CS335 - Milestone 1

Pranjal Singh, Dev Gupta, Deven Anil Gangwani

March 2024

1 Introduction

This document describes our implementation of milestone 1 of the course project of CS335 (The Manhatthon Project). Our parser recognises a statically typed subset of Python 3.12 and generates `dot` files corresponding to the abstract syntax tree (AST) of the input program.

2 Lexical Analysis

As per the problem specification, we use GNU `flex` for the lexical analysis of the input program. The utility generates a `yylex` function from the lexical specification in `lexer.l` and a C file containing it.

This is integrated with the parser in the next step using inbuilt features.

3 Syntax Analysis

We use the GNU `bison` parser generator to generate a C++ parser from a grammar specification `parser.y`. This can be compiled and linked with the lexer generated by `flex` into an executable binary.

4 Semantics of the Parser

We define the semantic values of grammar tokens (terminals and nonterminals) to be a custom class, `Node` defined in `classes.h` (Refer to section 3.4 of the `bison` manual - Defining Language Semantics). It contains an identification number and string for each token, and defines a constructor and helper function to create parent-child relationships.

We chose to maintain a vector containing the list of children of each node for each instance of `Node`, but have not used it in this phase of the compiler.

5 Generating ASTs

5.1 Generating AST Specification

In the aforementioned semantic description of the program, we used the constructor to generate nodes in the AST specification and the `Node.addchild()` helper function to generate edges. A global file object is used by the semantic actions, and the specification of a node/edge is printed directly to it.

5.2 Generating ASTs

As per the problem specification, we use the Graphviz package's `dot` command to generate PDFs. The syntax of the language is remarkably simple and intuitive (we used only `man` pages to learn it). Some flags and output redirection is needed to generate a PDF.

6 Compilation Commands

6.1 List of Commands

- Generating the parser (`parser.tab.c`) and header file for the lexer (`parser.tab.h`):
  `$ bison -d parser.y`
- Generating the lexer (`lex.yy.c`):
  `$ flex lexer.l`
- Compilation: (the `-lfl` flag may be needed)
  `$ g++ -o parser parser.tab.c lex.yy.c`
- Running the parser with input redirection (generates `ast.dot`):
  `$ ./parser < input.py`
- Generating the AST:
  `$ dot -Tpdf -Gordering=out ast.dot > ast.pdf`

6.2 Makefile Usage

To simplify the compilation procedure during programming and debugging, we used GNU `make`. We have defined the following targets:

- `$ make`: Same as `make parser`.
- `$ make parser` Compiles the lexical and grammar specifications into an executable (`parser`)
- `$ make test` Compiles `input.py`, generates `temp.pdf` and `ast.dot` and then cleans up.
- `$ make temp` Uses the existing binary to generate a DOT file and PDF. (Considerably faster, but compile the parser before use.)
- `$ make clean` Deletes intermediate files and the PDF.

7 Command-line Arguments

- `-input input_program`: Use input_program as the input. By default, the lexer reads from the console (stdin). Input redirection can also be used.
- `-output output_file`: Create/overwrite output_file with the DOT specification of the AST.
- `-help`: List command-line options and exit.
- `-verbose shift`: Filter shift operations from the inbuilt `bison` traces and prints to `stderr`. Useful to figure where an error is encountered.
- `-verbose reduce`: Filter and print reduce operations.
- `-verbose sr`: Filter and print shift and reduce operations.
- `-verbose all`: Copy the entire trace. (Runs into thousands of lines.)
- `-verbose srla`: Filter and print shift and reduce operations and lookahead token in each step. Exclude stack state.