

ConcurrX

Parallelized, Threaded, and Graphed for Performance Brilliance.

Project Manager: Varshini Jonnala

Language Guru: Anudeep Rao Perala

System Integrator: Tushita Sharva Janga

System Architect: Narsupalli Sai Vamsi

Tester: All

Outline

- Introduction
- Compiler Architecture
- Lexer
- Interface
- Parser
- Execution Flow
- Example



Introduction

- A language tailored for developers focusing on parallel computing and performance analysis
- It provides built-in features to parallelize tasks efficiently across multiple threads.
- The language includes tasks for benchmarking, allowing users to compare the execution time of them across different thread numbers.
- It supports smooth inclusion with popular parallel computing libraries, making it straightforward for the developers to use the existing tools and resources efficiently.

COMPILER ARCHITECTURE



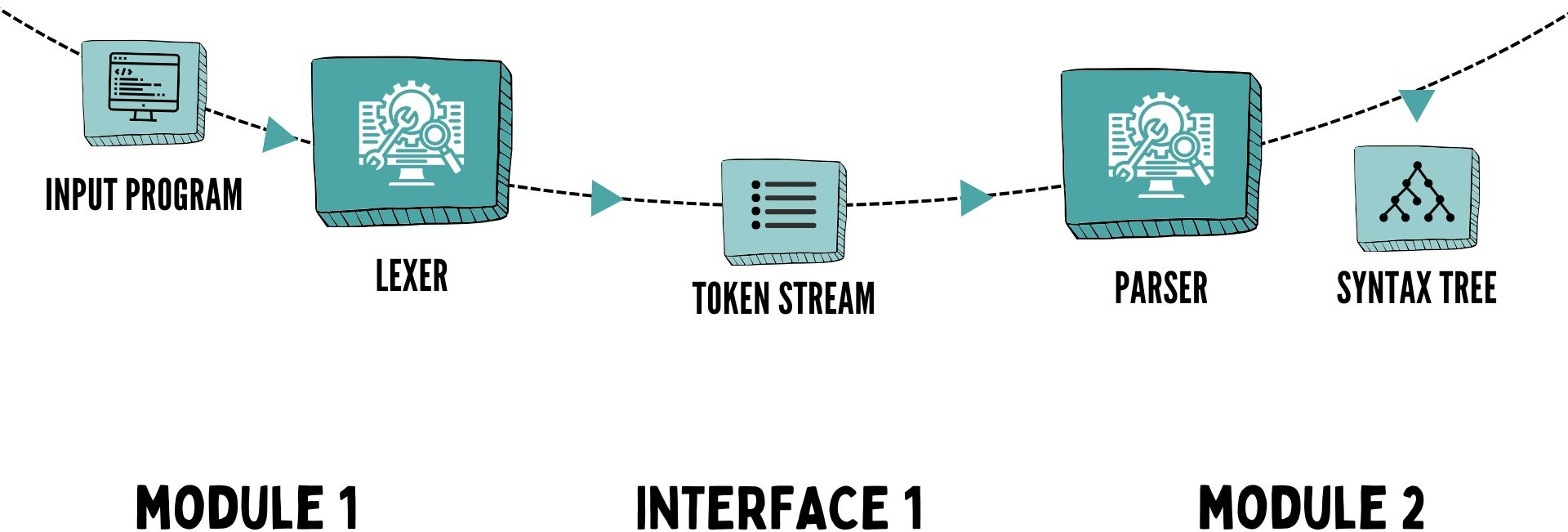
Compiler Architecture

We are using the Module-Interface architecture in our compiler for the DSL.

The architecture is as follows:

- A distinct and encapsulated unit of the compiler design that is responsible for a specific aspect of the compilation process called as “module”
- An “Interface” between every 2 modules, which acts as a point of communication between those 2 modules which defines how the information flows between the modules
- The input stream gets processed in the modules, flows through the interface after processing and finally we get the output .
- In these phase, we have 2 modules and 1 interface. They are Lexer, Parser and the TokenFlowNexus

FLOW

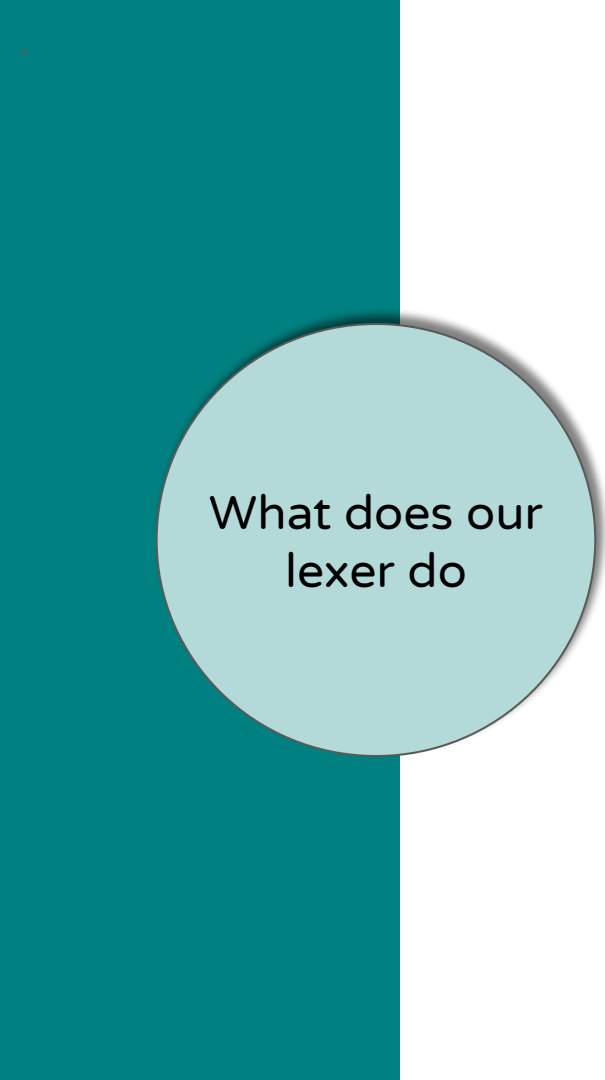


MODULE 1: LEXER



The lexer

- This module of our compiler reads the input source code and breaks it down into meaningful fragments called tokens
- Tokens represent the smallest units of meaning within the code. They are categorized elements, that has a specific role in the syntax and semantics of the programming language.
- The lexer's main role is to transforms the raw code into a token stream and then returns these tokens to the next module through the interface
- It does so by reading the source code character by character and groups them into meaningful chunks based on the regular expressions that we define.



What does our lexer do

- The lexer receives the .cx file as an input
- It removes the comments and white spaces
- Converts the file into a stream of tokens
- It returns all the valid tokens to the next module using the interface
- An error would be returned as a LEX_ERROR token which would stop the tokenizing and parsing
- The following are the different types of tokens that we have defined for our language.
- It also outputs a tokenstream file, which contains all the different

RESERVED KEYWORDS

Keywords that have special meaning in the program and can not be used directly as a variable name.

PUNCTUATIONS

Used for various purposes, including syntax and code structure. The punctuation marks used include ;,->

CONSTANTS

Constants are fixed values, like numeric literals or string literals,

SPECIAL SYMBOLS

In our case, special symbols refer to the various kinds of brackets that are used

TOKEN TYPES

IDENTIFIERS

The names we supply for variables, types, AND functions in the program

IDENTIFIERS FOR VARIABLE NAMES AND FUNCTIONS

IDENTIFIERS FOR TYPES

NON ATOMIC USER DEFINED

ATOMIC USER DEFINED

DATATYPES

Classification of data which tells the compiler or interpreter how the programmer intends to use the data.

NON ATOMIC SIMPLE

ATOMIC SIMPLE

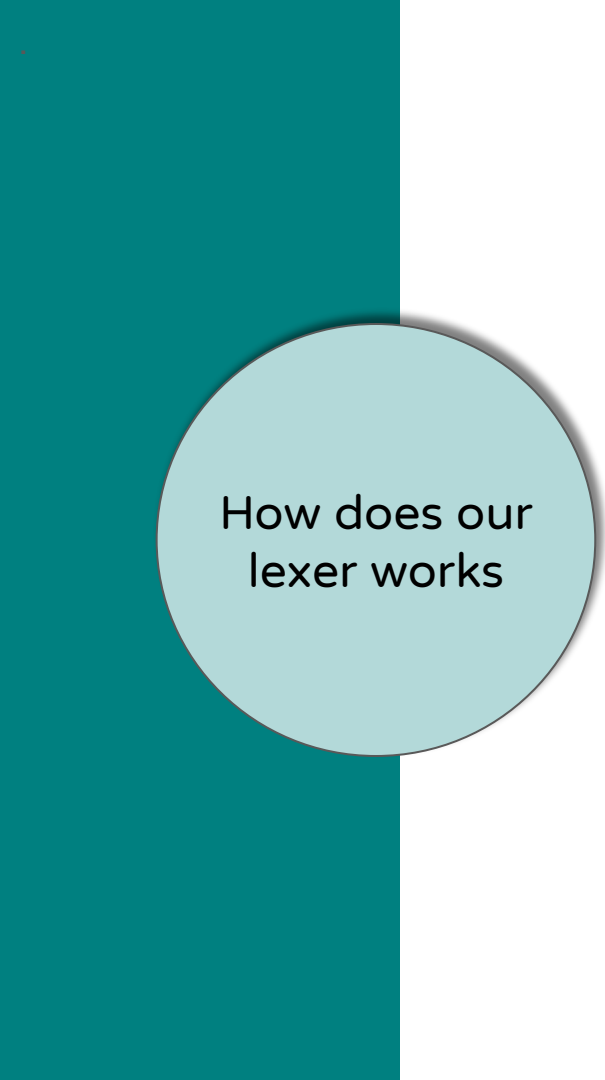
NON ATOMIC ARRAY

ATOMIC ARRAY

TO BE IGNORED

COMMENTS

WHITESPACE



How does our lexer works

- Each of the above defined tokens are recognized through the patterns that we have specified in the rules section of the lex.l file
- Rules are expressed using the regular expressions.
- Once the sequence is matched with a given regular expression according to the precedence given, the corresponding C++ functions would be executed and corresponding token is returned
- We write our lexical analyzer in flex and we use the following libraries for running it
- Libraries used in lex.l :
 1. cpp iostream, unordered_set, string
 2. We include bison.tab.h for importing the token types

INTERFACE 1: LEXER-PARSER INTERFACE



The Interface

- The interface acts as a communication link between the lexer and parser.
- Once the lexer has categorized the source code into tokens, the interface becomes the bridge for passing of these organized stream of tokens to the parser
- It ensures that the parser receives a well-defined set of language elements, allowing it to perform its syntax analysis effectively
- It involves a set of functions or methods that allow the parser to request the next token from the lexer and get to know about the type of the current token (Check this line once)

MODULE 2: PARSER

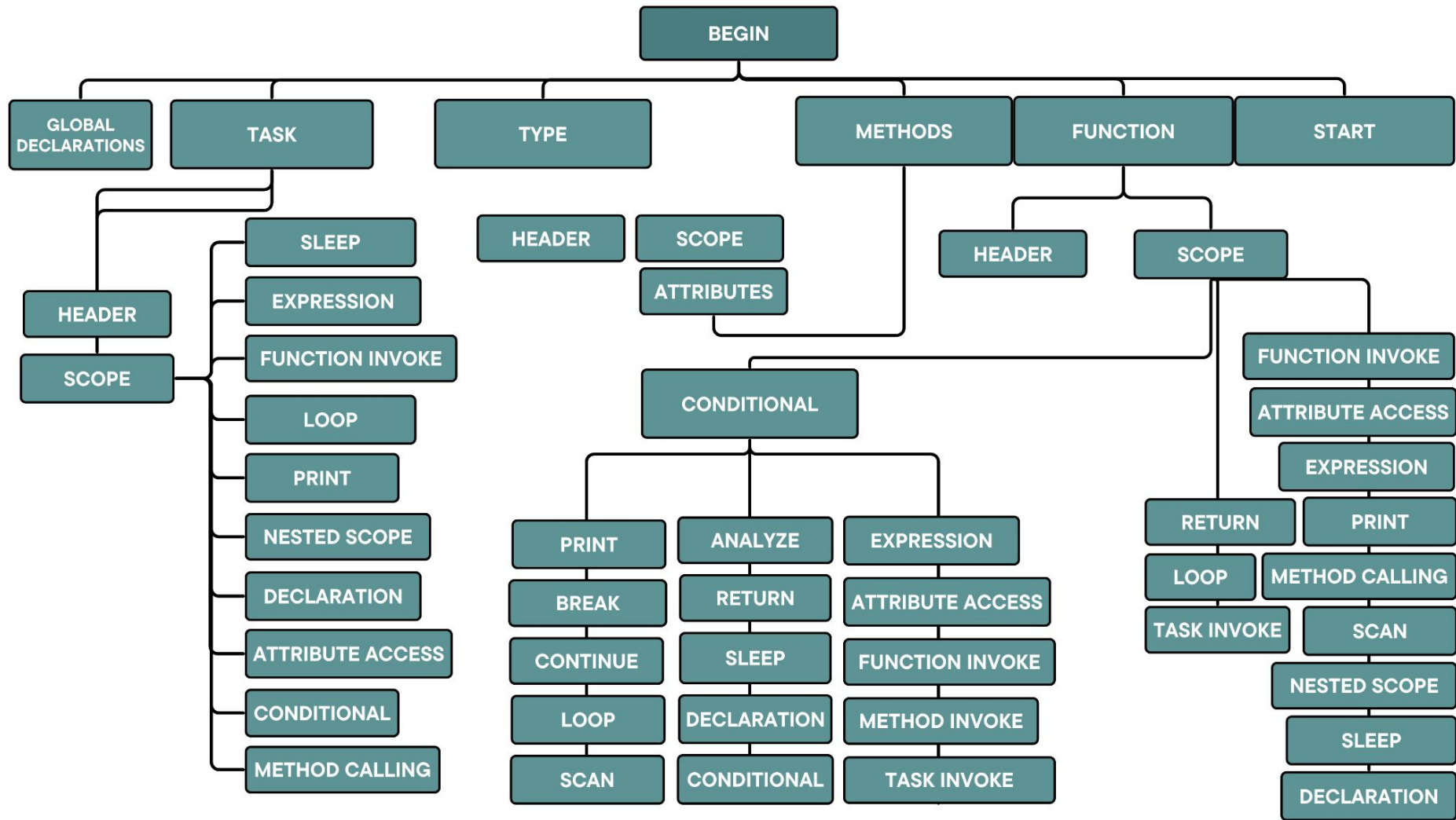


The Parser

- This module of our compiler transforms the organized tokens from the lexer into a structured blueprint, which would be used in further steps of compilation.
- It ensures whether the arrangement of tokens follows the grammar rules of the programming language
- The parser builds a tree called the Abstract Syntax Tree (AST) that shows how different parts of the code relate to each other. It helps the compiler understand how the code is put together.
- The following are the different sets of rules that we have come up for our DSL.

How does our Parser works

- The parser gets its tokens as input from the lexer file, through interface
- It contains predefined set of tokens that it accepts, by including `bison.tab.h`
- We used bison for generating our parser
- The parser contains grammar rules, these rules dictate how the code should be structured
- It then checks whether the token stream can be produced from the grammar rules
- Each of the grammar rules in the parser would help us in building the parse tree by checking whether it can be produced by one of the given rules
- The parser then converts the input token stream into an Abstract Syntax Tree(AST)
- Any error in the syntax would then be outputted to the standard output
- The following is the basic syntax tree that we get after parsing



Execution Flow

We run the bash file which contains the following order of execution

1. ***flex lex.l***: running the lex file would generate ***lex.yy.c***
2. ***bison bison.y***: running the parser file which would output ***bison.tab.c*** and ***bison.tab.h***
3. The ***bison.tab.h*** file will be used in ***lex.yy.c***
4. ***g++ -c main.cpp***: running this file would generate ***main.o***
5. ***g++ -o lex.yy.c bison.tab.c main.o -lfl***: Finally running this command would generate a compilers as an executable which tokenizes and parses the input source code
6. ***./exe 1***: We run this executional file/Compiler together with the input, which outputs a tokenized file, a parsed file and also points out any errors in our source code by checking whether the program fits each and every rule of the language

```
$ flex lex.l
$ bison -dtv bison.y
$ g++ -c main.cpp
$ g++ -o exe lex.yy.c bison.tab.c main.o -lfl
$ ./exe 1
```

