Limbaje formale si tehnici de compilare

Laborator 3

Implementarea analizorului lexical (ALEX)

Rolul ALEX este sa grupeze caracterele de intrare in atomi lexicali (AL, token). AL sunt unitati lexicale indivizibile din punctul de vedere al etapelor ulterioare ale compilatorului (ex: numere, identificatori, cuvinte cheie, …). Totodata vor fi eliminate toate secventele care nu mai sunt relevante in etapele ulterioare (comentarii, spatii, linii noi). Un AL este compus din:

- **nume (tip, cod)** – un nume simbolic (constanta numerica) ce permite identificarea unitatii lexicale respective. Exemple: INT, ID, STR, ADD. De multe ori cand se vorbeste despre un AL, de fapt se vorbeste despre codul lui.
- **atribute** – informatii asociate cum ar fi: caracterele propriu-zise din care este compus un identificator sau un sir de caractere, valoarea numerica pentru constante numerice, linia din fisierul de intrare, …. Daca unele dintre aceste atribute sunt mutual exclusive (valoarea numerica a unui numar nu se foloseste niciodata simultan cu sirul de caractere necesar pentru un identificator), ele se pot grupa intr-un *union*, pentru a ocupa mai putin spatiu in memorie.

O posibila structura care defineste un AL poate fi:

```
enum{ID, END, CT_INT, ASSIGN, SEMICOLON…};   // codurile AL

typedef struct _Token{
  int             code;         // codul (numele)
  union{
     char           *text;       // folosit pentru ID, CT_STRING (alocat dinamic)
     long int       i;           // folosit pentru CT_INT, CT_CHAR
     double         r;           // folosit pentru CT_REAL
     };
  int             line;         // linia din fisierul de intrare
  struct _Token *next;          // inlantuire la urmatorul AL
  }Token;
```

Campul **next** este folosit in cazul in care vom folosi o structura de tip lista pentru a mentine atomii lexicali. In acest caz vom considera variabila **tokens** ca fiind inceputul acestei liste si pentru usurinta adaugarii vom mai mentine inca o variabila **lastToken** care va pointa la ultimul AL din lista. Initial aceste doua variabile vor fi NULL. Putem implementa o functie care sa creeze si sa adauge un nou AL in lista:

```
Token     *addTk(int code)
{
  Token       *tk;
  SAFEALLOC(tk,Token)
  tk->code=code;
  tk->line=line;
  tk->next=NULL;
  if(lastToken){
     lastToken->next=tk;
     }else{
     tokens=tk;
     }
  lastToken=tk;
  return tk;
}
```

Aceasta functie mai are nevoie si de variabila **line** care contine numarul liniei curente. **SAFEALLOC** este un macro care aloca memorie pentru un tip dat si iese din program daca nu are memorie suficienta:

```
#define SAFEALLOC(var,Type) if((var=(Type*)malloc(sizeof(Type)))==NULL)err("not enough memory");
```

Functia **err** are un numar variabil de argumente si este folosita analogic functiei **printf** (permite oricate argumente si placeholder-e). **err** afiseaza argumentele sale sub forma unui mesaj de eroare, dupa care iese din program:

```
void	err(const char *fmt,...)
{
  va_list	va;
  va_start(va,fmt);
  fprintf(stderr,"error: ");
  vfprintf(stderr,fmt,va);
  fputc('\n',stderr);
  va_end(va);
  exit(-1);
}
```

Pentru afisarea erorilor care pot aparea la o anumita pozitie in fisierul de intrare se poate folosi o varianta a functiei **err** care mai primeste ca parametru si un AL, din care extrage linia corespunzatoare erorii:

```
void	tkerr(const Token *tk,const char *fmt,...)
{
  va_list	va;
  va_start(va,fmt);
  fprintf(stderr,"error in line %d: ",tk->line);
  vfprintf(stderr,fmt,va);
  fputc('\n',stderr);
  va_end(va);
  exit(-1);
}
```

Deoarece dorim ca fiecare modul al compilatorului sa fie pe cat posibil cat mai independent de celelalte module (decuplare), in faza de analiza lexicala vom crea lista completa cu atomii lexicali din fisierul de intrare. Ulterior, in faza de analiza sintactica, vom folosi doar aceasta lista.

Partea cea mai importanta a ALEX este implementata intr-o functie „int **getNextToken**()”, care la fiecare apel va adauga in lista de AL urmatorul AL din sirul de intrare si ii returneaza codul. Aceasta functie va fi apelata pana cand va returna codul atomului lexical END, ceea ce marcheaza sfarsitul fisierului. In final toti AL vor fi in lista **tokens**.

Implementarea getNextToken() cu stari explicite

Este o implementare simplu de realizat, desi presupune mai mult cod. Se pleaca de la diagrama de tranzitii (DT) care cuprinde toate definitiile lexicale. Algoritmul este urmatorul:

- Consideram o variabila care contine starea curenta, initial 0 (starea initiala)
- Intr-o bucla infinita, la fiecare iteratie:
    - Pentru starea curenta testam caracterul de consumat conform tuturor tranzitiilor posibile din acea stare
    - Daca s-a gasit o tranzitie care consuma caracterul, acesta este consumat si noua stare devine starea curenta conform tranzitiei urmate
    - Daca nu s-a gasit o tranzitie care consuma caracterul dar starea curenta are o tranzitie de tip „else”, starea destinatie a acestei tranzitii devine starea curenta, fara a consuma caracterul de intrare
    - Daca starea curenta este o stare finala se creeaza un nou AL, i se seteaza atributele sale si acesta este returnat

Daca consideram urmatoarele definitii din AtomC:

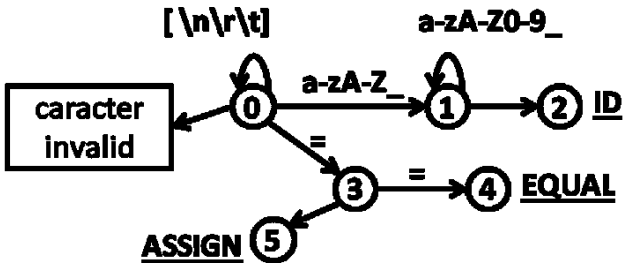
ID: [a-zA-Z_] [a-zA-Z0-9_]* ;

SPACE: [\n\r\t] ;

ASSIGN: '=' ;

EQUAL: '==' ;

O posibila DT poate fi urmatoarea:



Daca consideram ca deja am citit fisierul de intrare in memorie, am adaugat un terminator de sir (0) si variabila **pCrtCh** este un pointer la urmatorul caracter de procesat, implementarea **getNextToken** poate fi:

```
int     getNextToken()
{
int             state=0,nCh;
char            ch;
const char      *pStartCh;
Token           *tk;

while(1){    // bucla infinita
    ch=*pCrtCh;
    switch(state){
        case 0:             // testare tranzitii posibile din starea 0
            if(isalpha(ch)||ch=='_'){
                pStartCh=pCrtCh;        // memoreaza inceputul ID-ului
                pCrtCh++;               // consuma caracterul
                state=1;                // trece la noua stare
                }
            else if(ch=='='){
                pCrtCh++;
                state=3;
                }
            else if(ch==' '||ch=='\r'||ch=='\t'){
                pCrtCh++;       // consuma caracterul si ramane in starea 0
                }
            else if(ch=='\n'){  // tratat separat pentru a actualiza linia curenta
                line++;
                pCrtCh++;
                }
            else if(ch==0){     // sfarsit de sir
                addTk(END);
                return END;
                }
            else tkerr(addTk(END),"caracter invalid");
            break;
        case 1:
            if(isalnum(ch)||ch=='_')pCrtCh++;
                else state=2;
```

```
                break;
            case 2:
                nCh=pCrtCh-pStartCh; // lungimea cuvantului gasit
                // teste cuvinte cheie
                if(nCh==5&&!memcmp(pStartCh,"break",5))tk=addTk(BREAK);
                else if(nCh==4&&!memcmp(pStartCh,"char",4))tk=addTk(CHAR);
                // … toate cuvintele cheie …
                else{                   // daca nu este un cuvant cheie, atunci e un ID
                    tk=addTk(ID);
                    tk->text=createString(pStartCh,pCrtCh);
                    }
            return tk->code;
            case 3:
                if(ch=='='){
                    pCrtCh++;
                    state=4;
                    }
                else state=5;
                break;
            case 4:
                addTk(EQUAL);
                return EQUAL;
            case 5:
                addTk(ASSIGN);
                return ASSIGN;
            }
        }
    }
```

Se poate constata ca aceasta implementare urmeaza indeaproape diagrama de tranzitii si astfel este destul de simplu de realizat. La ID-uri s-a explicitat si cum se pot testa cuvintele cheie, care generic vorbind au aceeasi definitie ca un ID, dar au o semnificatie speciala. Evident la implementarile mai performante se pot folosi pentru aceste teste vectori asociativi sau chiar diagrame de tranzitie care cuprind si toate cuvintele cheie.

Functia **createString** creeaza o noua copie, alocata dinamic, pentru sirul care incepe cu **pStartCh** si se incheie cu **pCrtCh** (exclusiv). Este folosita pentru setarea atributului **text** al ID-urilor. Pentru setarea atributelor numerice ale CT_INT si CT_REAL se pot folosi functii care convertesc siruri de caractere la valori numerice (**strtol**, **atof**) sau aceste conversii se pot realiza direct din cod.

Observatie: mai exista si alte posibilitati de implementare a functiei **getNextToken**, de exemplu folosind stari implicite. In aceasta implementare nu mai este nevoie de DT ci se urmareste implementarea directa a logicii definitiilor regulate ale atomilor lexicali. Aceasta implementare de regula necesita mai putin cod de scris si e mai lizibila, dar este putin mai complexa.

Pentru testarea ALEX va exista o functie care afiseaza toti AL (**showAtoms**). Aceasta functie va itera lista **tokens** si va afisa codurile numerice ale tuturor atomilor. Daca unii atomi au si atribute (ID, CT_INT, …) , se vor afisa si acestea. Exemplu pentru enum-ul de mai sus si programul „speed=70;”: 0:speed 3 2:70 4 1.

Idei pentru testare si depanare:

- Pentru diversi AL se vor incerca mai multe combinatii corecte si gresite, pentru a se vedea daca rezultatele sunt cele asteptate. De exemplu pentru CT_INT se poate testa:
    - corect: 0, 25, 017, 9, 0xAFd9
    - gresit: 08, 0xG

- Daca atomii lexicali nu sunt recunoscuti corect si din DT sau din implementare nu se poate remedia problema, se poate pune in **getNextToken** un **printf** inainte de *switch*-ul starilor care sa afiseze atat starea cat si caracterul curent la inceputul fiecarei tranzitii. In acest fel se poate vizualiza clar drumul care este parcurs in interiorul DT.

In finalul executiei compilatorului, toata memoria alocata dinamic va trebui eliberata. Aceasta se poate realiza cu o functie **terminare**(), care elibereaza toate structurile de date folosite.

Aplicatia 3.1: *Sa se scrie codul getNextToken() corespunzator definitiilor pentru CT_INT si COMMENT. Pentru CT_INT se va obtine si atributul sau (valoarea propriu-zisa).*

Tema: sa se scrie analizorul lexical complet pentru limbajul AtomC si sa se apeleze folosind un fisier de intrare, pentru a se obtine si afisa toti atomii din acel fisier. Cerinte:

- In CT_STRING si CT_CHAR secventele ESCAPE vor fi inlocuite prin codul ASCII corespunzator (ex: '\\' 't' -> '\t')
- Atomii vor contine linia din fisierul de intrare de unde au fost formati
- CT_INT si CT_REAL vor avea caracterele lor transformate in valoare numerica
- La sfarsitul listei de atomi se va adauga atomul END
- Spatiile si comentariile nu formeaza atomi
- Atomii se afiseaza folosind numele codului. Pentru CT_* si ID se va afisa si informatia lexicala asociata. (ex: ID:tmp ASSIGN CT_INT:108 SEMICOLON END)
- Analizorul lexical se va testa folosind programul de test 8.c