# CPS2000: Compiler Theory and Practice

Assignment Report

Stefania Damato

May 12, 2018

# Contents

TASK I

# Table-Driven Lexer

The job of the lexer is to perform lexical analysis. This involves reading the source program, grouping the input characters into lexemes and converting them into a sequence of tokens. Here, we use a table-driven approach, which means that the lexer uses a lookup table holding language specific information to determine which action to take.

## 1.1 Constructing the deterministic finite state automaton

The first step in building the lexer was to look at the EBNF of the source language `MiniLang` and to construct a deterministic finite state automaton (DFA) based on the rules given in the EBNF. $S_0$ is the initial state from which every valid transition has to start. First, a DFA accepting integer and floating point values was constructed.

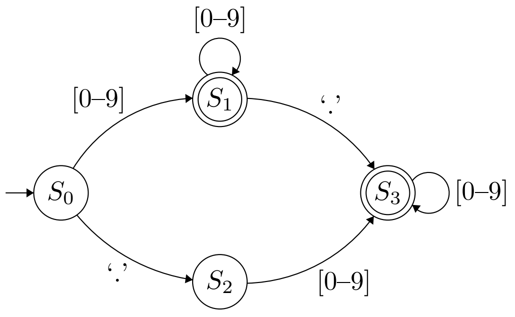

Figure 1.1: DFA accepting integer and floating point values.

Next, a DFA accepting binary and arithmetic operations, as well as the = sign used for variable assignment, was constructed.

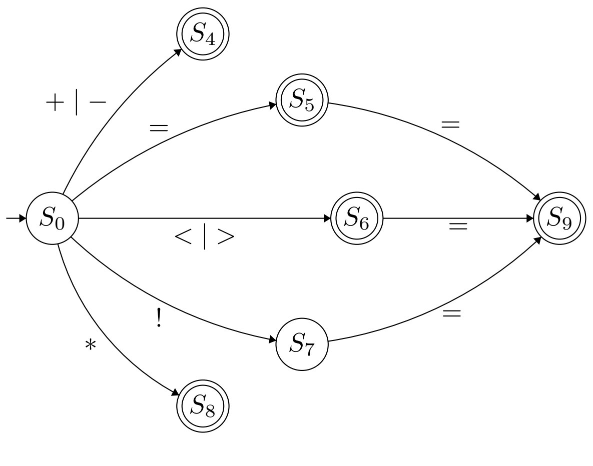

Figure 1.2: DFA accepting = sign and binary operators.

A DFA accepting identifiers was constructed next. We note that keywords forming part of the language, such as `def`, `if` and `return`, are also accepted by this automaton but will be treated differently later on.

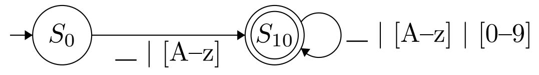

Figure 1.3: DFA accepting identifiers and keywords.

The next DFA accepts strings (surrounded by quotation marks), the end-of-file character, parentheses and punctuation marks. [\x20 –\x7E] refers to the ASCII characters having Hex values ranging between 20 and 7E. Since the character having Hex value 22 is the " character, it was removed from the loop of state $S_{12}$, because upon receiving that character we go to $S_{15}$ to end the string.

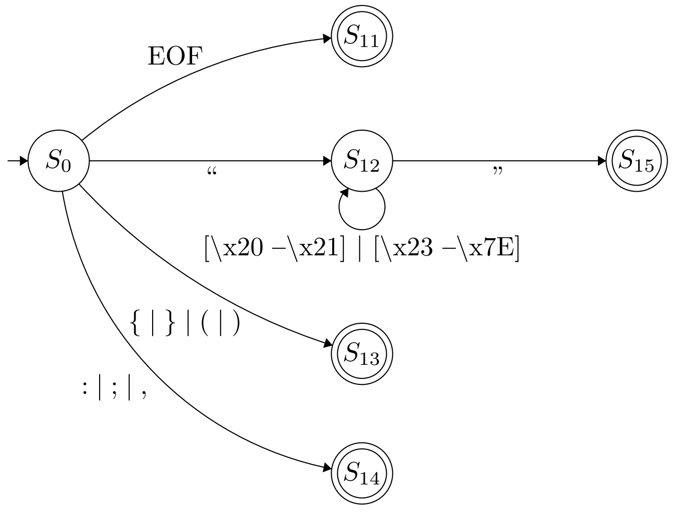

Figure 1.4: DFA accepting identifiers and keywords.

Finally, a DFA to encode single-line and multi-line comments was constructed. Similarly to what was explained above, the character * with Hex value 2A was excluded from the loop of state $S_{19}$, and * together with \ were excluded from the edge going from $S_{20}$ to $S_{19}$. This DFA also accepts the multiplicative operator /.

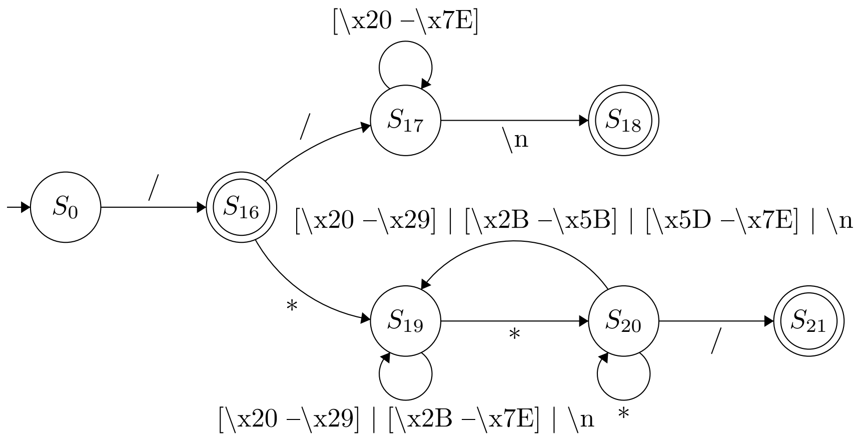
[\x20 –\x7E]
/
$S_{17}$
\n
$S_{18}$
$S_0$
/
$S_{16}$
[\x20 –\x29] | [\x2B –\x5B] | [\x5D –\x7E] | \n
*
$S_{19}$
*
$S_{20}$
/
$S_{21}$
[\x20 –\x29] | [\x2B –\x7E] | \n
*

Figure 1.5: DFA accepting comments.

The above DFAs can be merged into one bigger DFA to encode all the valid transitions, however this was split up to make visualising the automaton easier and to facilitate explanation.

## 1.2 Encoding the deterministic finite state automata as a table

Having completed the DFAs above, a table based entirely on them can be drawn up, which encodes the transitions in a table format. This table can be seen overleaf. The rows are the different classifier names used to identify the edge labels in the DFAs above. More specifically, we have the following:

| Classifier | What it represents |
|---|---|
| `DIGIT` | [0–9] |
| `POINT` | . |
| `ADDITIVE` | + \| - |
| `ASTERISK` | * |
| `EQUAL` | = |
| `LTGT` | < \| > |
| `EXCLAMATION` | ! |
| `UNDERSCORE` | _ |
| `PARENTHESIS` | { \| } \| ( \| ) |
| `PUNCTUATION` | : \| ; \| , |
| `QUOTES` | " |
| `FORWARDSLASH` | / |
| `NEWLINE` | \n |
| `ENDOFFILE` | EOF |
| `LETTER` | [A-z] |
| `PRINTABLE` | [\x20 −\x7E] |

Table 1.1 shows that if you are in state $j$ and have input $i$, you move to the state indicated in the entry $i, j$.

| Input | $S_0$ | $S_1$ | $S_2$ | $S_3$ | $S_4$ | $S_5$ | $S_6$ | $S_7$ | $S_8$ | $S_9$ | $S_{10}$ | $S_{11}$ | $S_{12}$ | $S_{13}$ | $S_{14}$ | $S_{15}$ | $S_{16}$ | $S_{17}$ | $S_{18}$ | $S_{19}$ | $S_{20}$ | $S_{21}$ | $S_E$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `DIGIT` | $S_1$ | $S_1$ | $S_3$ | $S_3$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{10}$ | $S_E$ | $S_{12}$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{17}$ | $S_E$ | $S_{19}$ | $S_{19}$ | $S_E$ | $S_E$ |
| `POINT` | $S_2$ | $S_3$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{12}$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{17}$ | $S_E$ | $S_{19}$ | $S_{19}$ | $S_E$ | $S_E$ |
| `ADDITIVE` | $S_4$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{12}$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{17}$ | $S_E$ | $S_{19}$ | $S_{19}$ | $S_E$ | $S_E$ |
| `ASTERISK` | $S_8$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{12}$ | $S_E$ | $S_E$ | $S_E$ | $S_{19}$ | $S_{17}$ | $S_E$ | $S_{20}$ | $S_{20}$ | $S_E$ | $S_E$ |
| `EQUAL` | $S_5$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_9$ | $S_9$ | $S_9$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{12}$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{17}$ | $S_E$ | $S_{19}$ | $S_{19}$ | $S_E$ | $S_E$ |
| `LTGT` | $S_6$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{12}$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{17}$ | $S_E$ | $S_{19}$ | $S_{19}$ | $S_E$ | $S_E$ |
| `EXCLAMATION` | $S_7$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{12}$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{17}$ | $S_E$ | $S_{19}$ | $S_{19}$ | $S_E$ | $S_E$ |
| `UNDERSCORE` | $S_{10}$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{10}$ | $S_E$ | $S_{12}$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{17}$ | $S_E$ | $S_{19}$ | $S_{19}$ | $S_E$ | $S_E$ |
| `PARENTHESIS` | $S_{13}$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{12}$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{17}$ | $S_E$ | $S_{19}$ | $S_{19}$ | $S_E$ | $S_E$ |
| `PUNCTUATION` | $S_{14}$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{12}$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{17}$ | $S_E$ | $S_{19}$ | $S_{19}$ | $S_E$ | $S_E$ |
| `QUOTES` | $S_{12}$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{15}$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{17}$ | $S_E$ | $S_{19}$ | $S_{19}$ | $S_E$ | $S_E$ |
| `BACKSLASH` | $S_{16}$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{12}$ | $S_E$ | $S_E$ | $S_E$ | $S_{17}$ | $S_{17}$ | $S_E$ | $S_{19}$ | $S_{21}$ | $S_E$ | $S_E$ |
| `NEWLINE` | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{18}$ | $S_E$ | $S_{19}$ | $S_{19}$ | $S_E$ | $S_E$ |
| `ENDOFFILE` | $S_{11}$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ |
| `LETTER` | $S_{10}$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{10}$ | $S_E$ | $S_{12}$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{17}$ | $S_E$ | $S_{19}$ | $S_{19}$ | $S_E$ | $S_E$ |
| `PRINTABLE` | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{12}$ | $S_E$ | $S_E$ | $S_E$ | $S_E$ | $S_{17}$ | $S_E$ | $S_{19}$ | $S_{19}$ | $S_E$ | $S_E$ |

Table 1.1: Transition table used in the lexer. States highlighted in green are final states.

## 1.3 The function `nextToken()`

The function `nextToken()` performs the main functionality of the lexer and consists of the scanning loop, the rollback loop and the returning of tokens. It takes a program and a starting position as arguments, then starting from the given position, it processes one character at a time until it reaches a character which upon adding it to the previous characters, would require an invalid transition. It then rolls back to the last valid transition which ended in a final state and returns a token depending on the string, referred to as the lexeme. This utilises a stack of states which is updated every time a new character is read.

The scanning loop calls the function `delta`, which acts exactly like a transition function for the DFAs shown previously. Given the current state and an input, it returns the state we end up in, which is either an integer between 1 and 21 (the number of states we have excluding the start state), or ERR, which would indicate that the transition is invalid.

The rollback loop takes care of going back to a previous final state.

After going through the rollback loop, a previous final state is obtained and a token is then returned depending on which final state we landed on, and, in some cases, what the lexeme was. If a previous final state is not reached, it means that the inputted code is invalid and `nextToken()` outputs a 'Lexical error' message.

The tokens used are the following:

| Value | Token | Current state |
|---|---|---|
| integer value | `TOK_IntLiteral` | $S_1$ |
| floating value | `TOK_FloatLiteral` | $S_3$ |
| + \| - | `TOK_AdditiveOp` | $S_4$ |
| = | `TOK_Equals` | $S_5$ |
| <\|> | `TOK_RelOp` | $S_6$ |
| * | `TOK_MultiplicativeOp` | $S_8$ |
| <= \|>= \| != \| == | `TOK_RelOp` | $S_9$ |
| `real` \| `int` \| `bool` \| `string` | `TOK_LangType` | $S_{10}$ |
| `if` | `TOK_If` | $S_{10}$ |
| `else` | `TOK_Else` | $S_{10}$ |
| `while` | `TOK_While` | $S_{10}$ |
| `def` | `TOK_Def` | $S_{10}$ |
| `return` | `TOK_Return` | $S_{10}$ |
| `set` | `TOK_Set` | $S_{10}$ |
| `var` | `TOK_Var` | $S_{10}$ |
| `print` | `TOK_Print` | $S_{10}$ |
| `and` | `TOK_And` | $S_{10}$ |
| `or` | `TOK_Or` | $S_{10}$ |
| `not` | `TOK_Not` | $S_{10}$ |
| `true` | `TOK_True` | $S_{10}$ |
| `false` | `TOK_False` | $S_{10}$ |
| identifier | `TOK_Identifier` | $S_{10}$ |
| EOF | `TOK_EOF` | $S_{11}$ |
| { | `TOK_OpenScope` | $S_{13}$ |

| Value | Token | Current state |
|---|---|---|
| } | `TOK_CloseScope` | $S_{13}$ |
| ( | `TOK_OpenParenthesis` | $S_{13}$ |
| ) | `TOK_CloseParenthesis` | $S_{13}$ |
| ; | `TOK_Delimeter` | $S_{14}$ |
| : | `TOK_Colon` | $S_{14}$ |
| , | `TOK_Comma` | $S_{14}$ |
| string | `TOK_String` | $S_{15}$ |
| / | `TOK_MultiplicativeOp` | $S_{16}$ |
| single line comment | `TOK_Comment` | $S_{18}$ |
| multi line comment | `TOK_Comment` | $S_{21}$ |

## 1.4 Testing the lexer

To test the lexer, an array called `token_names` was created with all the tokens in string format. This was only required for testing purposes, and was hence deleted after completion of the project.

Upon inputting the following statement

```
var x : int = 3;
```

we get the below output.

```
Lexeme: var         Token type: TOK_Var
Lexeme: x           Token type: TOK_Identifier
Lexeme: :           Token type: TOK_Colon
Lexeme: int         Token type: TOK_LangType
Lexeme: =           Token type: TOK_Equals
Lexeme: 3           Token type: TOK_IntLiteral
Lexeme: ;           Token type: TOK_Delimeter
Lexeme: �           Token type: TOK_EOF
```

The tokens are assigned to the correct lexemes and any whitespace in the input statement is skipped, as required. We note that the lexeme for the end-of-file character shows up as �. `TOK_EOF` is added manually to the end of every program in the `Lexer` class and the lexeme associated to it is `(char) EOF`, and since `EOF` is represented by -1 and the ASCII table values start from 0, when typecasting to `char` we get � which represents an unrecognised character.

When inputting

```
var s:string = "not closed;
```

we get a lexical error, as expected.

Finally, we input an adding function.

```
//Adding two real numbers
def add(x:real, y:real):real{
    return x+y;
}

print(add(4,7));
```

We get the following output.

```
Lexeme: //Adding two real numbers
Token type: TOK_Comment
Lexeme: def         Token type: TOK_Def
Lexeme: add         Token type: TOK_Identifier
Lexeme: (           Token type: TOK_OpenParenthesis
Lexeme: x           Token type: TOK_Identifier
Lexeme: :           Token type: TOK_Colon
Lexeme: real        Token type: TOK_LangType
Lexeme: ,           Token type: TOK_Comma
Lexeme: y           Token type: TOK_Identifier
Lexeme: :           Token type: TOK_Colon
Lexeme: real        Token type: TOK_LangType
Lexeme: )           Token type: TOK_CloseParenthesis
Lexeme: :           Token type: TOK_Colon
Lexeme: real        Token type: TOK_LangType
Lexeme: {           Token type: TOK_OpenScope
Lexeme: return      Token type: TOK_Return
Lexeme: x           Token type: TOK_Identifier
Lexeme: +           Token type: TOK_AdditiveOp
Lexeme: y           Token type: TOK_Identifier
Lexeme: ;           Token type: TOK_Delimeter
Lexeme: }           Token type: TOK_CloseScope
Lexeme: print       Token type: TOK_Print
Lexeme: (           Token type: TOK_OpenParenthesis
Lexeme: add         Token type: TOK_Identifier
Lexeme: (           Token type: TOK_OpenParenthesis
Lexeme: 4           Token type: TOK_IntLiteral
Lexeme: ,           Token type: TOK_Comma
Lexeme: 7           Token type: TOK_IntLiteral
Lexeme: )           Token type: TOK_CloseParenthesis
Lexeme: )           Token type: TOK_CloseParenthesis
Lexeme: ;           Token type: TOK_Delimeter
Lexeme: �           Token type: TOK_EOF
```

The lexer is working as required, and we now move on to the parser.

TASK II

# Recursive Descent Parser

## 2.1 How the parser works

A hand crafted top-down parser closely following the given EBNF was implemented. The parser's job is to take the list of tokens generated by the lexer and make sure that the tokens conform to the rules given by the EBNF.

Since a program is made up of a list of statements, the parser works by calling `parseStatement()` and checking the first token. Depending on this first token, the statement could be one of the following: a variable declaration, an assignment, a print statement, an if statement, a while statement, a return statement, a function declaration or a block. If the first token does not match any of the first tokens required for the mentioned statements, `parseStatement()` resorts to the default case and reports an error. If we match the first token of one of the statements, `parseStatement()` calls the respective funtion to parse that statement, which in many cases calls other functions to parse specific grammar rules.

For example, if the first token is `TOK_If`, `parseStatement()` calls `parseIfStatement()`. This checks for an open parenthesis, and if one is not found reports an error and stops execution. The function then calls `parseExpression()` to parse the if-condition. In order to parse an expression, the functions `parseSimpleExpression()`, `parseTerm()` and `parseFactor()` are called, and once they return (having found no syntax errors), `parseExpression()` checks for a `TOK_RelOp` and if found, it returns an `ASTBinaryExpressionNode`. Otherwise, it simply returns an `ASTExpressionNode`.

The parser and the lexer communicate through the functions `getNextToken()` and `getLookahead`. After the program is lexically analysed, its tokens are stored in an array, and a global variable is used to keep track of the current position in this array. The former function consumes a token from this array, while the latter simply 'peeks' at the next token without consuming it.

The only way in which the code deviates from the given EBNF is in ⟨`Unary`⟩. In our code, this is redefined as ⟨`Unary`⟩ ::= ('-' | '+' | '`not`') ⟨`Expression`⟩. This was done to accomodate statements like `set var x : int = +3;` while also simplifying the implementation of ⟨`Unary`⟩ (since + and - are both given the `TOK_AdditiveOp` token.)

## 2.2 The abstract syntax tree

The return types of the parse functions discussed above are instances of AST (abstract syntax tree) node classes. These are used to form the abstract syntax tree, which shows the structure of code written in MiniLang. The syntax is *asbtract* in that it does not represent every little detail present in the real syntax. For example, parentheses are omitted and if-condition-then expressions are denoted by a single node[1]. The hierarchy of the AST node classes is shown below.

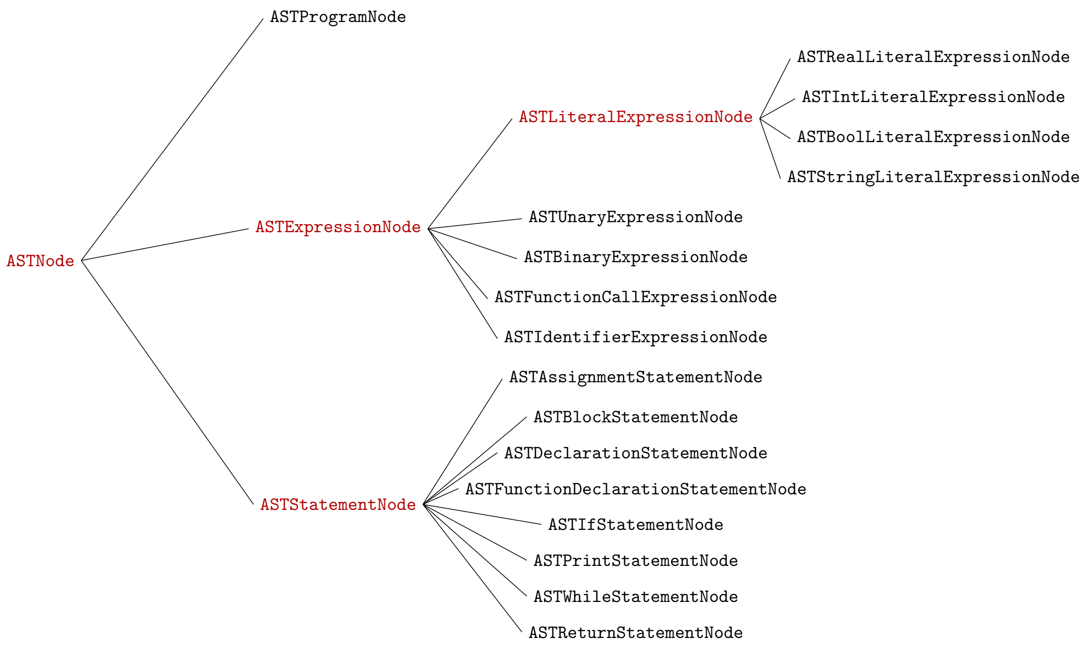
ASTNode
ASTProgramNode
ASTExpressionNode
ASTLiteralExpressionNode
ASTRealLiteralExpressionNode
ASTIntLiteralExpressionNode
ASTBoolLiteralExpressionNode
ASTStringLiteralExpressionNode
ASTUnaryExpressionNode
ASTBinaryExpressionNode
ASTFunctionCallExpressionNode
ASTIdentifierExpressionNode
ASTStatementNode
ASTAssignmentStatementNode
ASTBlockStatementNode
ASTDeclarationStatementNode
ASTFunctionDeclarationStatementNode
ASTIfStatementNode
ASTPrintStatementNode
ASTWhileStatementNode
ASTReturnStatementNode

Figure 2.1: AST class hierarchy. Nodes marked in red are abstract.

## 2.3 Testing the parser

To test the parser, we input incorrect programs and check that the parser correctly indicates the error. We will test correct programs later using Visitor classes and XML to verify that the generated abstract syntax tree is correct.

First, we input the following program, having a missing colon before the parameter list.

```
def funcSquare(x:real) real{
    return x*x;
}
```

The parser outputs

```
Unexpected token on line 1: expected ‘:’ after ‘)’.
```

When inputting the below program, which has a missing expression on line 6 after -, we get the output that follows.

```
var x:int = 3;
var y:int = 7;

while(y>x){
    print y;
    set y = y-;
}
```

```
Unexpected token on line 6: invalid expression.
```

Lastly, if we forget the delimeter at the end of a statement, as shown below,

```
var x:int = 6.4
```

the parser outputs the following.

```
Unexpected token on line 2: expected ‘;’ at the end of statement.
```

The error is reported on line 2 since the parser sees our program as the string `var x:int = 6.4\n`, however it ignores any whitespace or newline characters, therefore it searches for a ‘;’ after the newline character and does not find one, hence it ends up on line 2. In this case, outputting line 1 instead of 2 could be a possible improvement to be made to the parser.

TASK III

# Generating XML of the AST

## 3.1 The Visitor design pattern

In this task, we were required to implement the Visitor design pattern in order to generate XML code which describes the AST produced by the parser. An `accept()` function was added to every AST node class, but implemented only in the concrete ones (the black nodes in figure 2.1). An abstract class `Visitor` was created listing all the `visit()` functions for the concrete AST classes. These functions are then overriden in the concrete class `XMLGeneratorVisitor`.

## 3.2 Testing the parser using the generated XML

We now test the functionality of the parser by looking at the generated XML of a given program and making sure that the correct AST classes are called in the right order, and therefore verify that the resulting AST we get is correct.

We input the following program, given in the assignment specification.

```
var x : real = 8.1 + 2.2;
```

The following XML is generated.

```xml
<Program>
  <Declaration>
    <Identifier Type="real">x</Identifier>
    <BinaryExpressionNode Op="+">
      <Real>8.1</Real>
      <Real>2.2</Real>
    </BinaryExpressionNode>
  </Declaration>
</Program>
```

Then, we implemented a factorial function in MiniLang.

```
def funcFact(n:int) : int{
    if(n == 0){
        return 1;
    }

    else{
        return fact(n-1);
    }
}
```

The XML generated by this code is shown below.

```xml
<Program>
  <FunctionDeclaration Identifier="funcFact">
    <Identifier Type="int">n</Identifier>
    <Block>
      <If>
        <BinaryExpressionNode Op="==">
          <Identifier Type="real">n</Identifier>
          <Int>0</Int>
        </BinaryExpressionNode>
        <Block>
          <Return>
            <Int>1</Int>
          </Return>
        </Block>
        <Block>
          <Return>
            <FunctionCall Identifier="fact">
              <BinaryExpressionNode Op="-">
                <Identifier Type="real">n</Identifier>
                <Int>1</Int>
              </BinaryExpressionNode>
            </FunctionCall>
          </Return>
        </Block>
      </If>
    </Block>
  </FunctionDeclaration>
</Program>
```

TASK IV

# Semantic Analysis

To perform semantic analysis, we implemented another Visitor class. This traverses the AST and, among other things, performs type checking, makes sure that variables and functions are declared before being used and makes sure that functions always return no matter how they might branch. This behaviour is impossible to describe in the EBNF and thus semantic errors will not be detected by the lexer or the parser[2].

## 4.1 How semantic analysis was implemented

In order to perform semantic analysis, we need a symbol table holding the variables and functions that have already been defined and their corresponding types. The first class that we created was `Symbol`, which encodes a single entry in4 the symbol table. It takes care of both variables and functions. It has two constuctors, one accepting a type and an expression node in case it's a variable, and another accepting a return type and a vector of parameters in case it's a function. To differentiate between functions and variables, we defined an enum `IDENTIFIER` whose possible values are `VARIABLE` and `FUNCTION`. An `IDENTIFIER id` is kept to be able to check which of the two the current symbol is.

A class `Scope` was also declared, holding a symbol table which is simply a multimap with key `string` and value a `Symbol` pointer. This class contains a number of functions. Some of them check whether variables or functions have been declared in the current scope (and would therefore be found in the symbol table), others return the type of symbols already in the symbol table, and the rest add, edit and delete symbols from the symbol table. We therefore have a symbol table for every scope, containing what has already been declared in that scope. To complete the structure, we have a stack of `Scope` pointers in our `SemanticAnalyserVisitor`, so that we push a scope on this stack whenever we come across a block and pop that scope when we exit the block.

In `SemanticAnalyserVisitor` we also have a global variable `lastType`, which is updated by most `visit()` functions and is used to indicate what the type of the last node that has been visited is. We also make use of a `vector<pair<string, TYPE>>` called `currentParams` which we use for function declarations. This stores the name and type of the parameters in a function declaration, which are then used by the function body (which is a block) to type check. For example, if we declare the function `def f(x:int):real{}` and in the function body we return `x`, this should return an error since `x` is of type `int` but the function should be returning a `real`. This wouldn't be possible without adding `x` and its type to the symbol table of the function body. Another function that performs type checking is `returns()`. This takes an AST node pointer and recursively checks whether the statements inside this node return, regardless of any branching that could be happening. It is used when declaring functions to make sure that the functions always return.

## 4.2 Decisions taken about semantic analysis and testing our implementation

It was decided to adhere strictly to the definition of `int` and `real` so that no typecasting would occur. This means that we get a semantic error when trying to add a `real` and an `int` value:

```
Semantic error: binary expression where left node has type ‘int’ but
    right node has type ‘real’.
```

To declare a `real` variable, a floating point is required. This also allows us to declare two functions of the same name having the same number of parameters but which have different types and there being no ambiguity when these functions are called.

We allow functions to be defined within other functions. The below program, which approximates the value of $\pi$ using the Leibniz formula for $\pi$, is valid:

```
1  //Function approximating  the value of pi
2  def approxPi(m:real):real{
3
4      def pow(x:real, n:real): real{
5          var y:real = 1.0;
6          while(n>0.0){
7              set y = y * x;
8              set n = n - 1.0;
9          }
10
11         return y;
12     }
13
14     def denom(n:real):real{
15         return 2.0 * n + 1.0;
16
17     }
18
19     var pi:real = 0.0;
20     var i:real = 0.0;
21
22     while(i<m){
23         set pi = pi + 4.0 * pow(-1.0, i) / denom(i);
24         set i = i + 1.0;
25     }
26
27     return pi;
28 }
29
30 print approxPi(150.0);
```

We note that removing the return statement from line 11 gives us:

```
Semantic error: function ‘pow’ does not always return.
```

The following table shows which binary operators are allowed for different types.

| Binary Operator | real | int | bool | string |
|---|---|---|---|---|
| $+$ | ✓ | ✓ | | ✓ |
| - | ✓ | ✓ | | |
| $*$ | ✓ | ✓ | | |
| $/$ | ✓ | ✓ | | |
| $<$ | ✓ | ✓ | | |
| $>$ | ✓ | ✓ | | |
| $\leq$ | ✓ | ✓ | | |
| $\geq$ | ✓ | ✓ | | |
| $==$ | ✓ | ✓ | ✓ | ✓ |
| $!=$ | ✓ | ✓ | ✓ | ✓ |
| and | | | ✓ | |
| or | | | ✓ | |

So the code

```
print "Hello " + "world!";
```

is valid, however

```
print true + false;
```

yields the error

```
Semantic error: operator '+' cannot operate on type 'bool'.
```

TASK V

# Interpreter Execution

Yet another Visitor class was implemented, this time to perform the actions described by programs written in MiniLang. The interpreter execution pass executes the program and outputs results anytime a `print` statement is encountered.

## 5.1 How the interpreter was implemented

The implementation of the interpreter execution pass resembles that of the semantic analysis pass. The class `InterpreterScope` was created, this time having two maps, one map storing variables, their types and their current values, and one multimap storing functions, together with their return type and their declaration node (which was used for extracting certain information).

A `struct VarValue` was created to store the value of variables in the interpreter symbol table. This contains four members of type `float`, `int`, `bool` and `string`. In order to access the correct member which stores the actual value of the variable in our symbol table, two global variables are kept in `InterpreterVisitor`: `lastValue` which is of type `VarValue` and `lastType` which holds one of `REAL`, `INT`, `BOOL` or `STRING` (the types of our language MiniLang), and this would indicate which one of the members we should access to get the value of the variable. A possible improvement to this implementation would be to use a `union` instead of a `struct`, since only one of the four members of `VarValue` is ever used at a time, therefore a `union` suffices and takes up less memory. A `struct` was used here because `union`s do not allow members which have a non-trivial constructor like `string`.

The interpreter then has the job to declare variables and functions and store them in the respective map, perform function calls, assign new values to variables, work out expressions involving unary and binary operators and check the conditions of `if` and `while` statements every time they are run.

One thing to note is that although addition and subtraction have precedence over multiplication and division, the compiler works out expressions from left to right when it comes to multiplication and division themselves. So $6/2*3$ evaluates to 1 not 9, and we have to use parentheses and write $(6/2)*3$ in order to get 9.

## 5.2 Testing the interpreter

To test the interpreter, we wrote the program `funcPow()` given in the assignment sheet and shown below.

```
//Function definition for power
def funcPow( x : real , n : int) : real
{
    var y : real = 1.0;      //Declare y and set it to 1.0
    if ( n>0 )
    {
        while(n>0)
        {
            set y = y * x;  //Assignment y = y * x;
            set n = n - 1;  //Assignment n = n - 1;
        }
    }
    else
    {
        while (n<0) {
            set y = y / x;  //Assignment y = y / x
            set n = n + 1;  //Assignment n = n + 1
        }
    }
    return y;               //Return y as the result
}

print funcPow(5.0, 3);
```

The output we get is 125, as required.

We also ran the function `approxPi` defined previously, and with 150 iterations we got 3.13493 (which is sufficiently close to the value of $\pi$ considering the series converges quite slowly).

A recursive function `fact` which implements the factorial function was also implemented, in order to check whether scopes were being kept correctly and whether they return a value correctly.

```
def fact(n:int):int{
    if(n == 0){
        return 1;
    }

    else{
        return n * fact(n-1);
    }
}

print(fact(4));
```

This outputted 24, as required.

TASK VI

# Read-Eval-Print Loop

The way MiniLang was designed allows for user commands to be executed piecewise. To achieve this, a read-eval-print loop (REPL) was implemented.

## 6.1 Implementation of the REPL

The REPL was implemented entirely in the `main` class. The only change that was required in other classes was an addition of a non-trivial constructor in the `Scope` and `InterpreterScope` classes, to be able to pass an already declared scope and have functions and variables declared in user files stored in this scope's symbol table. The `main` consists of a `while` loop that keeps on running until the user decides to quit. The commands that can be used are:

- `#help` displays the available commands.
- `#load` loads a file from the same directory as the REPL executable and executes its contents. Variables and functions declared in the file will become available to use globally in the REPL. The file to be loaded is to be contained in quotation marks e.g. `#load "my_prog.txt"`
- `#st` shows functions and variables that have already been declared and which can be used in user commands.
- `#quit` exits the REPL.

The user can input single statements for execution, possibly including variables and functions that have already been declared, and declare functions directly from the REPL. The REPL allows for this by checking if there are any open curly brackets in the statement the user inputs once they hit enter, and if there are, it allows for more lines of code until the number of open and closed curly brackets match.

When loading a file to be executed by the REPL or declaring directly within the REPL, a temporary scope is used so that if the code has any errors, the variables and functions used in it are not loaded into the REPL global scope. If no error is produced, the declared variables and functions can then be stored in the global scope.

## 6.2 Demonstrating and testing the REPL

We now demonstrate some REPL features. First, we run the MiniLangI executable, and we are greeted with a welcome message. We then load the first MiniLang program in the assignment sheet, displayed below.

```
def funcSquare(x:real):real{
    return x*x;
}

def funcGreaterThan(x:real, y:real):bool{
    var ans:bool = true;
    if(y>x){set ans = false;}
    return ans;
}

var x:real = 2.4;
var y:real = funcSquare(2.5);
print y;
print funcGreaterThan(x, 2.3);
print funcGreaterThan(x, funcSquare(1.555));
```

After running `#st`, we get the following output.

```
stefaniatadama@steftop:~$./MiniLangI
-------------
MiniLangI 1.0
Type #load to load a program in MiniLang, #help for more information or
#quit to exit.


MLi> #load "test.prog"
6.25
true
false
MLi>#st
The functions and variables shown below are currently stored in the
symbol table.

--Variables--
real x = 2.4
real y = 6.25

--Functions--
funcGreaterThan(real,real)
funcSquare(real)

MLi>
```

We then use the declared functions and variables and work out some expressions.

```
MLi>x+y;
8.65
MLi>funcGreaterThan(x,x+y);
false
MLi>
```

Next, we declare a function from within the REPL and assign a new value to `x`, making sure both these changes are shown in the symbol table.

```
MLi> def funcLessThan(x:real, y:real):bool{
...    var ans:bool = true;
...    if(y<x){set ans = false;}
...    return ans;
...   }
MLi> set x = 3.5;
MLi> #st
The functions and variables shown below are currently stored in the
symbol table.

--Variables--
real x = 3.5
real y = 6.25

--Functions--
funcGreaterThan(real,real)
funcLessThan(real,real)
funcSquare(real)

MLi> funcLessThan(x,x+y);
true
MLi>
```

Finally, we make some mistakes to see how they're handled.

```
MLi>funcSquare(3);
Semantic error:  function ‘funcSquare’ has different parameters.
MLi> def f(x:int){return 0;}
Unexpected token on line 1:  expected ‘:’  after ‘)’.
MLi> set y = false;
Semantic error:  conflicting types - expected ‘real’ but expression is
of type ‘bool’.
MLi> #st
The functions and variables shown below are currently stored in the
symbol table.

--Variables--
real x = 3.5
real y = 6.25

--Functions--
funcGreaterThan(real,real)
funcLessThan(real,real)
funcSquare(real)

MLi>
```

A limitation encountered while testing the REPL is that programs ending with a single line comment result in a segmentation fault thrown by the lexer `nextToken()` function. This was handled using a try-catch in `nextToken()`. When inputting a program ending in a comment, we get the below output.

```
MLi> #load "test.prog"
Lexical error - cannot end program with single line comment.
MLi>
```

# Bibliography

[1] Wikipedia contributors. Abstract syntax tree (introduction). `https://en.wikipedia.org/wiki/Abstract_syntax_tree`.

[2] Wikipedia contributors. Semantic analysis (compilers). `https://en.wikipedia.org/wiki/Semantic_analysis_(compilers)`.

[3] Evan Wallace. Finite state machine designer. `http://madebyevan.com/fsm/`.

[4] Prof. Douglas Thain. The abstract syntax tree. `https://www3.nd.edu/~dthain/courses/cse40243/fall2016/chapter6.pdf`.

[5] Puppy & Matthieu M. Why compiler doesn't allow std::string inside union? `https://stackoverflow.com/questions/3521914/why-compiler-doesnt-allow-stdstring-inside-union?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa`.