Limbaje formale și tehnici de compilare

implementarea analizorului lexical

Implementarea analizorului lexical (ALEX)

Rolul ALEX este să grupeze caracterele de intrare în atomi lexicali (AL, token). AL sunt unități lexicale indivizibile din punctul de vedere al etapelor ulterioare ale compilatorului (ex: numere, identificatori, cuvinte cheie, …). Totodată vor fi eliminate toate secvențele care nu mai sunt relevante în etapele ulterioare (comentarii, spații, linii noi). Un AL este compus din:

- **nume (tip, cod)** – un nume simbolic (constantă numerică) ce permite identificarea AL respectiv. Exemple: INT, ID, STR, ADD. De multe ori când se vorbește despre un AL, de fapt se vorbește despre codul lui.
- **atribute** – informații asociate cum ar fi: caracterele propriu-zise din care este compus un identificator sau un șir de caractere, valoarea numerică pentru constante numerice, linia din fișierul de intrare, …. Dacă unele dintre aceste atribute sunt mutual exclusive (valoarea numerică a unui număr nu se folosește niciodată simultan cu șirul de caractere necesar pentru un identificator), ele se pot grupa într-un *union*, pentru a ocupa mai puțin spațiu în memorie.

O posibilă structură 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)
                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;
```

Atomii se păstrează sub forma unei liste simplu înlănțuite. Câmpul **next** indică atomul următor din listă. Vom considera variabila globală **tokens** ca fiind începutul acestei liste și, pentru ușurința adăugării, vom mai menține încă o variabilă **lastToken** care va pointa la ultimul AL din listă. Inițial aceste două variabile vor fi NULL. Putem implementa o funcție care să creeze și să adauge un nou AL în listă:

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

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;
}
```

Această funcție mai are nevoie și de variabila globală **line** care conține numărul liniei curente. **SAFEALLOC** este un macro care alocă memorie pentru un tip dat și iese din program dacă nu are memorie suficientă.
Funcția **err** are un număr variabil de argumente și este folosită analogic funcției **printf** (permite oricâte argumente și placeholdere). **err** afișează argumentele sale sub forma unui mesaj de eroare, după care încheie programul:

```
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 afișarea erorilor care pot apărea la o anumită poziție în fișierul de intrare, se poate folosi o variantă a funcției **err** care mai primește ca parametru și un AL, din care extrage linia corespunzătoare 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 să fie cât mai independent de celelalte module (decuplare), în faza de analiză lexicală vom crea lista completă cu atomii lexicali din fișierul de intrare. Ulterior, în faza de analiză sintactică, vom folosi doar această listă.
Partea cea mai importantă a ALEX este implementată într-o funcție „int **getNextToken**()”, care la fiecare apel va adăuga în lista de AL următorul AL din șirul de intrare și îi returnează codul. Această funcție va fi apelată până când va returna codul atomului lexical END, ceea ce marchează sfârșitul fișierului. În final toți AL vor fi în lista **tokens**.

Implementarea getNextToken() cu stări explicite

Este o implementare simplu de realizat, deși presupune mai mult cod. Se pleacă de la diagrama de tranziții (DT) care cuprinde toate definițiile lexicale. Algoritmul este următorul:

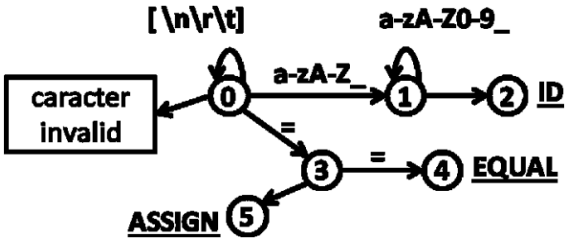
- Considerăm o variabilă care conține starea curentă, inițial 0 (starea inițială)
- Într-o buclă infinită, la fiecare iterație:

- Pentru starea curentă, testăm caracterul de consumat conform tuturor tranzițiilor posibile din acea stare
- Dacă s-a găsit o tranziție care consumă caracterul, acesta este consumat și starea finală a tranziției folosite devine starea curentă
- Dacă nu s-a găsit o tranziție care consumă caracterul, dar starea curentă are o tranziție de tip „else”, starea destinatie a acestei tranziții devine starea curentă, fără a consuma caracterul de intrare
- Dacă starea curentă este o stare finală, se creează un nou AL, i se seteaza atributele sale și acesta este returnat

Dacă considerăm următoarele definiții din AtomC:

ID: [a-zA-Z_] [a-zA-Z0-9_]* ;
SPACE: [\n\r\t] ;
ASSIGN: '=' ;
EQUAL: '==' ;

O posibilă DT poate fi următoarea:



Dacă considerăm că deja am citit fișierul de intrare în memorie, am adăugat un terminator de șir (\0) și variabila **pCrtCh** este un pointer la următorul caracter de procesat, implementarea **getNextToken** poate fi:

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

    for(;;){                                // 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 că această implementare urmează îndeaproape diagrama de tranziții, și astfel este destul de simplu de realizat. La ID-uri s-a explicitat și cum se pot testa cuvintele cheie, care generic vorbind au aceeași definiție ca un ID, dar au o semnificație specială. Evident la implementările mai performante se pot folosi pentru aceste teste vectori asociativi sau chiar diagrame de tranziție care cuprind și toate cuvintele cheie.

Funcția **createString** creează o nouă copie, alocată dinamic, pentru șirul care începe cu **pStartCh** și se încheie cu **pCrtCh** (exclusiv). Este folosită pentru setarea atributului **text** al ID-urilor. Pentru setarea atributelor numerice ale CT_INT și CT_REAL se pot folosi funcții care convertesc siruri de caractere la valori numerice (**strtol**, **atof**) sau aceste conversii se pot realiza direct din cod.

Observație: mai există și alte posibilități de implementare a funcției **getNextToken**, de exemplu folosind stări implicite. În implementarea cu stări implicite nu mai este nevoie de DT, ci se urmărește implementarea directă a logicii definițiilor regulate ale atomilor lexicali. Această implementare de regulă necesită mai puțin cod de scris și e mai lizibilă, dar este puțin mai complexă.

Pentru testarea ALEX va exista o funcție care afișează toți AL (**showAtoms**). Această funcție va itera lista **tokens** și va afișa liniile și codurile numerice ale tuturor atomilor. Dacă unii atomi au și atribute (ID, CT_INT, …) , se vor afișa și acestea. De exemplu pentru enum-ul de mai sus și programul „speed=70;”:

```
1 ID:speed
1 ASSIGN
1 CT_INT:70
1 SEMICOLON
1 END
```

Idei pentru testare și depanare:

- Pentru diverși AL se vor încerca mai multe combinatii corecte și greșite, pentru a se vedea dacă rezultatele sunt cele așteptate.
- Dacă atomii lexicali nu sunt recunoscuti corect, iar din DT sau din implementare nu se poate remedia problema, se poate pune în **getNextToken** un **printf** înainte de *switch*-ul stărilor, care să afișeze atât starea cât și caracterul curent la începutul fiecărei tranziții. În acest fel se poate vizualiza clar drumul care este parcurs în interiorul DT. Pentru *switch*-ul stărilor se poate pune o condiție de *default*, care afișează starea și caracterul curente. În acest fel se poate determina dacă o stare nu a fost implementată sau este vreo eroare de numerotare a stărilor.

În finalul execuției compilatorului, toată memoria alocată dinamic va trebui eliberată. Aceasta se poate realiza cu o funcție **terminare**(), care eliberează toate structurile de date folosite.

Aplicația 3.1: *Să se scrie codul getNextToken() corespunzător definițiilor pentru CT_INT și COMMENT. Pentru CT_INT se va obține și atributul său (valoarea propriu-zisă).*

Tema: să se scrie analizorul lexical complet pentru limbajul AtomC și să se apeleze folosind un fișier de intrare, pentru a se obține și afișa toți atomii din acel fișier. Cerințe:

- Atomii vor conține linia din fișierul de intrare de unde au fost formați
- CT_INT și CT_REAL vor avea caracterele lor transformate în valoare numerică
- La sfârșitul listei de atomi se va adăuga atomul END
- Spațiile și comentariile nu formează atomi
- Atomii se afișează folosind numele codului. Pentru CT_* și ID se va afișa și informația lexicală asociată. (ex: ID:tmp ASSIGN CT_INT:108 SEMICOLON END)
- Analizorul lexical se va testa folosind programul de test dat