INTERPRETOR PYTHON

Proiect LFT

GRUPA 30238
Matiș Oana-Antonia
Popesc Ariadna-Ioana

UNIVERSITATEA TEHNICĂ CLUJ-NAPOCA

Cuprins

1. Descrierea proiectului

Acest proiect implică dezvoltarea unui analizator lexical și sintactic (lexer și parser) pentru limbajul de programare Python, utilizând Lex și Yacc. Lexer-ul are rolul de a scana codul sursă și de a identifica componentele fundamentale ale limbajului, în timp ce parser-ul organizează aceste componente conform gramaticii limbajului Python.

2. Structura proiectului

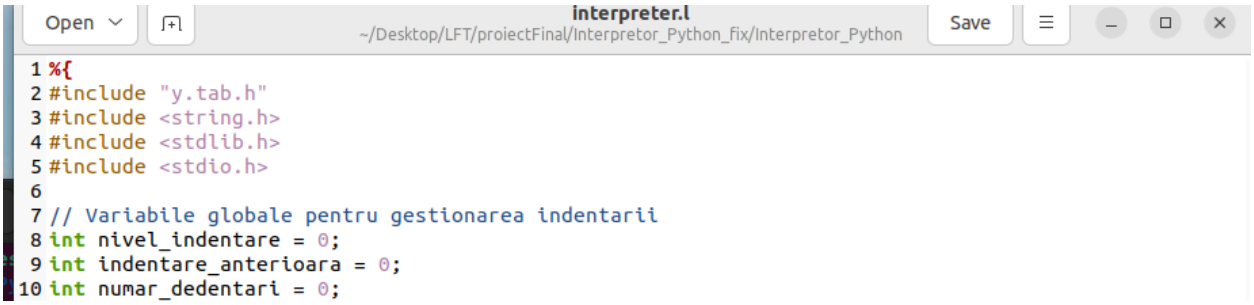
Proiectul constă din următoarele fișiere:

- interpreter.l: Fișierul sursă Lex care conține definițiile regulilor lexicale.
- interpreter.y: Fișierul sursă Yacc care conține definițiile regulilor gramaticale.
- types.h: Header file care conține definițiile structurilor și tipurilor de date utilizate.
- ---------- de adaugat !!!!!!!!!!!!!!!!!!!!!!

Fișierul lexer.l

Acest fișier conține regulile lexicale pentru identificarea diferitelor componente ale limbajului Python. Lexer-ul transformă secvențele de caractere din codul sursă în token-uri care vor fi utilizate de un parser (generat cu Bison) pentru a construi arborele de sintaxă abstractă.

Header-ul:



- **y.tab.h:** Include antetul generat de Bison care definește token-urile și alte structuri necesare.
- **string.h, stdlib.h, stdio.h:** Include anteturi standard pentru manipularea șirurilor de caractere și pentru funcții de alocare de memorie.
- **nivel_indentare, indentare_anterioara, numar_dedentari:** Variabile globale pentru gestionarea nivelurilor de indentare, esențiale pentru structura blocurilor de cod în limbajul asemănător Python.

Funcție de utilitate:

```
11
12 // Functie personalizata pentru duplicarea unui sir de caractere pana la n caractere
13 char *my_strndup(const char *s, size_t n) {
14     char *p = (char*)malloc(n + 1);
15     if (p) {
16         strncpy(p, s, n);
17         p[n] = '\0';
18     }
19     return p;
20 }
21 %}
22
```

- **my_strndup:** Funcție personalizată pentru a duplica un șir de caractere până la n caractere. Aceasta este folosită pentru a manipula șirurile de caractere găsite în codul sursă.

Reguli lexicale:

```
25 // Regula pentru a ignora caracterele spatiu si tab
26 [ \t]       { /* ignora spatiile albe de la inceput */ }
27
28 // Regula pentru gestionarea liniilor noi si incrementarea numarului de linii
29 \n           { yylineno++; return NEWLINE; }
30
31 // Regula pentru a gasi sirurile de caractere intre ghilimele duble
32 "\""[^\"\\]*"\"" {
33     yylval.string = my_strndup(yytext + 1, yyleng - 2); // elimina ghilimelele din jur
34     return STRING;
35 }
36
37 // Regula pentru gestionarea nivelurilor de indentare la inceputul unei linii
38 ^([ \t]+)    {
39     int indent = strlen(yytext);
40     if (indent > indentare_anterioara) {
41         indentare_anterioara = indent;
42         nivel_indentare++;
43         return INDENT;
44     } else if (indent < indentare_anterioara) {
45         numar_dedentari = (indentare_anterioara - indent) / 4;
46         indentare_anterioara = indent;
47         if (numar_dedentari > 0) {
48             numar_dedentari--;
49             unput('\n');
50             return DEDENT;
51         }
52     }
53 }
54
```

Lex Tab Width: 8 Ln 19, Col 14 INS

- **[\t]:** Ignoră spațiile și taburile de la începutul liniilor de cod.
- **\n:** Incrementează numărul de linii (yylineno) și returnează token-ul NEWLINE.
- **"\""[^\"\\]*"\"":** Găsește șirurile de caractere între ghilimele duble, elimină ghilimelele și returnează token-ul STRING.
- **^([\t]+):** Gestionează nivelul de indentare. Compara indentarea curentă cu cea anterioară și returnează token-urile INDENT sau DEDENT după caz.

Operatori:

```
55 // Reguli pentru diferiti operatori si cuvinte cheie
56 "=="          { return EQ; }
57 "!="          { return NE; }
58 "<="          { return LE; }
59 ">="          { return GE; }
60 "<"           { return LT; }
61 ">"           { return GT; }
62 "="           { return ASSIGN; }
63 "end"         { return END; }
64 "for"         { return FOR; }
65 "if"          { return IF; }
66 "else"        { return ELSE; }
67 "while"       { return WHILE; }
68 "print"       { return PRINT; }
69 "def"         { return FUNCTION; }
70 "return"      { return RETURN; }
71 "("           { return LPAREN; }
72 ")"           { return RPAREN; }
73 ":"           { return COLON; }
74 ","           { return COMMA; }
75 "in"          { return IN; }
76 "range"       { return RANGE; }
77
78 // Regula pentru a gasi identificatori
79 [a-zA-Z_][a-zA-Z0-9_]* { yylval.string = strdup(yytext); return NAME; }
80
81 // Regula pentru a gasi numere intregi
82 [0-9]+       { yylval.num = atoi(yytext); return NUMBER; }
83
84 // Regula implicita pentru a returna orice alt caracter unic
85 .            { return yytext[0]; }
86
87 %%
```

Această secțiune identifică operatorii și simbolurile specifice Python, returnând token-uri corespunzătoare pentru fiecare.

Reguli pentru identificatori și numere: Găsește și returnează token-uri pentru identificatori și numere întregi.

Regula implicită: Returnează orice alt caracter unic ca valoare ASCII.

```
88
89 // Functie apelata la sfarsitul intrarii pentru a indica terminarea
90 int yywrap() {
91     return 1;
92 }
93
```

- **`yywrap`:** Indică sfârșitul intrării. Returnează 1 pentru a semnaliza sfârșitul analizării.

Fișierul parser.y

Antet și variabile globale

Open | interpreter.y ~/Desktop/LFT/proiectFinal/Interpretor_Python_fix/Interpretor_Python | Save

```
%{

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "interpreter.h"
#include "y.tab.h"

// Declarațiile externe pentru funcțiile și variabilele utilizate
extern FILE *yyin;
extern int yylineno;
extern int yylex(void);

extern Expression *create_expression(int num, char *value);
extern Expression *create_expression_with_operator(Expression *left, char *op, Expression
*right);
extern void free_expression(Expression *expr);
extern void yyerror(const char *msg);

extern Statement *create_statement(int type, Expression *condition, Statement *body, Statement
*next);
extern void free_statement(Statement *stmt);
extern void execute_statement(Statement *stmt);

%}

%union {
    struct Expression *expression;
    struct Statement *statement;
    char *string;
    int num;
}
```

- **Anteturi:** Include fișierele antet standard și anteturile specifice proiectului (interpreter.h și y.tab.h).
- **Declarații externe**: Declarațiile pentru funcții și variabile utilizate în alte fișiere.

Definirea uniunii și token-urilor

```
%union {
    struct Expression *expression;
    struct Statement *statement;
    char *string;
    int num;
}

// Declaratii pentru token-uri și tipuri de date
%token <string> STRING
%token <num> NUMBER
%token <string> NAME
%token END
%token ASSIGN LT GT LE GE EQ NE IF ELSE WHILE PRINT FUNCTION RETURN LPAREN RPAREN COLON COMMA FOR
IN PLUS MINUS TIMES DIVIDE RANGE NEWLINE END INDENT DEDENT TAB
%type <expression> expression
%type <statement> statement statements while_statement for_statement

%%
```

- **Uniune:** Definirea uniunii yyunion pentru a specifica tipurile de date posibile pentru valorile token-urilor.
- **Token-uri:** Declarații pentru token-uri și tipurile de date asociate acestora.

Regulile gramaticale

```
43 // Definirea regulilor gramaticale
44 program : statements
45         | /* empty */
46         ;
47
48 statements : statement NEWLINE
49            | statements statement NEWLINE
50            ;
51
52 statement : assignment
53           | while_statement
54           | for_statement
55           | print_statement
56           | if_statement
57           | END {printf("END Found");}
58           ;
59
60 assignment : NAME ASSIGN expression
61            {
62                printf("Assignment: %s = %d\n", $1, $3->num);
63                set_variable($1, $3->num);
64                free($1);
65                free_expression($3);
66            }
67            ;
68
69 if_statement : IF expression COLON NEWLINE INDENT statements DEDENT END
70              {
71                  printf("Parsing if statement with condition\n");
72                  if (evaluate_expression($2)) {
73                      if ($6) {
74                          execute_statement($6);
75                      } else {
76                          printf("Warning: if body is NULL\n");
77                      }
78                  }
79              }
```

Definirea Programului:

- program este rădăcina arborelui sintactic și poate conține o secvență de instrucțiuni (statements) sau poate fi gol.

Definirea Instrucțiunilor (Statements):

- statements este o secvență de instrucțiuni, fiecare încheiată cu un nou rând. Poate conține o sau mai multe instrucțiuni.
- statement reprezintă o instrucțiune individuală și poate fi o atribuire (assignment), un ciclu while, un ciclu for, o instrucțiune de tip print, o structură if sau semnalează sfârșitul programului.

Definirea Atribuirilor:

- assignment definește o instrucțiune de atribuire a unei valori unei variabile. Atribuirea este urmată de evaluarea expresiei și actualizarea variabilei corespunzătoare.

Definirea Instrucțiunilor If:

- if_statement definește instrucțiunile if și if-else. Analizează condiția, execută blocul de instrucțiuni corespunzător dacă condiția este adevărată și poate include și un bloc de instrucțiuni pentru cazul fals (else).

Definirea Instrucțiunilor de Tip Print:

- print_statement definește instrucțiunile de tip print, care afișează valori pe ecran. Poate fi folosit pentru afișarea unor șiruri de caractere sau a rezultatelor evaluării unei expresii.

```
for_statement : FOR NAME IN RANGE LPAREN expression COMMA expression RPAREN COLON NEWLINE INDENT
    statements DEDENT
                {
                    int start = evaluate_expression($6);
                    int end = evaluate_expression($8);
                    char *loop_variable_name = $2;
                    set_variable(loop_variable_name, start);
                    for (int x = start; x < end; ++x) {
                        set_variable(loop_variable_name, x);
                        execute_statement($13);
                    }
                    free($2);
                    free_expression($6);
                    free_expression($8);
                }
                ;

expression : expression PLUS expression
           {
               printf("Expression: %d + %d\n", $1->num, $3->num);
               $$ = create_expression($1->num + $3->num, NULL);
               free_expression($1);
               free_expression($3);
           }
           | expression MINUS expression
           {
               printf("Expression: %d - %d\n", $1->num, $3->num);
               $$ = create_expression($1->num - $3->num, NULL);
               free_expression($1);
               free_expression($3);
           }
           | expression TIMES expression
           {
               printf("Expression: %d * %d\n", $1->num, $3->num);
               $$ = create_expression($1->num * $3->num, NULL);
               free_expression($1);
               free_expression($3);
           }
           | expression DIVIDE expression
           {
               printf("Expression: %d / %d\n", $1->num, $3->num);
               $$ = create_expression(0, NULL);
               int divisor = $3->num;
               if (divisor == 0) {
                   yyerror("Division by zero");
               } else {
                   $$->num = $1->num / divisor;
               }
               free_expression($1);
               free_expression($3);
```

Definirea Ciclurilor While și For:

- while_statement și for_statement definesc instrucțiunile pentru ciclurile while și for. Acestea evaluează condiția și execută blocul de instrucțiuni corespunzător în funcție de rezultat.

Definirea Expresiilor:

- expression reprezintă o expresie aritmetică sau logică. Poate fi formată din numere, variabile, operatori și paranteze. Aceste expresii sunt evaluate pentru a obține un rezultat numeric sau o valoare logică.

Definirea Operatorilor și Comparatori:

- Operatorii aritmetici (PLUS, MINUS, TIMES, DIVIDE) și operatorii de comparație (LT, GT, LE, GE, EQ, NE) sunt utilizați în definirea expresiilor pentru a realiza operații matematice și comparații între valori.

Aceste reguli gramaticale definesc structura și semantica limbajului interpretat, permițând parser-ului să analizeze și să interpreteze corect codul sursă.

```
187             }
188             | LPAREN expression RPAREN
189             {
190                 $$ = $2;
191             }
192             | NAME
193             {
194                 printf("Expression: variable %s\n", $1);
195                 $$ = create_expression(get_variable($1), NULL);
196                 free($1);
197             }
198             | NUMBER
199             {
200                 printf("Expression: number %d\n", $1);
201                 $$ = create_expression($1, NULL);
202             }
203             | expression LT expression
204             {
205                 printf("Expression: %d < %d\n", $1->num, $3->num);
206                 $$ = create_expression_with_operator($1, "<", $3);
207             }
208             | expression GT expression
209             {
210                 printf("Expression: %d > %d\n", $1->num, $3->num);
211                 $$ = create_expression_with_operator($1, ">", $3);
212             }
213             | expression LE expression
214             {
215                 printf("Expression: %d <= %d\n", $1->num, $3->num);
216                 $$ = create_expression_with_operator($1, "<=", $3);
217             }
218             | expression GE expression
219             {
220                 printf("Expression: %d >= %d\n", $1->num, $3->num);
221                 $$ = create_expression_with_operator($1, ">=", $3);
222             }
223             | expression EQ expression
224             {
225                 printf("Expression: %d == %d\n", $1->num, $3->num);
226                 $$ = create_expression_with_operator($1, "==", $3);
227             }
228             | expression NE expression
229             {
230                 printf("Expression: %d != %d\n", $1->num, $3->num);
231                 $$ = create_expression_with_operator($1, "!=", $3);
232             }
233             ;
234
235 %%
```

Funcție pentru gestionarea erorilor

```
237 // Functie pentru gestionarea erorilor
238 void yyerror(const char *msg) {
239     fprintf(stderr, "Syntax error at line %d: %s\n", yylineno, msg);
240 }
241
242 // Functie principala
243 int main(int argc, char *argv[]) {
244     if (argc != 2) {
245         fprintf(stderr, "Usage: %s input_file\n", argv[0]);
246         exit(EXIT_FAILURE);
247     }
248
249     yyin = fopen(argv[1], "r");
250     if (!yyin) {
251         fprintf(stderr, "Error: Couldn't open input file %s\n", argv[1]);
252         exit(EXIT_FAILURE);
253     }
254
255     printf("Input file opened successfully\n");
256
257     yyparse();
258
259     fclose(yyin);
260
261     free_variables();
262     return 0;
263 }
264
```

3. Rezultate

Pentru testarea codului :

```
1 x = 3
2 if x == 5:
3     print("Reached 5")
4 else:
5     print("Not 5")
```

Vom avea rezultatele:

```
popescariadna@So2023:~/Desktop/LFT/proiectFinal/Interpretor_Python_fix/Inte
or_Python$ ./parser input.txt
Input file opened successfully
Expression: number 3
Assignment: x = 3
Expression: variable x
Expression: number 5
Expression: 3 == 5
Reached 5
Syntax error at line 4: syntax error
popescariadna@So2023:~/Desktop/LFT/proiectFinal/Interpretor_Python_fix/Inte
or_Python$
```

Pentru testarea codului:

Open ~/Desktop/LFT/proiectFinal/Interpretor_Python_fix/Interpretor_Py

```
1 print("Hello, World!")
```

Va afișa:

```
popescariadna@So2023:~/Desktop/LFT/proiectFinal/Interpretor
or_Python$ ./parser input_print.txt
Input file opened successfully
Hello, World!
popescariadna@So2023:~/Desktop/LFT/proiectFinal/Interpretor
or_Python$
```

Pentru codul:

~/Desktop/LFT/proiectFinal/Interpretor_Python_fix/Interpretor_Python

```
1 for x in range(2, 6):
2     print(x)
```

Va afișa:

```
popescariadna@So2023:~/Desktop/LFT/proiectFinal/Interpretor_Python_fix/Interpr
or_Python$ ./parser input_for.txt
Input file opened successfully
Expression: number 2
Expression: number 6
Expression: variable x
-1
Syntax error at line 3: syntax error
popescariadna@So2023:~/Desktop/LFT/proiectFinal/Interpretor_Python_fix/Interpr
```

Și pentru codul de while:

Open | input_while.txt ~/Desktop/LFT/proiectFinal/Interpretor_Python_fix/Interpretor_Python | Save

```
1 i = 1
2 while i < 6:
3     print(i)
4     i = i + 1
```

Va afișa:

```
or_Python$ ./parser input_while.txt
Input file opened successfully
Expression: number 1
Assignment: i = 1
Expression: variable i
Expression: number 6
Expression: 1 < 6
Expression: variable i
1
Expression: variable i
Assignment: i = 1
Syntax error at line 4: syntax error
```

4. Compilare și Utilizare

1. Instalare flex și bison
2. Textarea lex lex interpreter.l
3. Testarea yacc: yacc interpreter.y
4. Testarea lex & yacc gcc lex.yy.c y.tab.c -o a.out -ll -ly
5. Compilarea ./parser <in.txt>(fără out pentru ca vor fi afișate în consolă)
6. Testarea pe fiecare input: ./parser input_while.txt/input_for.txt/input_print.txt/input_if.txt

5. Concluzii

Acest proiect demonstrează cum se poate construi un analizator lexical și sintactic pentru Python utilizând Lex și Yacc. Lexer-ul recunoaște cuvintele cheie, identificatorii, numerele și operatorii, și gestionează indentarea specifică Python-ului. Parser-ul organizează aceste componente conform gramaticii limbajului Python. Această structură poate fi extinsă pentru a crea un compilator sau interpret complet pentru Python.

6. Dezvoltări ulterioare

Adăugarea Funcționalității de Funcții Utilizator:

- Extinderea interpretorului pentru a permite definirea și apelarea de funcții de către utilizatori în codul sursă.
- Implementarea unei sintaxe simple pentru definirea și utilizarea funcțiilor, precum și gestionarea parametrilor și valorilor returnate.

Optimizarea Performanței Lexer-ului și Parser-ului:

- Identificarea și implementarea unor îmbunătățiri în algoritmul de analiză lexicală și sintactică pentru a crește viteza de analiză a codului sursă.
- Utilizarea unor tehnici de optimizare, cum ar fi memoizarea sau îmbunătățirea algoritmilor existenți, pentru a reduce timpul de procesare al lexer-ului și parser-ului.

Suport Pentru Fișiere Externe și Modularitate:

- Extinderea interpretorului pentru a permite importarea și utilizarea de fișiere externe în codul sursă pentru a împărți și organiza mai bine codul.
- Implementarea unui sistem modular care permite definirea și utilizarea de module externe pentru a extinde funcționalitățile interpretorului într-un mod modular și flexibil.

7. Bibliografie

https://stackoverflow.com/questions/33853707/parser-with-lex-and-yacc

https://www.geeksforgeeks.org/introduction-to-yacc/

https://www.dabeaz.com/ply/ply.html

https://www.w3schools.com/python/

https://www.w3schools.com/python/python_while_loops.asp

https://www.w3schools.com/python/python_for_loops.asp