Analizzatore lessicale o scanner

Lo scanner rappresenta un'interfaccia fra il programma sorgente e l'analizzatore sintattico o parser.

Lo scanner, attraverso un esame carattere per carattere dell'ingresso, separa il programma sorgente in parti chiamate token che rappresentano i nomi delle variabili, operatori, label, ecc.

Il parser genera un albero sintattico le cui foglie sono i simboli terminali della grammatica, ovvero i token che lo scanner ha estratto dal programma sorgente e passato al parser.

Il parser potrebbe fare direttamente anche l'analisi sintattica, ma non è conveniente in quanto la grammatica per i token è una grammatica regolare più semplice quindi di quella che tratta il parser.

Lo scanner può interagire con il parser in due modi differenti:

- Lavorare in un passo separato, producendo i token in una grossa tabella in memoria di massa;

- Interagire direttamente con il parser che chiama lo scanner quando è necessario il prossimo token nell'analisi sintattica (preferibile).

Lo scanner suddivide il programma sorgente in *token.*
Il tipo di token è rappresentato con un numero intero unico (esempio, variabile con il numero 1, costante 2, label 3).

Il token, che è una stringa di caratteri, è memorizzato in una tabella.
I valori delle costanti sono memorizzati in una *constant table,* mentre i nomi delle variabili in una *symbol table.*

Esempio:

```
SUM:    A=A+B;
        GOTO DONE;
```

Output dello scanner:

Token	Rapp. Interna	Locazione
SUM	3	1
:	11	0
A	1	2
=	6	0
A	1	2
+	5	0
B	1	3
;	12	0
GOTO	4	0
DONE	3	4
;	12	0

Si noti che gli spazi bianchi sono stati ignorati dallo scanner.

Il Progetto di uno Scanner e la sua Realizzazione

Compiti:

- Eliminare spazi bianchi, commenti ecc;ù
- Isolare il prossimo token dalla sequenza di caratteri in input;
- Isolare identificatori e parole-chiave;
- Generare la symbol-table.

I token possono essere descritti in diversi modi. Spesso si utilizzano le **grammatiche regolari**.

Esempio:

Grammatica regolare per generare i numeri naturali:

<unsigned integer>::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
0<unsigned integer> |
1<unsigned integer> |
...
9<unsigned integer>

Un altro modo per descrivere i token è in modo riconoscitivo piuttosto che generativo mediante **automi a stati finiti**.

Esempio di applicazione:

Diagramma a stati finiti per un linguaggio che consiste dei token <, <=, =, >=, >, <>, (,), +, -, *, /, :=, ;, identificatori, parole chiave, costanti, e stringhe (fra apici).

Commenti (/* e */) o spazi bianchi sono ignorati.

Algoritmo:

Implementa lo scanner specificato dall'automa a stati finiti seguente.

Procedure SCAN(PROGRAM, LOOKAHEAD, CHAR TOKEN, POS)

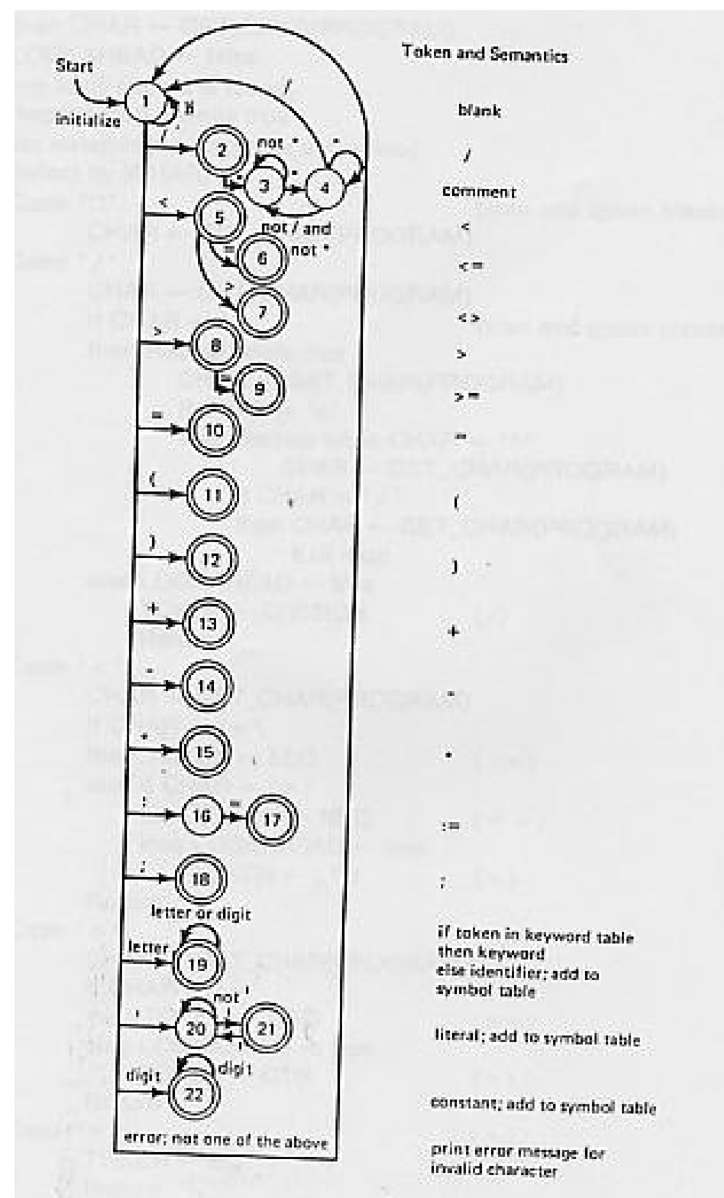
Data la stringa sorgente PROGRAM, questo algoritmo ritorna il numero di rappresentazione interna del TOKEN successivo nella stringa.
Se TOKEN rappresenta un identificatore, stringa o costante, la procedura ritorna anche la sua posizione numerica nella tabella POS.

CHAR rappresenta il carattere corrente su cui si sta facendo lo scanner. LOOKAHEAD è una variabile logica che ci dice se il simbolo in CHAR è stato usato nella chiamata precedente a SCAN. Un valore "false" denota che non è stato ancora controllato.

L' algoritmo utilizza le seguenti funzioni:

GET_CHAR(PROGRAM) che ritorna il prossimo carattere del programma sorgente;
INSERT(STRING,type) che inserisce un dato token STRING (se necessario) ed il suo tipo (cioè costante, stringa o variabile) nella tabella dei simboli;
KEYWORD(STRING) che ritorna il numero di rappresentazione interna del suo argomento se è una keyword, 0 altrimenti.
STRING contiene il token corrente (nome di variabile, costante, stringa).

Le variabili DIVISION, LEQ, NEQ, LTN, GEQ, GTN, EQ, LEFT, RIGHT, ADDITION, SUBTRACTION, MULIPLICATION, ASSIGNMENT, SEMICOLON, LITERAL, IDENTIFIER e CONSTANT contengono i numeri interni di rappresentazione dei token /, <=, <>,<,>=, >, =, (,), +, -, ~, :=, ;, stringhe, identificatori e costanti rispettivamente.



```
1. [Initialize]
     POS ← 0
2. [Get first character]
     If not LOOKAHEAD
     then CHAR ← GET_CHAR(PROGRAM)
     LOOKAHEAD ← false
3. [Loop until a token is found]
     Repeat step 4 while true
4. [Case statement to determine next token]
     Select by (CHAR)
     Case '□':                                    (scan and ignore blanks)
          CHAR ← GET_CHAR(PROGRAM)
     Case '/':
          CHAR ← GET_CHAR(PROGRAM)
          If CHAR = '*'                           (scan and ignore comments)
          then Repeat while true
                    CHAR ← GET_CHAR(PROGRAM)
                    If CHAR = '*'
                    then Repeat while CHAR = '*'
                              CHAR ← GET_CHAR(PROGRAM)
                         If CHAR = '/'
                         then CHAR ← GET_CHAR(PROGRAM)
                              Exit loop
          else LOOKAHEAD ← true
               TOKEN ← DIVISION                   (/)
               Return
     Case '<':
          CHAR ← GET_CHAR(PROGRAM)
          If CHAR = '='
          then TOKEN ← LEQ                        (<=)
          else If CHAR = '>'
               then TOKEN ← NEQ                   (< >)
               else LOOKAHEAD ← true
                    TOKEN ← LTN                   (<)
          Return
     Case '>':
          CHAR ← GET_CHAR(PROGRAM)
          If CHAR = '='
          then TOKEN ← GEQ                        (>=)
          else LOOKAHEAD ← true
               TOKEN ← GTN                        (>)
          Return
     Case '=':                                    (=)
          TOKEN ← EQ
          Return
```

```
Case '(':                                         (()
     TOKEN ← LEFT
     Return
Case ')':                                         ())
     TOKEN ← RIGHT
     Return
Case '+':                                         (+)
     TOKEN ← ADDITION
     Return
Case '-':                                         (-)
     TOKEN ← SUBTRACTION
     Return
Case '*':                                         (*)
     TOKEN ← MULTIPLICATION
     Return
Case ':':
     CHAR ← GET_CHAR(PROGRAM)
     If CHAR = '='
     then TOKEN ← ASSIGNMENT                      (:=)
          Return
     else
          CHAR ← GET_CHAR(PROGRAM)
          Write('UNKNOWN TOKEN ":"', CHAR,
            'IN SOURCE STRING')
Case ';':                                         (;)
     TOKEN ← SEMICOLON
     Return
Case '''':                                        (literal)
     STRING ← ''
     Repeat while true
        CHAR ← GET_CHAR(PROGRAM)
        If CHAR = ''''
        then CHAR ← GET_CHAR(PROGRAM)
             If CHAR ≠ ''''
             then LOOKAHEAD ← true
                  POS ← INSERT(STRING, LITERAL)
                  TOKEN ← LITERAL
                  Return
        STRING ← STRING∘CHAR
Default:
     If CHAR >= 'A' and CHAR <= 'Z'
     then STRING ← CHAR                           (identifier)
          CHAR ← GET_CHAR(PROGRAM)
          Repeat while  (CHAR >= 'A' and CHAR <= 'Z')
                     or (CHAR >= '0' and CHAR <= '9')
```

```
        STRING ← STRING∘CHAR
        CHAR ← GET_CHAR(PROGRAM)
    LOOKAHEAD ← true
    If KEYWORD(STRING) > 0
    then TOKEN ← KEYWORD(STRING)
            Return
    POS ← INSERT(STRING, IDENTIFIER)
    TOKEN ← IDENTIFIER
    Return
If CHAR >= '0' and CHAR <= '9'      (constant)
then STRING ← CHAR
    CHAR ← GET_CHAR(PROGRAM)
    Repeat while CHAR >= '0' and CHAR <= '9'
        STRING ← STRING∘CHAR
        CHAR ← GET_CHAR(PROGRAM)
    LOOKAHEAD ← true
    POS ← INSERT(STRING, CONSTANT)
    TOKEN ← CONSTANT
    Return
Write('ERROR--UNKNOWN CHARACTER', CHAR,
      'IN SOURCE STRING')
CHAR ← GET_CHAR(PROGRAM)
```

Nota:
In alcuni linguaggi non è possibile determinare sempre cos'è un token nel momento in cui è stato letto completamente.

Esempio:

FORTRAN (i bianchi sono ignorati completamente):

DO10I = 1,20
DO10I=1+20

Il primo è l'inizio di un loop e DO1OI consiste dei tre token DO, 10 e I, mentre nel secondo DO10I è un identificatore.

Si deve risolvere il problema andando avanti fino a che l'ambiguità è risolta.

I ***linguaggi regolari*** vengono riconosciuti in modo efficiente attraverso automi a stati finiti.

Tuttavia, sono linguaggi abbastanza ***limitati*** e con le loro grammatiche non si possono descrivere anche semplici costrutti dei linguaggi di programmazione.

L'utilizzo delle grammatiche regolari è perciò limitato alla costruzione (e riconoscimento) dei simboli base usati in un programma (quello che si indica generalmente come ***lessico****),* ad esempio gli *identificatori*.

Il riconoscimento di tali simboli base è la prima fase eseguita durante la compilazione di un programma e prende il nome di ***analisi lessicale****.*

Una volta terminata l'analisi lessicale sono stati individuati, ad esempio, le costanti, le variabili, etc. utilizzate nel programma e costruita quella che si chiama ***tabella dei simboli****.*

L'analisi del programma prosegue poi con tecniche più complesse (analisi sintattica e analisi semantica).

Le grammatiche regolari possono essere usate per rappresentare scanner che sono implementati come automi a stati finiti.

Poichè realizzare un automa a stati finiti è banale sono stati progettati programmi in grado di generare automaticamente degli scanner usando un metodo formale per specificare l'automa a stati finiti.

Un noto generatore di scanner è *Lex* (1975) che utilizza *espressioni regolari* per specificare lo scanner.

Utilizzando espressioni regolari ed associati segmenti di codice chiamati *azioni*, Lex genera una tabella delle transizioni e un programma che interpreta tale tabella.

L'azione è eseguita quando si riconosce un token generato dall'espressione regolare.

L'output di Lex è quindi un programma che simula un automa a stati finiti ed esegue le funzioni addizionali associate con le azioni.

Lo scanner generato da Lex può essere usato in congiunzione con un parser (YACC) per eseguire sia l'analisi lessicale che sintattica.