Compiler Design(CS 335), 2023-24
Indian Institute of Technology Kanpur
Milestone 1

Student Names: Abhinav Garg, Kriti, Ashutosh Kumar
Roll Numbers: 210029, 210534, 210221
Date: March 3, 2024

Milestone 1

Introduction

For milestone 1, we have constructed a scanner and a parser for a statically typed subset of the Python language. The output of your compiler is a graphical representation of the abstract syntax tree (AST) of the input program. The leaves of the AST are labeled by the token names and the lexemes within parentheses, e.g., ID (myVarName). We have used **Python 3.12 grammar** to implement the parser using the **LALR Bison** parser generator. This report file describes the tools that we used and includes compilation and execution instructions and all command line options.

Compiling and Executing Instructions

The optional commands provided by our program are given below:

- **-i :** Input file specification (default is command line) which the compiler uses for reading the Python program. The input file must immediately follow the -i tag.
- **-o :** Generates the AST corresponding to the input code in out.pdf
- **-v :** Uses YYDEBUG to show the states of the stack maintained for LALR Parsing
- **-h :** Instruction regarding usage instructions and options

By default, the program does not have an input file. The -o tag by default creates out.pdf. A sample execution example is provided below:

```
$ cd src

$ make

$ ./slytherin.o -i ../tests/test1.py -o

$ make clean
```

File Structure

- All the source files are in the milestone1/src directory.
- flex.l contains the lexer implementation in flex
- parser.y contains the paser implementation in Bison
- node.cpp contains the parse tree implementation
- The five test cases are in a directory milestone1/tests. The testcases are named as “test¡serialnumber¿.py”.
- The submission includes a PDF file under the milestone1/doc directory.

Building AST from Parse Tree

• There are productions (rules) that allow for empty or epsilon derivations in the grammar. When building a parse tree, epsilon productions result in nodes with no children (such as other terminals). Since these nodes do not contribute to the meaning of the program, they are removed during the conversion process to create a more compact and meaningful AST.

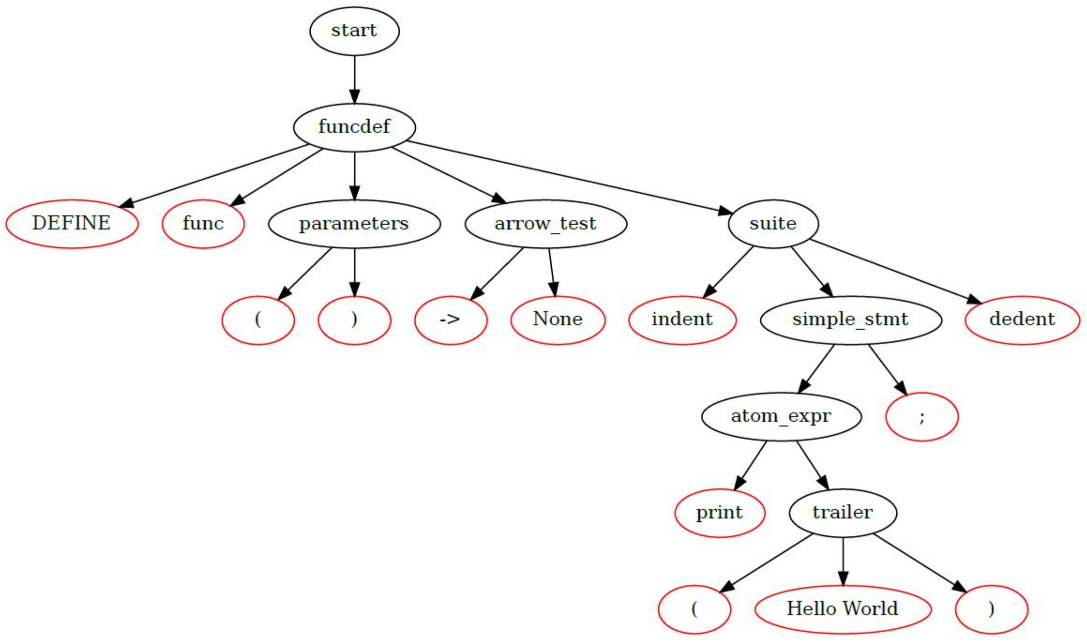
• After removing empty productions, some nodes in the AST have only a single child. While these nodes are not inherently problematic, they can add unnecessary depth to the parse tree, making it more complex than necessary. To simplify the parse tree, we collapsed these nodes by promoting their single child to replace the parent node, effectively reducing the tree's depth.

Sample

Consider the following Python program:

```
def func()->None:
    print("Hello World");
```

The generated AST for the above program is shown below:



All the terminals have been colored red.