UNIVERSITÀ
DI SIENA
1240

Language Processing Techniques project report

Author:
Barbara Corradini

Supervisors:
Prof. Marco Maggini
Dr. Andrea Zugarini

Contents

Chapter 1

Lex/Yacc parser

1.1 Text

Using lex/Yacc, implement a parser for the definition of functions, with the following rules:

1. The functions are defined as

   ```
    function name(par1,par2,...) {
           return par1 op par2 op par3...;
   }
   ```

   where `name` is the function name with the usual restrictions (an alphanumeric string beginning with a letter), `par1,par2,...` are the function parameters whose names follow the same rules as variables names, `op` is sum or product.

2. The function body contains only the return instruction that involves all the parameters.

3. Assume that only one function can be defined

4. After the function definition, there are the calls whose syntax is

   ```
   name(cost1,cost2,...);
   ```

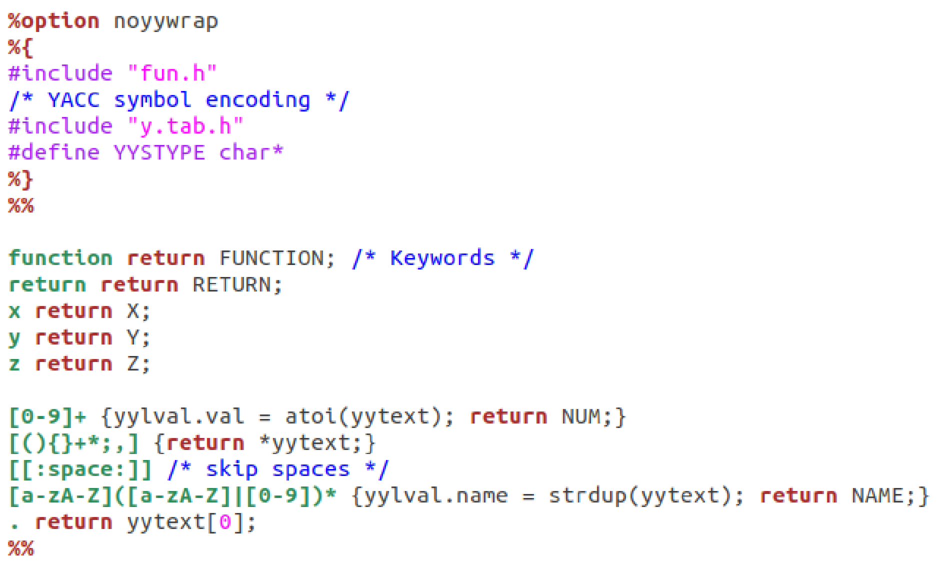
   where `name` is the name of a defined function, `cost1,cost2,...` are numeric constants in the same number as the function arguments.

5. Print the result of each function call

1.2 Implementation

1.2.1 Lex file

In file `fun.h`, keywords and regular expressions are defined. For each keyword, a non-terminal corresponding symbol is defined, so that when the parser reads a token it can make an association. Moreover, regual expressions allow us to respect the constraint on the function name.



```
%option noyywrap
%{
#include "fun.h"
/* YACC symbol encoding */
#include "y.tab.h"
#define YYSTYPE char*
%}
%%

function return FUNCTION; /* Keywords */
return return RETURN;
x return X;
y return Y;
z return Z;

[0-9]+ {yylval.val = atoi(yytext); return NUM;}
[(){}+*;,] {return *yytext;}
[[:space:]] /* skip spaces */
[a-zA-Z]([a-zA-Z]|[0-9])* {yylval.name = strdup(yytext); return NAME;}
. return yytext[0];
%%
```

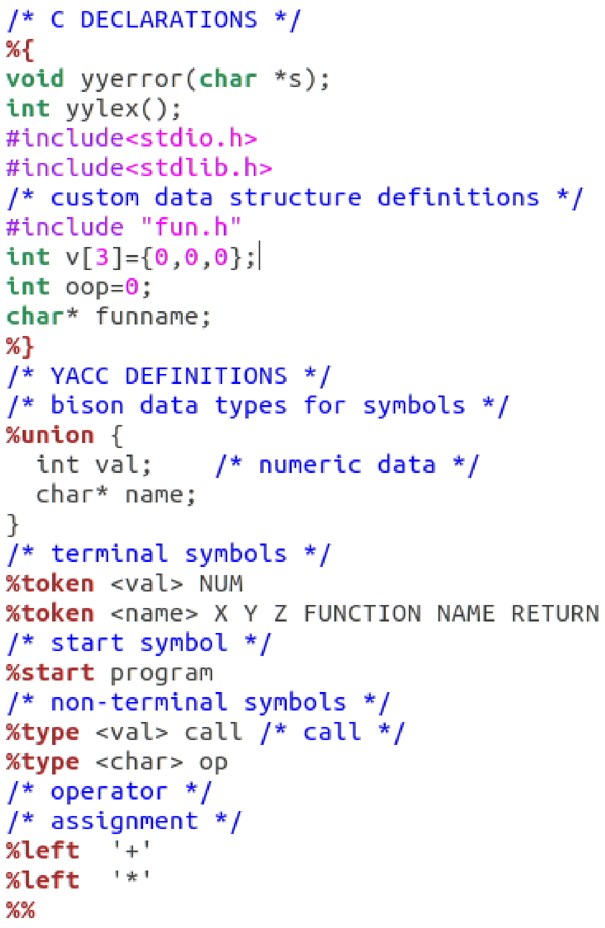
Figure 1.1: Lex file code

1.2.2 Yacc file

In the Yacc file, three C variables are declared:

- Vector `v`, containing three input parameters;
- `oop`, an integer associated to the operation to perform (`oop=2` if it is a product, `oop=1` if it is a sum, `oop=0` if it is undefined);
- `funname` is reserved to the function name.

Then, in the Yacc part terminal and non-terminal symbols are defined, together with start symbol and operators.
Among terminal symbols, tokens for `function`, `name` and `return` are introduced; while the non-terminal symbols are `call` and `op`.



```
/* C DECLARATIONS */
%{
void yyerror(char *s);
int yylex();
#include<stdio.h>
#include<stdlib.h>
/* custom data structure definitions */
#include "fun.h"
int v[3]={0,0,0};
int oop=0;
char* funname;
%}
/* YACC DEFINITIONS */
/* bison data types for symbols */
%union {
  int val;     /* numeric data */
  char* name;
}
/* terminal symbols */
%token <val> NUM
%token <name> X Y Z FUNCTION NAME RETURN
/* start symbol */
%start program
/* non-terminal symbols */
%type <val> call /* call */
%type <char> op
/* operator */
/* assignment */
%left  '+'
%left  '*'
%%
```

Figure 1.2: Yacc file code

The third part of the code is dedicated to grammar rules. The first to be considered is `program`, as the start symbol. What is worth to notice is the flow of the program: the `definition` stores the function name, and this is useful for comparing the function name in the `call` with the name in the function definition. If the comparison gives a bad result, the program gives an error; else, the input parameters are stored in vector `v` and, based on the value of `oop`, the program gives the result.

```
/* GRAMMAR RULES */
program: /*empty*/
        | program statement
        ;
statement: '\n'
        | definition
        | call
        | error statement {yyerrok;}
        ;
definition: FUNCTION NAME '(' X ',' Y ',' Z ')' body {funname=$2;};
op: '+' {oop=1;} | '*' {oop=2;};
body: '\n'|'{' RETURN X op Y op Z ';''}';
call: NAME '(' NUM ',' NUM ',' NUM ')'';'
        {
                if(strcmp($1, funname) != 0) {
                        printf("ERROR: call the right function!\n");
                }
                else if(strcmp($1, funname) == 0) {
                        v[0]=$3;
                        v[1]=$5;
                        v[2]=$7;
                        if(oop == 1) printf("RES OF SUM = %d", v[0]+v[1]+v[2]);
                        if(oop == 2) printf("RES OF PRODUCT = %d", v[0]*v[1]*v[2]);
                        if(oop == 0) printf("ERROR: sign not defined");
                }
        };
%%
/* error message handling */
void yyerror(char *s) {
  printf("%s\n",s);
}
/* file reading */
int yywrap()
{
        return 1;
}
int main(void) {
   return yyparse();
}
```

Figure 1.3: Yacc file code, pt. 2

Chapter 2

Author classifier based on ANN

2.1 Aim and dataset

The aim of the second project is to implement an author classifier based on Artificial Neural Networks. The dataset exploited counts about 300000 entries, each of which with the following attributes:

1. stanzas;
2. pos;
3. collection title;
4. author;
5. publication date;
6. head content;
7. poem scheme.

2.2 Pre-processing of data

The attributes of the dataset useful to reach the goal are just stanzas and author. Initially, the number of authors in the dataset is 613 but, due to the logic of the neural networks, authors with a small number of works can be neglected, since they are not significant for the classification.
After throwing these entries, **the number of authors for the classification task is 35 and the number of entries is 35421**.
The second step of the pre-processing phase is tokenize stanzas: Python tokenizer is used to associate an integer value to each word of the text (i.e.

stanzas text). Total number of tokens is 65144 and few of them follow, by way of example, with respective value:

- world, 27
- lie, 201
- kiss, 286
- rome, 412

2.3 Three models architecture

The following table summarizes the remarkable parameters for each of the three models used for classification task.

ANN model	# neurons	`EMBEDDING_DIM`	`MAX_NB_WORDS`	dropout
Bidirectional LSTM	64	250	40000	0.5
Standard LSTM	100	250	40000	0.2 + 0.2[1]
ANN model	`filters`	`kernel_size`	`hidden_dims`	dropout
CNN	250	3	250	0.2

2.4 Three models results

The following table summarizes results obtained by each of the three models in performing the classification task.

ANN model	`train_accuracy`	`train_loss`	`val_accuracy`	`val_loss`
Bidirectional LSTM	0.4569	1.2755	0.4013	2.2594
Standard LSTM	0.5393	1.6920	0.4207	2.1967
CNN	0.6864	1.0871	0.4569	2.2014

[1]There are two dropout contributes: 0.2 in LSTM layer and 0.2 in dropout layer