Analizator lexical pt C

Sa se studieze programul flex.

- nu prea stiu ce anume ar trebui sa fac aici asa ca o sa scriu lucrurile interesante pe care le intalnesc pe parcurs mai jos.

Wikipedia

- flex (fast lexical analyzer generator) e o alternativa la lex
- il gasesti usor pe linux
- foloseste automate finite deterministe in spate

Ce face

- primeste un fisier $.l$ care contine reguli si genereaza un fisier C care parseaza text
- pt fiecare regula executa un cod de C dar de user

Exemplu

```
%%
username    printf( "%s", getlogin() ); // inlocuieste fiecare aparitie a lui username cu apelul functiei getlogin() care returneaza un string cu usernameul de linux
```

Complexitate

- daca nu este folosit REJECT programul ruleaza in $\mathrm{O}(N)$
  - oarecum seamana cu varianta de RegEx folosita de golang(Go) care e se garanteaza ca e parsat in $\mathrm{O}(N)$ datorita lipsei unor feature-uri care il faceau exponential.

Manual

1. **Forma generala**

```
definitions
%%
rules
%%
```

```
user code
```

- **definitions**
  - **DIGIT [0-9]**
  - **ID [a-z][a-z0-9]***
- **rules**
  - **pattern action**
    - pattern = {DIGIT}+"."{DIGIT}*
    - action = printf(“numar float”);
- **user code**
  - e copiat in `lex.yy.c` asa cum e

Bonus

- liniile indendate sunt copiata asa cum sunt
- la fel si liniile intre `%{ … %}` iar `%{` si `%}` trebuie sa fie singule pe linii si neindentate

2. **Forma de RegEx**

- pare similara cu un RegEx simplu
- diferenta e ca daca se vrea sa se matchuiasca un nume se foloseste `{DIGIT}` ca sa matchuiasca `[0-9]` nu cuvantul explicit `DIGIT`
- o gramada de lucruri predefinite cum ar fi `[:alpha:]` pe care nu o sa le folosim

Building the lexer

1. Preprocesor tokens

```
^{WHITESPACE}*"#".*    {
    PrintLeadingWhitespaces();
    printf("[preprocessor token]");
}
%%
void PrintLeadingWhitespaces() {
    int i = 0;
    while (yytext[i] == ' ' || yytext[i] == '\t') {
        printf("%c", yytext[i]);
        i += 1;
    }
```

```
}
```

```
#include <stdio.h>
#ifdef __LOCAL__
    #define LOCAL 1
#else
    #define LOCAL 0
#endif
```

Gives

```
[preprocessor token]
[preprocessor token]
    [preprocessor token]
[preprocessor token]
    [preprocessor token]
[preprocessor token]
```

2. Keywords! - the complete list: here

- pentru fiecare keyword o sa scriem `[keyword]` dar o sa le structuram in grupe in functie de: daca sunt sau nu tipuri de ex;
  - Problema: pentru “int” matchuieste printf ca pr[keyword]f
  - Solutie: Nu facem nimic! O sa se rezolva cand o sa matchuim `printf` ca [identifier]

```
int main() {
    int a = 1337; // 1337
    long int b = 0b1100110; // 102
    unsigned int c = 0xf; // 15
    int d = 01337; // 735
    printf("%d\n", LOCAL); // depends on the compile option
    printf("%d\n", (int)(a + b + c + d)); // 2189
```

```
    float pi = 3.141592;
    double sqrt_2 = 1.4142;
    printf("%.3f\n", pi * sqrt_2);

    for (int i = 0; i < 10; i += 1) {
        printf("\t%d\n", i);
    }
}
```

Gives

```
[keyword] main() {
    [keyword] a = 1337; // 1337
    [keyword] [keyword] b = 0b1100110; // 102
    [keyword] [keyword] c = 0xf; // 15
    [keyword] d = 01337; // 735
    pr[keyword]f("%d\n", LOCAL); // depends on the compile opt
ion
    pr[keyword]f("%d\n", ([keyword])(a + b + c + d)); // 2189

    [keyword] pi = 3.141592;
    [keyword] sqrt_2 = 1.4142;
    pr[keyword]f("%.3f\n", pi * sqrt_2);

    [keyword] ([keyword] i = 0; i < 10; i += 1) {
        pr[keyword]f("\t%d\n", i);
    }
}
```

3. Numbers

- O sa tratam doar cazurile cu numbere scrise in diferite baze (2, 8, 10, 16) fara sa tratam numerele scrise in forma stintifica (`scientific notation`)

```
0[x|X][0-9|a-f|A-F]+        printf("[number in base 16]");
```

```
0[b|B][0|1]+                printf("[number in base 2]");
0[0-7]+                     printf("[number in base 8]");
[0-9]+\.[0-9]+              printf("[real number]");
[0-9]+                      printf("[number in base 10]");
```

4. String literals

- String literals in C sunt un pic dubiosi in sensul de exista `line break` care spune ca linia viitoare se continua pe asta, nu e o linie noua
- Asta afecteaza atat string-literals cat si chestiile de preprocesor

```
\"([^\"\\\n]|(\\\n)|(\\.))*\"        printf("[string litera
l]");
```

Logica:

- un string incepe cu `"` si se termina cu `"`
- el poate contine orice inauntru
- trebuie sa avem grija la caractere `"` escape-uite in interiorul stringului si la problema cu `line break`
- Astfel, eliminam din ‘orice’ urmatoarele elemente: `" \n \` ca mai apoi sa le punem dupa explicit
  - eliminam `"` ca sa avem o stare de oprire.
  - eliminam `\n` ca sa matchuim doar liniile care se termina cu `\`, semnificand un `line break`
  - eliminam `\` ca sa punem sa ne dam seama daca avem sau nu caractere escapeuite
- punem regula pentru `\\\n` semnificand o linie care se termina cu `\`
- punem sa match-uim orice caracter escapuit. e.g: `\n \t \" \\`
  - practic match-uim in plus `\\\n` dar nu lasam caracterul `"` daca nu are un `\` in fata

```
    printf("%.3f\n", pi * sqrt_2);


    printf("salutare asta e un string \
pe mai multe linii \
huhuhuh \" \n \' \" caractere speciale      \t      \' \"");
    printf("\n~~~~\n\n");
```

```
    for (int i = 0; i < 10; i += 1) {
        printf("\t%d\n", i);
    }
```

Gives

```
    [identifier]([string literal], [identifier] * [identifie
r]);


    [identifier]([string literal]);
    [identifier]([string literal]);


    [keyword] ([keyword] [identifier] = [number in base 10];
[identifier] < [number in base 10]; [identifier] += [number in
base 10]) {
        [identifier]([string literal], [identifier]);
    }
```

5. Identifiers

- Definim un identifier ca orice element pe care un user il poate numi:
  - functie
  - variabila
  - clasa
- Regulile C pentru identifieri sunt:
  - poate incepe cu `a-zA-Z` si `_`
  - dupa poate contine si `0-9` dar nu poate incepe cu asta

```
[a-zA-Z_][a-zA-Z_0-9]*        printf("[identifier]");
```

6. Comentarii

- singura linie:

```
"//"[^\n]*                          printf("[single-line-commen
t]");
```

- linii multiple
  - avem o problema similara cu cea de la string-literals deja

```
"/*"([^*]|(\*[^\/]))*"*/"        printf("[multiple-line-commen
t]");
```

Logica:

- un comentariu e de forma `/* … */`
- unde acel `…` poate contine si `*` sau `/`
  - o sa construim ceva care o sa matchuiasca cat de mult poate dar nu poate avea `*/` ca substring

```
    printf("%.3f\n", /* sa nu inmultim si cu e? */ pi * sqrt_
2);
    printf("uite cum se pune un comentariu: /* comentariu */
");


/*
    // cometariu micut si ascuns
    for (int i = 0; i < 10; i += 1) {
        printf("\t%d\n", i);
    }
*/
```

Gives

```
    [identifier]([string literal], [multiple-line-comment] [id
entifier] * [identifier]);
    [identifier]([string literal]);


[multiple-line-comment]
```

7. preprocessor-tokens done right

- problema de la string-uri se aplica si aici
- urmatorul cod e valid:

```
#ifdef \
    __LOCAL__
```

Magic, `^{WHITESPACE}*"#"(.|(\\\n))*` cuprinde si cazul descris mai sus .. desii spre surprinderea mea poate nu ar trebui.

Aparent daca matchuieste . poate sa mai matchuiasca si ceva ce a fost matchuit de `.`
Intrebarea e daca se poate nega un string?

8. operatori - lista completa ❤️

```
"+="|"-="|"*="|"/="|"^="|"&="|"|="|"%="|">>="|"<<="|"=" {
  printf("[assignment operators]");
}
"+"|"-"|"*"|"/"|"%"             printf("[math operator]");
"<<"|">>"|"&"|"|"|"^"           printf("[binary operator]");
"~"|"!"|"++"|"--"|"&" {
   printf("[unary operator]"); // somehow sizeof is a unary op
erator as well
}
"&&"|"||"                       printf("[logical operator]");
"<="|">="|"<"|">"               printf("[relational operato
r]");
```

9. finishing everything!

- Daca matchuim si paranteze + whitespace + ; + , si doar le afisam ar trebui sa putem matchui tot

```
"("|")"|"["|"]"|"{"|"}"|","|";"      ECHO;
" "|"\t"|"\n"                        ECHO;
.                                    printf("Bad token '%c'",
yytext[0]);
```

10. Lucruri care dau prost 😓

- nu merge inline if: `(a == 0) ? 1 : 0`
- `int 0identifier = 1;` da `[keyword] [number in base 10][identifier] [assignment operators] [number in base 10];` in loc de `bad token 0`
- `+++` da `[unary operator][assignment operators]` dar asta nu pare o eroare de lexer