the function definition was visited.

  This mechanism is encapsulated cleanly by a pair of methods called beginFunc() and endFunc(), which are called at the beginning and end of visit methods for FuncDeclStmt and TranslationUnit (which can be viewed as a function definition for the main function).

  Besides handling nested function definitions, this mechanism also encompasses regular function definitions, treating them as nested function definitions in the main function.
- A std::unique_ptr to a FrameIndexMap, called frameIndexMap. This keeps track of the entire frame stack (through the chain of parent pointers) during code generation.

  frameIndexMap is reset to a newly created frame every time that a StmtNode with a corresponding Scope (for example a ForStmt or IfElseStmt node) is visited. The old value of frameIndexMap becomes the parent of the new frame.

  After visiting the StmtNode corresponding to the new frame, the frame pointer is reset to point to the parent again.

This mechanism is encapsulated cleanly by a pair of functions `enterFrame()` and `exitFrame()`.

Aside from creating a new frame, `enterFrame()` also updates the `currentScope` pointer with the `Scope` of the visited `StmtNode` (using the symbol table), and fills in the new frame with indices for each of the variable symbols in `currentScope`.

`enterFrame()` and `exitFrame()` also issue instructions to the current basic block to open a new frame of the required size, and to close the current frame respectively.

Note that the `enterBlock()/exitBlock()` pair have a different functionality from `enterFrame()/exitFrame()`. The latter handles memory (frame stack) considerations for the generated code, while the former handles management of basic blocks.

However, since `enterFrame()/exitFrame()` issue code to the current basic block, it is important that within a single visit method, the methods are always used in the following order: `enterBlock()`, `enterFrame()`, `exitFrame()`, `exitBlock()`.

A careful symmetry is employed between `enterFrame()` and `exitFrame()` to ensure that there are no memory leaks when changing the pointers to the current `FrameIndexMap`.

Firstly, `enterFrame() releases` the current frame pointed to, and then resets `frameIndexMap` to point to the new instance of `FrameIndexMap` (which takes the released pointer as a constructer argument, and stores it as its parent).

Secondly, `exitFrame() resets frameIndexMap` to the parent of the current value, therefore `delete`ing the no longer needed frame.

`enterFrame()/exitFrame()` are not generic enough for use with `FuncDeclStmt` and `TranslationUnit` nodes. In order to handle visiting these kinds of nodes, two further pairs of functions are defined, called `enterFuncDefFrame()/exitFuncDefFrame()` and `enterMainFrame()/exitMainFrame()` respectively.

`enterFuncDefFrame()/exitFuncDefFrame()` do not have to generate code to open or close a new frame, as creating a frame is handled automatically by the `CALL` instruction, and a function's frames are closed before a `RET` instruction, rather than at the end of a function's generated code.

However, `enterFuncDefFrame()` still needs to update `frameIndexMap` to represent the automatically created frame.

In addition, when assigning indices to each variable in the new frame, special care is taken to ensure that indices are assigned to function parameters in the same order as they are declared in the source code, so that the calling convention used by the VM is respected[6].

Note that `enterFrame()` has no such consideration; it uses an arbitrary order to allocate frame indices to each variable in `currentScope`.

`enterFuncDefFrame()` also checks for local variables defined in the outermost scope of the function (excluding parameters), and issues an `ALLOC` instruction to allocate space for them if there are any.

`enterMainFrame()` simply allocates frame indices to variables defined in the outermost scope of the main function, and issues an `ALLOC` instruction if needed. This simplicity is possible because the main function has no parameters.

`exitMainFrame()` also issues a `HALT` instruction to the current basic block.

- A `std::stack` of `int`s called `frameLevels`.

  As mentioned above, code for a `ReturnStmt` must include `CFRAME` instructions to close all of the function's frames.

  For this reason, the top of `frameLevels` keeps track of the number of frames that have been created by a function at a code point. The `visit()` method for a `ReturnStmt` node issues a number of `CFRAME` instructions equivalent to the top of `frameLevels`.

[6] i.e. that arguments are assigned in order of declaration to the initial segment of the frame created by CALL.

A new item is pushed onto the stack by `enterFuncDefFrame()`, and the top of the stack is popped by `exitFuncDefFrame()`.

The top of the stack is incremented and decremented by `enterFrame()` and `exitFrame()` respectively.

Armed with this state and the corresponding methods to modify it, code generation is an easy task.

The `visit()` methods for expressions and simple statements issue instructions in a straightforward way to the block at the top of the stack. Listing 5 shows the `visit()` method for `FunctionCallNodes` as an example.

```
void CodeGenerator::visit(ast::FunctionCallNode &node) {
  rvisitChildren(&node);
  addInstr({PixIROpcode::PUSH, std::to_string(node.children().size())});
  addInstr({PixIROpcode::PUSH, "." + node.funcName});
  addInstr({PixIROpcode::CALL});
}
```

Listing 5: `visit()` method for `FunctionCallNodes`. Note the use of `rvisitChildren()`, so that in the generated code each argument is pushed onto the stack in reverse order of declaration, as required by the semantics of the PixAR VM.

The `visit()` methods for more complex statements use the `enter*/exit*` pairs and `terminateBlock()` method described above to handle memory allocation and creation of new basic blocks. Listing 6 shows the `visit()` method for `FuncDeclStmt` as an example.

```
void CodeGenerator::visit(ast::FuncDeclStmt &node) {
  beginFunc(node.funcName);
  enterFuncDefFrame(node);
  visitChildren(&node);
  exitFuncDefFrame();
  endFunc();
}
```

Listing 6: `visit()` method for `FuncDeclStmts`.

### 6.2.1 Handling of lvalues

Special consideration must be taken in handling lvalue expressions. `IdExprNodes` have an `isLValue` boolean field which is set by the parser if the corresponding node appears on the LHS of an assignment statement (even if the node is part of a more complex array access lvalue).

In the `CodeGenerator`'s `visit()` method for `IdExprNode`, this boolean field is checked. If it is `false`, a `PUSH [x:y]` instruction is issued with the frame/index address of the corresponding variable.

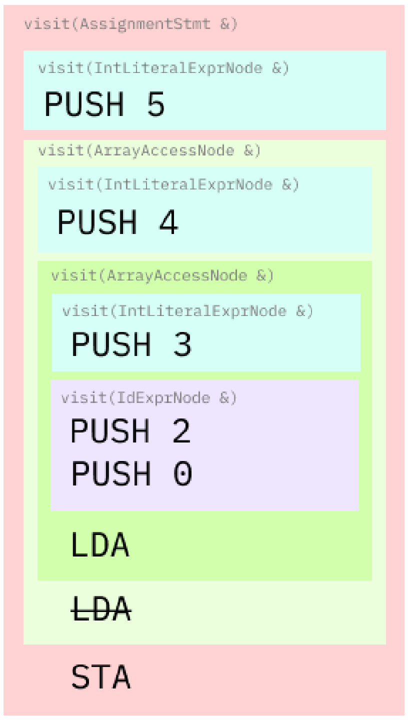
Otherwise, two `PUSH` instructions with the index and frame of the corresponding variable are issued.

If the entire lvalue in an assignment is an identifier, an `ST` instruction is issued when `visit`ing the corresponding `AssignmentStmt`, and that is the whole story. The combination of the two `PUSH` instructions and `ST` stores the assigned value to the frame/index slot allocated to the variable.

Handing array access lvalues is more complicated, however. The `visit()` method for `ArrayAccessNode` unconditionally visits the index expression and the array expression (in that order), and then issues a `LDA` (load array) instruction.

The `visit()` method for an `AssignmentStmt` whose lvalue is an array access pops the last instruction issued to the current basic block (which is always a `LDA` generated by `visit`ing `ArrayAccessNode`), and issues a `STA` (store to array) instruction.

It is best to illustrate why this scheme works with an example. Consider the program fragment `arr[3][4] = 5;`. According to our scheme, this is translated into the following bytecode:



where we have assumed (arbitrarily) that the `arr` variable gets allocated to index 2 in the frame at the top of the stack.

In the PixAR VM, the frame slot allocated to an array contains a pointer to its head. Multidimensional arrays are arrays containing pointers to the heads of other arrays. Hence, the first `LDA` instruction effectively pushes a *pointer* to `arr[3]` onto the work stack, which is the array which we want to store to.

If an `ArrayAccessNode` is used as an rvalue, a second `LDA` would load the required element from the multidimensional array. In this case, however, we remove this second `LDA`, and replace it with an `STA` instruction, effectively storing 5 to the 4th location of the array at the top of the working stack, which is `arr[3]`.

### 6.2.2 Linearization of the produced code

While translating control flow statements, it is often the case that a jump instruction needs to be issued which redirects control flow to the beginning of another `BasicBlock`. This can be implemented by pushing a PC-relative offset onto the work stack, and then calling one of the jump instructions.

However, there are scenarios where we do not know the offset when we need to issue the jump instruction. For example, consider the translation of an `IfElseStmt`. The jump instruction needs to be issued after visiting the branch condition expression, but before visiting the if/else body. At this point we do not know “how far forward” we need to jump if the branch condition is not met, and hence cannot compute the required offset.

To solve this problem, `PUSH` instructions containing relative jump offsets are not issued by the code generator at all. Instead a `PUSH` instruction is issued with the corresponding data field set to point to the `BasicBlock` that we want to jump to.

After `CodeGenerator` has finished visiting all nodes in the AST, a `linearizeCode()` function can be called on the `PixIRCode` generated.

In one pass of the generated program, this function builds a `std::map` which maps `BasicBlock` pointers to the PC offset of their first instruction in the overall program.

In a separate pass, the function then uses the built map to convert `BasicBlock` pointers in any `PUSH` instructions to the corresponding PC offset.

This scheme is also advantageous for optimization passes which process the generated code, as instructions within a basic block can be transformed, and entire basic blocks or functions can be removed before linearization without invalidating any computed PC offsets.

In order to keep code generation simple, `CodeGenerator` may produce some empty basic blocks. `linearizeCode` has a final pass over the generated code where it removes these empty blocks. Note that this is safe to do after linearization, because an empty block has the same PC offset as its immediate successor.

## 6.3 Optimization passes

The implemented compiler includes three optimization passes which may improve the quality of the code produced. These passes can be enabled individually, and will now be described.

### 6.3.1 Loop rotation

The loop rotation optimization can be enabled using the `-frotate-loops` flag.

It is an optimization which changes the code generated for `ForStmt` and `WhileStmt` nodes in such a way that only one jump instruction has to be executed per iteration of the loop.

In the regular translation of a loop, code for the loop's condition check is placed in a basic block before the body of the loop. This basic block (called the loop head) checks the condition, and jumps to the basic block immediately succeeding the loop body if the condition is not met. The last instruction in the last block of the loop body unconditionally jumps to the beginning of the loop head.

While this gives the required semantic behaviour, there are two jump instructions per iteration of the loop. For a tight loop[7] the extra jump instruction may pose a significant overhead on execution.

Loop rotation duplicates the code for the condition check, and appends it to the last block in the loop body. A jump is also added which redirects control flow to the beginning of the loop body if the loop condition is met. In this way, a single iteration of the loop will only execute a single jump. Note that the loop head must be kept in place in order to handle the case where the loop condition fails immediately, since otherwise the loop body will always execute at least once. This means that the overall code size of the generated program will be larger, with this being the trade-off made for speed up.

Figure 4 shows control flow graphs for rotated and unrotated versions of a `while` loop.

In the implementation, loop rotation is enabled/disabled for an entire program using a `boolean` option in the `CodeGenerator`. The `visit()` methods for `ForStmt` and `WhileStmt` check this field, and generate code accordingly.

### 6.3.2 Dead code elimination

[7]A tight loop is one which has a body consisting of a single basic block with only a few instructions.

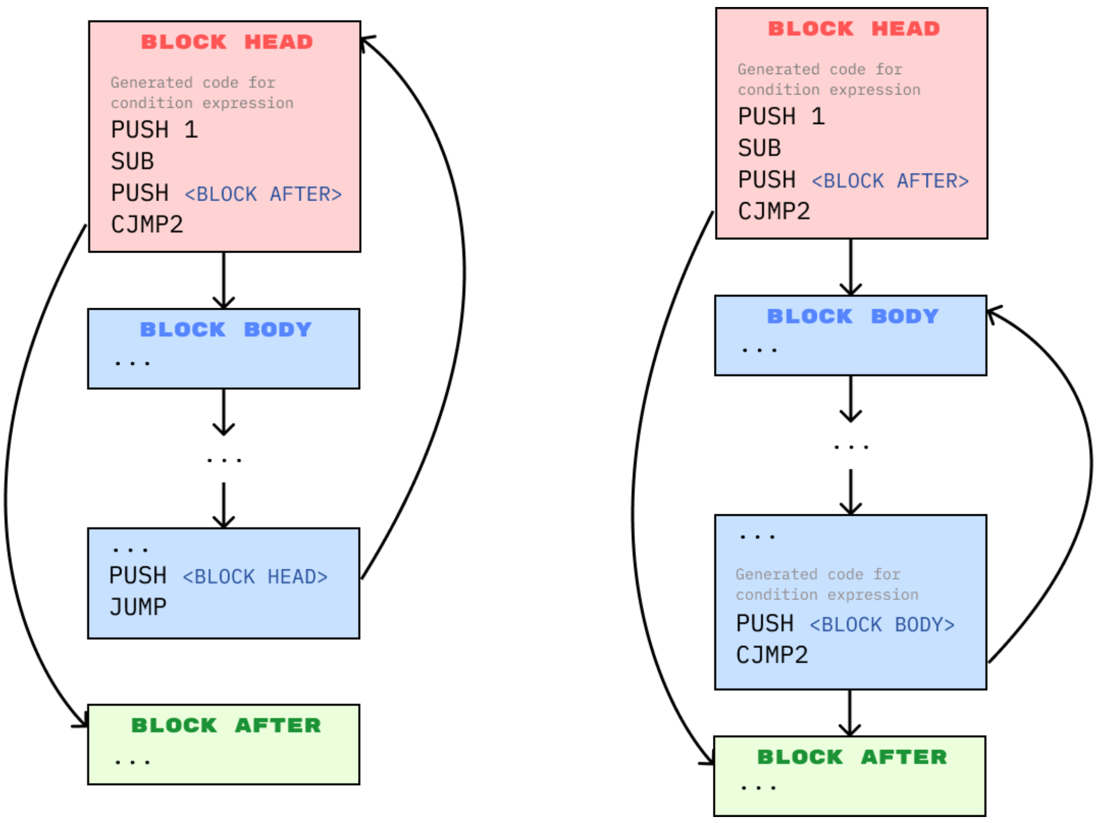


Figure 4: Control flow graphs for rotated (right) and unrotated (left) translations of a while statement. The rotated version contains only one jump instruction per iteration of the loop.

Dead code elimination can be enabled using the `-felim-dead-code` flag. It transforms the generated code by removing unreachable instructions in a basic block, as well as dead functions.

Instructions in a basic block are deemed unreachable if they appear after a RET (return) instruction in the same basic block. In this case, the instructions can be removed safely. This transformation is carried out in a function called `eliminateDeadCodeAfterReturn()`.

It is important that this pass is performed before the code linearization described in Section 6.2.2, as removing instructions from a basic block is likely to change PC relative offsets to the start of basic blocks following the modified block.

The other transformation performed by dead code elimination is the removal of code for dead functions. A function is considered dead if it is unreachable from the main function.

In order to remove dead functions, we first use an instance of the worklist algorithm[2] to find reachable functions. Pseudo code for this worklist algorithm is shown in Algorithm 2.

Essentially, we keep a `std::vector` of pointers to `PixIRFunction` instances (our worklist) which starts out with just the main function. While our worklist is non empty, we find callees of each `PixIRFunction` in our worklist, add the ones we have never encountered before to the worklist, and remove the processed worklist item, placing it in an `std::unordered_set` which will hold all reachable functions. The worklist algorithm is encapsulated in the `findReachable()` method of a class called `DeadFunctionEliminator`, whose constructor takes a reference to `PixIRCode` to be transformed.

Callees of a function are found by looking for PUSH instructions in the function body where the push operand starts with a "`.`" character, and then extracting the function name from this operand.

In order to add the corresponding `PixIRFunction` to the worklist, we need a fast way of mapping function names to the corresponding `PixIRFunction`. A `std::unordered_map` field in `DeadFunctionEliminator` is used for this purpose. During construction of an instance of this class, a single pass is performed over the `PixIRCode` to be transformed, adding the mapping for each `PixIRFunction` in the code to this field.

**Algorithm 2:** Pseudocode for the worklist algorithm used to find reachable functions.

**Data:**

- *workList* is a list of `PixIRFunction`s.
- *reachable* is the result: A set containing the names of reachable functions.
- *mainFunc* is a pointer to the `PixIRFunction` representing the main function.

```
$workList \leftarrow [mainFunc]$;
while length($workList$) $> 0$ do
    $workListTail \leftarrow$ pop($workList$);
    for $func \in$ calleesOf($workListTail$) do
        if name($func$) $\notin reachable$ then
            insert($reachable$, name($func$));
            push($workList$, $func$);
        end
    end
end
```

The `eliminate()` method of `DeadFunctionEliminator` first calls `findReachable()` to obtain a set of the names of all reachable functions. It then iterates over the code to be transformed, removing every `PixIRFunction` whose name is not in this set.

### 6.3.3 Peephole optimization

The peephole optimizer transforms small runs of instructions within a single basic block. This may lead to optimization because there are situations where the `CodeGenerator` may generate bad code due to locality (each `visit()` method views a very small fragment of the AST).

One example of this occurs when translating an expression such as `1 + x`. The ideal translation pushes the value of `x` onto the work stack, uses the `INC` instruction to increment the top of the stack, and then stores the value back to `x`.

However, there is no clean way[8] for the `CodeGenerator` to know that one of the operands is an `IntLiteralExprNode` with a value of 1. Therefore, code based on the more generic `ADD` instruction will be generated. The peephole optimizer can detect this situation in the generated bytecode, and transform it into code using `INC`.

The peephole optimizer implemented is not very powerful, and is based on detecting exact matches for small runs of instructions, and replacing them with another run.

Small runs of instructions are represented by a type called `CodePeephole`, which is a type alias for a `std::list` of `PixIRInstructions`.

A run of instructions intended to be matched is encapsulated in a class called `PixIRPattern`. Aside from a `CodePeephole` field containing the instructions to be matched, this class also has `match()` and `match_and_replace()` methods, which can check for matching runs of instructions and replace them with a given substitute.

The `match()` method takes an iterator range, and checks whether the initial segment of this range matches the sequence of instructions. A boolean is returned to indicate success or failure.

The `match_and_replace()` method takes an iterator range, the `std::list` this range is taken from (which will be the list of instructions in the basic block being optimized), as well as a `CodePeephole`

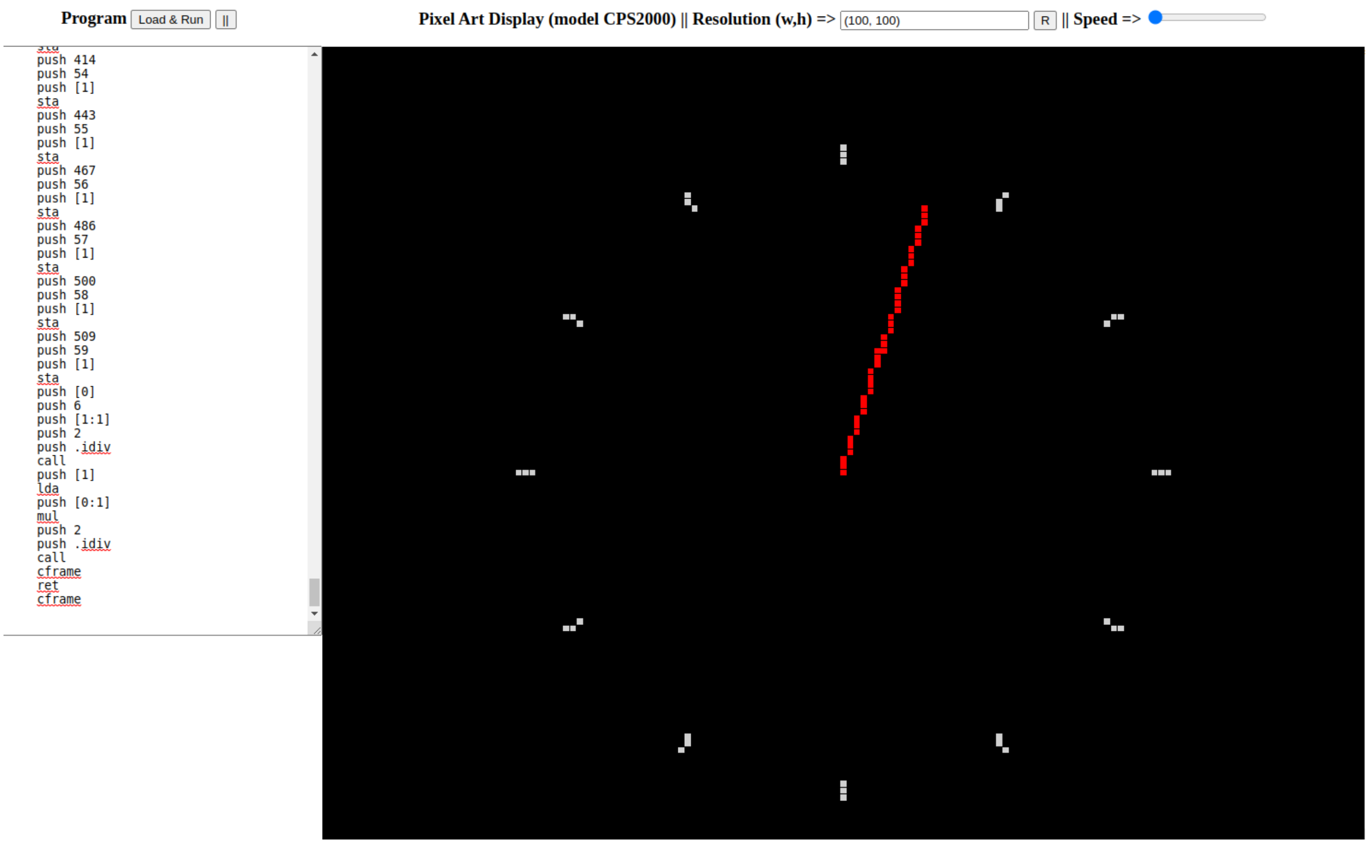
[8]`dynamic_cast` could be used to check the specific type of an `ExprNode` operand in the `visit()` method for `BinaryExprNode`. This is considered bad practice, however, and would greatly increase complexity of the code generator.

containing the run of instructions that will be substituted for a match. The `match()` method is used to check whether the initial segment of the iterator range contains the required pattern. If this is the case, the `erase()`/`insert()` methods of `std::list` are used to replace the matched run of instructions with the substitute.

A global, constant `patterns` variable maps `PixIRPatterns` to the corresponding substitute run. For example, the `PixIRPattern` containing the run of instructions `PUSH 1; ADD` is mapped to the substitute `INC`.

The `peepholeOptimize()` function puts everything together: it takes a `PixIRCode` instance and iterates over each of its basic blocks' list of instructions, calling `match_and_replace()` for every pattern in `patterns` at each location in the list.

This peephole optimizer has some obvious limitations. The largest shortcoming stems from the fact that it can only match exact runs of instructions. So for example any program of the form `e + 1` where *e* is an arbitrary integer expression will not be optimized to use `INC`, because there is no way to match the instructions generated for every arbitrary *e*.



```
push 414
push 54
push [1]
sta
push 443
push 55
push [1]
sta
push 467
push 56
push [1]
sta
push 486
push 57
push [1]
sta
push 500
push 58
push [1]
sta
push 509
push 59
push [1]
sta
push [0]
push 6
push [1:1]
push 2
push .idiv
call
push [1]
lda
push [0:1]
mul
push 2
push .idiv
call
cframe
ret
cframe
```

Figure 5: Screenshot of the rendered wall clock, shown here with VM display dimensions of $100 \times 100$.

## 7 Some example programs used to test the compiler

This section describes some of the more creative programs implemented in the PixAR language in order to test the PixAR compiler.

### 7.1 Wall clock

This example program, which can be found at `examples/wall_clock.pix` in the source code, renders a wall clock on the VM's display, with a single hand which ticks periodically. A snapshot of the rendered clock is shown in Figure 5.

Rendering a circular clock face using the PixAR VM is challenging for several reasons. Calculation of coordinates is easiest to do in polar coordinates, but this requires some version of the trigonometric functions. In addition, integer division and modulo are needed; for example, the radius of the clock face is dynamically computed as

$$r = \left\lfloor \frac{\min(\texttt{\_\_width}, \texttt{\_\_height})}{2} \right\rfloor - 2$$

Integer division is not natively supported by the language, as the division operator `/` returns a `float`, and there is no way to convert a `float` to an `int`. This is not an inherent limitation of the language, but of the VM, which only has a float division instruction (`DIV`), and has no instructions for rounding or flooring floats.

Integer division and modulo were implemented using Euclidian division[3], which is slow but simple. Pseudocode is shown in Algorithm 3.

The trigonometric functions were implemented in a much simpler way. Two arrays, each of size 60, were filled with hardcoded values for **round**$(512 \times \cos(x))$ and **round**$(512 \times \sin(x))$, with $x = \frac{2\pi i}{60}$ for $0 \leq i < 60$.

**Algorithm 3:** Pseudocode for the Euclidian division algorithm used to implement integer division in a PixAR program.

**Data:**

- $n$ is the dividend.
- $d$ is the divisor; this is assumed to be non-negative.
- $q$ is the result of division, i.e. $\lfloor n/d \rfloor$ if $n \geq 0$ and $\lceil n/d \rceil$ otherwise.
- $r$ is the remainder (modulo result) of division, and is always $\geq 0$.

```
$isNegative \leftarrow n < 0$;
if $isNegative$ then
    $n \leftarrow -n$;
end
$r \leftarrow n$;
$q \leftarrow 0$;
while $r \geq d$ do
    $r \leftarrow r - d$;
    $q \leftarrow q + 1$;
end
if $isNegative$ then
    $q \leftarrow -q$;
end
```

The reason for using a multiplier is that the values of sin($x$) and cos($x$) lie in the range $[-1, 1]$, and hence cannot be represented faithfully by integers. The PixAR code involving hardcoded values was auto-generated using a simple Python script.

Since the trigonometric functions are only used for converting polar coordinates to Cartesian ones, two functions `rsin_theta` and `rcos_theta` were defined which take a radius $r$ and an angle $\theta \in [0, 360)$ and compute

$$\textbf{idiv}\left(r \times \texttt{sin\_table}\left[\left\lfloor \frac{\theta}{6} \right\rfloor\right], 512\right)$$

and

$$\textbf{idiv}\left(r \times \texttt{cos\_table}\left[\left\lfloor \frac{\theta}{6} \right\rfloor\right], 512\right)$$

respectively[9]. The order of operations used ensures that the division by 512, which scales the computed value to account for the multiplier in the hardcoded trigonometric values, causes the smallest possible loss in precision.

With these functions in hand, it is easy to render the clock face; the radius of the face is computed using the formula mentioned above, and the Cartesian coordinates to colour in are found using `rsin_theta` and `rcos_theta` with the appropriate arguments. The $x$ and $y$ coordinates computed are shifted by $\lfloor \texttt{\_\_width}/2 \rfloor$ and $\lfloor \texttt{\_\_height}/2 \rfloor$ respectively to center the clock face).

The clock hand is rendered inside a `for` loop which performs one iteration per tick of the clock.

The angle of the clock's hand is determined using a `clock_pos` variable, which is updated every tick (loop iteration) using the formula

$$\texttt{clock\_pos} = \textbf{imod}(\texttt{clock\_pos} + 1, 60)$$

[10]where **imod** serves to keep its value in the range $\{0, ..., 59\}$. The effective angle $\theta$ which is passed to `rsin_theta` and `rcos_theta` is then $6 \times$ `clock_pos`.

[9]**idiv** is integer division implemented using Euclidian division, as defined above.
[10]**imod** is the modulo result of integer division implemented using Euclidian division.

To render the whole hand, the radius $r$ passed to `rsin_theta` and `rcos_theta` is incremented from 1 to a little less than the clock radius, keeping $\theta$ constant, and each coordinate returned is coloured in.

After a fixed delay, the colour of the hand's coordinates is set back to the background colour of the clock face, effectively rendering the hand invisible, and a new iteration of the loop begins.

Unfortunately, due to the slow nature of the VM, the clock could not be synchronized to tick with a specified duration of time.

## 7.2 Text rendering

A program was also developed which is capable of rendering text. This can be found at `examples/text.pix` in the source code.

The main function used to render text is `render_letter`. This takes $x$, $y$ coordinates and a colour, as well as an integer representing an encoded letter. It then renders the letter in a $3 \times 5$ rectange with the top left corner specified by the $x$, $y$ coordinates, and the colour specified by the corresponding argument.

Only uppercase alphabetic letters can be rendered, and these are encoded by mapping $A - Z$ directly to the numerical range $0 - 25$. "." and "," characters can be rendered as well, and are encoded as the numbers 26 and 27.

If a number outside of the range $0 - 27$ is passed to `render_letter`, nothing happens.

Note also that `render_letter` unconditionally returns 0, as the PixAR language does not support functions which return nothing.

The `render_text` function takes $x$, $y$ coordinates, a colour, and an array of integers meant to represent an encoded stream of characters.

This stream of characters is rendered using `render_letter`, with a single pixel being used for both vertical and horizontal inter-character spacing.

Spaces are encoded in the stream using 28 by convention. This causes `render_letter` to do nothing, while `render_text` skips ahead by a single character position, "rendering" a space.

Note also that `render_text` implements line wrapping, but not word wrapping (this would be very difficult). This means that when rendering a character would cause the current $x$ position to go out of bounds, `render_text` resets $x$ to 1 and starts rendering text in a new line (which has a $y$ position of 6 less than the previous line).

An example run of this text rendering program is shown in Figure 6. The character streams used were encoded and hardcoded into the program using a simple Python script.

## 7.3 Rule 110 automaton

Another program developed simulates the Rule 110 automaton, and can be found in `examples/rule110.pix`.

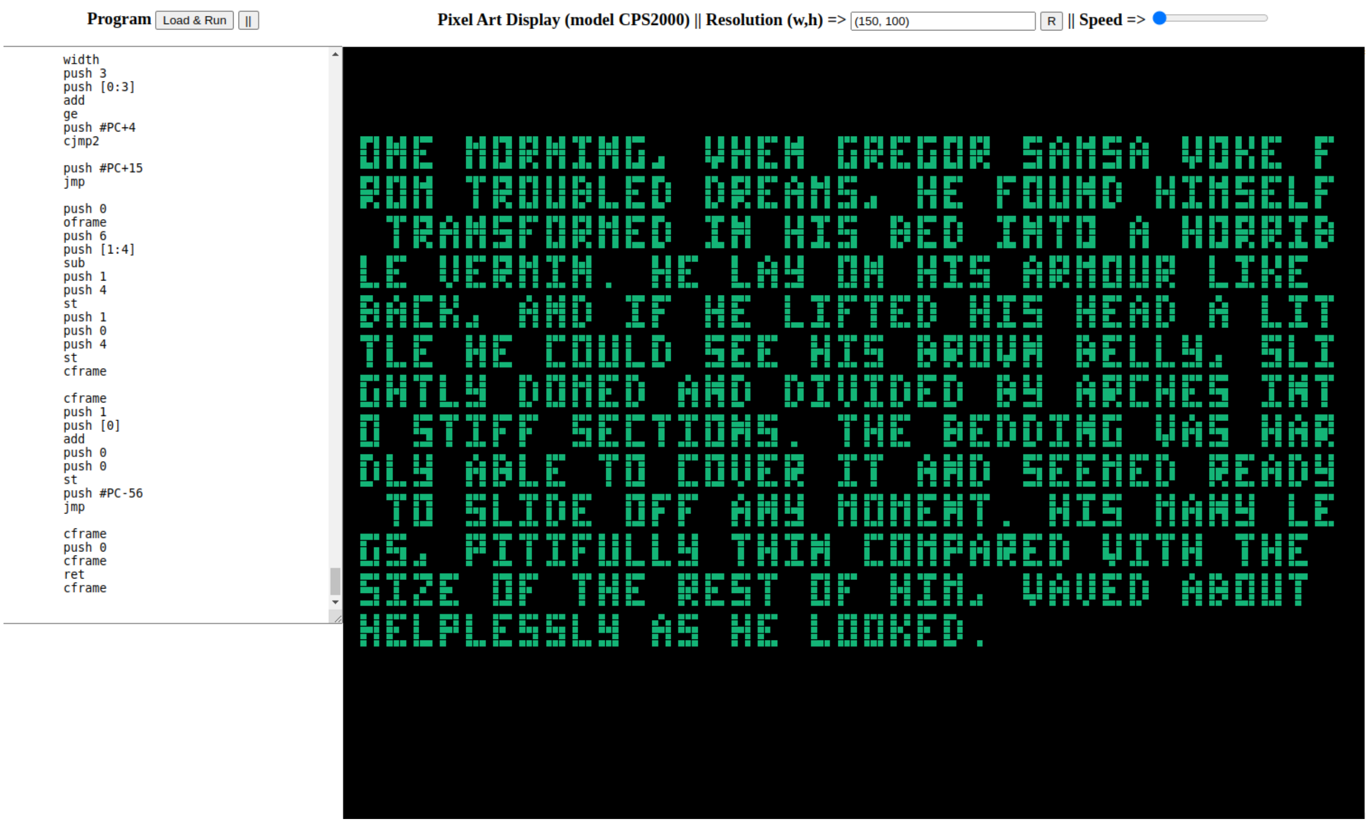
The Rule 110 automaton is a one-dimensional cellular automaton[4] whose state is updated in a series of steps. At each step, the state of one cell is updated according to some rule based on the previous states of itself and its neighbours.

In the case of the Rule 110 automaton, the rule for updating the state is shown in Table 3.

The program developed renders each state of the automaton on a seperate line. The initial state is rendered on the bottom line of the display, and updated states are rendered on progressively higher lines until the entire display is full.

Dead cells are coloured black, while the colour of live cells is randomly picked from a palette of 4 colours.

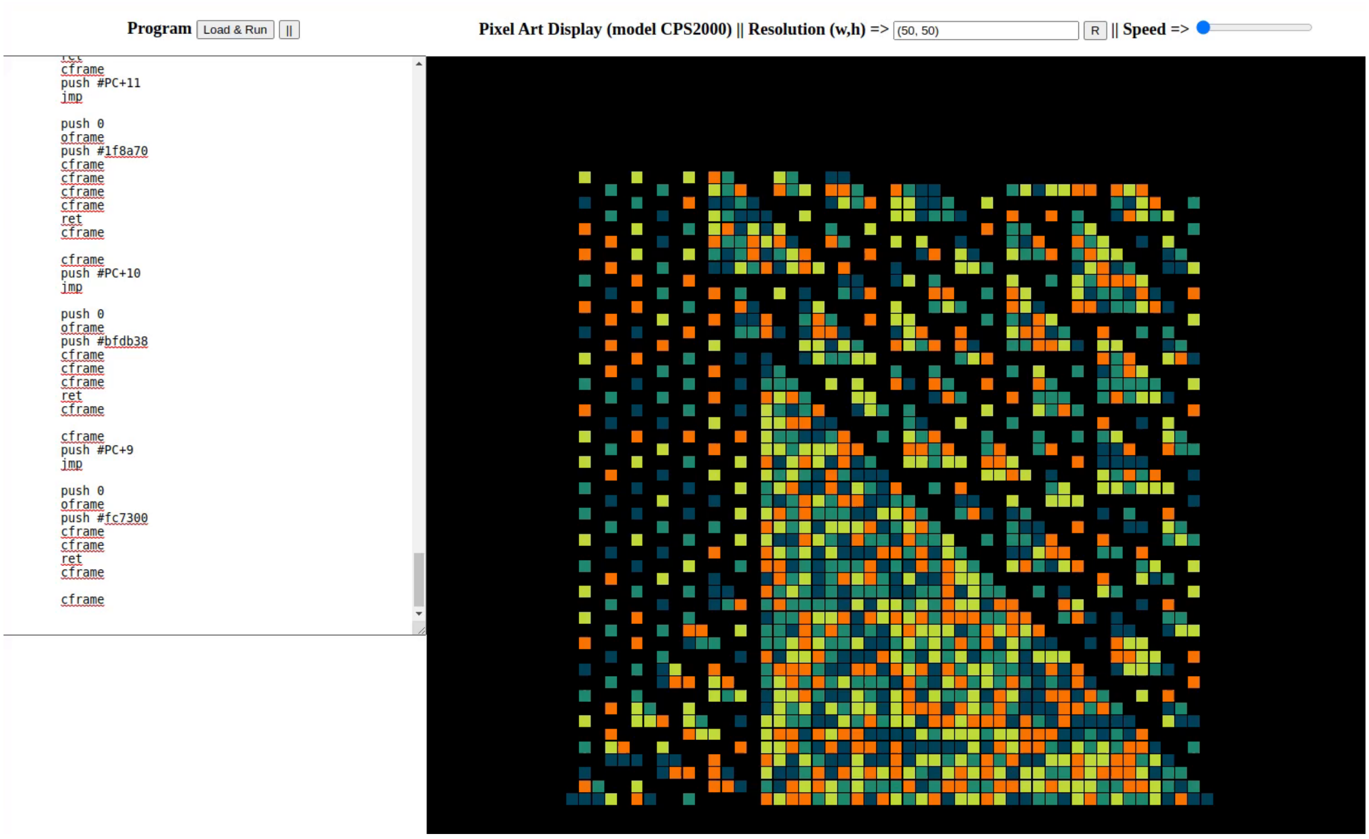
An example run of this program is shown in Figure 7.



```
width
push 3
push [0:3]
add
ge
push #PC+4
cjmp2

push #PC+15
jmp

push 0
oframe
push 6
push [1:4]
sub
push 1
push 4
st
push 1
push 0
push 4
st
cframe

cframe
push 1
push [0]
add
push 0
push 0
st
push #PC-56
jmp

cframe
push 0
cframe
ret
cframe
```

```
ONE MORNING, WHEN GREGOR SAMSA WOKE F
ROM TROUBLED DREAMS, HE FOUND HIMSELF
 TRANSFORMED IN HIS BED INTO A HORRIB
LE VERMIN. HE LAY ON HIS ARMOUR LIKE
BACK, AND IF HE LIFTED HIS HEAD A LIT
TLE HE COULD SEE HIS BROWN BELLY, SLI
GHTLY DOMED AND DIVIDED BY ARCHES INT
O STIFF SECTIONS. THE BEDDING WAS HAR
DLY ABLE TO COVER IT AND SEEMED READY
 TO SLIDE OFF ANY MOMENT. HIS MANY LE
GS, PITIFULLY THIN COMPARED WITH THE
SIZE OF THE REST OF HIM, WAVED ABOUT
HELPLESSLY AS HE LOOKED.
```

Figure 6: Screenshot of an invocation of the text rendering program. Line wrapping can be seen in action.



```
cframe
push #PC+11
jmp

push 0
oframe
push #1f8a70
cframe
cframe
cframe
cframe
ret
cframe

cframe
push #PC+10
jmp

push 0
oframe
push #bfdb38
cframe
cframe
cframe
ret
cframe

cframe
push #PC+9
jmp

push 0
oframe
push #fc7300
cframe
cframe
ret
cframe

cframe
```

Figure 7: Screenshot of an invocation of the Rule 110 automaton program.

| State of left neighbour | State of cell | State of right neighbour | New state of cell |
|---|---|---|---|
| Dead | Dead | Dead | Dead |
| Dead | Dead | Alive | Alive |
| Dead | Alive | Dead | Alive |
| Dead | Alive | Alive | Alive |
| Alive | Dead | Dead | Dead |
| Alive | Dead | Alive | Alive |
| Alive | Alive | Dead | Alive |
| Alive | Alive | Alive | Dead |

Table 3: Rule used to update the state of the Rule 110 automaton. Cells at the extremal ends (ones which only have one neighbour) are not updated.

## References

[1] K. Cooper and L. Torczon, *Engineering a Compiler*. Elsevier Science, 2011.

[2] F. Nielson, H. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer Berlin Heidelberg, 2004.

[3] I. Herstein, *Topics in Algebra*. Wiley, 1991.

[4] S. Wolfram, *A New Kind of Science*. Wolfram Media, 2002.