

Analiza lexicală pentru o gramatică dată folosind Flex

Cuprins

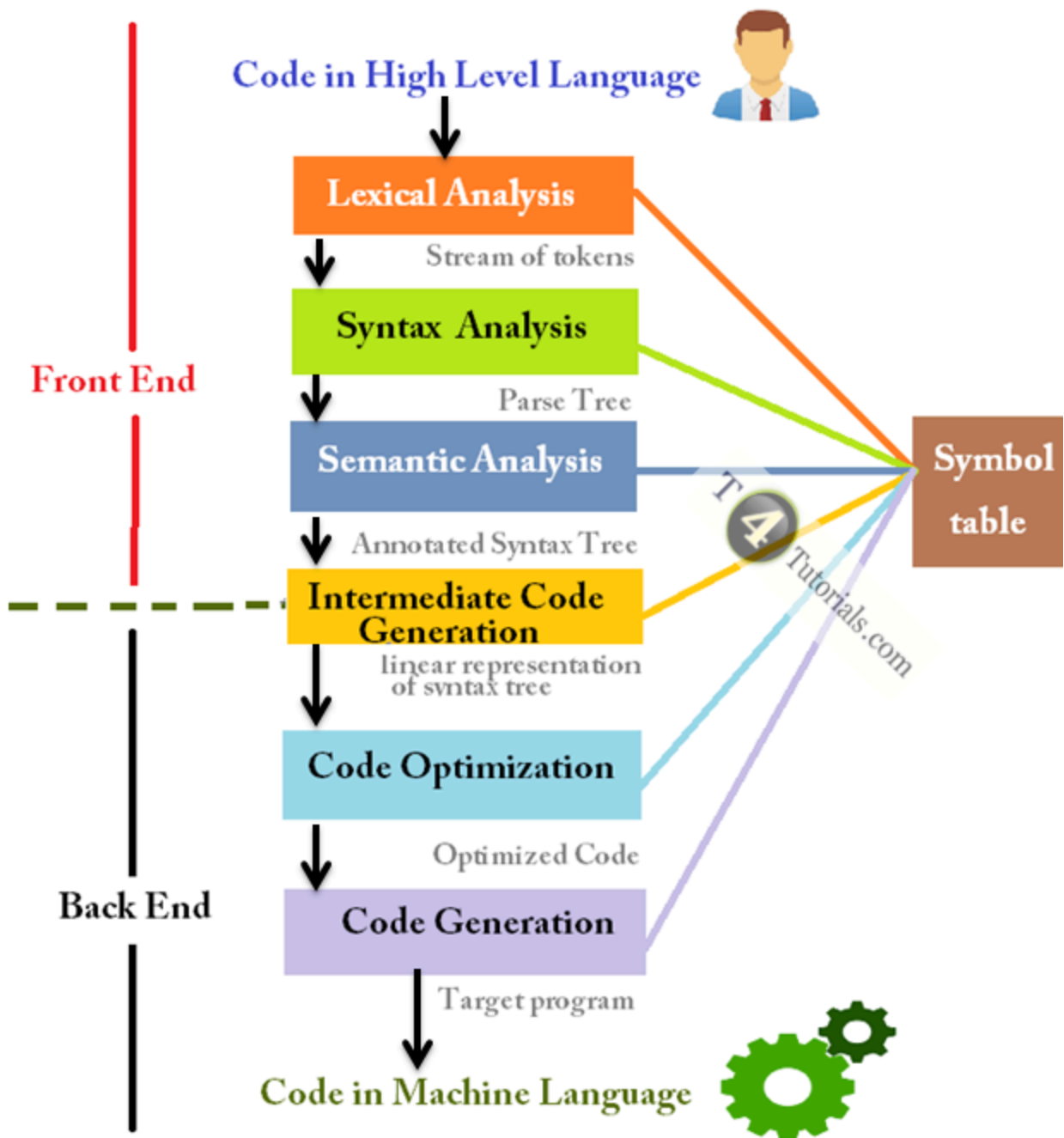
1. Aspecte teoretice
2. Obiective
3. Programul Flex dezvoltat
4. Rulare și automatizare
5. Concluzii și perspective

1. Aspecte teoretice

1.1 Definiție și importanță

Analiza lexicală reprezintă primul pas în procesul de prelucrare a unui program sursă, fiind efectuată de un component denumit **analizator lexical** sau **lexer**. Scopul acestuia este de a parcurge textul codului sursă, caracter cu caracter, și de a-l transforma într-o secvență de **tokenuri** – elemente atomice care descriu componente semantice precum **cuvinte cheie**, **identificatori**, **operatori** sau **constante numerice**.

Această etapă este esențială pentru procesele ulterioare de compilare, deoarece reduce complexitatea codului sursă, transformându-l într-un format mai ușor de procesat de către analizatorul sintactic și semantic.



Etapele design-ului unui compilator

1.2 Diferențe față de alte etape de analiză

Pentru a înțelege mai bine specificul analizei lexicale, este util să o comparăm cu alte etape din procesul de compilare:

- **Analiza lexicală:**
 - Se concentrează pe identificarea corectă a simbolurilor din cod.
 - Exemplu: recunoașterea cuvântului cheie `for` sau a identificatorului `sum`.
- **Analiza sintactică:**
 - Verifică dacă secvența de tokenuri respectă regulile gramaticale ale limbajului.
 - Exemplu: asigurarea faptului că bucla `for` este construită corect, cu toate parantezele și expresiile necesare.
- **Analiza semantică:**
 - Evaluează sensul logic al codului, asigurându-se că expresiile sunt valide din punct de vedere semantic.
 - Exemplu: verificarea compatibilității tipurilor de date într-o expresie precum `int x = "string";`.

1.3 Flex în procesul de analiză lexicală

Flex este o unealtă software utilizată pentru generarea automată a analizatorilor lexicali. Folosind un fișier de specificație cu extensia `.l`, utilizatorul poate defini reguli de potrivire bazate pe expresii regulate. Aceste reguli permit lexer-ului generat să identifice tokenurile din codul sursă conform următorilor pași:

1. **Definirea regulilor de potrivire:** fiecare expresie regulată este asociată unui token specific.
2. **Generarea codului C:** Flex produce un fișier C care implementează analizatorul lexical.

3. **Integrarea în procesul de compilare:** analizatorul lexical generat poate fi folosit ca primă etapă în prelucrarea codului.

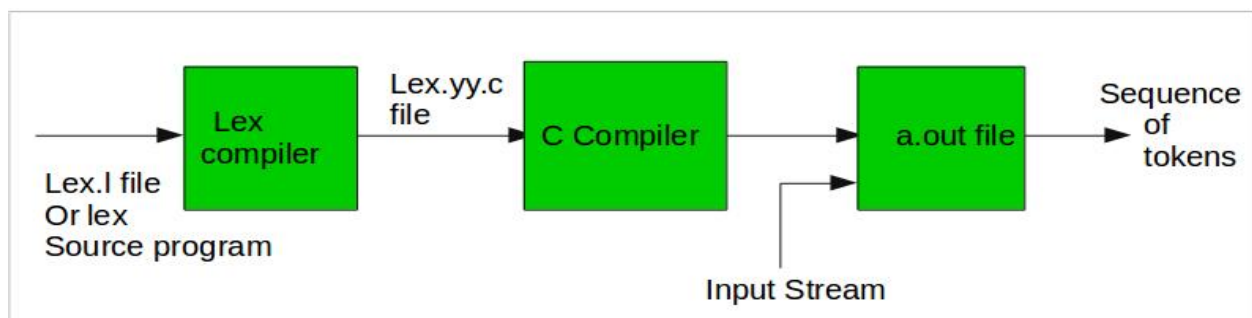
Flex oferă avantajul unei **automatizări eficiente**, eliminând necesitatea de a scrie manual cod pentru lexer, ceea ce reduce posibilitatea de erori și accelerează procesul de dezvoltare.

Exemplu simplu de specificație Flex

Un fișier Flex tipic include reguli pentru identificarea tokenurilor. De exemplu:

```
1  /* Header/Declarations/Prologue */
2  %{
3  #include <stdio.h>
4  %}
5
6  /* Definitions */
7  DIGIT  [0-9]
8
9  /* Token Rules/Patterns */
10 %%
11 if      { printf("Cuvânt cheie: IF\n"); }
12 [0-9]+  { printf("Constanta numerica: %s\n", yytext); }
13 [a-zA-Z_][a-zA-Z0-9_]* { printf("Identificator: %s\n", yytext); }
14 .       { printf("Caracter necunoscut: %s\n", yytext); }
15 %%
16
17 /* Auxiliary C code */
18 int main() {
19     yylex();
20     return 0;
21 }
```

Acest exemplu arată cum **Flex** poate recunoaște cuvintele cheie **if**, constantele numerice și identificatorii, raportând, de asemenea, simbolurile nevalide. Lexer-ul generat este integrat ulterior în compilator pentru a analiza codul sursă.



2. Obiective

Construirea unui analizator lexical pentru pseudo-C (gramatica aleasă)

- Crearea unui fișier Flex specific pentru un subset al limbajului C, denumit pseudo-C.
- Definirea regulilor lexicale utilizând expresii regulate adaptate gramatical pentru pseudo-C.
- Maparea expresiilor regulate către acțiuni specifice pentru identificarea corectă a tokenurilor, precum identificatori, constante numerice, cuvinte cheie, operatori și simboluri.

Generarea și integrarea analizatorului lexical

- Utilizarea Flex pentru a genera cod sursă în limbajul C (e.g., fișierul `lex.yy.c`).
- Compilarea analizatorului lexical generat folosind un compilator C (e.g., GCC).

Testarea și validarea analizatorului lexical

- Crearea unui set diversificat de fișiere de intrare care să respecte gramatica pseudo-C, incluzând atât exemple corecte, cât și scenarii de eroare.
- Rularea analizatorului lexical pentru a valida corectitudinea identificării tokenurilor și raportarea corespunzătoare a erorilor lexicale.
- Verificarea robusteții analizatorului prin scenarii de intrare complexe, incluzând comentarii, spații albe și simboluri nepermise.

3. Programul Flex dezvoltat

3.0 Instalare Flex (linie de comandă)

```
sudo apt-get update
sudo apt-get install flex
```

3.1 Blocul de declarații / Header / Prologue

```
1  %{
2  #include <stdio.h>
3  |   ...
4  #define COLOR_RESET      "\033[0m"
5  |   ...
6  #define COLOR_BRIGHT_BLUE "\033[94m" /* Boolean literals */
7  |   ...
8  |
9  /* To track line and column */
10 int yycolumnno = 1;
11
12 /* To track token metrics */
13 int token_count = 0;
14 int keyword_count = 0;
15 int identifier_count = 0;
16
17 /* Gets color based on token type */
18 const char* get_color(const char* type) {
19 |   ...
20 }
21
22 /* Prints token to console and output file */
23 void print_token(const char* type) {
24 |   ...
25 }
26
27 /* Prints string to console and output file */
28 void print_to_console_and_output_file(const char* format, ...) {
29 |   ...
30 }
31
32 /* Prints token metrics summary */
33 void print_token_metrics() {
34 |   ...
35 }
36 %}
```

Această secțiune definește funcționalități auxiliare în C, inclusiv:

- **Coduri de culoare ANSI** pentru afișarea colorată a tokenurilor în consolă.
- Variabile globale pentru gestionarea liniei, coloanei și statisticilor tokenurilor.
- Funcții pentru afișarea tokenurilor și metricilor.

3.2 Definiții de simboluri și expresii regulate

```
1  /* Definitions */
2  DIGIT      [0-9]
3  LETTER     [a-zA-Z]
4  IDENTIFIER {LETTER}{LETTER}|{DIGIT})*
5  NUMBER     {DIGIT}+(\.{DIGIT})?
6  STRING     \"[^\"]*\"
7  BOOL       \"true\"|\"false\"
8  %x COMMENT
```

Definirea șabloanelor de bază pentru tokenuri:

- **DIGIT** și **LETTER** sunt folosite pentru a construi identificatori și numere.
- **IDENTIFIER**: Nume de variabile sau funcții.
- **NUMBER**: Constante numerice (cu sau fără punct zecimal).
- **STRING**: Șiruri de caractere între ghilimele.
- **BOOL**: Literalii booleani (**true/false**).
- **%x COMMENT**: Mod pentru gestionarea comentariilor multiline.

3.3 Reguli pentru tokenuri și procesare

```

1  %%
2  \n+ {
3      int newline_count = yyleng;
4      yylineno += newline_count;
5      yycolumnno = 1;
6      printf("\n");
7  }
8
9  "int"|"float"|"string"|"bool"|"if"|"else"|"while"|"for"|"return" {
10     token_count++;
11     keyword_count++;
12     print_token("KEYWORD");
13     yycolumnno += yyleng;
14 }
15
16 "=="|"!="|"<"|>"|<="|>=" {
17     token_count++;
18     print_token("RELATIONAL_OPERATOR");
19     yycolumnno += yyleng;
20 }
21
22 "+"|"-"|"*"|"/" {
23     token_count++;
24     print_token("ARITHMETIC_OPERATOR");
25     yycolumnno += yyleng;
26 }
27
28 "(" { print_token("LEFT_PARENTHESIS"); yycolumnno += yyleng; token_count++; }
29 ")" { print_token("RIGHT_PARENTHESIS"); yycolumnno += yyleng; token_count++; }
--

```

Reguli principale pentru identificarea și procesarea tokenurilor:

- **Salt la linie nouă (\n+):** Actualizează linia curentă și resetează coloana.
- **Cuvinte cheie:** Detectează și contorizează cuvintele precum `int`, `if`.
- **Operatori relaționali:** Detectează `==`, `<`, `>=`.
- **Paranteze:** Identifică simboluri precum `(` și `)`.

3.4 Gestionarea comentariilor


```

1  /*
2  | printf(COLOR_CYAN"%d:%d" COMMENT START: /*s\n", yylineno, yycolumnno, COLOR_RESET);
3  | fprintf(output, "[%d:%d] COMMENT START: /*\n", yylineno, yycolumnno);
4  |
5  | yycolumnno += yyleng; // Update column number
6  | BEGIN(COMMENT); // Enter COMMENT state
7  | }
8  | <COMMENT>{
9  |     /*
10 |     | {
11 |     | printf(COLOR_CYAN"%d:%d" COMMENT END: /*s\n", yylineno, yycolumnno, COLOR_RESET);
12 |     | fprintf(output, "[%d:%d] COMMENT END: /*\n", yylineno, yycolumnno);
13 |     |
14 |     | yycolumnno += yyleng; // Update column number
15 |     | BEGIN(INITIAL); // Return to INITIAL state
16 |     | }
17 |     | [^*\n]+ {
18 |     | printf(COLOR_CYAN"%d:%d" COMMENT: %s\n", yylineno, yycolumnno, yytext, COLOR_RESET);
19 |     | fprintf(output, "[%d:%d] COMMENT: %s\n", yylineno, yycolumnno, yytext);
20 |     |
21 |     | yycolumnno += yyleng; // Update column number
22 |     | }
23 |     | /*
24 |     | {
25 |     | printf(COLOR_CYAN"%d:%d" COMMENT: /*s\n", yylineno, yycolumnno, COLOR_RESET);
26 |     | fprintf(output, "[%d:%d] COMMENT: /*\n", yylineno, yycolumnno);
27 |     |
28 |     | yycolumnno += yyleng; // Update column number
29 |     | }
30 |     | \n {
31 |     | yylineno++; // Increment line number
32 |     | yycolumnno = 1; // Reset column number
33 |     | }
34 | }
35 | <COMMENT><<EOF>> {
36 |     error_count++;
37 |     printf(COLOR_RED"%d:%d" ERROR: Unterminated multi-line comment\n", yylineno, yycolumnno);
38 |     BEGIN(INITIAL); /* Reset state */
39 | }

```

Reguli pentru comentarii:

- **Comentarii pe o linie:** // urmat de orice text.
- **Comentarii multiline:** Începute cu /* și închise cu */, gestionate cu un mod și condiții exclusive (%x COMMENT).

3.5 Funcția principală main

```

1  int main(int argc, char** argv) {
2      output = fopen(argv[1] ? argv[1] : default_output_filename, "w");
3      if (!output) {
4          perror("Error opening output file");
5          return 1;
6      }
7
8      if (argc == 2) {
9          printf("\nInteractive Mode: Type input and press Enter to process, Ctrl+D (EOF) to stop.\n");
10         yyin = stdin;
11     } else if (argc == 3) {
12         FILE *file = fopen(argv[2], "r");
13         if (!file) {
14             perror("Error opening input file");
15             return 1;
16         }
17         yyin = file;
18     } else {
19         fprintf(stderr, "Usage: %s [output_file] [input_file]\n", argv[0]);
20         return 1;
21     }
22
23     yylineno = 1;
24     yycolumnno = 1;
25
26     print_to_console_and_output_file("\nStart lexical analysis:\n\n");
27     yylex();
28     return 0;
29 }

```

Funcția principală:


- Inițializează fișierul de ieșire și modul de intrare (interactiv sau din fișier).
- Rulează funcția `yylex()` pentru analiza lexicală.

3.6 Închiderea fluxului de analiză

```

1  int yywrap() {
2      print_token_metrics();
3      return 1;
4  }
5
6

```



Funcția `yywrap()` este apelată la sfârșitul intrării pentru a afișa un rezumat al metricilor tokenurilor.

4. Rulare și automatizare

Comenzi explicite pentru fiecare etapă

1. Generarea fișierului C folosind Flex

```
flex lex_source.l
```

- Flex procesează fișierul sursă `lex_source.l` și generează fișierul `lex.yy.c`.

2. Compilarea fișierului C generat

```
gcc lex.yy.c -o lexer
```

- Compilatorul GCC compilează fișierul `lex.yy.c` și creează executabilul `lexer`.

3. Rulare în mod interactiv

```
./lexer output.txt
```

- Rulează lexer-ul în mod interactiv, citind intrarea de la tastatură (`stdin`).
- Tokenurile generate sunt scrise în `output.txt`.

4. Rulare cu fișier de intrare

```
./lexer output.txt input.txt
```

- Rulează lexer-ul folosind `input.txt` ca fișier de intrare.
- Tokenurile sunt scrise în `output.txt`.

5. Ștergerea fișierelor generate

```
rm -f lex.yy.c lexer output.txt
```

- Șterge fișierele temporare și executabilele generate:
 - `lex.yy.c`: Fișier C generat de Flex.
 - `lexer`: Executabilul lexer-ului.
 - `output.txt`: Fișierul de ieșire.

M Makefile

```
1 CC = gcc                # Compilator utilizat pentru codul C.
2 LEX = flex              # Utilitarul Flex pentru generarea lexer-ului.
3 LEXER = lexer           # Numele executabilului rezultat.
4 LEXER_C = lex.yy.c      # Fișierul C generat de Flex.
5 LEX_SOURCE = lex_source.l # Fișierul sursă Flex.
6 INPUT = input.txt       # Fișierul de intrare pentru testare.
7 OUTPUT = output.txt     # Fișierul de ieșire unde vor fi scrise tokenurile.
8
9 # default target
10 all: run
11
12 run-interactive:
13     | ./$(LEXER) $(OUTPUT)
14
15 run: build
16     | ./$(LEXER) $(OUTPUT) $(INPUT)
17
18 build: $(LEX_SOURCE)
19     | $(LEX) $(LEX_SOURCE)
20     | $(CC) $(LEXER_C) -o $(LEXER)
21
22 clean:
23     | rm -f $(LEXER_C) $(LEXER) $(OUTPUT)
```

Automatizarea procesului de construire, rulare și curățare a lexer-ului este realizată printr-un **Makefile**, un instrument standard pentru gestionarea proceselor de compilare în proiecte C. Acest Makefile definește țintele și comenzile necesare pentru a simplifica utilizarea lexer-ului.

1. Ținta implicită: **all**

- Setează ținta **run** ca fiind cea implicită.
- Executarea comenzii **make** fără argumente va construi și rula lexer-ul folosind fișierul de intrare specificat.

2. Ținta **run-interactive**

- Rulează lexer-ul în **mod interactiv**, citind intrarea direct din consolă (**stdin**).
- Rezultatele sunt salvate în fișierul de ieșire specificat.

3. Ținta **run**

- Asigură că lexer-ul este construit (**build**) înainte de rulare.
- Rulează lexer-ul folosind fișierul de intrare specificat (**input.txt**).

- Scrie rezultatele analizei lexicale în fișierul de ieșire (`output.txt`).

4. Ținta **build**

- Construiește lexer-ul în două etape:
 1. **Flex** procesează fișierul sursă (`lex_source.l`) și generează un fișier C (`lex.yy.c`).
 2. **GCC** compilează fișierul C generat într-un executabil (`lexer`).

5. Ținta **clean**

- Șterge fișierele generate (`lex.yy.c`, `lexer` și `output.txt`).
- Această țintă este utilă pentru a curăța proiectul și a începe de la zero.

5. Concluzii

Proiectul de analiză lexicală pentru pseudo-C, realizat cu ajutorul Flex, demonstrează importanța acestui pas fundamental în procesul de compilare. Analizatorul lexical creat a îndeplinit cu succes obiectivele propuse, oferind o soluție robustă pentru identificarea și clasificarea tokenurilor dintr-o gramatică specifică unui subset al limbajului C.

Realizări principale

1. **Automatizarea procesului de analiză lexicală:**
 - Am utilizat Flex pentru a genera automat codul necesar procesării tokenurilor, reducând timpul și complexitatea implementării manuale.
2. **Detectarea corectă a tokenurilor:**
 - Lexer-ul a identificat corect componentele de bază ale limbajului pseudo-C, inclusiv cuvinte cheie, identificatori, operatori, literalii numerici, șiruri de caractere și booleani.
3. **Gestionarea erorilor lexicale:**
 - Au fost implementate reguli pentru raportarea simbolurilor necunoscute, permițând utilizatorului să identifice rapid problemele din codul sursă.
4. **Output vizual îmbunătățit:**

- Utilizarea culorilor pentru afișarea diferitelor tipuri de tokenuri a făcut procesul de analiză mai intuitiv și ușor de urmărit.

5. Automatizare prin Makefile:

- Makefile-ul creat a simplificat procesul de construire, rulare și curățare a lexer-ului, făcând proiectul ușor de utilizat pentru diferite scenarii.

Beneficiile analizei lexicale cu Flex

- **Scalabilitate:** Flex permite extinderea ușoară a regulilor lexicale pentru a adăuga noi tokenuri.
- **Eficiență:** Automatizarea procesului de generare a lexer-ului reduce timpul necesar implementării manuale.
- **Integrare:** Lexer-ul generat poate fi integrat fără dificultate într-un compilator complet, fiind pregătit pentru a transmite tokenuri analizatorului sintactic.

Limitări și posibile îmbunătățiri

1. Complexitatea gramaticală:

- Lexer-ul creat este limitat la analiza lexicală a unui subset al limbajului C. Extinderea sa pentru a include reguli mai complexe ar necesita modificări ale definițiilor și regulilor.

2. Gestionarea erorilor complexe:

- Raportarea erorilor lexicale ar putea fi îmbunătățită prin includerea unor sugestii de corectare.

3. Testare extensivă:

- Testarea lexer-ului pe un set mai variat de coduri sursă ar putea asigura o acoperire mai largă a scenariilor posibile.

Acest proiect a demonstrat că Flex este un instrument puternic pentru construirea analizatorilor lexicali. Rezultatele obținute arată că analiza lexicală nu doar simplifică prelucrarea codului sursă, dar și oferă o bază solidă pentru etapele următoare de analiză sintactică și semantică. Cu o întreținere și extindere adecvată, lexer-ul creat poate evolua într-o componentă esențială a unui compilator complet.