Corso di Linguaggi di Programmazione (II anno)

Progetto Java – Consegna 5 giugno 2001

a.a. 2000/01

Si consideri il linguaggio ToyTeX per l'elaborazione di testi, definito nel modo seguente.

Alfabeto

`document itemize quotation plain comment \begin \end \par \item \quote { }`

più i simboli in *Char*. Si può supporre che *Char* contenga tutti i caratteri che hanno codice Unicode compreso tra `\u0000` e `\u00FF` a parte il carattere \ (backslash).

Grammatica

Program	::=	`\begin{document}` *TextSpecList* `\end{document}`
TextSpecList	::=	Λ | *TextSpec TextSpecList*
TextSpec	::=	*Command* | *Environment* | *Text*
Command	::=	`\par` | `\item` | `\quote`
Environment	::=	`\begin { itemize }` *TextSpecList* `\end { itemize }` |
		`\begin { quotation }` *TextSpecList* `\end { quotation }` |
		`\begin { plain }` *TextSpecList* `\end { plain }` |
		`\begin { comment }` *TextSpecList* `\end { comment }`
Text	::=	Λ | *Char Text*

Un programma ToyTeX è una sequenza di specifiche di testo racchiusa tra i delimitatori `\begin{document}` e `\end{document}`. Una specifica di testo può essere un comando o un ambiente o una stringa. Un esempio di programma è il seguente:

```
\begin{document}
\begin{comment} Un piccolo esempio di programma in ToyTeX \end{comment}
Fabrizio De Andre' nasce a Genova Pegli il 18 febbraio 1940, figlio
di genitori della borghesia agiata.
\par
Inizia la sua carriera come chitarrista jazz, in omaggio al suo idolo Jim Hall.
Debutta nel 1958, con il singolo Nuvole barocche, la prima affermazione arriva
nel 1965 quando, appena sposato, firma la storica Canzone di Marinella
all'epoca portata al successo da Mina.
\par ... \par
A riguardo del mare, una volta disse:
\begin{quotation}
...davanti al mare si sono placate le  furie di Attila... una volta arrivato al
mare ti siedi li‘ e lo contempli e ti passano tutte le voglie strane...
\quote Fabrizio De Andre'
\end{quotation}
... \par
DISCOGRAFIA:
\begin{itemize}
 \item ...
 \item LA BUONA NOVELLA
   \begin{itemize}
    \item LAUDATE DOMINUM
    \item L'INFANZIA DI  MARIA
    \item IL RITORNO DI GIUSEPPE
```

```
    \item ...
   \end{itemize}
 \item ...
\end{itemize}
\end{document}
```

L'interpretazione di questo programma elabora i comandi, gli ambienti e i frammenti di testo contenuti in esso e produce come risultato il seguente testo (ossia una stringa):

```
Fabrizio De Andre' nasce a Genova Pegli il 18 febbraio 1940, figlio di genitori
della borghesia agiata.

Inizia la sua carriera come chitarrista jazz, in omaggio al suo idolo Jim Hall.
Debutta nel 1958, con il singolo Nuvole barocche, la prima affermazione arriva n
el 1965 quando, appena sposato, firma la storica Canzone di Marinella all'epoca
portata al successo da Mina.

...

A riguardo del mare, una volta disse:

          ...davanti al mare si sono placate le furie di Attila... una
          volta arrivato al mare ti siedi li‘ e lo contempli e ti pass
          ano tutte le voglie strane...
                              Fabrizio De Andre'

...

DISCOGRAFIA:

-)   ...
-)   LA BUONA NOVELLA

     -)   LAUDATE DOMINUM
     -)   L'INFANZIA DI MARIA
     -)   IL RITORNO DI GIUSEPPE
     -)   ...

-)   ...
```

Semantica informale Un ambiente (environment) determina come il testo e i comandi in esso contenuti devono essere elaborati. Un ambiente può contenere dei sotto-ambienti. Notare che l'interpretazione di un testo o di un comando dipende dall'ambiente più interno che li contiene. Ad esempio in

```
\begin{itemize}
1 Questo e‘ un elenco.
\begin{quotation}
2 Questa e‘ una citazione.
\end{quotation}
\end{itemize}
```

Il testo che inizia con `1` viene elaborato secondo le regole dell'ambiente `itemize`, mentre quello che inizia con `2` viene elaborato secondo le regole dell'ambiente `quotation`.

- L'ambiente `document` è quello più esterno e non può essere un sotto-ambiente. In tale ambiente il testo viene elaborato nel seguente modo:
  - ogni carattere il cui codice Unicode sia compreso tra `\u0000` e `\u0020` è considerato uno spazio bianco (quindi, ad esempio, le andate a capo sono semplicemente interpretate come spazi bianchi). Sequenze non vuote di spazi bianchi vengono interpretate come un unico spazio bianco; ossia, se tra le parole `BUONA` e `NOVELLA` lasciamo 1 o 100 spazi bianchi, il testo prodotto dal programma non cambia e contiene sempre un solo spazio tra le due parole `BUONA` e `NOVELLA`. Tutti gli altri caratteri (ossia quelli non compresi tra `\u0000` e `\u0020`) vengono interpretati per quello che sono.

- Ogni linea di testo prodotta non deve contenere più di *MAX*, dove *MAX* è una costante opportunamente definita, caratteri (escluso il carattere di fine linea che può essere in posizione $MAX + 1$). Le andate a capo (a parte il comando `\par` definito più sotto) sono automatiche: dopo aver prodotto il carattere alla posizione *MAX* della linea corrente viene inserita un'andata a capo e viene iniziata una nuova linea. Le andate a capo non devono tenere conto delle norme grammaticali (vedi l'esempio sopra). Notare che un carattere di fine linea nel programma non produce una nuova linea nel testo prodotto dal programma.

  L'unico comando valido nell'ambiente `document` è `\par`: esso forza due andate a capo (ossia viene inserita una linea vuota).

- L'ambiente `itemize` viene utilizzato per le elencazioni. Ogni volta che viene incontrato tale ambiente, vengono inserite due andate a capo nel testo prodotto; inoltre, il testo contenuto in esso viene spostato verso sinistra di una certa quantità (ossia viene introdotto un margine bianco sinistro). La larghezza di tale margine viene determinato dall'espressione $liv * TAB$ dove *liv* indica il *livello* dell'ambiente e *TAB* è una costante maggiore o uguale a 2, opportunamente definita, che indica un certo numero di spazi bianchi (contigui). Ogni ambiente *env* di tipo `itemize` ha un livello *liv* definito dalle seguenti regole:

  - se l'ambiente immediatamente esterno all'ambiente *env* non è di tipo `itemize`, allora $liv = 1$;
  - altrimenti, sia l il livello dell'ambiente (necessariamente di tipo `itemize`) immediatamente esterno ad *env*; allora $liv = l + 1$.

  Il testo viene interpretato come nell'ambiente `document` a parte il fatto che esiste un margine bianco sinistro (la lunghezza di una linea è sempre di *MAX* caratteri, ma in questo caso sono inclusi nei *MAX* caratteri anche gli spazi bianchi del margine sinistro).

  L'unico comando valido nell'ambiente `itemize` è `\item`: esso produce un'andata a capo, immediatamente seguita da $(liv - 1) * TAB$ spazi bianchi, immediatamente seguiti dai due caratteri `-)`; il frammento di testo che segue il comando `\item` deve cominciare in corrispondenza con la colonna $(liv * TAB) + 1$.

- L'ambiente `quotation` viene utilizzato per le citazioni. Ogni volta che viene incontrato tale ambiente, vengono inserite due andate a capo nel testo prodotto; inoltre, il testo contenuto in tale ambiente viene centrato introducendo un margine bianco sinistro e uno destro, ognuno dei quali deve essere largo $2 * TAB$. Il testo viene interpretato come nell'ambiente `document` a parte il fatto che esiste sia un margine sinistro che uno destro (la lunghezza di una linea è sempre di *MAX* caratteri, ma in questo caso sono inclusi nei *MAX* caratteri anche gli spazi bianchi dei due margini).

  L'unico comando valido nell'ambiente `quotation` è `\quote`: esso produce un'andata a capo, immediatamente seguita da $6 * TAB$ spazi bianchi.

- L'ambiente `plain` permette di lasciare inalterato il testo in esso contenuto così come appare, ossia non viene effettuata alcuna elaborazione. Come per `itemize` e `quotation`, ogni volta che viene incontrato l'ambiente `plain`, vengono inserite due andate a capo nel testo prodotto. Chiaramente tale ambiente può produrre linee di lunghezza arbitraria. In tale ambiente non è valido alcun comando.

- L'ambiente `comment` serve per introdurre commenti nel programma. Qualsiasi stringa contenuta in esso viene semplicemente ignorata e (quindi) non viene prodotta alcuna stringa in uscita. Per definizione, tale ambiente non può contenere dei sotto-ambienti.

Errori statici

- Ogni comando può essere utilizzato correttamente solo in certi ambienti; ad esempio, `\item` non è valido fuori da un ambiente `itemize`.
- Il livello massimo per gli ambienti `itemize` è *LEV*, dove *LEV* è una costante opportunamente definita.

Errori dinamici In alcuni ambienti (ad esempio `itemize`), a seconda della definizone di *TAB*, è possibile che i due margini bianchi sinistro e destro si sovrappongano occupando così tutta la linea fino alla posizione *MAX* e oltre (in modo che la lunghezza disponibile per il testo sia nulla). In questi casi deve essere sollevato un errore a tempo di esecuzione.

Problema Scrivere un interprete Java per il linguaggio ToyTeX, cioè un programma (collezione di classi e interfacce) Java che, preso in input (da file) un programma ToyTeX, produca in output (su file) il testo prodotto dall'esecuzione di tale programma (se non ci sono errori).
L'esercitazione comprende:

1. Stesura della specifica del problema (comprende la descrizione formale della semantica -statica e dinamica- del linguaggio ToyTeX.
2. Stesura di una collezione di classi Java che implementino la precedente specifica.

Guida metodologica alla stesura NB: si raccomanda di seguire **accuratamente** le istruzioni qui sotto. Programmi che risolvono il problema ma si discostano dall'approccio descritto saranno considerati insoddisfacenti.
I moduli (classi o interfacce) del programma devono corrispondere alla specifica in 1; in particolare dovranno essere presenti almeno i seguenti moduli:

- un modulo per ogni categoria sintattica, che fornisca l'implementazione dell'insieme di oggetti sintattici (termini sulla sintassi astratta, quindi indipendenti dalla sintassi concreta del linguaggio) corrispondenti: ad esempio, una classe per ogni tipo di ambiente;
- un modulo di parsing.

Nello strutturare il programma, si ricordi che in un linguaggio o.o. i tipi unione vanno implementati con interfacce o classi astratte in modo da poter utilizzare, ove possibile, il binding dinamico: sempre allo stesso scopo sarà opportuno distribuire nella varie classi il codice corrispondente alla fase di interpretazione e di scrittura (opportuna per il testing).
Il metodo main della classe principale dovrà prendere come parametro il nome del file contenente il programma in input; una volta letto tale programma, dovrà prima verificarne la correttezza sintattica (fase di parsing), poi quella statica (da definire), infine simularne l'esecuzione, producendo al termine su video lo stato finale.
Si richiede la gestione tramite eccezioni delle seguenti situazioni di errore:

- errori di compilazione (il programma in input non è corretto rispetto alla BNF data);
- errori statici (come definiti precedentemente);
- errori dinamici (come definiti precedentemente).

Si raccomanda di definire un'opportuna gerarchia di eccezioni secondo i principi metodologici visti a lezione. Non si lascino eccezioni non gestite (cioè, il `main` non deve sollevare eccezioni). L'utilizzo, quando possibile, di classi predefinite delle librerie Java è consigliato ed apprezzato.
Si raccomanda l'uso di costanti simboliche e del modificatore `private` per i campi quando possibile.
Per quel che riguarda la fase di parsing, si noti che la sintassi è tale che è in genere sufficiente l'esame del primo simbolo letto per accertare quale è la produzione da applicare; si raccomanda di applicare le tecniche di parsing "diretto dalla grammatica" viste a lezione, riassunte qui sotto. Per ogni non terminale A, andrà previsto un metodo `parseA` che, esaminando via via una sequenza di simboli del linguaggio (prodotti dallo scanner, per il quale si suggerisce di adattare classi predefinite Java), restituisce un oggetto di tipo `A` se la sequenza corrisponde alla parte destra di una produzione con parte sinistra A. Questo metodo per prima cosa deciderà, guardando il primo simbolo, qual'è la produzione applicabile: sia $A ::= u_0 B_1 u_1 \dots u_{n-1} B_n u_n$, con $B_1, \dots, B_n$ non terminali e $u_0, \dots, u_n$ stringhe di terminali. Allora il codice successivo seguirà lo schema:

- controllo che ci sia u_0; se sì
- richiamo il metodo `parseB1`; se questo termina con successo (quindi mi restituisce un oggetto o_1 di tipo `B1`)
- controllo che ci sia u_1; se sì
- ...
- controllo che ci sia u_{n-1}; se sì
- richiamo il metodo `parseBn`; se questo termina con successo (quindi mi restituisce un oggetto o_n di tipo `Bn`)
- controllo che ci sia u_n; se sì
- restituisco l'oggetto ottenuto componendo $o_1, \dots, o_n$.

Si noti che, applicando questo schema, quando viene invocato il metodo `parseA` il simbolo corrente è sempre il primo simbolo "utile" (cioè il primo simbolo della stringa di tipo A da riconoscere), e quando termina l'esecuzione di `parseA` il simbolo corrente è sempre il primo successivo all'ultimo simbolo "utile" (quindi il successivo `parseB` invocato si trova in situazione analoga).
Per il testing si raccomanda di usare un approccio modulare ossia di testare separatamente i vari moduli (ad esempio, verificare a parte che il modulo di parsing legga correttamente un programma).

Traccia di specifica formale

```
class Program {
     void check () throws StaticErrorException{}
     //POST: il programma e' staticamente corretto (da precisare)

     //PRE: il programma e' staticamente corretto
     Text run () throws DynamicErrorException {}
     // POST: RES = testo ottenuto eseguendo il programma
}
```