Progetto Informatica Teorica 2005/06 Scanner e Parser

Dott. Claudio Guidi

Progetto Scanner e Parser

- Obbligatorio per superare l'esame
- Sviluppato in linguaggio Java
  - Esistono diversi tool in laboratorio
    - Usate preferibilmente NetBeans
  - In gruppi formati da 2/3 persone
- Quando si consegna?
  - Consegna libera durante la sessione. La scadenza per ciascuna sessione verrà esposta sulla homepage (20/2/06): http://www.cs.unibo.it/~cguidi

Modalità di consegna

- Per mail:
  - all'indirizzo cguidi@cs.unibo.it con oggetto "IT – consegna progetto"
  - Il testo deve contenere la descrizione del gruppo (nome, cognome e matricola degli elementi del gruppo)
  - In allegato il file .zip (altri formati non verranno accettati) del progetto con:
    - Documentazione (file doc.html)
    - Sorgenti progetto
    - File di esempio

Documentazione

- Deve contenere 3 sezioni ben distinte
  - l'elenco dei componenti del gruppo che hanno partecipato al progetto, con indirizzo di posta elettronica (del corso di laurea)
  - Una breve descrizione dell'architettura del vostro progetto, identificando i gruppi di classi che formano moduli logici ben definiti
    - Espressioni regolari implementate
    - Eventuali modifiche apportate alla grammatica qualora sia necessario disambiguare.
    - Tipologia di parser utilizzata.
  - Note sulla vostra implementazione (tecniche particolari utilizzate, problematiche incontrate nell'uso del paradigma object-oriented e/o del linguaggio Java, ecc.).

Il Progetto

- Consiste nell'implementare, in Java, di uno scanner ed un parser per il linguaggio funzionale MiniScheme che è un sottolinguaggio derivato da Scheme
- Il progetto è pensato per essere complementare a quello sviluppato nel corso di Linguaggi di Programmazione

Valutazione del progetto

- Valutato in laboratorio con tutti gli elementi del gruppo
- Si basa sull'effettivo funzionamento di scanner e parser e sulla loro struttura, verranno anche considerate
  - la chiarezza del codice
  - la corretta applicazione dei meccanismi fondamentali dei linguaggi object-oriented (incapsulamento, polimorfismo ed ereditarietà)
  - l'uso di classi della libreria di Java

Il linguaggio MiniScheme

Non terminali

$$
\begin{array}{lcll}
\langle program \rangle & ::= & \langle define \rangle_1 \ \cdots \ \langle define \rangle_n & \\
 & & n \geq 1 & \\
\langle define \rangle & ::= & \texttt{(define } id \ \langle expr \rangle \texttt{)} & \\
 & | & \texttt{(define (} id_1 \ \cdots \ id_n \texttt{)} \ \langle expr \rangle \texttt{)} & \\
 & & n \geq 1 & \\
\langle const \rangle & ::= & \texttt{\#t} \mid \texttt{\#f} & \\
 & | & int & \\
 & | & string &
\end{array}
$$

Valori booleani true e false

Il linguaggio MiniScheme (2)

$$
\begin{array}{lcll}
\langle expr \rangle & ::= & \langle const \rangle & \\
 & | & id & \\
 & | & \texttt{(and } \langle expr \rangle_1 \ \cdots \ \langle expr \rangle_n \texttt{)} & n \geq 0 \\
 & | & \texttt{(or } \langle expr \rangle_1 \ \cdots \ \langle expr \rangle_n \texttt{)} & n \geq 0 \\
 & | & \texttt{(cond } \langle branch \rangle_1 \ \cdots \ \langle branch \rangle_n \texttt{)} & n \geq 1 \\
 & | & \texttt{(cond } \langle branch \rangle_1 \ \cdots \ \langle branch \rangle_n \texttt{ (else } \langle expr \rangle \texttt{))} & n \geq 0 \\
 & | & \texttt{(local (} \langle define \rangle_1 \ \cdots \ \langle define \rangle_n \texttt{)} \ \langle expr \rangle \texttt{)} & n \geq 1 \\
 & | & \texttt{(lambda (} id_1 \ \cdots \ id_n \texttt{)} \ \langle expr \rangle \texttt{)} & n \geq 0 \\
 & | & \texttt{(} \langle expr \rangle_1 \ \cdots \ \langle expr \rangle_n \texttt{)} & n \geq 1 \\
\langle branch \rangle & ::= & \texttt{(} \langle expr \rangle \ \langle expr \rangle \texttt{)} &
\end{array}
$$

Il linguaggio MiniScheme (3)

- Il programma è costituito da una serie di definizioni.
- Tramite le definizioni si associano a degli identificativi delle espressioni che possono sia essere costanti che espressioni più complesse:
  - and, or, condizioni (cond), funzioni (lambda), scope annidati (local)

Il linguaggio MiniScheme (4)

- Altre funzioni base quali il +, *, ecc. si assume siano già definite all'interno dell'ambiente globale. Non vanno quindi considerati come token ma come identificativi a tutti gli effetti.
- Es. (+ 2 3) è la funzione con identificativo + con argomenti 2 e 3.

Esempio

- Il programma che segue definisce l'algoritmo di inversione degli elementi di una lista

```
(define (reverse l)
      (local ((define (aux l1 l2)
                (cond ((null? l2) l1)
                (else (aux (cons (car l2) l1)
                           (cdr l2))))))
(aux (list) l)))

(define (main args)
(reverse (list 1 2 3 4 5 6 7 8 9 10)))
```

- Output: (10 9 8 7 6 5 4 3 2 1)

È una funzione che testa la fine lista

Seconda componente di una coppia (coda della lista)

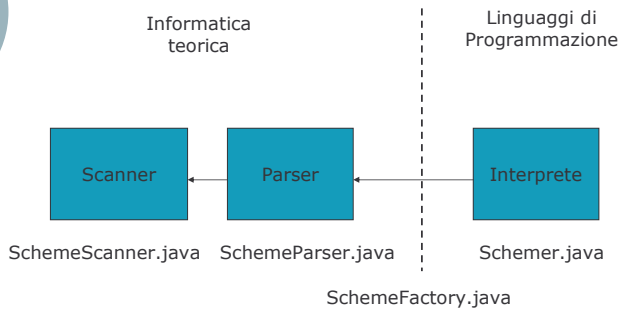
Prima componente di una coppia (testa)

Crea una coppia

Lista vuota

List crea una lista

Schema di progetto MiniScheme



Schemer

- Questo componente rappresenta l'interprete che chiama il parser chiedendo di volta in volta una definizione per interpretarla ed eseguirla
- Noi considereremo una versione che effettua solo la chiamata al parser
- Non esattamente complementare al progetto di Linguaggi di programmazione
  - Necessarie piccole modifiche

Scheletro interprete

```
public class Schemer {
...

    public static void parse(String source, InputStream stream,
                             SchemeEnvironment env) throws java.io.IOException,
    SchemeException {
        final SchemeScanner scanner = new SchemeScannerImpl(source, stream);
        final SchemeParser parser = new SchemeParserImpl(scanner,
                                        new SchemeFactoryImpl());
        while (scanner.getToken() != SchemeScanner.EOF) {
                    final SchemeDefinition def = parser.parseDefine();
                    def.declare(env);
                    def.define(env);
            }
    }

    public static void main(String[] args) throws java.io.IOException,
                    SchemeException {
    ...
    }
}
```

Scanner

- Lo scanner può essere implementato "a mano" oppure facendo uso del tool Jlex:
  - http://www.cs.princeton.edu/~appel/modern/java/JLex/
- Definire le espressioni regolari per i token
- Generazione dello scanner secondo le specifiche di progetto. Libertà di aggiungere metodi per gestire la comunicazione tra parser e scanner

Interfaccia Scanner

```
public interface SchemeScanner {
   static public final int EOF = 0;                  \\ fine file
   static public final int BOOL = 1;                 \\ booleano

   static public final int INT = 2;                  \\ intero

   static public final int STRING = 3;               \\ stringa di caratteri, (libertà
                                                     di scelta per l'espressione
                                                     regolare, racchiusa tra
                                                     virgolette)
   static public final int OPEN = 10;                \\ parentesi aperta

   static public final int CLOSE = 11;               \\ parentesi chiusa

   static public final int ID = 12;                  \\ identificativo

   public int getToken();        ...                 \\ unico metodo da rispettare
}                                                    (restituisce il token corrente
                                                     senza spostare la testina dello
                                                     scanner), per il resto libertà di
                                                     aggiungerne
```

Costruttore dello scanner:

- Per compatibilità verso il progetto di Linguaggi di Programmazione l'implementazione dello scanner ed il suo costruttore devono essere definiti nel seguente modo:

```
class SchemeScannerImpl implements SchemeScanner {
   …
   public SchemeScannerImpl(String n, InputStream s)
   throws java.io.IOException {…}
```

Source Name

Parser

- Il parser non può essere generato automaticamente
  - Analisi della grammatica e modifica della stessa qualora presenti delle ambiguità
  - Generazione del parser secondo le specifiche di progetto

Interfaccia Parser

- Il parser restituisce, quando invocato dall'interprete, una definizione per volta.
- Interfaccia:

```
public interface SchemeParser {
   public SchemeDefinition parseDefine() throws
   java.io.IOException;
}
```

- Come si vede c'è un unico metodo che restituisce un oggetto SchemeDefinition

Costruttore Parser

- L'implementazione ed il costruttore del parser, per compatibilità verso il progetto di Linguaggi di programmazione devono seguire le seguenti specifiche:

```
class SchemeParserImpl implements SchemeParser {
   …
   public SchemeParserImpl(SchemeScanner s,
   SchemeFactory f) {…}
```

Costruttore Parser

- L'implementazione ed il costruttore del parser, per compatibilità verso il progetto di Linguaggi di programmazione devono seguire le seguenti specifiche:

Scanner

```
class SchemeParserImpl implements SchemeParser {
   …
   public SchemeParserImpl(SchemeScanner s,
   SchemeFactory f) {…}
```

Oggetto necessario alla definizione dell'albero sintattico

SchemeFactory

- SchemeFactory consente al parser di rilasciare una struttura di oggetti legati alle produzioni della grammatica.
- L'oggetto SchemeDefinition rilasciato dal parser contiene una definizione sintatticamente corretta.

L'interfaccia di SchemeFactory

```
interface SchemeFactory {
public SchemeDefinition createDefinition(String name, SchemeExpression expr);
public SchemeBranch createBranch(SchemeExpression test, SchemeExpression e);
public SchemeExpression createApplyExpression(List exprs);
public SchemeExpression createLambdaExpression(List params, SchemeExpression
  expr);
public SchemeExpression createAndExpression(List exprs);
public SchemeExpression createOrExpression(List exprs);
public SchemeExpression createCondExpression(List branches, SchemeExpression e);
public SchemeExpression createLocalExpression(List bindings, SchemeExpression e);
public SchemeExpression createIdExpression(String id);
public SchemeExpression createBoolExpression(boolean v);
public SchemeExpression createIntExpression(int v);
public SchemeExpression createStringExpression(String v);
public SchemeExpression createSymbolExpression(String v);   // ignorare
}
```

Legame tra SchemeFactory e la grammatica

- C'è un legame preciso tra gli oggetti restituiti dai metodi di factory ed i simboli della grammatica
- schemeDefinition è legato alla produzione <define>
- schemeBranch è legato alla produzione <branch>
- schemeExpression è legato alle produzioni relative a <expr>, in particolare la produzione: <expr>->(<expr>…<expr>) è legata al metodo createApplyExpression

Da notare…

- La definizione
  (define (f x1 · · · xn) E)
  è un'abbreviazione di
  (define f (lambda (x1 · · · xn) E)
- Quindi la factory prevede solo quest'ultimo tipo di definizione
  - il parser deve provvedere ad espandere adeguatamente ogni definizione del primo tipo in una del secondo tipo con un opportuno lambda.

Esempio

- Il parser restituisce all'interprete una definizione con il seguente codice:
  - return factory.createDefinition(…));
  - dove factory è l'oggetto di tipo SchemeFactory passato al costruttore del parser.

Gestione errori

- Sfruttare la classe:

```
class SchemeSyntaxError extends Error {
public SchemeSyntaxError(String sourceName, int line,
String msg) {
  ...}}
```

- L'errore sintattico deve riportare il nome della sorgente, la linea in cui si è verificato l'errore, ed il messaggio di errore vero e proprio.
- Valutare l'opportunità di effettuare Error Recovery limitatamente ad ogni singola definizione.

Implementazione per il test

- Ci allontaniamo dalla compatibilità con il progetto di Linguaggi al fine di poter gestire il test dell'elaborato.
- Implementare le classi SchemeFactoryImpl, SchemeDefinition, SchemeBranch e SchemeExpression al fine di poter visualizzare l'albero di parsing di ciascuna definizione.
- Libertà di modificare Schemer.java e di ridirezionare l'output e l'input a piacere.