The PixARLang compiler

CPS2000 Compiler Theory and Practice

Mark Mizzi

Last edited: May 17, 2023

Contents

1 Lexing

1.1 Design considerations

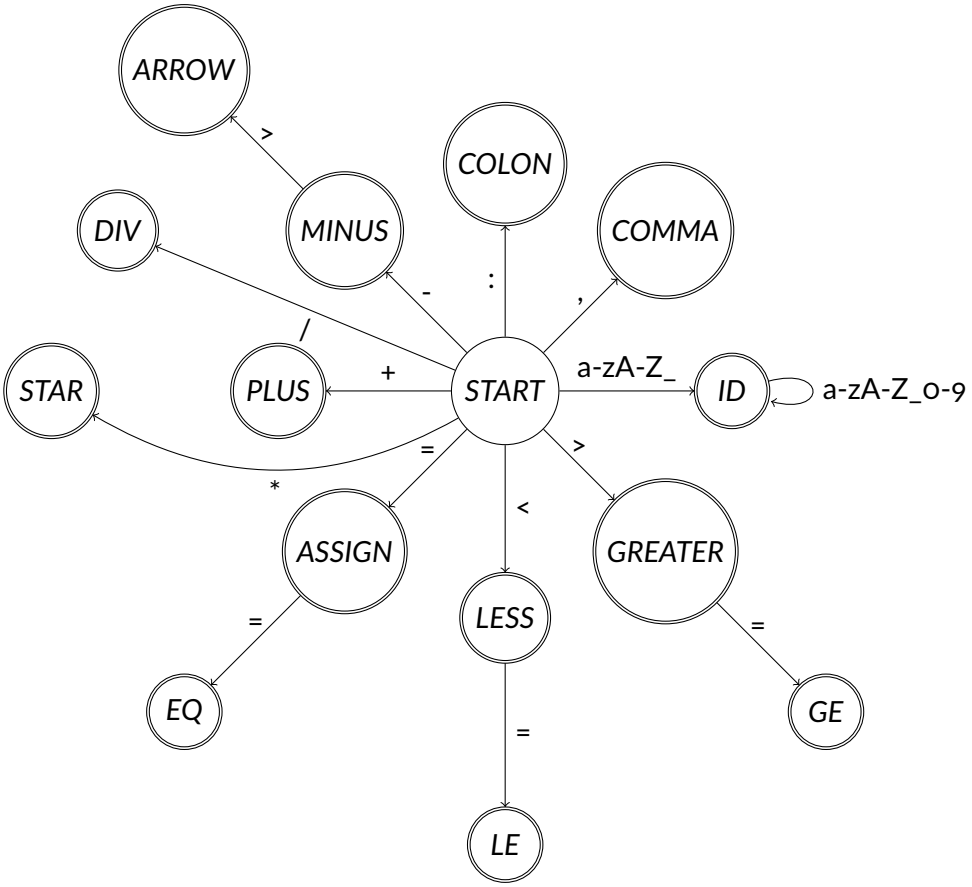
THere are several approaches available to the implementer when it comes to writing a hand-coded lexer.

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 1.

Figure 1: 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 1.

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 1 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

Algorithm 1 Pseudo-code for the "skeleton" function at the heart of the table-driven lexer

```
Require: n ≥ 0
  if Eof() then
      return Token(type = END)
  end if
  while !Eof() do
  end while
  c ← nextChar()
  X ← x
  N ← n
  while N ≠ 0 do
      if N is even then
          X ← X × X
          N ← N/2                                  ▷ This is a comment
      else if N is odd then
          y ← y × X
          N ← N − 1
      end if
  end while
```

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.

1.2 Code organization

2 Abstract Syntax tree, Visitors, and Parsing

2.1 Abstract syntax trees

Abstract syntax trees are represented by an inheritance hierarchy shown in Figure 2.

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.

2.2 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):

```
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:

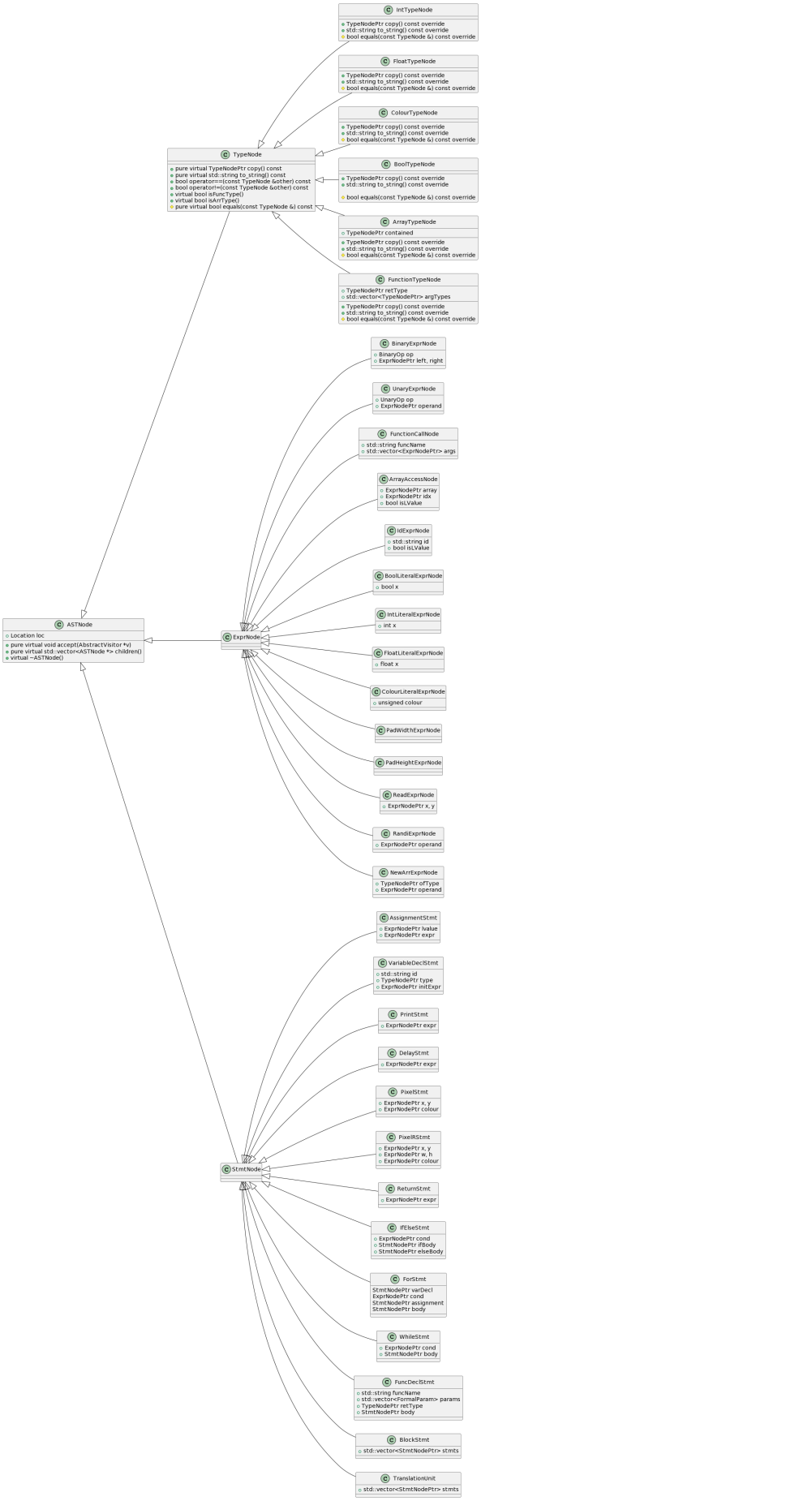


Figure 2: Inheritance hierarchy for AST nodes. Some methods are ommitted for brevity.

```
  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.

2.3 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.

3 XML generation

XML generation is implemented using a subclass of `AbstractVisitor` called `XMLVisitor`. The code is contained in `xml_visitor.hh/xml_visitor.cc`.

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 `string` from the `stringstream` member.

Below is the XML output from the compiler for the simple program

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

that prints a red rectangle in the bottom left corner of the PixAR VM screen:

```
<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>
```

4 Semantic checking

5 Code generation and Optimizations