Implementazione di un compilatore da “adding calculator” (ac) a “desktop calculator” (dc)

Paola Giannini

Implementazione dello scanner per **ac**

Un programma sorgente

```
Programma Sorgente ac

int tempa;
tempa = 5;
float tempb = tempa + 3.2;
tempb = tempb - 7;
print tempb;
```

Definizione informale di ac

In ac ci sono

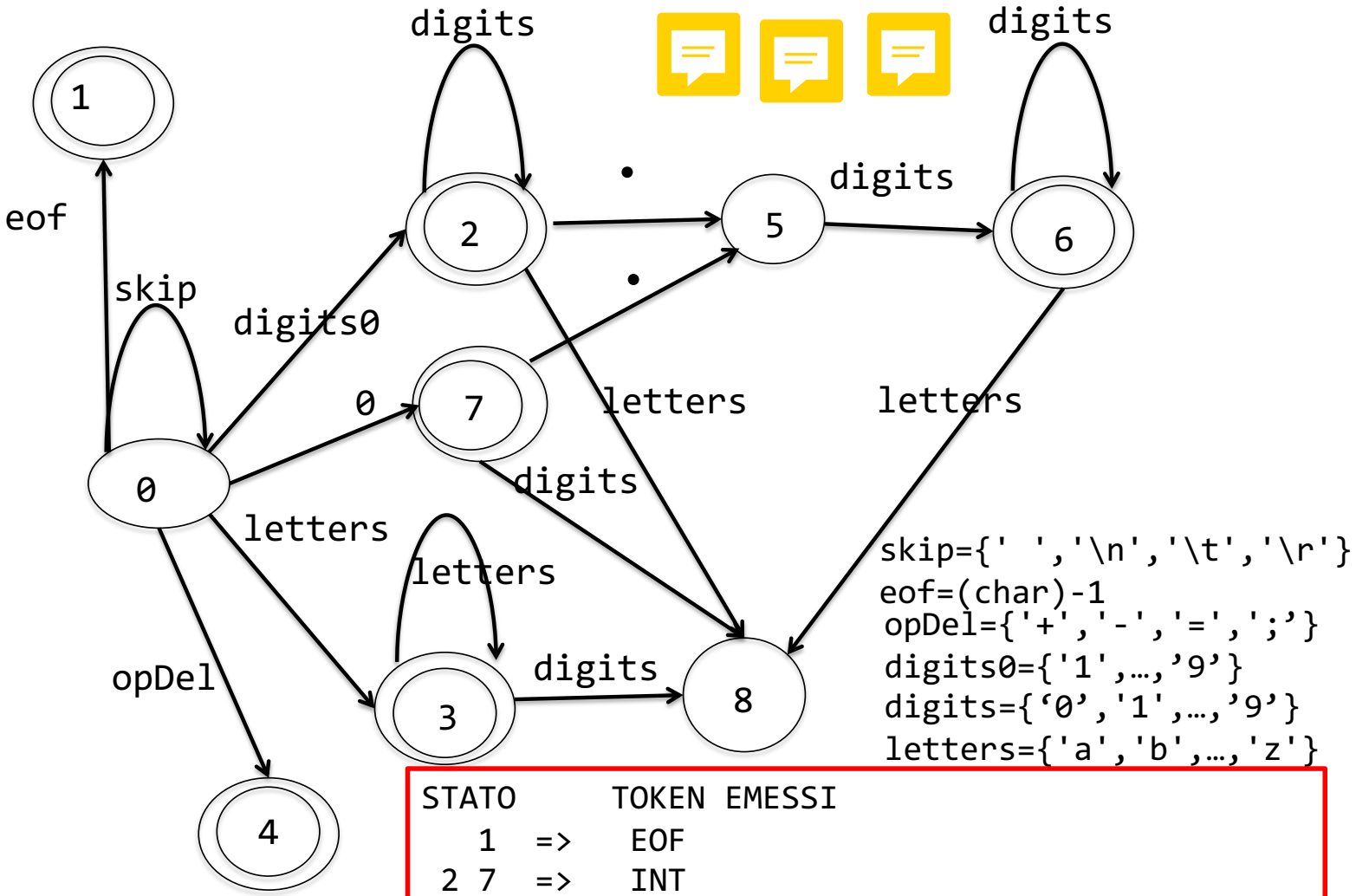
- 2 **tipi** di dato interi e floating point. Un letterale
  - intero è una sequenza di cifre;
  - floating point è una sequenza (non vuota) di cifre seguita da "." seguita da almeno 1 cifra e al più 5 cifre; se volete usare usare una definizione più restrittiva fatelo pure!
- **variabili** che sono stringhe contenenti solo i 26 caratteri dell'alfabeto inglese minuscoli. Una variabile deve essere dichiarata prima di poter essere usata (in una espressione);
- **dichiarazioni**: `float` o `int` seguito da una variabile
- 2 **istruzioni**: *assegnamento* e *stampa*.
  - sintassi dell'assegnamento: variabile "=" espressione
  - sintassi della stampa: `print` variabile
- Le **espressioni** possono essere letterali (interi o floating point), variabili oppure somma e sottrazione di espressioni.
- Una espressione di tipo `int` può essere **convertita automaticamente** a `float` (se necessario) e nessun altra conversione è possibile.

I token e i pattern associati

Token	Pattern	Classe rappresentata
INT	0 | [1-9][0-9]*	Costante/Letterale
FLOAT	(0 | [1-9][0-9]*).[0-9]{1,5}	Costante/Letterale
ID	[a-z]+	Identificatore
TYINT	int	Parola chiave
TYFLOAT	float	Parola chiave
ASSIGN	=	Operatore
PRINT	print	Operatore
PLUS	+	Operatore
MINUS	-	Operatore
SEMI	;	Separatore/Delimitatore
EOF	(char) -1	Fine Input

- Notate che INT e FLOAT hanno la stessa espressione regolare iniziale
- CARATTERI DA IGNORARE: ' ', '\n', '\t', '\r'

L'automa che riconosce i pattern dei Token (1)



STATO		TOKEN EMESSI
1	=>	EOF
2 7	=>	INT
3	=>	ID PRINT TYFLOAT TYINT
4	=>	PLUS MINUS SEMI ASSIGN
6	=>	FLOAT (cifre dopo . <= 5)

L'automa che riconosce i pattern dei Token (2)

- Notate che lo stato 8 è una sorta di stato di errore. Da questo stato non possiamo più leggere niente.
- Arriviamo nello stato 8 se
  - partendo a riconoscere un identificatore troviamo che questo contiene un numero, oppure
  - partendo a riconoscere un intero o un float troviamo che questo contiene questo è seguito da una lettera, oppure
  - `0` è seguito da una cifra.

UPO
UNIVERSITÀ DEL PIEMONTE ORIENTALE

Packages

Per il momento definiamo 3 packages:

1. token: contiene le classi per l'implementazione dei token
2. scanner: contiene la classe `Scanner` e le eventuali classi di eccezioni per l'analisi lessicale
3. test: classi di test per le componenti del compilatore
4. test.data : classi per i files per i test

UPO
UNIVERSITÀ DEL PIEMONTE ORIENTALE

Implementazione Token

```
public enum TokenType {FLOAT, INT, ID, ......}

public class Token {
     private int riga; // riga del codice in cui si trova il token
     private TokenType tipo;// il tipo del token
     private String val;// per identificatori e numeri contiene
                        // la stringa matchata
       ..............
}
```

Metodi e costruttori

Definite la classe `Token` in un package `token`. La/Le classe/i devono contenere

- costruttori
- il metodo `toString()` che produce una stringa in cui sono contenuti il tipo, la riga a cui è definito e il valore (se c'è) del token.
- i getters dei campi privati della classe

Il `toString()` dei token prodotti per il programma sarà:

```
<INT,r:1><ID,r:1,tempa><SEMI,r:1>
<ID,r:2,tempa><ASSIGN,r:2><INUM,r:2,5><SEMI,r:2>
<FLOAT,r:3><ID,r:3,tempb><ASSIGN,r:3><ID,r:3,tempa><PLUS,r:3><FNUM,r:3,3.2><SEMI,r:2>
<ID,r:4,tempb><ASSIGN,r:4><ID,r:4,tempb><PLUS,r:4><INUM,r:4,7><SEMI,r:4>
<PRINT,r:5><ID,r:5,tempb><SEMI,r:5><EOF,r:5>
```

ho messo la sequenza dei token divisi per riga solo per chiarezza!

UPO
UNIVERSITÀ DEL PIEMONTE ORIENTALE

Scanner

Nel package `scanner` sarà definita

- la classe `Scanner` che avrà
  - un costruttore che prende come input il nome di un file e costruisce un `PushbackReader`: memorizzato in un campo privato, `buffer`,
  - il metodo `nextToken()` che restituisce il prossimo token dell'input,
  - metodi privati necessari a strutturare bene i codice (es: scanNumber, scanID, readChar, peekChar, inizializza, ...)
  - per scrivere il codice in maniera più strutturata sono utili i seguenti campi:
    1. una lista dei caratteri di "skip",
    2. una lista delle "lettere" e
    3. una lista dei "numeri"
    4. una associazione fra stringhe che denotano parole chiave e e il corrispondente `TokenType`
    5. una associazione fra caratteri che denotano operatori assegnamento e il delimitatore e il corrispondente `TokenType`
  - come implementate liste e associazioni?

UPO
UNIVERSITÀ DEL PIEMONTE ORIENTALE

Schema dello Scanner

Lo trovate nello scheletro della classe `Scanner`

Test

Definiamo in un package `test`

- Una classe di test `TestToken` che testa la costruzione dei `Token` e il metodo `toString()`.
- Una classe di test `TestScanner` che testa che il metodo `nextToken()` ritorni i token corretti (nel caso di input corretto) e che si accorga degli errori e li segnali nel modo corretto.
  - Sviluppate in modo incrementale, prima solo caratteri di skip e identificatori e parole chiave, poi aggiungete gli operatori e infine i numeri (che sono la parte più complicata).
  - Usate i files di test, aggiungendone dei vostri se volete.
  - La classe di test avrà la creazione degli scanner nei metodi `@BeforeEach`, così i test testeranno solamente il corretto funzionamento della `nextToken()`.

UPO
UNIVERSITÀ DEL PIEMONTE ORIENTALE