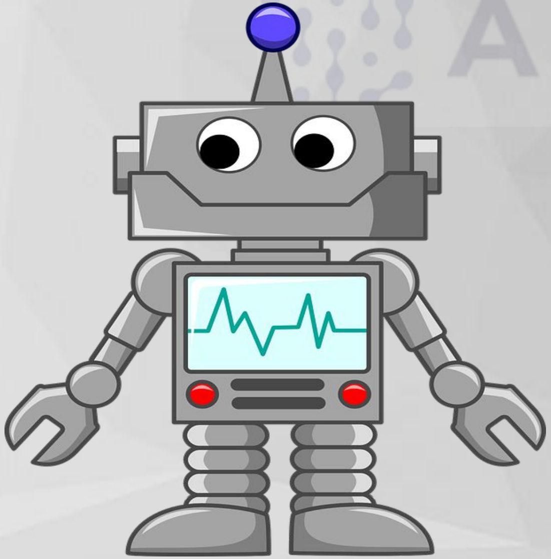
AUTOMATON



GROUP 6

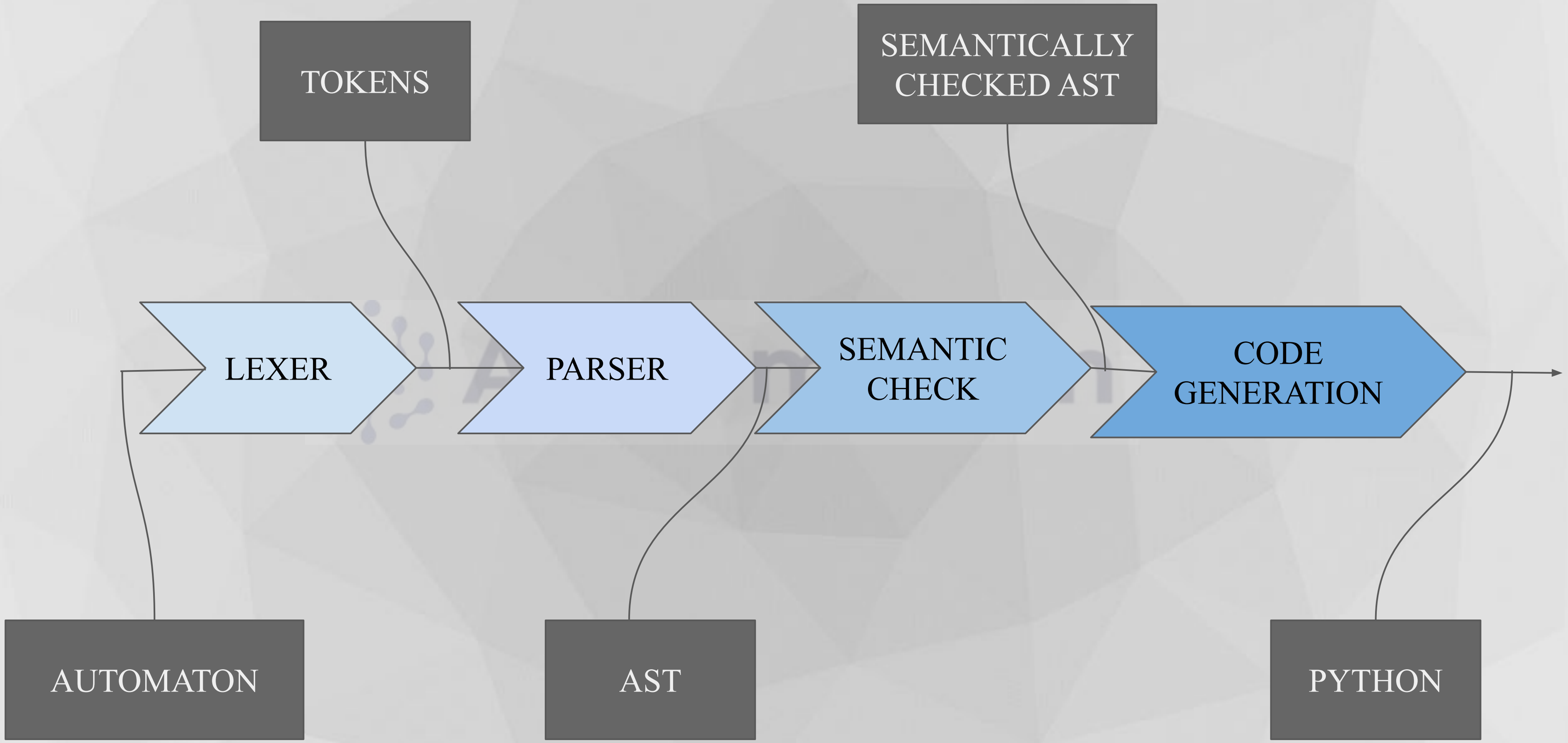
V. Sathwik - Project Manager
D. Sarat Chandra Sai - System Architect
Sushma Nalari - System Integrator
Pochetti Rashmitha - System Architect
L. Harsha Vardhan Reddy - Language Guru
K. Lakshmi Sravya - Tester
K. Sri Teja - Tester

INTRODUCTION

- **Idea:** To enable a programming language that emulates DFA (Deterministic Finite Automata).
- Automaton is organized and executed similar to an Automata diagram,emphasizes modularity and organization of code into short nodes that transition to each other until reaching one of the end states.
- Our language blends the functional and imperative programming styles to allow programmers to abstract away implementation details.

States in Automaton

- Automaton programs contain nodes/states, and the transition between these states takes place based on the conditions we check.
- Transition nodes include transition statements, which check the expressions, and use them if the statement is true. All transition nodes should be kept up to date. End node.
- The state of an automata is defined by name, followed by flower brackets , which contains multiple functions inside it, with either the transition or return statements.
- **Start state:**All the Automaton codes must begin with the “start” state, where the 1st state to be chosen is given.
- **Accepting state:**If a finite state machine finishes an input string and is in an accepting state, the string is considered to be valid


TOKENS
SEMANTICALLY CHECKED AST
LEXER
PARSER
SEMANTIC CHECK
CODE GENERATION
AUTOMATON
AST
PYTHON

First Phase: LEXER

LEXICAL ANALYSIS

- A single element of a programming language is considered as a **token**. It consists of constants, identifiers, separators, keywords, and operators.
- Lexical Analysis is used to tokenize a given source code. It returns all the valid tokens and mentions about the invalid tokens like invalid operators or invalid characters.
- It also removes the comments and white spaces.
- LEXER is used for lexical analysis. The code for lexer is written using OCAML.

LEXER

- “input.am” source code is used to take input stream for the lexer.
- The generated tokens are taken by the parser and an Abstract Syntax Tree is generated (will be used later).

Working of Lexer:

- The tokens of our source code are pre-defined in “lexer.mll”. Using Ocamllex, “lexer.ml” is generated from the above file.
- The corresponding identifier/operator/keyword names are printed for a valid token.
- In case of invalid tokens, “invalid token” is returned.
- Without writing grammar, just empty fields are defined in “parser.mly”.

Second Phase: PARSER

PARSER

- PARSER is used for checking/verifying the grammar of our source language.
- Parsing functions take a lexical analyze and a lexer buffer as arguments, which is a function from lexer buffers to tokens and return the semantic attribute of the corresponding entry point.
- Our parser is built using OCAMLYACC.

- **GRAMMAR:** A program which is generally represented as a sequence of ASCII characters is changed into a syntax tree using grammar rules, which describes the ordering of symbols in the language.
- **Ocamlyacc:** The *ocamlyacc* command is used to generate the parser for a given context free grammar.

Parser Code Explanation :

At the start of the code we arrange the tokens according to the preferences, %left and %right are used for associative precedences.

It usually checks all data provided to ensure it is sufficient to build a data structure in the form of a parse tree.

It checks data types,variable types to see whether declared properly or not.

It checks whether the statement written in example are declared properly or not.

IMPLEMENTATION

- Context-free grammar specification is defined in “parser.mly”. Using Ocamlyacc, “parser.mli” and “parser.ml” are generated.
- “ast.ml” file is the result of the syntax analysis phase of the compiler. The structure of our program code is represented in this file.

Third Phase: SEMANTIC CHECK

SEMANTIC ANALYSIS

- Semantic Analysis is the process of drawing meaning from a text i.e, it interprets sentences so that computers can understand them and helps us in maintaining the semantic correctness of the program.
- Functions of Semantic Analysis are Type Checking, Label Checking and Flow Control Check
- It occurs during compile time(static semantics) and run time(dynamic semantics).
- Syntax tree of the parser phase and the symbol table are used to check the consistency of the program in accordance with the language definition.
- It stores the gathered information in either syntax tree or symbol table which would be used by compiler during the intermediate-code generation.
- Errors recognized by semantic analyzer are Type mismatch, Undeclared Variables and Reserved identifier misuse.
- The output of semantic phase is the annotated tree syntax.

ABSTRACT SYNTAX TREE:

- The AST (Abstract Syntax Tree) is a hierarchical tree that represents the source code and the nodes of the tree represent construct of the code.
- The compiler generates symbol tables on the AST during semantic analysis.
- **Ast explanation:**
  - ➢ information preserved in ast are like variable type which contains int,float and binop which consist of arithmetic operations like add,sub,mult.
  - ➢ order and definition of executables statements present in ast with name type stmt,type decl.

Semantic Explanation:

- Program execution starts from check-program as we wrote that in main.ml saying starts from semantic.check-program.
- node list passes through the corresponding function and checks whether it is valid or not.if its valid it goes to next function or raises an error.
- It checks whether the data types got matched or not and perform the corresponding operation after that.

IMPLEMENTATION

- Using Ocamlc command, “semantic.cmo” file is generated from “semantic.ml” and “ast.mi” files.
- All of the above which are combination of lexer,parser, ast, semantic generate object file named “semantic”, which on executing gives out the names of tokens to stdout, taking input from stdin or “input.am”.

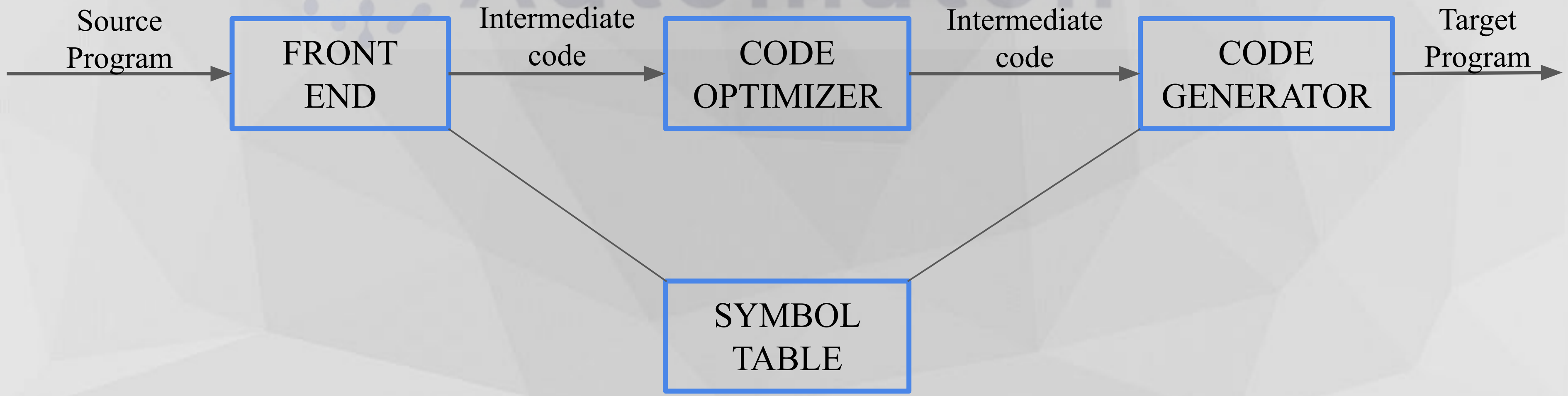
Fourth Phase: CODE GENERATION

CODE GENERATION

- The work of Code generator is to take the semantically checked program and convert it into Python code.
- It converts
  - ➢ DFA into a class
  - ➢ States into class methods.
- 3 Primary tasks of Code Generator are
  - ➢ Instruction Selection: choosing target-machine instructions to implement the IR statements.
  - ➢ Register Allocation and Assignment: deciding the values that the registers need to keep.
  - ➢ Instruction Ordering: deciding the order in which the instructions need to be executed.
- It also generates a fair amount of pre-established pure python code that is used to do built-in language functions (wrapped as DFAs), as well as set up the architecture for the main function to run itself.

Working of Code Generator:

- It takes input from the IR(intermediate representation) with supplementary information in symbol table of the source program and produces an equivalent target program as Output.
- Front End is nothing but the lexer, parser and semantic analysis.
- Intermediate code will be in the form of linear representation like a postfix expression which is free from type checking errors since it is passed through the semantic file.
- IR is passed into itself multiple times to produce more efficient code.



IMPLEMENTATION

- Using ocamlc -c comand with gen_python.ml file and ast.cmi, gen_python.cmi file will be generated.
- After running "make " to produce the compiler executable, we compile our .am file with the following Command:

```
$ ./gen_python "name of output file" < "path to your .am file"
```

- This compiles our Automaton program and produce python code named "name of output file".py.
- We then run this file using the command:

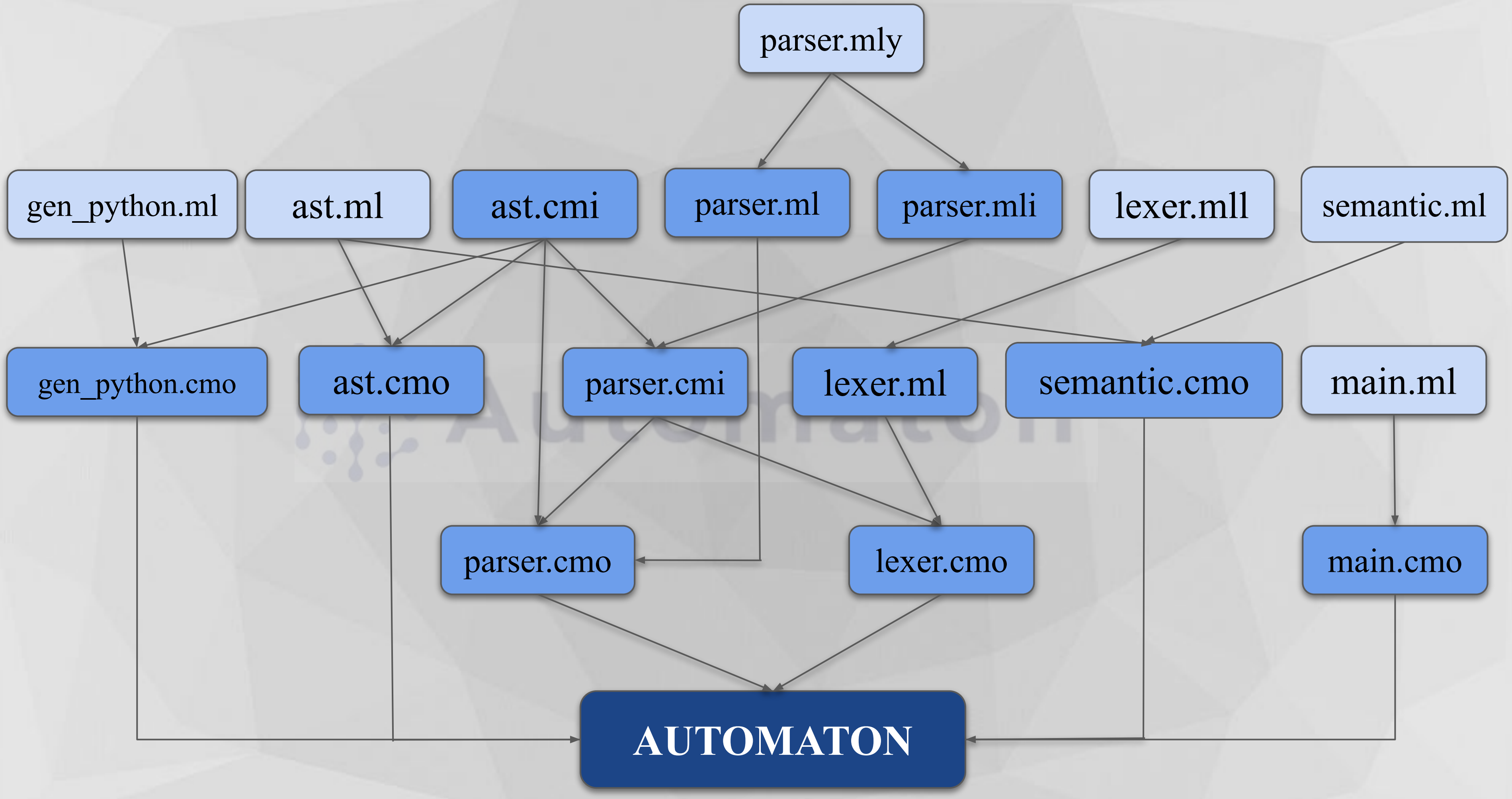
```
$ python "name of output file".py
```

- Stacks can be passed in a command line by separation via commas.
- No spaces should exist between the elements of a stack.
- To pass in a string as a stack of strings, with each string consisting as a single character of the string, surround the string to be passed with “’ (double quotes then single quotes).

```
python outputName.py a,b,c  || python outputName.py[a,b,c] || python outputName.py “’bitbybit’”
```

HOW TO COMPILE AND RUN

- The folder “Automaton” consists of the following files:
    - lexer.mll
    - parser.mll
    - ast.ml
    - semantic.ml
    - gen_python.ml
    - main.ml
    - input.am
    - makefile
- “make” command is used to run these files and generate the required output.(Refer to the next slide for better understanding)


parser.mly
gen_python.ml
ast.ml
ast.cmi
parser.ml
parser.mli
lexer.mll
semantic.ml
gen_python.cmo
ast.cmo
parser.cmi
lexer.ml
semantic.cmo
main.ml
parser.cmo
lexer.cmo
main.cmo
AUTOMATON

THANK YOU!!!