Corso: Fondamenti, Linguaggi e Traduttori

Paola Giannini

Costruzione AST

Indice

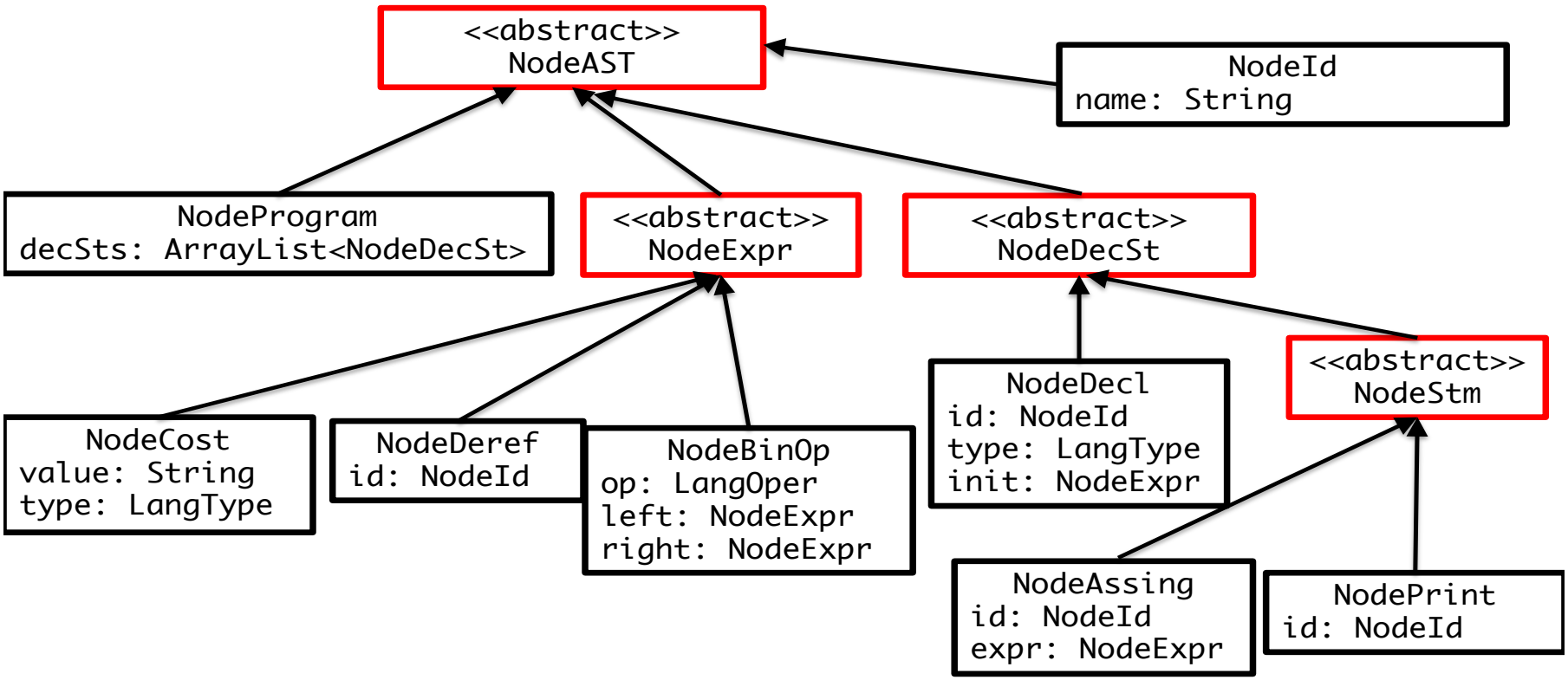
Aggiungiamo 1 packages al progetto

- ast che conterrà le classi per abstract syntax tree

Definiamo i nodi del AST per il linguaggio

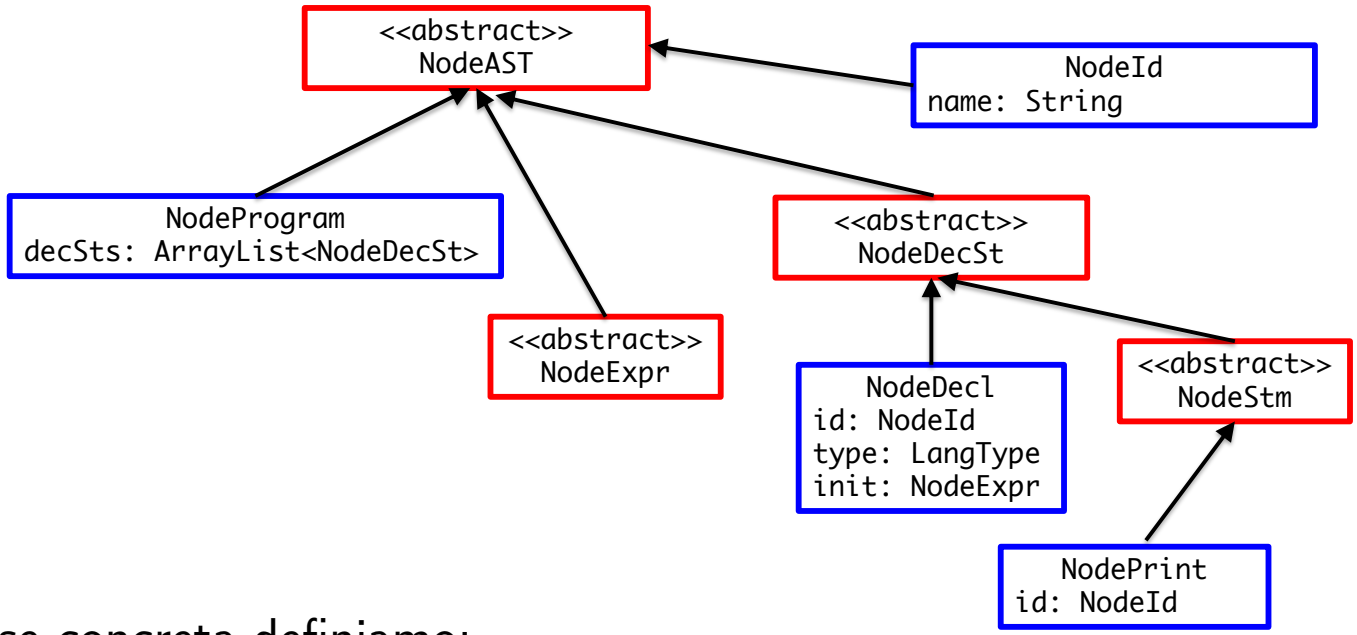
- Definiamo i Nodi da rappresentare attraverso una gerarchia di classi (di cui alcune astratte)
- Definiamo il tipo enumerato `LangOper` che usiamo per rappresentare gli operatori nel nodo delle espressioni binarie (alternativamente si possono avere 4 sottoclassi una per ogni operatore!)
- Definiamo il tipo enumerato `LangType` che usiamo per rappresentare i tipi delle variabili nel nodo che rappresenta le dichiarazioni

Gerarchia di classi dei nodi



Indice

Inizialmente definiamo SOLO questi nodi



Per ogni classe concreta definiamo:

- costruttori opportuni
- `toString()`
- getters

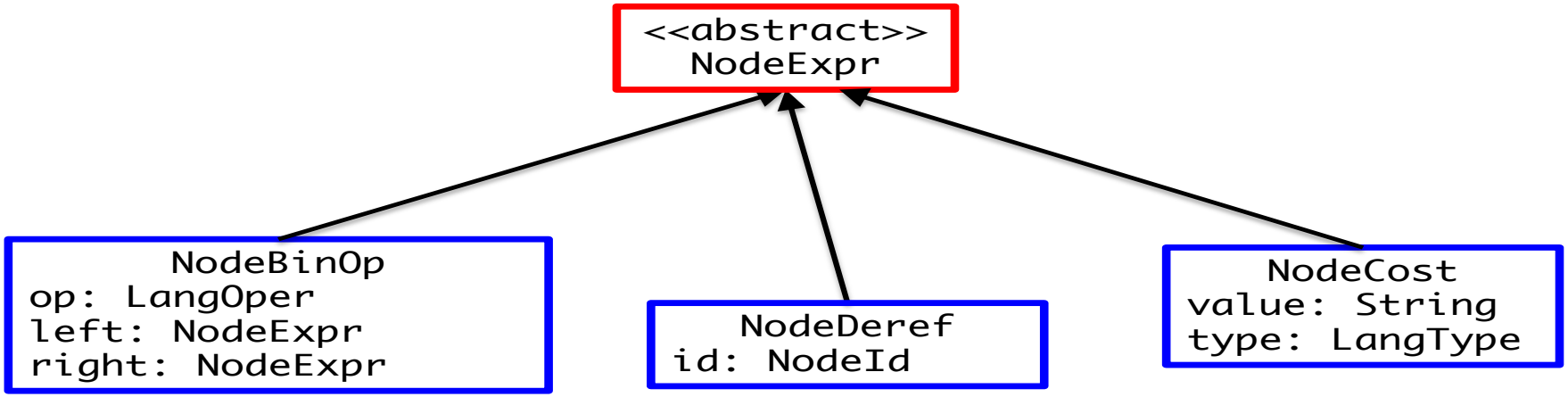
Costruzione del AST per programma, dichiarazioni (senza inizializzazione) e istruzione `print`

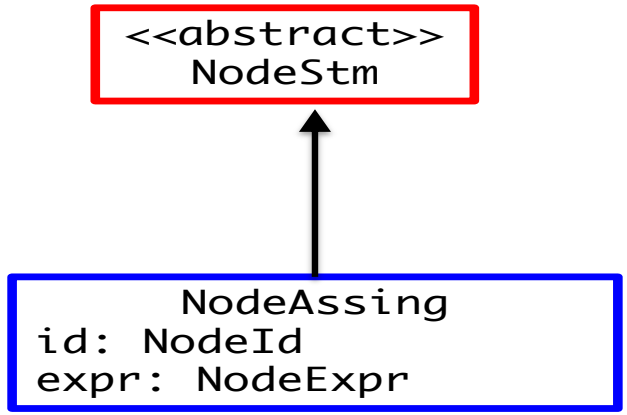
Incorporate la costruzione del AST nel parser:

1. i metodi `parse` e `parseProgram` ritornano un nodo di tipo `NodeProgram` e (`parseProgram` fa la costruzione del nodo)
2. il metodo `parseDSs` ritorna un `ArrayList<NodeDecSt>`
3. il metodo `parseDcl` fa la costruzione e ritorna `NodeDecl`
4. il metodo `parseStm` ritorna `NodeStm` e per il momento ritorna un `NodePrint` in caso la produzione usata è `Stm` $\rightarrow$ `print id` e `null` se la produzione è un assegnamento.
5. i metodi `parseExp`, `parseExpP`, `parseTr`, `parseTrP`, `parseVal`, ritornano `NodeExp` e per il momento ritornano `null`.
6. Testare la costruzione partendo da un file che contiene solo dichiarazioni senza inizializzazione e istruzioni `print`, confrontando il risultato di `parse` con il `toString` dell'albero restituito.

Indice

Aggiungiamo le classi concrete per i nodi `NodeExpr` e la classe `NodeAssign`





Definiamo:

- costruttori opportuni
- `toString()`
- getters

Produzioni per espressioni (termini e valori)

$$
\begin{aligned}
&Exp \rightarrow Tr\ \ ExpP \\
&ExpP \rightarrow +Tr\ \ ExpP \\
&ExpP \rightarrow \text{-}Tr\ \ ExpP \\
&ExpP \rightarrow \epsilon
\end{aligned}
$$

$$
\begin{aligned}
&Tr \rightarrow Val\ \ TrP \\
&TrP \rightarrow /Val\ \ TrP \\
&TrP \rightarrow *Val\ \ TrP \\
&TrP \rightarrow \epsilon
\end{aligned}
$$

$$
\begin{aligned}
&Val \rightarrow \text{intVal} \\
&Val \rightarrow \text{floatVal} \\
&Val \rightarrow \text{id}
\end{aligned}
$$

Come costruire i nodi per i non terminali *ExpP* e *TrP*

- Notate che le produzioni per *ExpP* e *TrP* iniziano con un operatore binario, quindi i metodi `parseEprP` e `parseTrP` dovranno produrre un oggetto di tipo `NodeBinOp`.
- Ma da dove arriva l'operando a sinistra dell'operatore?
- Notate che in una derivazione prima di avere il non terminale *ExpP* dobbiamo aver usato la produzione

$$Exp \rightarrow Tr\ ExpP$$

  (stessa cosa per *TrP*) quindi qua l'operando arriva dal `nodeExpr` restituito da `parseExp` e
- Quindi i metodi per *ExpP* e *TrP* avranno anche un input di tipo `NodeExpr`.
- Vediamo i prototipi dei metodi che costruiscono espressioni.

Prototipo dei metodi di parsing dei non terminali per le espressioni

```
private NodeExpr parseExp() throws ..... {
}

private NodeExpr parseExpP(NodeExpr left) throws ..... {
}


private NodeExpr parseTr() throws ..... {
}

private NodeExpr parseTrP(NodeExpr left) throws ..... {
}


private NodeExpr parseVal() throws ..... {
}
```

I metodi per i nonterminali *Exp* e *ExpP* (quelli per *Tr* e *TrP* sono simili)

```
private NodeExpr parseExp() throws SyntaxException {
  Token tk;
  try {
    tk = this.scanner.peekToken();
  } catch (LexicalException e) throw new SyntaxException("Lexical Exception", e);
  switch (tk.getTipo()) {
    case INT, FLOAT, ID:
      NodeExpr left = parseTr();
      NodeExpr exp1 = parseExpP(left);
      return ??????;
    default:
      throw new SyntaxException("ErroreSintattico: ......");
  } }

private NodeExpr parseExpP(NodeExpr left) throws SyntaxException {
  Token tk;
  try {
    tk = this.scanner.peekToken();
  } catch (LexicalException e) throw new SyntaxException("Lexical Exception", e);
  switch (tk.getTipo()) {
    case PLUS:
      match(TokenType.PLUS);
      NodeExpr exp1 = parseTr();
      NodeExpr exp2 = parseExpP(?????);
      return ?????;
    case MINUS:
        .......
    case SEMI:
      return ?????;
    default:
      throw new SyntaxException("ErroreSintattico: ......");
  } }
```