Compiler Construction: Project Final Report

WizuAll : A custom data visualization language

Siddharth Chaitra Vivek - 2022A7PS0569G

27th April 2025

Overview

This reports summarizes the design and implementation of a domain-specific language (DSL) called *WizuAll* , meant for data visualization. The language can be used to extract data from tabular sources (directly from CSV Files or from a PDF File containing a table using by preprocessing it) and help visualize the data. The Pipeline consists of Preprocessing of the data, lexical analysis, parsing of the tokenized code and finally the generation of the target code and its execution. the target code here is Python and it uses Matplotlib for visualization and Pandas for data handling.

1 The Components in the ToolChain

The code is written from scratch and not modeled off CalciList but maintains all the required functionality while implementing the main objective of data visualization.

The code can be run by executing the following steps:

1. Downloading and extracting the zip file contents
2. Running the command *bash setup.sh* to download dependancies
3. Running the command *make show* to run everything and save the output plots to the same directory

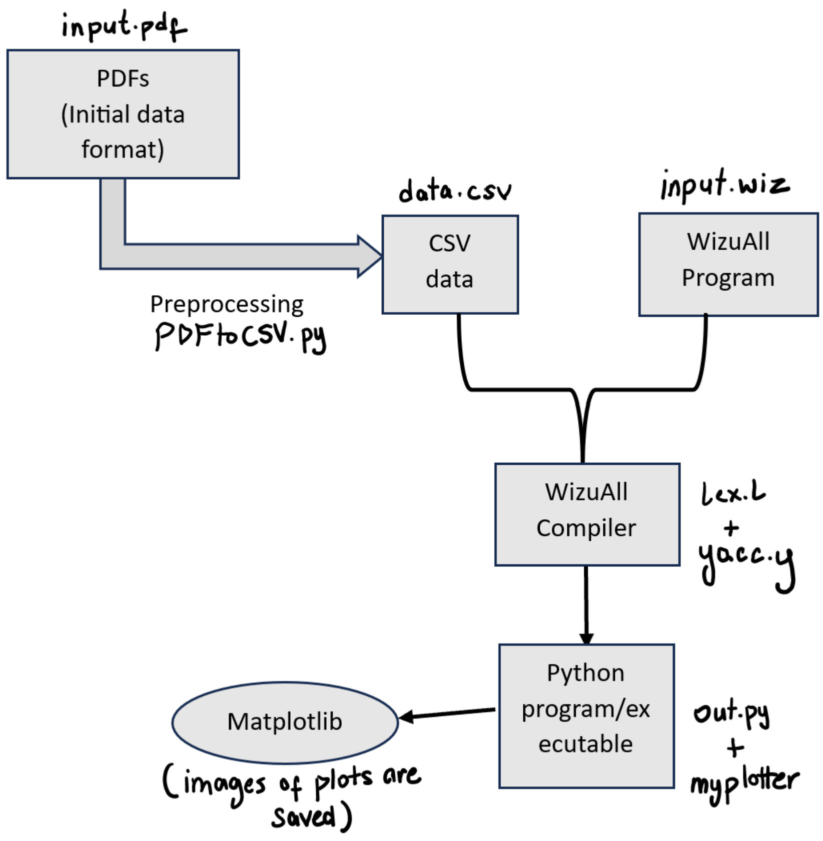
Preprocessor:
Converts tabular data from PDFs into CSV files which are then accessed by the target code during runtime to generate the visualizations. The preprocessing is done using a python library called tabula. The code for the same is written in *PDFtoCSV.py.*

***WizuAll* Compiler:**
A compiler written using flex and bison that reads the programmers code written in *input.wiz* and generates the target language code. The compiler *WizuAll* compiler consists of two files: *lex.l* and *yacc.y* and generated the target code in *out.py*

Visualizer:
Uses Python libraries Matplotlib and Pandas for data visualization and data handling respectively. The vsiualizations get saved as png files to the same directory after running the target code generated.

2 Scheme and Pipeline

The following flowchart shows the scheme and pipeline of the entire process



3 Program Structure

- `Wizuall-Language/`
  - `input.pdf`
  - `input.wiz`
  - `lex.l`
  - `makefile`
  - `PDFtoCSV.py`
  - `setup.sh`
  - `yacc.tab.c`
  - `yacc.tab.h`
  - `yacc.y`

4 *WizuAll* Syntax

4.1 Operators:

- Assignment: `=`
- Arithmetic: `+`, `-`, `*`, `/`
- Comparison: `>`, `<`, `>=`, `<=`, `==`, `!=`

4.2 General Notes:

- **Semicolons** (`;`) must terminate every statement.
- **Curly braces** (`{ }`) are used for blocks (functions, conditionals, loops).
- **Parentheses** (`( )`) are used for conditions and function arguments.
- **Square brackets** (`[ ]`) are used for dataframe column access.

4.3 Accessing Data:

```
OPEN <dataframe_variable> "<filename>";
```

Loads a CSV file into a named variable.

```
<new_variable> = <dataframe_variable>["<ColumnName>"];
```

Accesses a column from the dataframe and assigns it.

4.4 Functions

```
function <function_name>(<param1>, <param2>, ...) {
    <statements>;
    return <value>;
}
```

4.5 Function Calls:

```
<variable> = <function_name>(<arguments>);
```

Or if no assignment:

```
<function_name>(<arguments>);
```

4.6 Conditionals

```
if (<condition>) {
    <statements>;
}
```

4.7 Loops:

For Loop

```
for <variable> from <start_value> to <end_value> {
    <statements>;
}
```

While Loop:

```
while (<condition>) {
    <statements>;
}
```

4.8 Plotting Commands

```
LINE <Column1> <Column2>;
SCATTER <Column1> <Column2>;
BAR <Column>;
```

Special keywords for different chart types. Column names (variables) are provided as arguments.

4.9 Some other functions

```
print(<expression>);
```

Prints the expression in terminal

```
sum(<list>);
```

Returns the sum of all the elements in a list

```
avg(<list>);
```

Returns the average value of a list

5 Semantics and Implementation

Here the *WizuAll* compiler utilizes Lex and Yacc for compiling a custom domain-specific language into Python code. The Lex file (lex.l) is responsible for tokenizing the input, identifying keywords (e.g., OPEN, LINE, PRINT), operators (==, !=, =, etc.), identifiers, string literals, and numbers. Tokens are passed to the Yacc parser for syntactic analysis.

The Yacc file (*yacc.y*) defines a grammar that maps high-level commands into valid Python constructs. For instance, commands like OPEN data ”file.csv”; are translated into data = pd.readcsv(”file.csv”), and plotting commands like LINE x y; generate Matplotlib code. The parser supports basic control structures (if, for, while), function definitions, variable assignments, and common data processing operations. The result is a Python script that integrates pandas and matplotlib functionality, providing a streamlined way to script data operations using a simplified syntax.

Tokens recognized by lex.l:

```
%%

"OPEN"             { return OPEN; }
"LINE"             { return LINE; }
"BAR"             { return BAR; }
"SCATTER"          { return SCATTER; }
"if"               { return IF; }
"for"              { return FOR; }
"while"            { return WHILE; }
"print"            { return PRINT;}
"sort"             { return SORT;}
"function"         { return FUN;}
"from"             { return FROM;}
"avg"              { return AVG;}
"to"               { return TO;}
"return"           {return RETURN;}
"=="               { return EQ; }
"!="               { return NEQ; }
">="               { return GTE; }
"<="               { return LTE; }
">"                { return GT; }
"<"                { return LT; }
"("                { return LPAR; }
")"                { return RPAR; }
"{"                { return LCURL; }
"}"                { return RCURL; }
"["                { return LBRACK; }
"]"                { return RBRACK; }
"="                { return ASSIGN; }
"+"                { return PLUS; }
"-"                { return MINUS; }
"*"                { return MUL; }
"/"                { return DIV; }
";"                { return SC; }
","                { return COMMA; }


\"[^"]*\"          { yylval.str = strdup(yytext); return STR; }
[0-9]+             { yylval.num = atoi(yytext); return NUM; }
[a-zA-Z_][a-zA-Z0-9_]*  { yylval.str = strdup(yytext); return ID; }

[ \t\n]+           {}
"%%".*         {}

.                  { printf("Unknown character: %s\n", yytext); }

%%
```

As explained earlier this `yacc.y` file defines a parser that translates a custom simple language into Python code using Pandas and Matplotlib. It consists of the following parts:

The `yacc.y` code is given at the end in the code section

Tokens and Union Definition

- `%union` defines possible data types tokens can carry: integers (`num`) and strings (`str`).
- Tokens declared:
  - `NUM` (`<num>`), `ID` and `STR` (`<str>`).
  - Keywords and operators such as `OPEN`, `LINE`, `BAR`, `IF`, `FOR`, `WHILE`, etc.
- `%start program` sets the starting non-terminal.

Grammar Rules Section

- **program**:
  - A sequence of statements or a function definition.
- **statement**:
  - Handles different commands:
    - `OPEN` - Read CSV into a DataFrame.
    - `LINE`, `BAR`, `SCATTER` - Generate plots.
    - `PRINT` - Print expressions.
    - `FOR`, `WHILE` - Loop structures with indentation management.
    - `IF` - Conditional execution.
    - `Assignment` and `Function Calls`.
- **func_defn**:
  - Defines Python functions.
- **program_with_return**:
  - Allows a function body ending with a return statement.
- **return_stmt**:
  - Handles `return expr;` syntax.
- **assignment**, **parameters**, **p_items**, **expr_list**:
  - Handle variable assignment and function parameters.
- **list**, **list_items**:
  - Handle creation of Python lists.
- **condition**:

  - Comparison operators: `==`, `!=`, `>`, `<`, `>=`, `<=`.

- **expr**:
  - Arithmetic operations: `+`, `-`, `*`, `/`.
  - Variable references, string literals, numbers.
  - List sorting: `SORT`.
  - Array access: `ID[STR]`.
  - Function calls: `ID(expr_list)`.
  - Average calculation: `AVG(ID)`.

6 Example of Target Code Generation

6.1 Input.wiz

Following is an example code written in *WizuAll* by the programmer. This code can be changed in the repository and corresponding output can be checked by running `make clean` to reset and then `make show` to generate output

```
OPEN df "data.csv" ;
salary = df["Salary"];
exp = df["Exp"];
age = df["Age"];

avg_salary = avg(salary);
avg_age = avg(age);


if (avg_salary > 0){
  print(avg_salary);
}

sorted_ages = sort age;
print(sorted_ages);

LINE Name Salary ;
SCATTER Salary Exp ;
BAR Height ;

for i from 3 to 7 {
  print(i);
}

i = 5;
while (i > 2){
    i = i - 1;
    print(i);
}

function test(a,b){
  print(a);
  print(b+2);
  return a;
}
new = test(sorted_ages,30);
print(new);
```

The target code generated from this *WizuAll* code is:

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv("data.csv")
salary = df["Salary"]
exp = df["Exp"]
age = df["Age"]
avg_salary = sum(salary) / len(salary)
avg_age = sum(age) / len(age)
if avg_salary > 0:
    print(avg_salary)
sorted_ages = sorted(age)
print(sorted_ages)
plt.figure(figsize=(8, 4))
plt.plot(df['Name'], df['Salary'])
plt.xlabel('Name')
plt.ylabel('Salary')
plt.savefig('line.png')
plt.figure(figsize=(8, 4))
plt.scatter(df['Salary'], df['Exp'])
plt.xlabel('Salary')
plt.ylabel('Exp')
plt.savefig('scatter.png')
plt.figure(figsize=(8, 4))
plt.hist(df['Height'])
plt.xlabel('Height')
plt.ylabel('Frequency')
plt.savefig('bar.png')
for i in range(3, 7+1):
    print(i)
i = 5
while i > 2:
    i = i - 1
    print(i)
def test(a, b):
    print(a)
    print(b + 2)
    return a
new = test(sorted_ages, 30)
print(new)
```

The output of the following code can be seen by running make show in terminal after unzipping the code uploaded. The output visualizations will be saved as files into the same directory

7 Appendix: *Yacc.y* Program Code

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

FILE *output;
void yyerror(const char *s);
int yylex(void);
int indent = 0;
void print_indent() { for (int i = 0; i < indent; i++) fprintf(output, "
    ↪     "); }

%}

%union {
    int num;
    char* str;
}

%token <num> NUM
%token <str> ID STR
%type <str> parameters p_items statement func_defn condition list
    ↪ list_items expr expr_list assignment return_stmt
    ↪ program_with_return

%token OPEN LINE BAR SCATTER IF FOR WHILE PRINT SORT FUN FROM TO AVG
    ↪ ASSIGN PLUS MINUS MUL DIV SC COMMA LPAR RPAR LCURL RCURL LBRACK
    ↪ RBRACK EQ NEQ GTE LTE GT LT RETURN

%start program

%%
program:
    program statement
  | statement
  | func_defn
  ;

statement:
  |  OPEN ID STR SC {
        print_indent();
        fprintf(output, "%s = pd.read_csv(%s)\n", $2, $3);
    }
  | LINE ID ID SC {
        print_indent();
        fprintf(output, "plt.figure(figsize=(8, 4))\n");
        print_indent();
        fprintf(output, "plt.plot(df['%s'], df['%s'])\n", $2, $3);
        print_indent();
        fprintf(output, "plt.xlabel('%s')\n", $2);
        print_indent();
        fprintf(output, "plt.ylabel('%s')\n", $3);
        print_indent();
        fprintf(output, "plt.savefig('line.png')\n");

    }
  | BAR ID SC {
        print_indent();
        fprintf(output, "plt.figure(figsize=(8, 4))\n");
        print_indent();
        fprintf(output, "plt.hist(df['%s'])\n", $2);
        print_indent();
        fprintf(output, "plt.xlabel('%s')\n", $2);
        print_indent();
        fprintf(output, "plt.ylabel('Frequency')\n");
        print_indent();
        fprintf(output, "plt.savefig('bar.png')\n");
        print_indent();

    }
  | SCATTER ID ID SC {
        print_indent();
        fprintf(output, "plt.figure(figsize=(8, 4))\n");
        print_indent();
        fprintf(output, "plt.scatter(df['%s'], df['%s'])\n", $2, $3);
        print_indent();
        fprintf(output, "plt.xlabel('%s')\n", $2);
        print_indent();
        fprintf(output, "plt.ylabel('%s')\n", $3);
        print_indent();
        fprintf(output, "plt.savefig('scatter.png')\n");

    }
  | PRINT LPAR expr RPAR SC {
        print_indent();
        fprintf(output, "print(%s)\n", $3);
    }
  |  FOR ID FROM expr TO expr LCURL {
        print_indent();
        fprintf(output, "for %s in range(%s, %s+1):\n", $2, $4, $6);
        indent++;
    } program RCURL {
        indent--;
    }
  | WHILE LPAR condition RPAR LCURL {
        print_indent();
        fprintf(output, "while %s:\n", $3);
        indent++;
    } program RCURL {
        indent--;
    }
  | ID ASSIGN expr SC {
        print_indent();
        fprintf(output, "%s = %s\n", $1, $3);
    }
  | ID ASSIGN list SC {
        print_indent();
        fprintf(output, "%s = %s\n", $1, $3);
    }
  | ID LPAR expr_list RPAR SC{
        print_indent();
        fprintf(output, "%s(%s)\n", $1, $3);
    }
  | ID ASSIGN ID LPAR expr_list RPAR SC {
        print_indent();
        fprintf(output, "%s = %s(%s)\n", $1, $3, $5);
    }
  | IF LPAR condition RPAR LCURL {
        print_indent();
        fprintf(output, "if %s:\n", $3);
        indent++;
    } program RCURL {
        indent--;
    }
  | func_defn
  ;

func_defn:
    FUN ID LPAR parameters RPAR LCURL {
        print_indent();
        fprintf(output, "def %s(%s):\n", $2, $4);
        indent++;
    }
    program_with_return
    RCURL {
        indent--;
    }
;

program_with_return:
    program return_stmt
    {
        fprintf(output, "    return %s\n", $2);
    }
;

return_stmt:
    RETURN expr SC {
        $$ = $2; // This can be used to reference the return expression
    }
;
```

```
assignment:
    ID ASSIGN expr {
        char *buf = malloc(strlen($1) + strlen($3) + 4);
        sprintf(buf, "%s = %s", $1, $3);
        $$ = buf;
    }
  ;

parameters:
    p_items {
        $$ = $1;
    }
  | {
        $$ = strdup("");
    }
  ;

p_items:
    ID {
        $$ = strdup($1);
    }
  | p_items COMMA ID {
        char *buf = malloc(strlen($1) + strlen($3) + 3);
        sprintf(buf, "%s, %s", $1, $3);
        $$ = buf;
    }
  ;

expr_list:
    expr {
        $$ = strdup($1);
    }
  | expr_list COMMA expr {
        char *buf = malloc(strlen($1) + strlen($3) + 3);
        sprintf(buf, "%s, %s", $1, $3);
        $$ = buf;
    }
  | {
        $$ = strdup("");
    }
  ;

list:
    LBRACK list_items RBRACK {
        char *buf = malloc(strlen($2) + 3);
        sprintf(buf, "[%s]", $2);
        $$ = buf;
    }
  ;
```

```
list_items:
    expr {
        $$ = strdup($1);
    }
  | list_items COMMA expr {
        char *buf = malloc(strlen($1) + strlen($3) + 3);
        sprintf(buf, "%s, %s", $1, $3);
        $$ = buf;
    }
  ;

condition:
    expr EQ expr     { asprintf(&$$, "%s == %s", $1, $3); }
  | expr NEQ expr    { asprintf(&$$, "%s != %s", $1, $3); }
  | expr GT expr     { asprintf(&$$, "%s > %s", $1, $3); }
  | expr LT expr     { asprintf(&$$, "%s < %s", $1, $3); }
  | expr GTE expr    { asprintf(&$$, "%s >= %s", $1, $3); }
  | expr LTE expr    { asprintf(&$$, "%s <= %s", $1, $3); }
  ;

expr:
    expr PLUS expr   { asprintf(&$$, "%s + %s", $1, $3); }
  | expr MINUS expr  { asprintf(&$$, "%s - %s", $1, $3); }
  | expr MUL expr    { asprintf(&$$, "%s * %s", $1, $3); }
  | expr DIV expr    { asprintf(&$$, "%s / %s", $1, $3); }
  | ID               { $$ = strdup($1); }
  | STR           { $$ = strdup($1); }
  | NUM           {
        char buf[16]; sprintf(buf, "%d", $1);
        $$ = strdup(buf);
    }
  | SORT ID {
        asprintf(&$$, "sorted(%s)", $2);
    }
  | ID LBRACK STR RBRACK {
        asprintf(&$$, "%s[%s]", $1, $3);
    }
  | ID LPAR expr_list RPAR {
        asprintf(&$$, "%s(%s)", $1, $3);
    }
  | AVG LPAR ID RPAR {
        asprintf(&$$ , "sum(%s) / len(%s)" , $3,$3);
  }
  ;

%%

int main() {
    output = fopen("out.py", "w");
    fprintf(output, "import pandas as pd\nimport matplotlib.pyplot as
    ↪ plt\n\n");
    yyparse();
    fclose(output);
    return 0;
}

void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}
```