Graphical user interface, text

Description automatically generated

**CPS2000 – Compiler Theory and Practice**

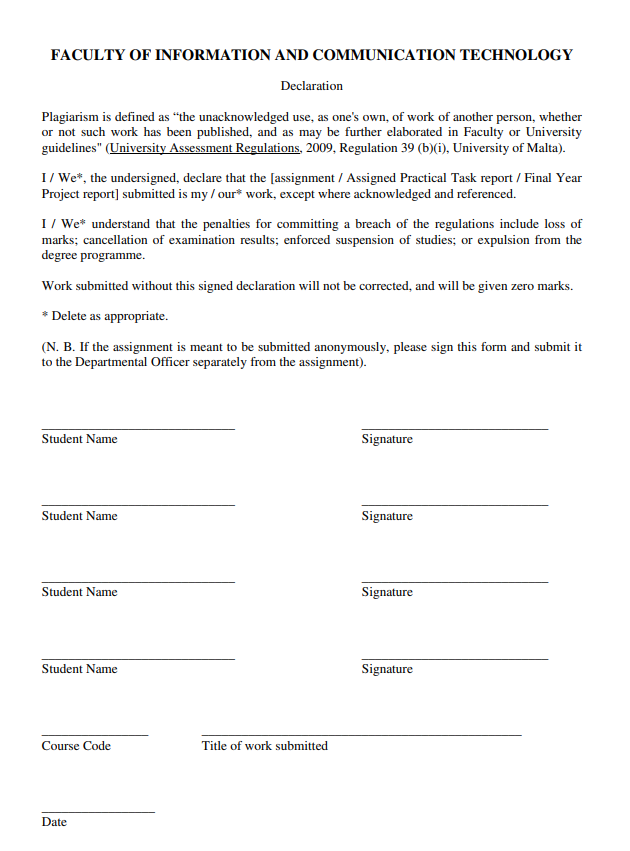
Cristina Cutajar\* (230802L)

\*B.Sc. (Hons) Artificial Intelligence

Study-unit: **Compiler Theory and Practice**

Code: **CPS2000**

Lecturers: **Dr Sandro Spina**



CPS2000 – Compiler Theory and Practice

CPS2000

19/06/2022

Cristina Cutajar

## Implementation

The aim of this project is to implement a lexer, parser, semantic analyser, and interpreter for the TinyLang programming language. This needs to be done by first developing a Table-driven lexer, then developing a Hand-crafter LL(k) parser and then developing 3 visitor classes. These 3 visitor classes consist of an AST XML Generation Pass, a Semantic Analysis Pass and an Interpreter Execution Pass.

In order to complete this project, I developed the code in C++.

### Task 1: Table-driven lexer

The aim of this task was to develop a lexer for the TinyLang language. In order to do this, I first extracted a set of tokens from the given EBNF. The tokens I chose to implement are the following: Boolean, int, float, char, identifier, multiplicative operator, additive operator, relational operator, equals sign, comment, punctuation, keyword, end of file and error. These were implemented in Tokens.h with type Token\_ID as be seen:

Text

Description automatically generated

The comments that can be seen in the above code correspond to the DFA state that the token corresponds to. In the tokens.h file, I also developed a struct for the Tokens with a variable to store the token type and a variable to store the toxen’s contents (lexeme).

Text

Description automatically generated

After choosing the tokens, I created the following DFA:

Diagram

Description automatically generated

I developed the DFA in order to be able to convert it into executable code. This was required to be done using the table-driven approach and so I created a transition table. The transition table consists of all the states seen in the above DFA (S0-S22) as well as the error state (Se) as rows. The columns of the transition table were as follows: letter, digit, printable characters, ‘\n’, ’.’, ‘”’, ‘\_’, ‘\*’, ‘/’, ‘+’, ‘-‘, ‘<’, ‘>’, ‘!’, ‘=’, punctuation characters and EOF character. I separated the characters in this way in order to make sure that everything can correspond to the given EBNF without any errors or mistakes. The transitive table I created can be seen:

Table

Description automatically generated

The numbers in the above table refer to the states and all the empty spaces refer to the error state. I implemented the transition table as the above, in the states.h file.

In the states.h file I also created a struct for the States. The struct contains an int variable to store the state id, a bool variable to store whether or not the state is a final state and a variable of type Token\_ID to store the token type of the state. I also created a State class which contains a init\_DFSA() function. In this function, I hardcoded a list with the final state’s ids. Then, the function will loop according to the set global variable number of states, which was set to 23 and in each loop, it will set the state id and set the state’s final\_state variable to either true or false depending on if the id is found in the final\_states list. Afterwards, I hardcoded the token type of each state according to the DFA.

For this task, I also created the lexer.cpp and lexer.h files which contain the Lexer class. This class contains all the functions that are needed for the lexer. This class contains a private vector to store the program’s input as well as the following private functions: get\_char\_pos(), get\_next\_state(), check\_if\_keyword() and get\_next\_token(). The class also has the char\_index and line\_index public int variables that will be used to keep track of the character and line indexes of the file. It also has the following public functions: read\_program(), see\_next\_token(), return\_next\_token(), clear\_program() and set\_cmd\_code().

The get\_char\_pos() function is used to get the position of the given character from the transition table by checking what the character is. The get\_next\_state() function takes in the current state and current character and gets the next state from the transition table. It does so by calling the get\_char\_pos() function to get the position of the character in the transitive table and then it will return the transition table state reached after applying the character on the current state. The check\_if\_keyword() function takes in a string and checks if it is a keyword by comparing the given lexeme to the keywords in the keywords list. The get\_next\_token() function is used to continue reading from the input file and return the next token. This function is implemented according to the given table-driven lexer pseudocode in the Lexical Analysis notes:

Diagram

Description automatically generated

The read\_program() function is used to open the given file, read the program input and set it into the private vector of the Lexer class. The see\_next\_token() function returns the next token from the program without changing the positions of the index. It does so by keeping track of the original line and char indexes and then it will call the return\_next\_token(). After the token is returned, the function will reset the line and char indexes to their original values. The return\_next\_token() function calls the get\_next\_token() function to get the next token. It will then check if the token is a comment and if so, it will keep calling the get\_next\_token() function until a token which is not a comment is returned. Then it will return the token. The clear\_program() function is used to clear the private vector of the Lexer class. Finally, the set\_cmd\_code() is a function which calls the clear\_program() function to clear the private vector and then loops through the command line input to store the code.

### Task 2: Hand-crafted LL(k) parser

The aim of this task is for a hand-crafter predictive parser to be developed for the TinyLang language. I implemented this in such a way that the Parser makes use of the Lexer’s line\_index variable as well as the see\_next\_token() and return\_next\_token() functions in order to get the next valid token from the lexer as well as to read one lookahead character.

Firstly, I designed the structure of the abstract syntax tree. The AST\_node class is the root node of the AST. Then, the AST\_node\_expression, AST\_node\_operator and AST\_node\_statement classes are all subclasses of the AST\_node class.

The AST\_type, AST\_formal\_param and AST\_formal\_params classes, which can be found in the ‘others’ folder, are also subclasses of the AST\_node class.

The ‘expressions’ folder contains the AST\_actual\_params, AST\_expression, AST\_node\_factor, AST\_simple\_expression and AST\_term classes which are all subclasses of the AST\_node\_expression class. The ‘expressions’ folder also contains the ‘factors’ folder. This folder contains the AST\_function\_call, AST\_identifier, AST\_node\_literal, AST\_sub\_expression and AST\_unary classes which are all subclasses to the AST\_node\_factor class. The ‘factors’ folder contains the ‘literals’ folder which has the AST\_bool, AST\_char, AST\_float and AST\_int classes as subclasses of the AST\_node\_literal class.

The ‘operators’ folder contains the AST\_additive\_operator, AST\_multiplicative\_operator, AST\_relational\_operator and AST\_unary\_operator classes as subclasses of the AST\_node\_operator class.

The ‘statements’ folder contains the AST\_assignment, AST\_block, AST\_for, AST\_function\_declaration, AST\_if, AST\_print, AST\_return, AST\_variable\_declaration and AST\_while classes. These classes are all subclasses of the AST\_node\_statement class.

Each class has a .h and .cpp file with the necessary variables and functions.

The Parser class was also developed to be able to parse through the input of the program and produce a correctly structured abstract syntax tree. The Parser class can be found in the parser.h and parser.cpp files. This class contains a variable of type Token class called current\_tok which is used to keep track of the current token during parsing, as well as an object called lexer of the Lexer class which is used to use the Lexer class functions. It also contains functions which parse all the factors, expressions, operators, statements, type, formal parameter and formal parameters, separately, as needed. The Parser class also contains the Parser() constructor function which is used to create a new AST as well as a destructor function which is used to clear the AST. It also has the parse\_AST() function which is used to start parsing the program input by calling the parse\_statement() function which in turn calls the appropriate parse functions, until the entire program is parsed. Each parse function will either call another function if it has child class variables that need to be parsed or it will return a new object of the function type. This is done in order to construct the program AST structure. If there is an error during parsing, an error message is displayed and a nullptr will be returned, terminating the parser. Afterwards, if no error occurs during parsing, the final AST is returned describing the structure of the input program.

### Task 3: AST XML Generation Pass

For this task, it was required to implement the Visitor design pattern to take in the AST produced by the parser in Task 2 and output a properly indented XML representation of it.

In order to achieve this, I first created the Visitor class in the visitor.h file. This class contains all the virtual visit methods which will be used for Task 3, 4 and 5 for all the non-abstract AST root node subclasses.

I then created the XML\_Visitor class as a subclass of the Visitor class. The code for this class can be found in the XML\_Visitor.h and XML\_Visitor.cpp files. This class contains multiple visit functions, one for each of the literals, factors, expressions, statements, and operators. As well as visit functions for type, formal parameter, and formal parameters. This class also contains a private string variable named ‘indentation’, a public constructor XML\_Visitor(), which initialises the indentation variable with “” and a public visit\_AST() function which is used to visit every node in the AST. All the visit functions will first output the initial indentation display the opening tag based on their respective parameter type, the variable value and the closing tag. Some visit functions however, whose node is made up of other nodes, will first display the node’s opening tag with the initial indentation and then they will increase the indentation and call the other nodes’ accept() functions which calls the node’s visit function for the node to be visited. After the required accept() functions are called, the added indentation is removed and the closing tag is displayed. Some examples of this are the visit() functions for the AST\_function\_call and AST\_sub\_expression nodes:

Text

Description automatically generated

Some nodes contain optional variables. For these nodes, the visit() function uses if statements to check if the optional variable is included or not. If the variable’s value is empty, then the accept() function for that variable’s node is not called. An example of this is the visit() function for the AST\_if node where the else block is optional:

Text

Description automatically generated

Some of the visit() functions whose node’s value is made up of a vector of other nodes, a loop is performed through the node’s components and the accept() function is called for each component. An example of this is the visit() function for the AST\_block node:

Text

Description automatically generated

### Task 4: Semantic Analysis Pass

For this task, it was required to implement another visitor class which traverses the AST to perform type-checking. In order to develop the requirements, a symbol table needed to be implemented. In order to implement this, I first implemented the Value class. For this class, I also created an enum called Val\_Type which will be used to store whether the value of a variable is a float, int, bool, char or a function. The Value class has a variable of type Val\_Type called ‘id\_type’ as well as float, int, bool and char variables to store the values and a vector to store any function parameters if the variable is a function. This class also contains a constructor and destructor as well as some other functions. The constructor sets the id\_type to the given type and sets the other variables to nullptr:

Text

Description automatically generated

Meanwhile, the destructor, deletes the variables and clears the vector. The others functions set the id\_type to the appropriate type and set the given value to the appropriate variable. The other variables are set to the nullptr.

Then, I implemented the Symbol\_Table class in the symbol\_table.h and symbol\_table.cpp files. This class has a private vector map to store the contents of the symbol table. The class also contains a constructor which creates the symbol table and call the open\_scope() function. A destructor function can also be found which clears the symbol table. Apart from these, this class also contains the following functions: open\_scope(), close\_scope(), search(), insert\_variable(), assign\_variable(), check\_function\_call() and return\_value().

The open\_scope() function adds a new map element at the end of the symbol table vector and the close\_scope() function deletes the last element from the symbol table vector.

The search() function is used to check if the given identifier is present in the symbol table. If it is, its’ position will be returned. Otherwise, -1 will be returned.

The insert\_variable() function will first call the search() function and if -1 is returned, the function will insert the identifier and its’ value into the symbol table and the function will return 2 to show that it succeeded. If the search() function finds the identifier in the symbol table, then the insert\_variable() function will return -1.

The assign\_variable() function will call the return\_value() function to get the value of the identifier. It will then check if the value is not null and if it has the same type as the given value. If so, its appropriate value will be set and 2 is returned. Otherwise, -1 will be returned.

The check\_function\_call() function calls the return\_value() function to get the value of the given identifier and if it is not a nullptr, if will check if it’s function parameters value is a nullptr. If so, an error will be displayed and false will be returned. Otherwise, the function will check that the size of the function parameters value corresponds to the given parameters size. If not, an error will be displayed and false will be returned. It will then check to make sure that the parameters correspond and if so, it will return true. Otherwise, if at least one parameter does not correspond, false will be returned.

The return\_value() function calls the search() function to get the position of the given identifier. If the returned position is not -1, then the function will return the value found in the symbol table at the returned position. Otherwise, a nullptr will be returned.

Then, I created the semantic\_analysis\_visitor.h and semantic\_analysis\_visitor.cpp files in order to implement all the necessary functions to perform the semantic analysis. I created the SA\_Visitor class which contains a Symbol\_Table object, a bool flag to be used to make sure that every function has a return statement, a map used to store the function declaration parameters and two variables of type Val\_Type which will be used to store the type of the current variable and the return type of the current function. This class also contains a constructor which initialises a new symbol table and sets the flag to false, as well as a destructor which deletes the symbol table. I also implemented the visit\_AST() function which visits every node in the AST to display that no errors occurred during semantic analysis. This class also contains multiple visit functions, one for each of the literals, factors, expressions, statements, and operators. As well as visit functions for type, formal parameter, and formal parameters. These were implemented so that semantic analysis is performed for all the nodes of the AST.

The visit() functions for the literals, sets the current type variable to the appropriate type. The other visit() functions, will make use of if conditions and for loops in order to check that the types are all correct, as well as calling the accept() function for any node that is part of the current node’s value. For example, the visit() function for the AST\_unary node will first call the accept() function for the expression node so that the expression node’s visit() function is called to check that the expression has the correct types as well as to get the type of the expression. Then it will make use of an if condition to check if the given operator is a ‘not’ or a ‘-‘ as required in the EBNF. If the operator is a ‘not’, the function will check the current type of the expression and if the expression is not of type bool, an error will be returned. Similarly, if a ‘-‘ operator is found and the expression is of type bool, then an error will be returned as the ‘-‘ operator can only be applied to float or int variables. This can be seen here:

Text

Description automatically generated

Apart from this, there are certain visit() functions which are left empty. This is due to their node’s type already being handled in another node’s visit function(). For example the visit() function for the AST\_additive\_operator node, is empty as the type that can be used with the additive operators has already been checked in the AST\_simple\_exppression node’s visit() function.

This is the empty AST\_additive\_operator node’s visit() function.

Text

Description automatically generated

This is a part of the AST\_simple\_expression node’s visit() function which shows how the types used with the additive operators are checked:

Text

Description automatically generated

### Task 5: Interpreter Execution Pass

Due to time constraints, I was unable to develop the visitor class which traverses the AST to simulate an interpreter which executes the test program.

## References

1. S. Spina. Compiler Theory and Practice [PowerPoint Slides]. Available: <https://www.um.edu.mt/vle/course/view.php?id=59226> (Accessed May. 25, 2022)
2. https://cplusplus.com/reference/vector/vector/push\_back/