CoPro: Parser

T. Rohan - CS20BTECH11064
I. Rajasekhar - CS20BTECH11020
P. Vikas - CS20BTECH11037
S. Satvik - CS20BTECH11048
B. Revanth - CS20BTECH11007
J. Vamsi Preetham - CS20BTECH11058
G. Yagnavalkya - CS20BTECH11019

What's After Lexer?

- We have seen how a lexical analyzer can divide an input file into a token stream.
- However, a lexical analyzer cannot check the syntax of the given code using just regular expressions.
- This gives rise to the need for the next phase of the compiler, something which takes the tokens from the lexical analyzer and then checks its syntax.

Parser

- **Second phase of a compiler.**
- **Tokens → Syntax Tree**
  - The parser takes tokens as an input from the lexical analyzer.
  - It then checks the format of the tokens against the given grammar rules.
  - The output we get after this phase is an parser tree.

Parser

Bison

- The parser takes tokens as input from the lexer, and then analyzes it against the production rules mentioned.
- It then outputs a syntax tree.
- Our parser has been written using bison.
- The tree would then be passed on to the next stage of semantic analysis.

Implementation

- In parser, we get tokens as input from the lexer file.
- Inside the parser file, we first declare all tokens which could be an output from the lexer file.
- Then we wrote the grammar of CoPro. The grammar consists of rules to check the syntax of the language. It then goes on to build a syntax tree for the given code, checking if its errorless.
- The grammar starts building the tree from *'Translation Unit'*, which acts as a start node for our grammar.

Implementation

- We then write the C code for the parser file. This code is used to execute functionalities in the parser.
- We have built a symbol table to store variables, keywords, constants and functions.
- Each entry in the symbol table has 4 attributes: symbol, datatype, type, line number.
- We have then built the syntax tree, binary in our case, for the given code, with root being *'Translation Unit'*. This will be the 'head' of our tree.

Implementation

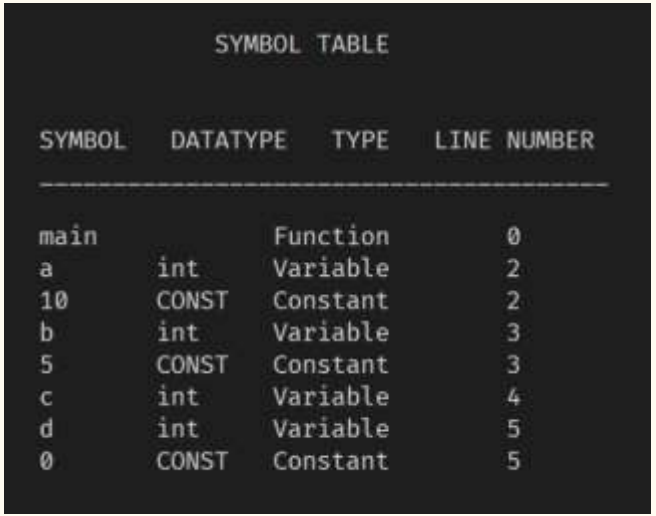
- For this tree, the node contains the name of the token along with the left and right subtrees.
- We have then added functions to add nodes to the tree and to print the preorder traversal of the tree.
- The creation of nodes and setting up the tree is done in bits of C code attached to each match of the corresponding grammar rule.
- In this way, we continue to build a syntax tree for the given code, and print out a preorder traversal of the tree as the output.
- We do this for 5 test cases given in ‘testCases’ folder, printing the traversal in the terminal.

Symbol Table Example

.cop file-

```
main -> int
<<
    int a = 10;
    int b = 5;
    int c = a + b;
    int d = 0;
>>
```

Symbol Table-



SYMBOL TABLE

SYMBOL	DATATYPE	TYPE	LINE NUMBER
main		Function	0
a	int	Variable	2
10	CONST	Constant	2
b	int	Variable	3
5	CONST	Constant	3
c	int	Variable	4
d	int	Variable	5
0	CONST	Constant	5

Parser Tree Example

.cop file-

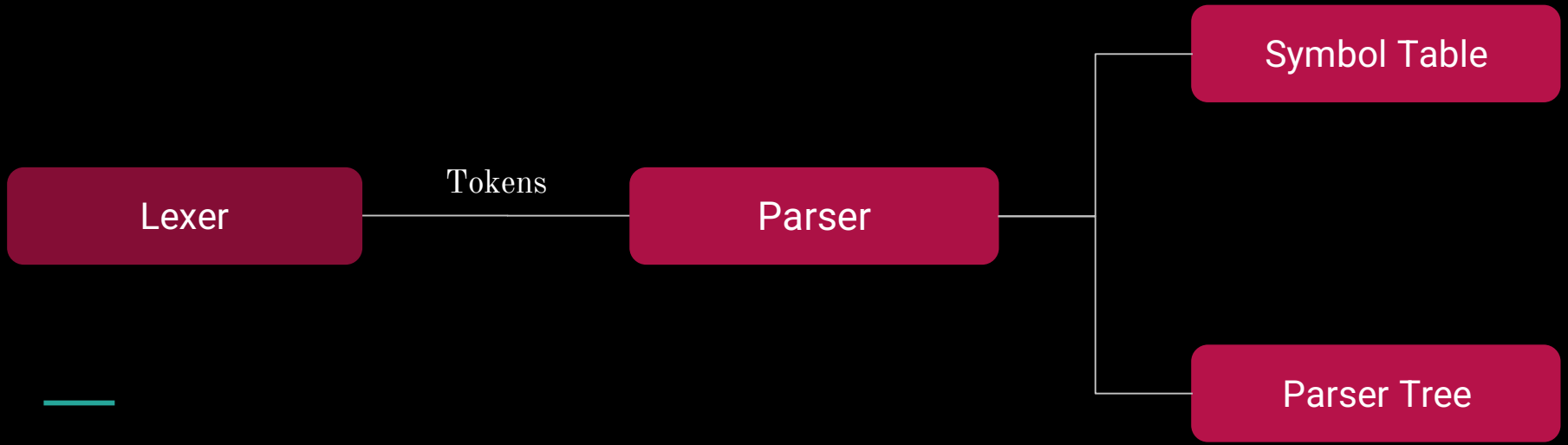
```
main -> int
<<
    int a = 10;
    int b = 5;
    int c = a + b;
    int d = 0;
>>
```

Parser Tree-

```
Preorder traversal of the Parse Tree:

TRANSLATION_UNIT, EXTERNAL_DECLR, main, OPTIONS, COMPUND STATEMENT, DECLR_LIST, DECLR_LIST,
DECLR_LIST, DECLR, DECLR_SPCIF, INIT_DECLR_LIST, INIT_DECLR, DECLR, a, INITIALIZER, ASS_EXPR
, COND_EXPR, LOGI_OR_EXPR, LOGI_AND_EXPR, EQ_EXPR, RELATIONAL_EXPR, ADDITIVE_EXPR, MUL_EXPR,
 CAST_EXPR, UNARY_EXPR, POSTFIX_EXPR, 10, DECLR, DECLR_SPCIF, INIT_DECLR_LIST, INIT_DECLR, D
ECLR, b, INITIALIZER, ASS_EXPR, COND_EXPR, LOGI_OR_EXPR, LOGI_AND_EXPR, EQ_EXPR, RELATIONAL_
EXPR, ADDITIVE_EXPR, MUL_EXPR, CAST_EXPR, UNARY_EXPR, POSTFIX_EXPR, 5, DECLR, DECLR_SPCIF, I
NIT_DECLR_LIST, INIT_DECLR, DECLR, c, INITIALIZER, ASS_EXPR, COND_EXPR, LOGI_OR_EXPR, LOGI_A
ND_EXPR, EQ_EXPR, RELATIONAL_EXPR, +, ADDITIVE_EXPR, MUL_EXPR, CAST_EXPR, UNARY_EXPR, POSTFI
X_EXPR, a, MUL_EXPR, CAST_EXPR, UNARY_EXPR, POSTFIX_EXPR, b,
```

Control Flow



Thank You