LPO Riassunto per Esame

Tokenizer & Parser

Il tokenizer, “spezzo” una stringa (stmt) in token da passare successivamente al parser che creerà l’albero della sintassi astratta

Example:

```
int x = 1;
```

A lexer or tokeniser will split that up into tokens 'int', 'x', '=', '1', ';'.

A parser will take those tokens and use them to understand in some way:

- we have a statement
- it's a definition of an integer
- the integer is called 'x'
- 'x' should be initialised with the value 1

Appunti

Elementi caratterizzanti della programmazione object oriented sono:

- Astrazione: realizzabile usando le classi, inheritance e polimorfismo
- Encapsulation: realizzabile usando le classi, inheritance e polimorfismo
- Gerarchia: realizzabile usando le classi, inheritance e polimorfismo
- Modularità: serve qualcosa che permetta di raggruppare più classi in un medesimo dominio e di partizionare un problema. Questi sono i package: uno strumento usato per raggruppare tipi in qualche modo legati fra di loro. Servono a classificare ed organizzare le classi .

Semantica Statica: si intende il suo sistema dei tipi, ovvero l'insieme delle regole che consentono di dare un tipo ad espressioni, comandi ed altri costrutti. Prende in input un programma e se coretto, prova a generare l’albero della sintassi astratta. Se anche questo va a buon fine si passa al controllo delle semantica dinamica.

Controllo dei tipi, quindi si implementa *typecheck*. Le regole di tipo si implementano sfruttando paradigma funzionale (che è un po’ più astratto). La specifica della sematica è in **ocaml** (file. ml).

Gestisce la nozione di scope, prozione di programma in cui una certa dichiarazione di variabile è valida.

Praticamente si fa una visita dell’albero della sintassi astratta, per fare questo si sfutta il design pattern **visitor pattern**, conveniente a visitare alberi. Questo ci permette di riutilizare codice in modo proficuo perchè anche l’implementazione della semantica dinamica corrisponde ad effettuare una visita dell’albero sintattico astratto (AST), ovviamentecorrisponde ad effettuare una visita diversa a scopo di valutare espressioni, ottenere valori, eseguire stmt modificando l’almbiente dinamico delle variabili. Nell’ambiente statico le variabili sono associate al loro tipo statico, in quello dinamico al tipo corrente

TypeCheck: la classe typecheck implementa l'interfaccia `Visitor` che è generica nel tipo che corrisponde a che valore è restituito dalla visita stessa. Sono gli oggetti che permettono di visitare l'albero per controllare regole di sematica statica. I metodi di visita (`Visit...()`) hanno parametri che corrispondo alle sottocomponenti del nodo corrispondente eccezione per la visita della foglia `varIdent` che identifica una variabile che non ha figli e prende come parametri l'identificatore stesso che è chiave dell' hash map che implementa l'ambiente statico.

> Le istanze di TypeCheck sono "visitor"

C'è una visita per ogni nodo e se restituisce qualcosa è Type (un tipo vero e proprio che viene restituito solo quando vengono analizzate delle espressioni, perchè sono associate a valori poi ottenuti a runtime).

ENV: è l'ambiente utilizzato dal visitor per verificare se le variabili sono dichiarate e che tipo hanno. Poi sarà gestito sia in lettura che scrittura. Esempio:

```
env.dec(ident, exp.Accept(this))
```

Accept(): a seconda del tipo di "this" va a selezionare (richiamare) la "visitor" corretta per il tipo.

Il controllo si scarica sulle primitive , si agisce sull'ambiente env e tramite il metodo `dec()` permette di aggiungere al livello corrente di annidamento la dichiarazione di ident con corrispondente tipo che viene dedotto automaticamente dal tipo dell'espressione. Il risulato è il tipo dell'espressione che si va ad associare all'identificatore dentro l'ambiente env.

> il metodo accept(this) viene chiamato ricorsivamente su exp.
>
> Per esempio se 'exp' avesse come nodo radice l'addizione, andrebbe a chiamare sul visitor this il "visitAdd"

> chiamata di metodo restituisce metodo di tipo "type"

Semantica Dinamica: assegna un significato ai programmi, in base al loro comportamento durante l'esecuzione. Quindi sono vincoli determinabili solo a runtime

Token: forniscono un' astrazione della sintassi concreta dei lessemi (**unità sintattiche primitive**) con la nozione di token type.

Tokenaizer: un **lexer** che ricostruisce i lessemi e genera i token corrisponenti.

Parser: parte dedicata all'analisi sintattica. Riconosce le sequenze di token generate da un tokenaizer. (Le regole sintattiche vengono formalmente definite attraverso una grammatica [i file ocaml (?)]). Il parser lavora con ricorsione topdown ossia parte dalla radice verso le foglie.
Questo genera un parser tree, ossia una rappresentazione concreta del programma (ALBERELLIIII).

Quando vogliamo scrivere un parser, possiamo estendere la classe `Parser` che implementa un parser vero e proprio con i vari metodi associati alle varie categorie sintattiche definiti **TokenType**.

Usando `var` il compilatore deduce direttamente il tipo dello *Stmt* generale rappresentato da un' interfaccia e, per vedere se c'è parte opzionale o meno, viene chiesto di che tipo sono il prossimo tokentype e il prossimo simbolo. Successivamente con `return new ...` si restituisce il risultato, costruito attraverso un nodo dell' AST.

Se il tipo non fosse corretto viene restuito un ndo di tipo diverso.

`consume():` verifica che il token sia del tipo giusto e in caso affermativo va a leggere il prossimo token.

Il return deve restituire un nodo del tipo corretto definito nel AST dove ci sono tutte le classi che rappresentano i vari tipi di nodi con i costruttori che servono alla classse *BufferParser* per costruire i vari nodi dell'albero. Per esempio: `return new PrintStmt(parseExp())` non c'è bisogno di definire variabili locali in cui memorizzare parseExp() perchè si può passare direttamente il risultato al costruttore. Le varibili locali di supporto servono quando ci sono più dati da fornire al costruttore come argomenti.

ParseExp() : categoria sintattica delle espressioni costruite a partire dall'operatore che ha meno precedenza ovvero operatore binario `AND`. Le chiamate di parsing fanno riferimento alla categoria sintattica sottostante. Esempio:

- `parseExp()`: rappresenta espressioni dove viene considerato come operatore principale l'operatore che segue nella tabella delle precedenze.
- `var exp = parseEq()` : ????
- `var val = buf_tokenizer.mValue()`: chiede al tokenizer, tramite il metodo di query *intValue* qual è il vlore del token riconosciuto. Va al prossimo token e restituisce il nodo foglia e al costruttore viene dato l'attributo sintattico associato al nodo foglia. Le parentesi forzano solo come deve essere formato l'albero.

AST: abstract sytax tree ovvero una rappresentazione più astratta che può essere sia scritta a mano o generata automaticamente da una grammatica definita con strumenti software definiti.

Esempio: se i nodi di un albero hanno 2 figli, avremo una *BinaryOP* (come la moltiplicazione).

Polimorfismo Parametrico: espressioni che possono rappresentare valori di diversi tipi. Quindi può ricevere un tipo come parametro invece che conoscerlo a priori.

CLASSI

Object: sono definiti e creati ttraverso le classi, che forniscono un' implementazione per oggetti dello stesso tipo. Gli oggetti possono essere creati dinamicamente dalle classi.

ACCESS MODIFIER

- **private**: visibile solo nella classe
- **public (method)**: visibile solo all'esterno della clase (????)
- **public (class)**: visibile ovunque nel programma

Instance methods: invoca l'oggetto (this)

Class method: invoca la classe --> non serve oggetto. `static`

Final (class, instance, local var): sola lettura: ha sempre lo stesso valore.

Se in una dichiarazione di una classe viene usato private / public, di default, l'accesso è package, ossia accessibile solo all'interno del package.

Any other objcet type is a subtype of object.

Shallow copy: this.a = l.a; this.b = l.b; Deep copy: this.a = new Point(l.a); this.b = new Point(l.b);

Conversione implicita tra tipi primitivi e object

- **boxing**: da primitivo a object type
- **unboxing**: dai tipi object ai tipi primitivi I tipi object consentono di gestire i valori uniformemente tramite riferimento.

SUPER: serve a chiamare un metodo ereditato da una superclasse

Method overriding

@ override : riscrittura di metodo ereditato. Se viene usato, il compilatore controlla le regole di override

this(..) : invocazione di costrutto della stessa classe.

ABSTRACT CLASS

(es: binaryOp ha metodi astratti ereditati da Exp) Sono definizioni incomplete che devono essere completate da sottoclassi. Come le interfacce, non possono essere usate per creare oggetti.

Differentemente dalle classi concrete possono avere metodi astratti.

Utilità:

- migliorare struttura di codice
- funzionalità condivise dalle sottoclassi per migliorare il riutilizzo. Colmano il divario tra interfacce e classi concrete.

Metodi generici

```
<T> TrequireNonNUll(T obj)
```

è un metodo generico (polimorfismo parametrico).

Errori

```
throw e
```

in semantica statica: ogni eccezione deve essere sottotipo di Throwable (T <= Throwable)

in semantica dinamica: il normale flusso di esecuzione viene interrotto, l'errore è propagato ai chiamanti ed eventualmente gestito, oppure il programma viene interrotto bruscamente con errore. Se l'errore è nullo alora viene alzato un NullPointerException.

Token

Il tokenizer definisce la seguente procedura:

- `nextToken()`: viene letto il prossimo token lookhaed
- `tokenType()`: ritorna il tipo del token corrente lookhaed
- `chekcTokenType(type)`: il tipo del corrente lookhaed è selezionato; è sollevata un' eccezione se il controllo fallisce, e si passa al prossimo token.

Differenza tra estendere ed implementare

- **extends**: serve per estendere una classe ed è fondamentale il concetto di ereditarietà. La nuova classe che si dichiara avrà tutti i metodi della classe padre. L'ereditarietà è singola, cioè non si possono fare 2 extends.
- **implements**: implementa tutti i metodi di un' interfaccia nella classe che stai creando

---

Esempio aggiunta operazione a LAB

1. Aggiungo l' ENUM "DIV" nel file: `Parser/TokenType.java`

   ```
   DIV
   ```

2. Aggiungo il char '/' ai token nel file: `Parser/Bufferedtokenizer.java`

   ```
   symbols.put("/", DIV);
   ```

3. Mi sposto nel file `parser/BufferedParser.java` e aggiungo il metodo `parseDiv`

```
  private Exp parseDiv() throws ParserException {
    var exp = parseAtom();
    while (buf_tokenizer.tokenType() == DIV) {
      nextToken();
      exp = new Div(exp, parseAtom());
    }
    return exp;
  }
```

4. dentro alla cartella `Parser/ast` si crea una classe (che può estendere classi come BinaryOp o UnaryOp) e si fa l'override della funzione `accept(this)` che avrà il compito di chiamare il `visitor()` corretto per il tipo di dato

```
package ProgettoFinale_G52.parser.ast;
import ProgettoFinale_G52.visitors.Visitor;

public class Div extends BinaryOp {
  public Div(Exp left, Exp right) {
   // if(accept(right) == 0)
  //    throw new Excpetion();
    super(left, right);
  }

  @Override
  public <T> T accept(Visitor<T> visitor) {
    return visitor.visitDiv(left, right);
  }
}
```

2. Nella cartella `Visitors` aggiungo all'interfaccia `Visitor` la nuova `visitDiv()`

```
package ProgettoFinale_G52.visitors;
public interface Visitor<T> {
  T visitAdd(Exp left, Exp right);
  /* ...... */
  T visitDiv(Exp left, Exp right);
}
```

3. Andiamo nella cartella `visitors/evaluation` e nella classe `Eval.java` aggiungiamo l'implementazione del metodo `visitDiv(Exp left, Exp right)`:

```
package ProgettoFinale_G52.visitors.evaluation;
public class Eval implements Visitor<Value> {
```

```
  private final GenEnvironment<Value> env = new GenEnvironment<>();
  private final PrintWriter printWriter; // output stream used to print values
  public Eval() { printWriter = new PrintWriter(System.out, true);}
  public Eval(PrintWriter printWriter) {this.printWriter =
requireNonNull(printWriter);}

  // dynamic semantics for programs; no value returned by the visitor
  @Override
  public Value visitProg(StmtSeq stmtSeq) {
    try {stmtSeq.accept(this);}
    catch (EnvironmentException e) throw new EvaluatorException(e);
    return null;
  }

  // dynamic semantics for statements; no value returned by the visitor
  /* .......... */
  @Override
  public IntValue visitDiv(Exp left, Exp right) {
    return new IntValue(left.accept(this).toInt() / right.accept(this).toInt());
  }
```

4. Ci spostiamo in `visitors/typechecking/TypeCheck.java` e anche qui andiamo a ridefinire la nuova `visitDiv`:

```
package ProgettoFinale_G52.visitors.typechecking;
public class TypeCheck implements Visitor<Type> {
  private final GenEnvironment<Type> env = new GenEnvironment<>();

  // useful to typecheck binary operations where operands must have the same type
  private void checkBinOp(Exp left, Exp right, Type type) {
    type.checkEqual(left.accept(this));
    type.checkEqual(right.accept(this));
  }

  // static semantics for programs; no value returned by the visitor
  @Override
  public PrimType visitDiv(Exp left, Exp right) {
    checkBinOp(left, right, INT);
    return INT;
  }
```

Cosa c'è nelle carte e nei files? il codiceee yaaay

Progetto_Finale_G52:

- Environments - Package:
  - *Environment* - Interfaccia: contiene intestazione di funzioni per creare e modificare l'albero sintattico (`dec(id,payload)`: crea una nuova foglia, `update(id, payload)`: aggiorna la foglia con id=id, `lookup(id)`: cerca il valore associato all' id)
  - *GenEnvironment* - Classe: dichiara il corpo delle funzioni introdotte nell'interfaccia
  - *EnvironmentException* - ExceptClass : una classe con eccezzioni riguaranti l'environment. User defined.
- Parser - Package:
  - AST:
    - *AST* - interfaccia: intestazione della sola funzione `T accept(Visitor v)`
      - Altre interfacce: *Exp, Prog, Stmt, StmSeq, VarIdent (`String getName`)*
    - 5x Classi Astratte: *Single, PrimLiteral, More, BinaryOp, AbstractAssignStmt, UnaryOp* –> definiscono i costruttori + ovveride di `toString()`
    - 24x Classi: troviamo tutte le classi relative ai nodi dell'AST con le relative override di `accept(Visitor v)`
  - *Parser* - Interfaccia: dichiara un metodo `parseProg()`
  - *Tokenizer* - Interfaccia: dichiara metodi utili a riconoscere il tipo di token (`tokenType()`) o a passare al prossimo token (`next()`) oltre che altre funzioni d'appoggio
  - *TokenType* - Enum: enumerazione di simboli, keywords, categories e EndOfFile per permettere al tokenizer di riconoscerli
  - *BufferedParser* - Classe: oltre a definire il metodo dell'interfaccia Parser, crea metodi parser per ogni token type
  - *BufferedTokenizer* - Classe: contiene 2 hashmap per l'AST una con keyword e valore, l'altra con symbols e valore. Inoltre definisce funzioni per: controlare se esiste un next token (`hasNext()`), assegnare un gruppo al token (`assignTokenType`), passare al prossimo token (`next`)
  - 2x Exception file uno per token, uno per parser
- Visitors - Package:
  - EVALUATION:
    - *Value* - Interfaccia: dichiarazione di metodi di conversione per tipi (`default PairValue toProd(){}; default int toInt(){};`)
    - *PrimValue* - Classe Astratta: inizializza tipi primitivi (int e bool)
    - *BoolValue & IntValue* - Classi: creano oggetti di tipo int e bool
    - *PairValue* - Classe: crea oggetti di tipo "coppia" (esempio: << 1, "arg">>)
    - *RangeValue* - Classe: crea oggetti di tipo range (esempio: [1:7])
    - ***Eval* - Classe:** sono definiti i metodi per visitare tutto il programma, dalla `visitProg()` che accetta una sequenza di statement, alla `visitNot()` che ritorna il valore booleano ricevuto in input, negandolo
  - TYPECHECKING:

        - *Type* - Interfaccia: contiene dichiarazioni di funzioni attue a controllare che il tipo di token sia quello giusto e funzioni getter che servono a prendere i valori dei tipi (es: `getFstRangeType(), checkIsProdType()...`)
        - *TypeCheck* - Classe: implementa tutte le `Visit...()` (le visit..() chiamano accept() al loro interno)
        - *ProdType* - Classe: definisce un tipo per il prodotto (come operazione)
        - *RangeType* - Classe: definisce un tipo per il range (for in...)
        - *PrimType* - Enum: enum dei tipi primitivi INT e BOOL
        - *TypeChekcerException* - Excpetion Class: definisce eccezioni per typecheck
    - *Visitor* - Interfaccia: dichiarazione dei metodi per visitare i vari token