Python Biella Group

Serata 3

Obiettivi

- Creare un interprete di un linguaggio: partiamo dai condizionali
- Proviamo a scrivere un nuovo linguaggio PBGLang
- Uso “avanzato”: un esempio da Internet

Creare un interprete - Condizioni

Grammatica per Condizioni

Vogliamo un interprete che gestisca un sottoinsieme di un linguaggio di programmazione che gestisca solo *if / else (opzionale) / print*; casi di esempio:

```
if 2 < 3 print 8 else print 5
>5

if 5 < 6 print 2
>2

if 6 < 5 print 2
>
```

Come sempre, partiamo dalla grammatica !

Grammatica per Condizioni

```
grammar Condizioni;

root: action+ EOF;

//('else' action) è opzionale
action: 'if' expr action ('else' action)? # Condition
    | 'print' expr                        # Print
    ;



expr: expr GT expr # Gt
    | expr LT expr # Lt
    | NUM          # Value
    ;

GT: '>';
LT: '<';
NUM: [0-9]+;
WS: [ \t\r\n]+ -> skip;
```

Grammatica per Condizioni

Generiamo helpers e visitor:

```
java org.antlr.v4.Tool -Dlanguage=Python3 -no-listener -visitor Condizioni.g
```

Customizziamo il visitor:

`InterpreteVisitor.py`

Scriviamo l’interprete che legge i comandi da file:

`interprete.py`

Creare il nuovo linguaggio “PBGLang”

Grammatica per PBG-Lang

Scriviamo la grammatica, step by step: `PBGLang.g4`
Iniziamo con la base, l'assegnamento a variabili

```
grammar PBGLang;

program     : statement+;
statement   : let | show ;
let         : VAR '=' INT;
show        : 'mustra' (INT | VAR) ;

VAR         : [a-z]+ ;
INT         : [0-9]+ ;
WS          : [ \n\r\t]+ -> skip ;
```

Esempio:

```
a = 100
mustra a
>100
```

Grammatica per PBG-Lang

Generiamo helpers con listener incluso:

```
java org.antlr.v4.Tool -Dlanguage=Python3 PBGLang.g4
```

Customizziamo il **PBGLangListener**.py:

`InterpreteVisitor.py`

Scriviamo l’interprete che legge i comandi da file:

`interprete.py`

Esercizio:

1) aggiungiamo le espressioni di calcolo tra gli statements e tra le variabili

Esempio “avanzato”

Utilizzi avanzati - Chat

`Chat.g4`
da https://github.com/gabriele-tomassetti/antlr-mega-tutorial

Nella grammatica:

Utilizzo di **fragment** per rendere la grammatica più leggibile
Sono elementi costitutivi riutilizzabili per le regole lexer.
Li definisci e poi ti riferisci a loro nella regola lexer. Se li definisci ma non li includi nelle regole lexer, semplicemente non hanno alcun effetto.

Utilizzo di codice python per disambiguare nella grammatica,
Abbiamo aggiunto un **predicato semantico** al token TEXT, scritto tra parentesi graffe e seguito da un punto interrogativo.

Questo è importante non solo per la sintassi stessa, ma anche perché target diversi potrebbero avere campi o metodi diversi, ad esempio LA restituisce un int in python, quindi dobbiamo convertire il char in un int.

Generiamo helpers con listener incluso:

```
java org.antlr.v4.Tool -Dlanguage=Python3 Chat.g4
```

Utilizzi avanzati - Chat

Quando predicato semantico vero, abilita la regola TEXT

```
TEXT: {self._input.LA(-1) == ord('[') or self._input.LA(-1) == ord('(')}? ~[\])]+ ;
```

Usiamo `self._input.LA(-1)` per controllare il carattere prima di quello corrente, se questo carattere è una parentesi quadra o la parentesi aperta, attiviamo il token `TEXT`.

È importante ricordare che questo deve essere un codice valido nella nostra lingua di destinazione, in questo caso “`python`”, perchè il codice viene aggiunto nel lexer `ChatLexer.py`

```
    def TEXT_sempred(self, localctx:RuleContext, predIndex:int):
            if predIndex == 0:
                return self._input.LA(-1) == ord('[') or self._input.LA(-1) == ord('(')
```

Traduttore HTML:
`HtmlChatListener.py`

Utilizzi avanzati - Chat

Vediamo il traduttore in azione:

```
python -m antlr.py input.txt
```

Unit test con gestione custom degli errori:

`ChatTests.py`

Potremmo semplicemente leggere il testo emesso dal listener di errori predefinito, ma c'è un vantaggio nell'usare la nostra implementazione, vale a dire che possiamo controllare più facilmente ciò che accade.

`ChatErrorListener.py`

Test:

```
python -m unittest discover -s . -p ChatTests.py
```

Grammatica
Embedded actions / Semantic predicates

- **Embedded actions**
  - A volte abbiamo bisogno di un maggiore controllo e possiamo farlo incorporando codice (azioni) nella grammatica, codice che verrà copiato nel parser generato.

- **Semantic predicates**
  - Ci consentono di abilitare o disabilitare le regole in una grammatica in base a condizioni
  - Possono essere utili per analizzare grammatiche sensibili al contesto

Attenzione: rendono la grammatica “target language” dependent