Corso di Laurea Magistrale in Ingegneria Informatica
A.A. 2013-2014

Linguaggi Formali e Compilatori

Analisi lessicale (scanner)

Giacomo PISCITELLI

Distinzione terminologica

- stringa lessicale o lessema = sequenza significativa di caratteri
- simbolo o token = astrazione di una classe di stringhe lessicali:
  esempio → **id** token associato alla classe {istanze di lessemi "**id**entificatori"}
- pattern = regole per descrivere come è fatto un lessema della classe

Simbolo	*Istanze*	*Pattern*
while	`while`	`while`
begin	`begin`	`begin`
relop	`<, <=, >, >=, !=, =, ==`	{`<, <=, >, >=, !=, =, ==`}
id	`partenza, tempo, m24, X2`	lettera seguita da lettere e/o cifre
num	`3, 25, 3.5, 4.37E12`	costante numerica
strconst	`"Hello world!"`	sequenza di caratteri racchiusa tra "

Il **token** rappresenta un insieme di stringhe descritte da un **pattern**.

id rappresenta un insieme di stringhe che iniziano con una lettera e continuano con lettere e cifre. La stringa reale (**`newval`**) è chiamata **lessema**.

Compito dell' Analizzatore lessicale

alfa = (beta * 3) ⇒ ⟨id, ↑alfa⟩ ⟨assign, ⟩ ⟨left, ⟩ ⟨id, ↑beta⟩ ⟨times, ⟩ ⟨num, 3⟩ ⟨right, ⟩

Compito di un analizzatore lessicale:

- data una sequenza di caratteri di un alfabeto (stringa di caratteri costituenti lo statement), verificare se la sequenza può essere decomposta in una sequenza di lessemi (simboli ammessi dal linguaggio)
- in caso positivo, restituire in uscita la sequenza di token per ciascun lessema del programma sorgente
- in caso negativo, restituire un errore lessicale

Per ogni lessema, lo scanner produce in uscita un token.

Ruolo dell' Analizzatore lessicale

Il **lessico** descrive le parole o elementi lessicali che compongono le frasi.

Nei linguaggi artificiali gli elementi lessicali possono essere assegnati a:

Parole chiave: sono particolari parole fisse che caratterizzano vari tipi di frasi o strutture. Ad es.: **`if`** , **`begin`**, **`subprogram`**, **`write`**. Le parole chiave sono indeclinabili.

Delimitatori, operatori e caratteri compositi: come i precedenti sono delle parole fisse composte però di caratteri anche non alfabetici. Ad es. i commenti sono preceduti nel linguaggio Ada dai due trattini **`--`** ; l'operatore "maggiore o eguale" è scritto come **`>=`** ed in altri linguaggi come **`.GTE.`** , acronimo di *greater or equal*.

Classi lessicali aperte: queste comprendono un numero illimitato di elementi lessicali, che devono avere la struttura di un linguaggio regolare ossia a stati finiti. Sono esempi tipici di classi aperte i seguenti:

- nomi o identificatori di variabili, di sottoprogrammi, o in generale di varie entità del linguaggio; ad es. gli identificatori di molti linguaggi sono definiti dalla espressione regolare:
  `identificatore = lettera (lettera | cifra)*`
- costanti quali i numeri interi o reali o le stringhe alfanumeriche.

Ruolo dell'Analizzatore lessicale

Vi è una importante differenza tra le parole chiave e le classi lessicali aperte:

- **le parole chiave** non hanno altra informazione che il proprio nome;
- **le classi lessicali** sono solitamente delle stringhe appartenenti ad un linguaggio formale del tipo regolare. Tanto gli identificatori quanto le costanti denotano delle entità che hanno un valore o certe altre proprietà, che più avanti saranno chiamate degli attributi semantici.

Token, pattern, lessico e attributi

Un token è descritto tramite una 3-tuple:

1. **Nome del Token** (id, keyword, etc.)
2. **Attributo del token** (può essere un valore o un puntatore alla tabella dei simboli)
3. **Posizione del token** (opzionale)

Token	Lessico	Pattern
id	count a123	stringa che inizia con una lettera e contiene lettere e numeri
num	123 345.67 2.6e10	qualsiasi costante numerica
if	if	la keyword "if"

- la stringa a123 è il valore di un token il cui tipo è identificatore
- la stringa 123 è il valore di un token il cui tipo è intero

Token, pattern, lessico e attributi

Qualche volta il valore è ignorato, per esempio una keyword oppure un simbolo aritmetico può essere specificato solo dal suo nome.

Token	PATTERN	Attributi		
TOKEN_if	PAROLA CHIAVE IF	NESSUN ATTRIBUTO		
TOKEN_while	PAROLA CHIAVE WHILE	NESSUN ATTRIBUTO		
…				
TOKEN_SOMMA	+	NESSUN ATTRIBUTO		
TOKEN_ASSEGN	=	NESSUN ATTRIBUTO		
…				
TOKEN_ID	Sequenza di caratteri che inizia con una lettera	indice	Altri attributi	
TOKEN_ID	Sequenza di caratteri che inizia con una lettera	X	Altri attributi	
…				
TOKEN_NUM	Sequenza di cifre	2	00000010	Altri attributi
TOKEN_NUM	Sequenza di cifre	12	00001100	Altri attributi
…				

Token e tabella dei simboli

Esempi di token

1. Keywords (e.g., IF)
2. Segni di punteggiatura (e.g. , ;)
3. Operatori costituiti da caratteri singoli e multipli (e.g., =, ==, <=)
4. Identificatori
5. Numeri
   - Interi
   - Reali
6. Commenti

Dato che un token può corrispondere ad uno o più elementi lessicali, altre informazioni possono essere aggiunte per specificarlo. Queste informazioni addizionali sono chiamati **attributi** di natura semantica.

Per semplicità, un token può avere un singolo attributo che raggruppa tutte le informazioni relative al token.

Per gli identificatori questo attributo è un *puntatore alla tabella dei simboli*, e la tabella dei simboli contiene i reali attributi del token.

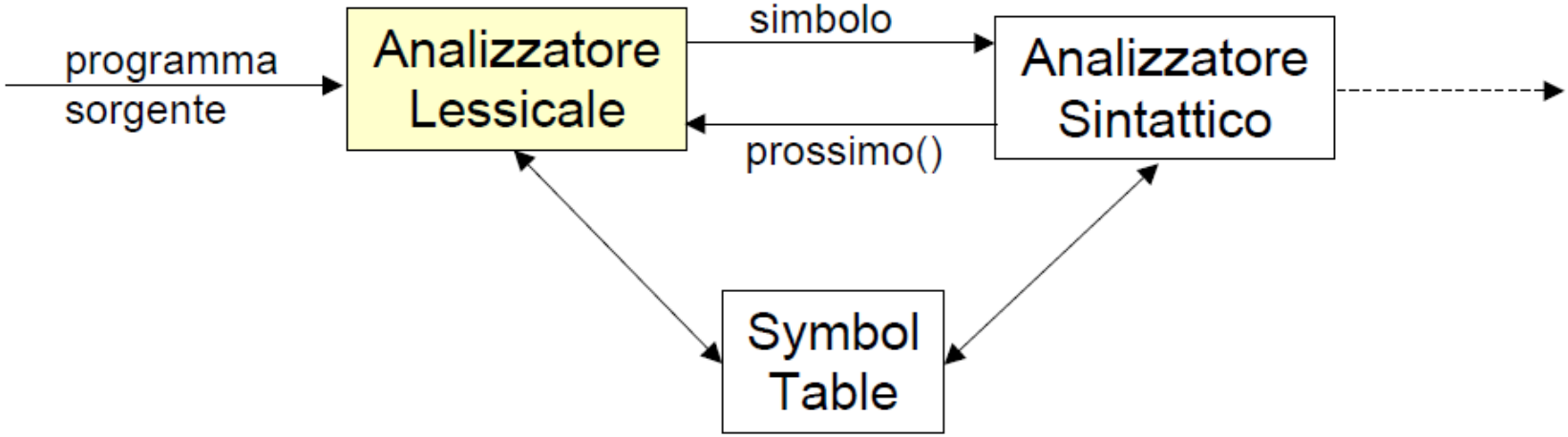
alfa = (beta * 3) ⇒ ⟨id, ↑alfa⟩ ⟨assign, ⟩ ⟨left, ⟩ ⟨id, ↑beta⟩ ⟨times, ⟩ ⟨num, 3⟩ ⟨right, ⟩

Ruolo dell' Analizzatore lessicale

Ruolo primario = processo di astrazione: ⟨*caratteri*⟩ → ⟨*simboli*⟩

Non esiste alcuna necessità di separare l'analisi lessicale dall'analisi sintattica, se non la maggiore semplicità di gestione dei moduli in modo separato.

Quando lo scanner e il parser agiscono nello stesso tempo, un analizzatore lessicale non ritorna una lista di token, ma ritorna un token alla volta quando il parser richiede un token.



- Vantaggi nella separazione tra analisi sintattica e analisi lessicale:
  1. Semplicità di design
  2. Efficienza
  3. Portabilità

Ruolo dell' Analizzatore lessicale

Poiché lo scanner è la parte del compilatore che legge il "sorgente", esso può svolgere (oltre al compito d'identificazione dei lessemi e quindi dei token da fornire al parser) quelli di:

- Eliminare gli spazi bianchi e i commenti
- Inserire i simboli nella tabella dei simboli
- Cancellare/inserire/sostituire caratteri nell'input o trasporre due caratteri adiacenti
- Numerare le linee di codice, così da associare ad un eventuale messaggio di errore il numero di linea di codice corrispondente
- Fornire un modo per isolare le regole di basso livello dalle strutture che costituiscono la sintassi del linguaggio

Funzionamento dell' Analizzatore lessicale

- L'analizzatore lessicale riconosce come token la stringa più lunga possibile.
  Esempio **newval** -- **n ne new newv newva newval**

- Normalmente non è definito un carattere di fine token.
  Che cosa è la fine di un token? Esiste un carattere che demarca la fine di un token?
  Se il numero di caratteri del token è fissato non ci sono problemi: (esempio **+ -** in Pascal, ma non in C)

- Delimitatori/spaziatori: caratteri sui quali una stringa cessa di rappresentare un simbolo ("confini" dei token)

  → **(blank | tab | newline | comment)**$^+$

  Sono pertanto necessari dei *lookhead*, cioè un numero massimo di caratteri entro il quale deve essere trovato il carattere che demarca la fine di un token.

Funzionamento dell' Analizzatore lessicale

Essendo, come è noto, un linguaggio formale del tipo regolare riconoscibile dagli automi a stati finiti (o macchine sequenziali), il programma che tratta gli elementi lessicali, detto analizzatore lessicale o scanner, è un algoritmo che realizza la funzione di transizione di un automa finito.

Più precisamente l'analizzatore lessicale non deve solo verificare se una sottostringa del testo sorgente corrisponde ad un elemento lessicale valido, ma deve anche tradurla in una **opportuna codifica** che faciliti la successiva elaborazione (analisi sintattica) da parte del traduttore o interprete.

La codifica deve contenere due informazioni: l'identificativo della classe lessicale a cui l'elemento appartiene e l'attributo semantico (nel caso ve ne sia uno associato a tale classe).

Ad es. la stringa 3.14159 è riconosciuta come costante numerica reale e tradotta nella coppia ('costante_reale', valore_della_costante).

Realizzazione di uno scanner

Per realizzare uno scanner vengono adoperati tre approcci:

1. **Realizzazione procedurale** o a controllo di programma (***hand-coded***):
   un programma ad-hoc per la grammatica G
   grammatica regolare → programma ad hoc
   espressione regolare → programma ad hoc

2. **Realizzazione tabulare interpretata:**
   una struttura dati rappresenta un DFA riconoscitore della grammatica G e un programma indipendente dalla grammatica G
   grammatica regolare → DFA
   espressione regolare → DFA

3. **Automaticamente con uno SCANNER GENERATOR**

Realizzazione di uno scanner

Le espressioni regolari sono una notazione formale sufficientemente potente per descrivere la varietà di token richiesti dai moderni linguaggi di programmazione.

In più, esse possono essere usate come specifica per la generazione automatica di automi finiti, che riconoscono insiemi regolari, cioè gli insiemi che le espressioni regolari definiscono.

Questa interpretazione delle espressioni regolari è la base dei **generatori di scanner**, programmi che producono uno scanner funzionante, una volta che gli venga data la specifica dei token che lo scanner deve riconoscere.

Ovviamente tale programma è un prezioso tool nella costruzione di un compilatore.

Alternativamente, per amore della semplicità, gli scanner possono essere **codificati a mano** (senza l'ausilio di tool), cioè costruiti per riconoscere i token di un particolare linguaggio.

L'uso di questo approccio è giustificato dal fatto che a volte si necessita di piú lavoro e tempo per imparare a usare un generatore di scanner, che per scrivere lo scanner direttamente.

Realizzazione hand-coded vs. table-driven

Gli scanner codificati a mano (***hand-coded***) sono stati una pratica comune fino a tempi recenti, in quanto si riteneva che gli scanner ***table-driven*** creati dai generatori fossero piú lenti rispetto a quelli hand-coded. Ogni overhead extra per uno scanner puó essere significativo, in quanto la scansione rappresenta una sostanziale frazione del processo di compilazione.

Per esempio vi sono report di compilatori, che spendono il 20% del loro tempo solo per saltare gli spazi.

Recenti sviluppi hanno dimostrato che gli scanner table-driven possono essere sempre piú veloci rispetto a quelli codificati senza ausilio di tool. Inoltre, il vantaggio degli scanner table-driven è che **lo stesso driver puó essere usato in una varietà di scanner**, cambiando solo le tabelle.

In conclusione possiamo dire che, una volta imparato ad usare un generatore di scanner, la creazione di uno scanner si semplifica.

Realizzazione hand-coded

Espressione regolare

↓

Codice per l'analisi della grammatica

Si scrive:

- ✓ una sequenza per ogni concatenazione
- ✓ un test per ogni unione
- ✓ un ciclo per ogni stella di Kleene

Esempio

r = c (c | 0)* | 0

↓

```
. . . .
{ leggi(carattere);
  if (carattere == 'c')
     { leggi(carattere)
       while(carattere == 'c' || carattere == '0')
            leggi(carattere)
       return}
  if (carattere == '0') return
  return errore
}
                                        . . . .
```

Realizzazione hand-coded

Grammatica regolare

↓

Codice per l'analisi della grammatica

- ✓ Si scrive una funzione per ogni non terminale
- ✓ Si scrive un test per ogni alternativa
- ✓ Si richiama la funzione per ogni non-terminale che compare nella parte destra di una derivazione

Esempio

V = {S, A}
T = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
P = { S → 0 ⊥
S → 1 A | 2 A | 3 A | 4 A | 5 A | 6 A | 7 A | 8 A | 9A
A → ⊥ | 0 A |1 A | 2 A | 3 A | 4 A | 5 A | 6 A | 7 A | 8 A | 9A
}

Realizzazione hand-coded

```
int S ()
{ leggi(carattere)
  if (carattere == 0)
        { leggi(carattere)
          if (carattere == ⊥ )
            return OK
          else
            return ERRORE
        }
  else
      if (carattere == 1..9)
         return A
      else
         return ERRORE
}
```

```
int A ()
{ leggi(carattere)
  if (carattere == ⊥ )
      return OK
  else
      if (carattere == 0..9)
         return A
      else
         return ERRORE
}
```

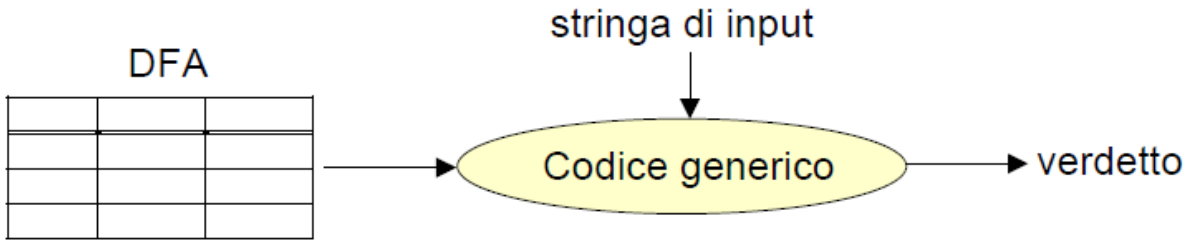
Realizzazione table driven

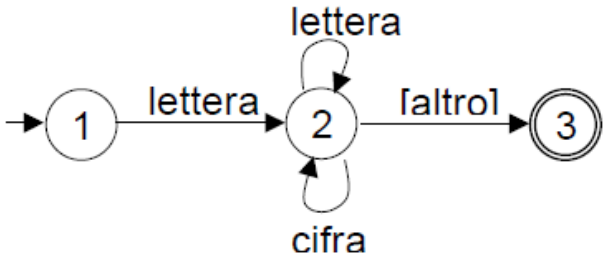
Espressione regolare

↓

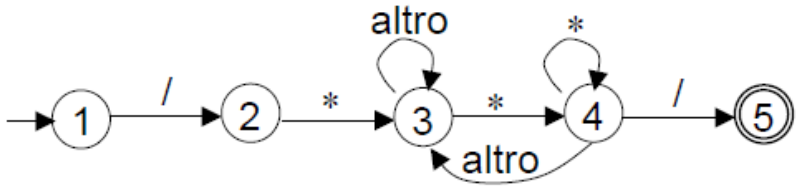
Automa a stati finiti deterministico

Realizzazione table driven con NDFSA





	c			
	lettera	*cifra*	*altro*	*F*
1	2			
2	2	2	[3]	
3				•



	c			
	/	*	altro	*F*
1	2			
2		3		
3		4	3	
4	5	4	3	
5				•

- Assunzioni:
  - Caselle bianche → transizioni non specificate
  - s_0 = il primo nell'elenco
  - Arricchimento informativo
    - stati ∈ F
    - consumo di *c*

Lexical Error Recovery

Una sequenza di caratteri per la quale non esiste un token valido è un errore lessicale.

Gli errori lessicali non sono comuni ma devono comunque essere gestiti dallo scanner. Non è opportuno bloccare il processo di compilazione per un tale errore.

Gli approcci alla gestione degli errori lessicali sono:

- Cancellare i caratteri letti fino al momento dell'errore e ricominciare le operazioni di scanning
- Eliminare il primo carattere letto dallo scanner e riprendere la scansione in corrispondenza del carattere successivo.

Di solito, un errore lessicale è causato dalla comparsa di qualche carattere illegale, soprattutto all'inizio di un token. In questo caso i due approcci sono equivalenti.