The PixARLang compiler

CPS2000 Compiler Theory and Practice

Mark Mizzi

Last edited: May 31, 2023

Contents

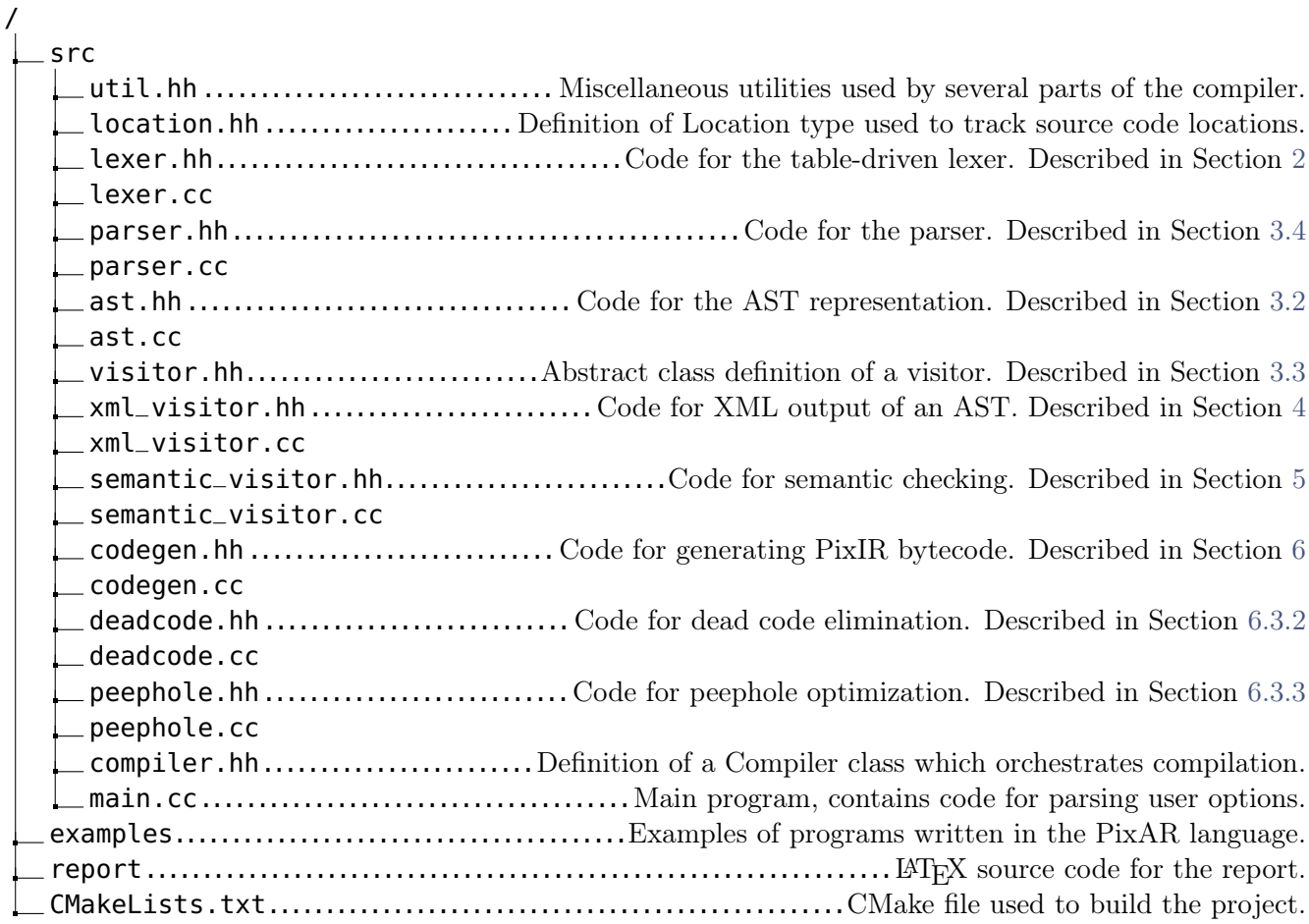


Figure 1: Organization of source code into files.

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
```

The 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 Design considerations

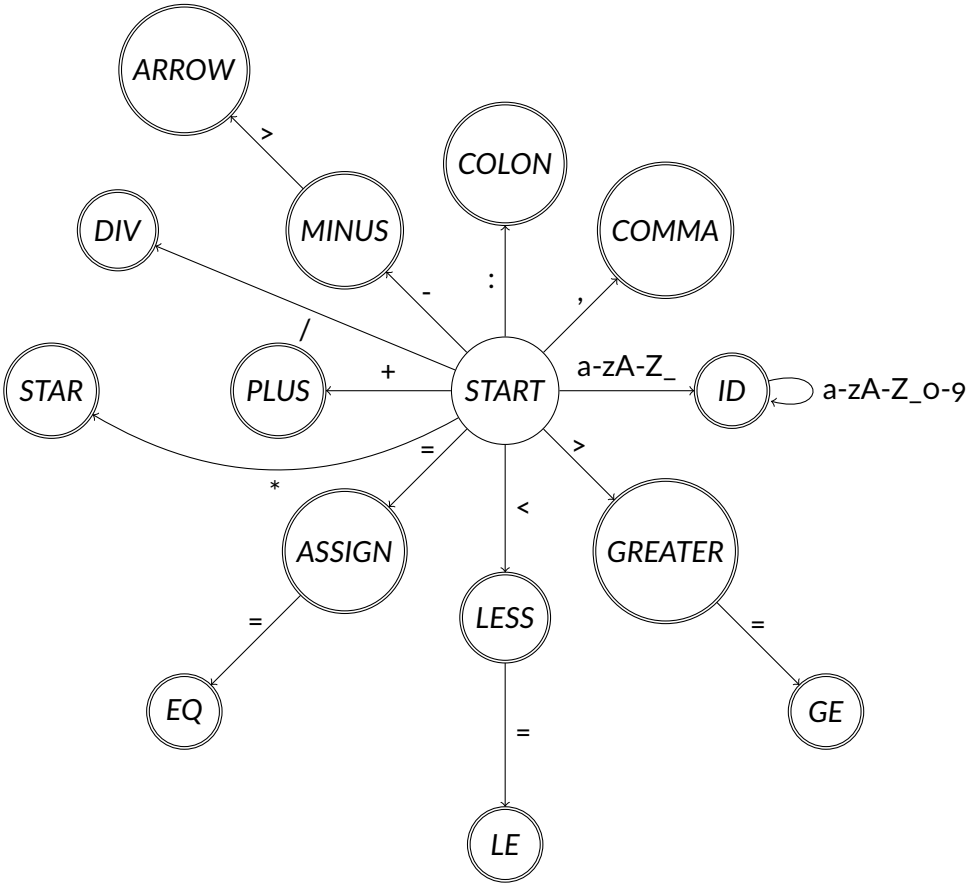
THere are several approaches available to the implementer when it comes to writing a hand-coded lexer.

The implementation uses a table-driven approach to lexing. This approach assumes that the micro-syntax 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) that accepts the language specified by the following regular expression:

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

The DFSA emulated by this compiler's lexer is shown in Figure 2.

Figure 2: DFSA that lexes tokens in the PixAR language.



The transition table of the deterministic finite state automaton is stored in an auxiliary data structure, and a "skeleton" function uses this table to carry out lexing itself. Pseudo-code for this function is shown in Algorithm **??**.

Several simplifying decisions were made in designing the lexer. Firstly, each valid input character was classified into a character class. The 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 $0-9$ are classified into a single group. 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 for $a-fA-F$ 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 table. This allows us to encode the partial DFSA shown in Figure 2 directly. Transitions that would lead to an error state are ommitted from this map, and the skeleton function throws a `LexerError` if it cannot make a transition while the current state is not an accepting state. The use of an array for storing the table would require transitions to an error state to be declared explicitly.

Finally, keywords are not handled directly by the table-driven portion of the lexer. Instead keywords are lexed as regular identifiers, and then filtered by another method which wraps the skeleton function. As a side effect of this design decision, the former function 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.

2.2 Code organization

3 Abstract Syntax tree, Visitors, and Parsing

3.1 Modifications made to eBNF

3.2 Abstract syntax trees

Abstract syntax trees are represented by an 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. 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 their child nodes.

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

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

Type nodes are used extensively in semantic checking. In order to aid this `TypeNode` has additional methods `copy` and `to_string` which produces a deep copy of the node, and prints it in human readable form respectively.

Note that deep copying is necessary in the semantic checker due to the use of `unique_ptr`. However, copying is rarely needed, and is considered a small cost compared to the convenience of using `unique_ptr`.

In addition, type nodes also implement the comparision operator ==. Implementing comparision over a type hierarchy requires an unusual pattern. The implementation of `operator==` in `TypeNode` compares the `typeid` of the two nodes being compared, and returns `false` if they are not the same (dealing with the condition where the two `TypeNodes` are of different types). Otherwise, a virtual `equals` method is called.

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

For more complex subclasses, such as `ArrayTypeNode`, the `equals` method statically casts its argument to the type of the subclass. This is guaranteed to be type safe provided `equals` is only called by `operator==`. The `equals` method can then perform subclass-specific equality checks, (in this case making sure the type of array elements is the same).

A subclass of `TypeNode` called `FunctionTypeNode` is used to represent function types during semantic checking. Nodes of this type are not produced during parsing, but are used to represent the type of function symbols.

Instances of `TypeNode` also have `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 an abstract syntax tree data structure 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`, which is passed as an argument (by reference):

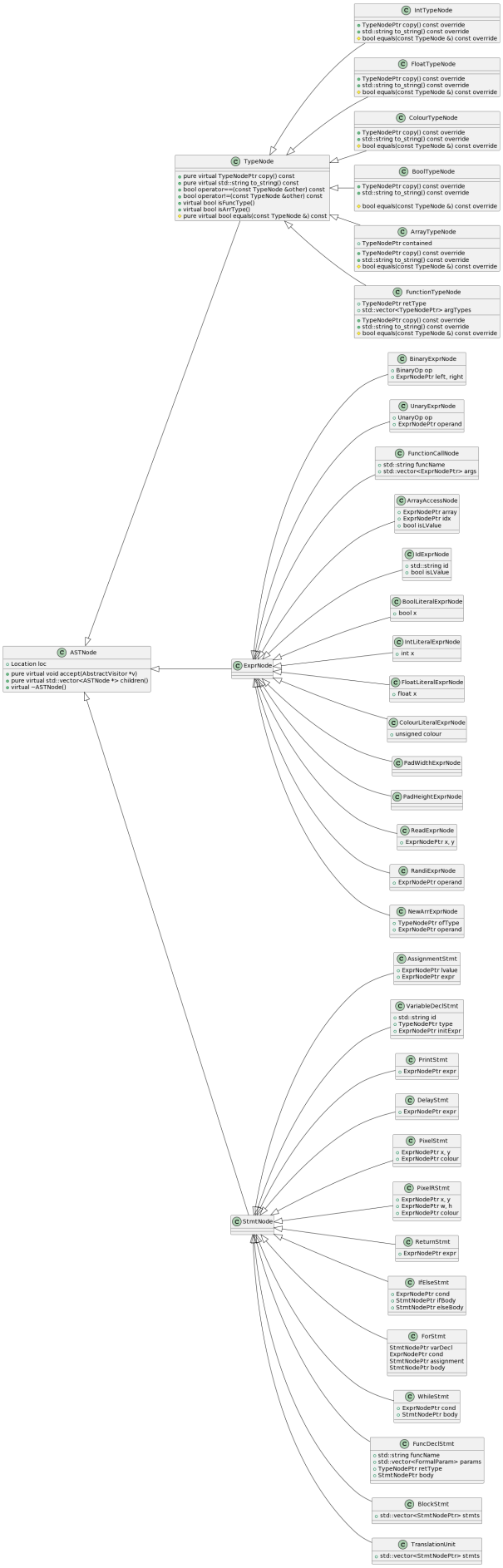


Figure 3: Inheritance hierarchy for AST nodes. Some methods are ommitted for brevity.

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

`ASTNode` contains a pure virtual method called `accept`, which takes a pointer to an `AbstractVisitor` instance. Each concrete derived class of `ASTNode` implements `accept` to call the version of the `visit` method that accepts its type:

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

In addition `AbstractVisitor` has concrete methods `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 children). `rvisitChildren` traverses the `vector` of children in reverse order (this proves useful during code generation).

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

3.4 Design considerations for the parser

The compiler uses a recursive descent parser which directly produces the AST.

Parse errors are handled by `throw`ing 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. This class 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 gets the required tokens from the lexer and pushes them onto the queue, returning the last one.

Similarly `consume` checks the internal queue. If it 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.

The parsing functions interact with the token stream through these functions. 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), or 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 check the type of a token with an expected type, and throw an appropriate `ParserError` if the types 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 consumes a token without checking it, assuming that it is a `while` keyword token.

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

The following snippet shows the implementation of block parsing, which demonstrates the features described above:

```
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));
}
```

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 macros, `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 `>`. This allows external programs, such as a browser, to correctly parse the XML produced.

Once an 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 2 shows the XML generated by the compiler for the example program in Listing 1.

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

Listing 1: 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 2: XML generated for the program shown in Listing 1

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 error message and the location of the error in the source code, which is obtained from the AST.

Semantic error subclasses `CompilationError`, and simply calls 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

The purpose of a symbol table is to map 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 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 inside a `std::map` field that maps `std::string` (identifiers) to a `SymbolTableEntry` struct.

Aside from symbol information, `Scopes` 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` method proves useful during semantic checking, chiefly when checking that a variable is defined at a point where it is referenced or assigned to.

`Scopes` also contain a `std::optional<FunctionTypeNode>` field, which contains nothing for most `Scopes`, but contains the function type (signature) for 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. 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 `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.

[1] Array comparision is based on reference equality.

[2] Denotes the universe of valid PixAR types

[3] 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`
==, !=[1]	`[]`T	`[]`T	`bool`

Operator	Operand	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` $\mathcal{T}$[2] `int`	`[]`T[3]
`__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 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:

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

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 `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 3.

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

Listing 3: 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 in the code generation code. For this reason, a number of auxiliary structs were defined which support the task of translation. These will now be described.

6.1.1 Opcodes and instructions

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

Instructions in the generated `PixIR` 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 in a form that the VM can parse.

6.1.2 Basic blocks and Functions

Basic blocks are runs of instructions where control flow runs 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 backpointer 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 below).

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, and hence the use of `unique_ptr` is justified.

An entire PixIR program is represented by `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 (relative to the current top of the frame stack) and index.

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.

The task requires keeping track of several state variables, which are listed below.

- 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 as that populated by the SemanticVisitor.

  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. The top of this stack is always used to add new instructions as the visitor visits new nodes (via an addInstr() method).

  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.

  A new frame is created every time that a StmtNode with a corresponding Scope (for example a ForStmt or IfElseStmt node) is visited. The pointer to FrameIndexMap is used as the parent of the new frame, and is reset to point to it.

  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 function 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 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()` releasees 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 deleteing the no longer needed frame.

Since `enterFrame()/exitFrame()` generate code, they are not suitable 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 this is handled automatically by the `CALL` instruction.

However, `enterFuncDefFrame()` still needs to create a new `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[4].

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

`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.

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 4 shows the `visit()` method for `FunctionCallNodes` as an example.

The `visit()` methods for more complex statements use the `enter*/exit*` pairs described above, as well as `terminateBlock()` to handle memory allocation and creation of new basic blocks.

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

```
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 4: `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 PixAR VM semantics.

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).

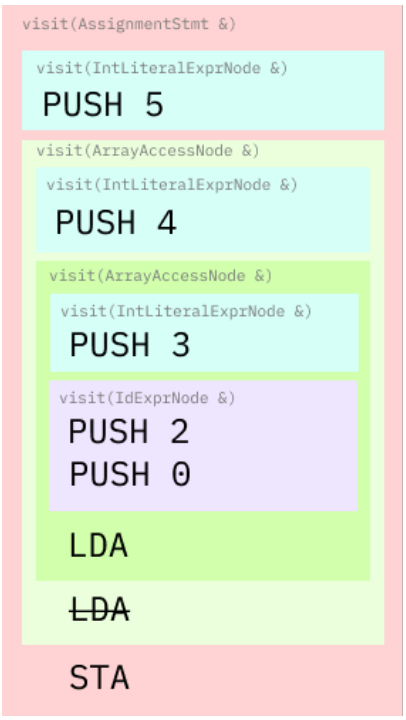
In the `visit()` method of `IdExprNode`, this boolean field is checked. If it is `false`, a `PUSH` instruction is issued with the frame/index address of the corresponding variable.

Otherwise, two `PUSH` instructions for 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 `visiting` 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 corresponding `AssignmentStmt` pops the last instruction issued to the current basic block, and issues a `STA` (store to array) instruction.

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



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

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 4^{th} 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. Hence we do not know "how far forward" we need to jump if the branch condition is not met.

To solve this problem, `PUSH` instructions with relative 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 `PixIRInstructions` 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 generate some empty 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 be described below.

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.

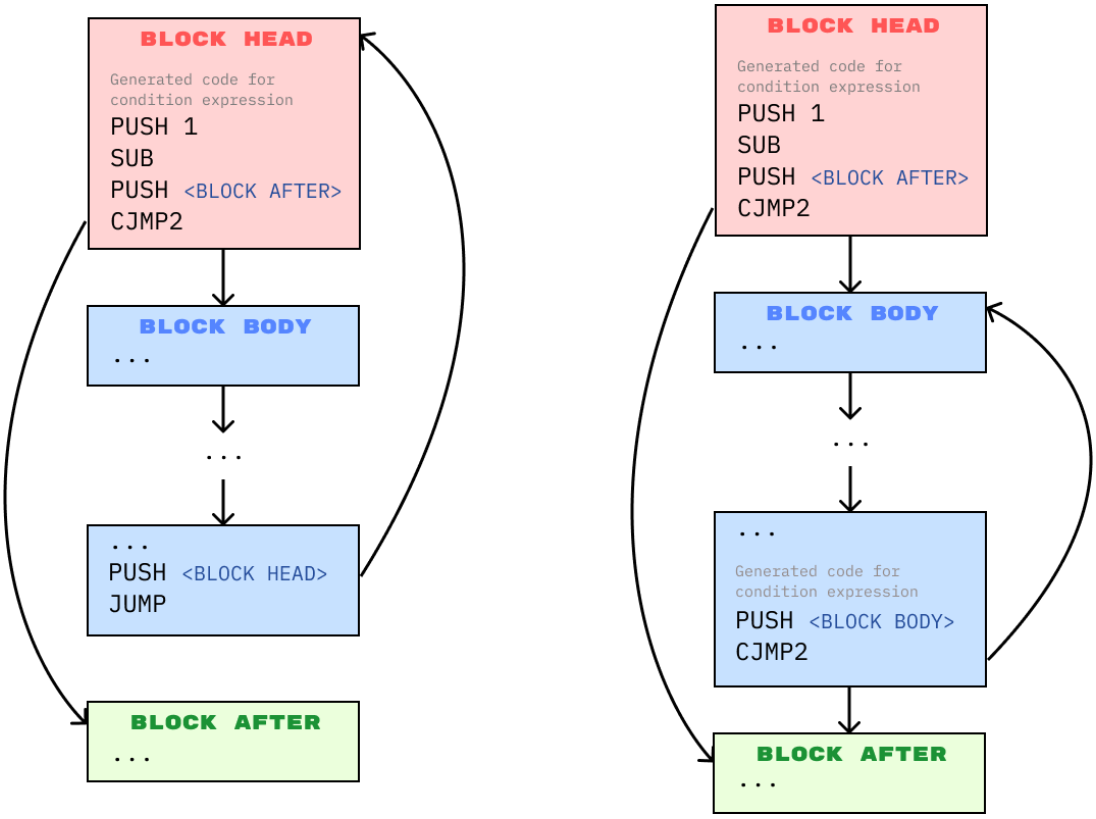


Figure 4: Control flow graphs for rotated (right) and unrotated (left) translations of a while statement. The rotated version contains only one jump instruction per iteration of the loop.

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[5] the extra jump instruction may pose a significant overhead on execution.

Loop rotation duplicates the code for the condition check, and adds 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

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

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

Code in a basic block is deemed unreachable if it appears after a `RET` (return) instruction. In this case, the instructions can be removed safely from the basic block. 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[1] to find reachable functions. Pseudo code for this algorithm is shown in Algorithm 1.

Algorithm 1: Pseudocode for the worklist algorithm used to find reachable functions.

Data:

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

```
workList ← [mainFunc];
while length(workList) > 0 do
    workListTail ← pop(workList);
    for func ∈ calleesOf(workListTail) do
        if name(func) ∉ reachable then
            insert(reachable, name(func));
            push(workList, func);
        end
    end
end
```

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 processed 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 their `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 map field.

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[6] 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 an 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 pattern field. 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` 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 expression will not be optimized to use `INC`, because there is no way to match the instructions for the arbitrary expression in the generated code.

[6] `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.

7 Some example programs used to test the compiler

7.1 Wall clock

7.2 Text rendering

7.3 Rule 110 automaton

References

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