Un po' di pratica

...

Linguaggi di Programmazione 2021/2022 @ Unibo - modulo 1

Arnaldo Cesco
arnaldo.cesco2@unibo.it

Outline

- Lezione mista: metà demo, metà ripasso
- Costruiremo un linguaggio da zero
- Tool di costruzione:
  - ANTLR 4 (parser generator)
  - Java (~~Rust o Haskell sono meglio, ma abbiamo una lezione e non un semestre~~)
- Linguaggio interpretato
  - Strongly, dinamically typed
- Costruzione incrementale:
  - Espressioni aritmetiche
  - Variabili
  - Statement
  - Procedure
  - Un costrutto concorrente (se rimane tempo)

Recap: un interprete

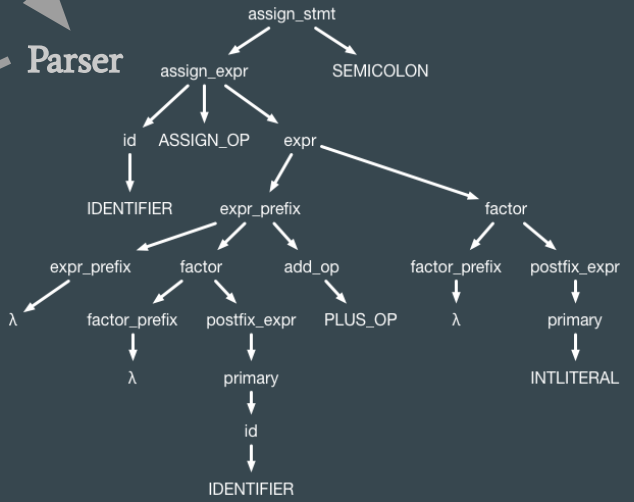
Sorgente

```
            if not hasattr(self, '_headers_buffer'):
                self._headers_buffer = []
            self._headers_buffer.append(("%s %d %s\r\n" %
                    (self.protocol_version, code, message)).encode(
                        'latin-1', 'strict'))

    def send_header(self, keyword, value):
        """Send a MIME header to the headers buffer."""
        if self.request_version != 'HTTP/0.9':
            if not hasattr(self, '_headers_buffer'):
                self._headers_buffer = []
            self._headers_buffer.append(
                ("%s: %s\r\n" % (keyword, value)).encode('latin-1', 'strict'))

        if keyword.lower() == 'connection':
            if value.lower() == 'close':
                self.close_connection = True
            elif value.lower() == 'keep-alive':
                self.close_connection = False
```

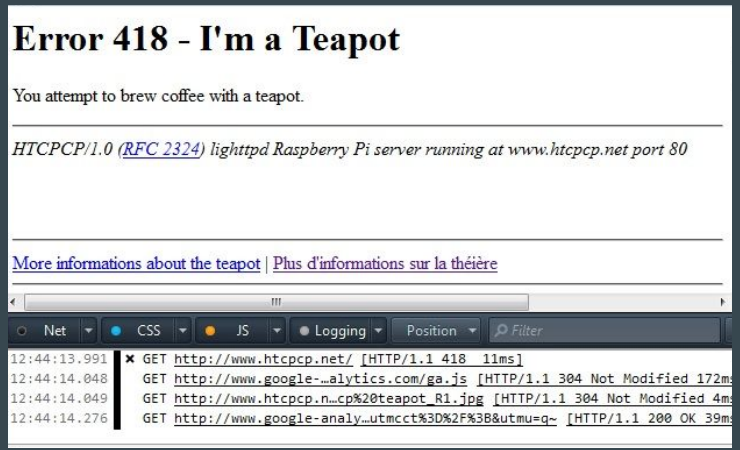
Lexer

```
Tokens:
ID: 0   Type: IDEN   Line: 1[1 ]   Symbol: a
ID: 1   Type: =      Line: 1[3 ]   Symbol:
ID: 2   Type: INT    Line: 1[5 ]   Symbol: 5
ID: 3   Type: ;      Line: 1[6 ]   Symbol:
ID: 4   Type: IDEN   Line: 2[1 ]   Symbol: a
ID: 5   Type: +=     Line: 2[3 ]   Symbol:
ID: 6   Type: INT    Line: 2[6 ]   Symbol: 10
ID: 7   Type: *      Line: 2[9 ]   Symbol:
ID: 8   Type: INT    Line: 2[11]   Symbol: 9
ID: 9   Type: ;      Line: 2[12]   Symbol:
ID: 10  Type: IDEN   Line: 3[1 ]   Symbol: println
ID: 11  Type: (      Line: 3[8 ]   Symbol:
ID: 12  Type: IDEN   Line: 3[10]   Symbol: a
ID: 13  Type: )      Line: 3[12]   Symbol:
ID: 14  Type: ;      Line: 3[13]   Symbol:
```

Parser



Esecuzione



Lexer - esempio

- Stringa di input:

\t if (x == y) \n \t \t z = 1; \n \t else \n \t \t z = 2;

- Lessemi (19!)

\t | if | (| x | == | y |) | \n\t\t | z | = | 1 | ; | \n\t | else | \n\t\t | z | = | 2 | ;

- Sequenza di token

IF, LPAR, ID("x"), EQUALS, ID(“y"), RPAR . . .

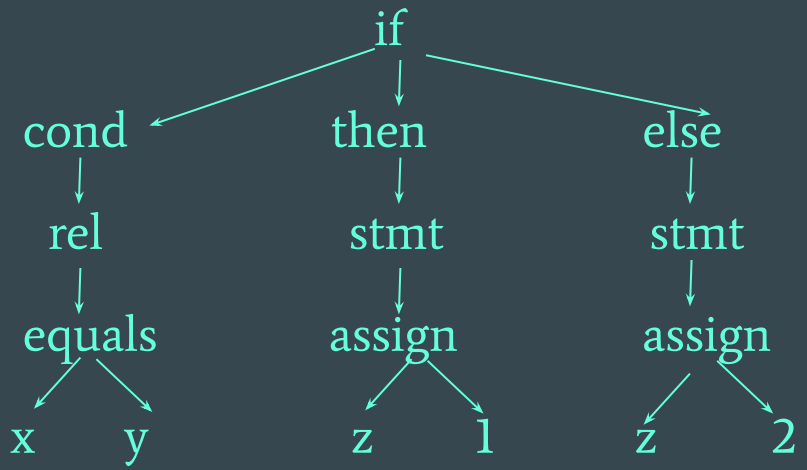
```
if (x == y)
    z = 1;
else
    z = 2;
```

Parser - esempio

- Sequenza di token

IF LPAR ID(“x”) EQUALS ID(“y”) RPAR ID(“z”)
ASSIGN CONST(1) ELSE ID(“z”) ASSIGN CONST(2)

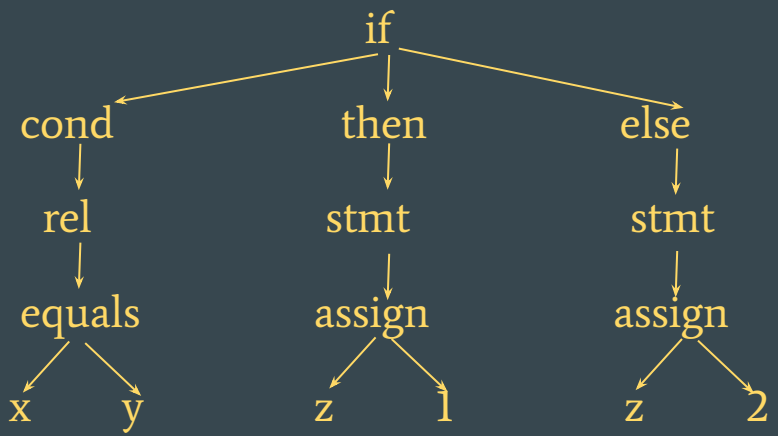
- Albero di sintassi astratto




```
if (x == y)
    z = 1;
else
    z = 2;
```

Esecuzione - esempio

- Albero di sintassi astratto




```
if (x == y)
    z = 1;
else
    z = 2;
```

- Supponiamo che $\sigma = \{x \mapsto 5, y \mapsto 4, z \mapsto 0\}$: l'esecuzione genera

$$\sigma' = \{x \mapsto 5, y \mapsto 4, z \mapsto 2\}$$

ANTLR 4: Generatore di parser LL(*)

- Scritto in Java
- Usa parser simili a LL(1)
- Top down
- Lettura da sinistra a destra (L)
- Ricostruisce gli step della derivazione leftmost (L)
- Può fare lookahead di tutti i token che servono a prevedere quale produzione usare (*)
- Segue ordinatamente le regole
- Sintassi simil-BNF
- Genera automaticamente il codice di lexer e parser, data una grammatica in input
  - Target Java, Go, Python, Javascript...

Grammatica delle espressioni (in simil-ANTLR)

```
exp::= arithmExp | boolExp
arithmExp ::=
    '(' arithmExp ')'
    | '-' arithmExp
    | arithmExp arithm_op arithmExp
    | variable
    | integer
boolExp ::=
    '(' boolExp ')'
    | 'NOT' boolExp
    | boolExp bool_op boolExp
    | arithmExp  compare_op arithmExp
    | variable
    | bool
variable ::= ID;
integer ::= NUMBER;
bool ::= TF;
```

Left recursive!

Grammatica del lexer

Esercizio: rimuovere la ricorsione sinistra

```
arithmExp ::=
    '(' arithmExp ')'
    | '-' arithmExp
    | arithmExp arithm_op arithmExp
    | variable
    | integer;
```

Considerando `'('`, `')'`, `'-'`, `arithm_op`, `variable`, `integer` come terminali

Soluzione: rimuovere la ricorsione sinistra

```
arithmExp::=
    '(' arithmExp ')' arithmExp'
    | '-' arithmExp arithmExp'
    | variable arithmExp'
    | integer arithmExp'
arithmExp'::=
    arithm_op arithmExp arithmExp'
    |ε
```

Demo 1

Espressioni aritmetiche e booleane

```
exp: arithmExp | boolExp;

arithmExp:
   '(' arithmExp ')'   # baseArithmeticExp
   | '-' arithmExp     # negArithmeticExp
   | left = arithmExp op = ('*' | '/') right = arithmExp   # binArithmeticExp
   | left = arithmExp op = ('+' | '-') right = arithmExp   # binArithmeticExp
   | variable                                              # varArithmeticExp
   | integer                                               # valArithmeticExp;

boolExp:
   '(' boolExp ')'                                                    #baseBoolExp
   | 'NOT' boolExp                                                    #negBoolExp
   | left = boolExp op = ('AND' | 'OR' | 'XOR') right = boolExp   #binBoolExp
   | left = arithmExp op = (
       '=='
       | '!='
       | '>'
       | '<'
       | '>='
       | '<='
   ) right = arithmExp # arithmeticComparisonExp
   | left = boolExp op = ('==' | '!=') right = boolExp # boolComparisonExp
   | variable                                          # varBoolExp
   | bool                                              # valBoolExp;

variable: ID;
integer: NUMBER;
bool: TF;
```

E adesso? Semantica!

SOS!

Per fare funzionare il codice Demo, è necessario specificare come

eseguire il codice.

Useremo lo stile della Structural Operational Semantics (Plotkin):

- Data una funzione di assegnamento, detta store, $\sigma : \mathbb{V}ar \mapsto (\mathbb{N} \cup \mathbb{B})$
- Dato un insieme di espressioni $\mathbb{E}xp$
- Dato un insieme di stati $\mathbb{S} \subseteq \{ < e, \sigma > \mid e \in \mathbb{E}xp,\ \sigma \text{ store}\}$
- Esecuzione come una relazione $\rightarrow \subseteq \mathbb{S} \times \mathbb{S}$

Esempio: moltiplicazione

$$\frac{<e_0, \sigma> \text{ -> } <e_0', \sigma'>}{<e_0 * e_1,\ \sigma> \text{ -> } <e_0' * e_1,\ \sigma'>}\ (\text{Mul1})$$

$$\frac{<e_1, \sigma> \text{ -> } <e_1', \sigma'>}{<n * e_1,\ \sigma> \text{ -> } <n_0 * e_1', \sigma'>}\ (\text{Mul2})$$

$$\frac{n_0 * n_1 = m}{<n_0 * n_1,\ \sigma> \text{ -> } <m,\ \sigma>}\ (\text{Mul3})$$

$$\frac{}{<v,\ \sigma> \text{ -> } \mathbf{<\sigma(v),\ \sigma>}}\ (\text{Var})$$

Digressione: SOS small-step vs big-step

Il prof. Gorrieri ha mostrato la semantica small-step, dove si descrive ogni passaggio della computazione (vedi slide precedente). È molto dettagliata e può esprimere facilmente costrutti avanzati, ma... non si mappa bene su un interprete.

La semantica big-step è una relazione $\Downarrow \subseteq \mathbb{S} \times \mathbb{S}$ che può essere intuitivamente vista come “partendo da uno stato, $\Downarrow$ arriva direttamente allo stato finale della corretta sequenza di applicazioni di $\rightarrow$”. Infatti $\Downarrow$ sse $\rightarrow^*$.

Esempio: moltiplicazione

$$\frac{\langle e_0, \sigma\rangle \Downarrow \langle n_0, \sigma'\rangle \qquad \langle e_1, \sigma'\rangle \Downarrow \langle n_1, \sigma''\rangle \qquad n_0 * n_1 = m}{\langle e_0 * e_1,\ \sigma\rangle \Downarrow \langle m,\ \sigma''\rangle}\ \text{(Mul-Big)}$$

$$\frac{}{\langle v,\ \sigma\rangle \Downarrow \langle \sigma(v),\ \sigma\rangle}\ \text{(Var-Big)}$$

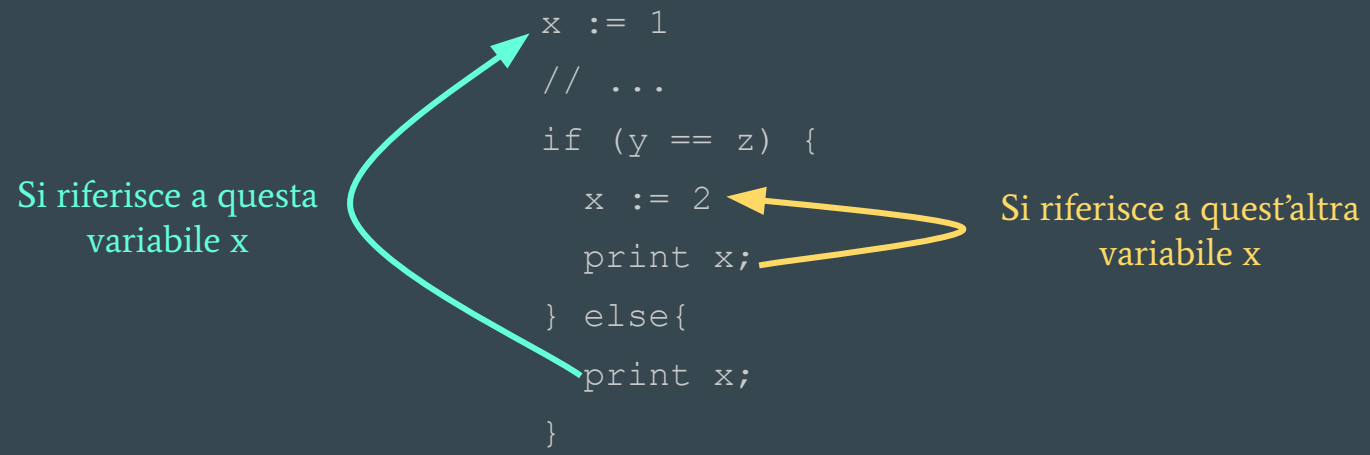
Dalla SOS al codice: esempio

$$\frac{\langle e_0,\sigma\rangle \Downarrow \langle n_0,\sigma'\rangle \qquad \langle e_1,\sigma'\rangle \Downarrow \langle n_1,\sigma''\rangle \qquad n_0 * n_1 = m}{\langle e_0 * e_1,\ \sigma\rangle \Downarrow \langle \mathbf{m},\ \boldsymbol{\sigma''}\rangle}\ \text{(Mul-Big)}$$

```
class Mul extends Exp{
    Exp leftHandSide;
    Exp rightHandSide;
    ...
    public Pair<Int, Env> eval(Env env){
        leftVal = eval(leftHandSide, env);
        rightVal = eval(rightHandSide, env);
        return (leftVal * rightVal);
    }
}
```

L’ambiente

- In teoria, σ è una funzione $\sigma : \mathbb{V}ar \mapsto (\mathbb{N} \cup \mathbb{B})$. Ma...
- ...nella pratica, è spesso rappresentata come una mappa da stringhe a valori per motivi di performance
- In più, spesso si accetta anche la possibilità di shadowing...
  - Ad esempio, questo spezzone di codice è valido



L’ambiente

- ...quindi la mappa è divisa in parti, una per ogni blocco di codice (detto scope)
- Le parti più recenti “nascondono” le parti più antiche
- Nel nostro caso, i valori possibili sono interi o booleani
  - Useremo i valori di Java per comodità
- Quindi la nostra funzione σ è rappresentata così nell’interprete:

```
    private LinkedList<HashMap<String, Either<Integer, Boolean>>> store;
// ...
// Dichiarazione
    public void newVariable(String id, Either<Integer, Boolean> value)
// Lettura
    public Either<Integer, Boolean> getVariable(String id)
// Scrittura
    public void setVariable(String id, Either<Integer, Boolean> value)
```

Demo 2

Eseguire espressioni

(dopo pausa)

Da espressioni a statement

- Gli statement (in genere) modificano l'ambiente
  - Stato terminale contenente solo l'ambiente:
  - Nell'ambiente σ, lo statement s esegue e produce l'ambiente σ'
- Divisione spesso didattica, che si affievolisce nei linguaggi più moderni
  - Ad esempio Scala, Elixir, ...
- Introdurre alcuni statement renderà il nostro linguaggio Demo decisamente più espressivo
  - Turing-completo, perfino
- Infine, aggiungeremo una funzione di libreria: print

Grammatica degli statement

```
statement ::=
   print | ifthenelse | assignment | declaration | loop
print ::= 'print' exp ';'
ifthenelse ::= 'if' '(' boolExp ')' block 'else' block
assignment ::= variable '=' exp ';'
declaration ::= variable ':=' exp ';'
loop ::= 'while' '(' boolExp ')' block
block: '{' statement* '}'
```

Demo 3

Grammatica completa del linguaggio Demo

```
block: '{' statement* '}';
statement:
   print
   | ifthenelse
   | assignment
   | declaration
   | loop ;
print: 'print' exp ';';
ifthenelse:
   'if' '(' boolExp ')' then = block
'else' els = block;
assignment: variable '=' exp ';';
declaration: variable ':=' exp ';';
loop: 'while' '(' boolExp ')' block;
```

Semantica di Demo: alcuni casi interessanti

- Dichiarazione

$$\text{(Declare)}\ \frac{\langle e, \sigma\rangle \Downarrow \langle n, \sigma\rangle}{\langle v := e, \sigma\rangle \to \sigma \cup \{v \mapsto n\}} \quad \text{se } v \notin \text{dom}(\sigma)$$

- Assegnamento

$$\text{(Assign)}\ \frac{\langle e, \sigma\rangle \Downarrow \langle n, \sigma\rangle}{\langle v = e, \sigma\rangle \to \sigma[v/n]} \quad \text{se } v \in \text{dom}(\sigma)$$

N.B: dato che il contesto è abbastanza chiaro, stiamo facendo overload dei simboli $\to$ e $\Downarrow$; bisogna però tenere sempre presente che la relazione $\to_{exp}$ è **diversa** dalla relazione $\to_{stm}$

Semantica di Demo: alcuni casi interessanti

- Block: utile per raggruppare sintatticamente, ma semantica poco interessante
  - Riassume l'operatore sequenza ; all'interno di comandi più complessi
- Sequenza

$$\frac{\langle s_1, \sigma\rangle \to \langle s_1', \sigma'\rangle}{\langle s_1; s_2, \sigma\rangle \to \langle s_1'; s_2, \sigma'\rangle}\ (\text{Seq-1})$$

$$\frac{\langle s_1, \sigma\rangle \to \sigma'}{\langle s_1; s_2, \sigma\rangle \to \langle s_2, \sigma'\rangle}\ (\text{Seq-2})$$

(la semantica dell'operatore ; è ancora più chiara usando la variante big-step)

Semantica di Demo: alcuni casi interessanti

- If: 2 casi!

$$\frac{\langle e, \sigma \rangle \Downarrow \langle true, \sigma \rangle}{\langle \texttt{if } e \texttt{ then } s \texttt{ else } s', \sigma \rangle \rightarrow \langle s, \sigma \rangle} \text{ (If-true)}$$

$$\frac{\langle e, \sigma \rangle \Downarrow \langle false, \sigma \rangle}{\langle \texttt{if } e \texttt{ then } s \texttt{ else } s', \sigma \rangle \rightarrow \langle s', \sigma \rangle} \text{ (If-false)}$$

- Print

$$\frac{\langle e, \sigma \rangle \Downarrow n \quad n \text{ stampato a console}}{\langle \texttt{print } e, \sigma \rangle \rightarrow \sigma} \text{ (Print)}$$

Semantica degli statement: alcuni casi interessanti

- While: 2 casi!

$$\frac{\langle e, \sigma \rangle \Downarrow \langle true, \sigma \rangle}{\langle while\ e\ do\ s, \sigma \rangle \rightarrow \langle s; while\ e\ do\ s, \sigma \rangle}\ (While\text{-}true)$$

$$\frac{\langle e, \sigma \rangle \Downarrow \langle false, \sigma \rangle}{\langle while\ e\ do\ s, \sigma \rangle \rightarrow \sigma}\ (While\text{-}false)$$

Implementare la semantica

- La relazione → è implementata con una funzione che passa ricorsivamente top-down tutti i nodi dell'AST
- Gli errori sono implementati da eccezioni che si propagano

```
public abstract Either<Integer, Boolean> eval(Environment env) throws DemoException;
```

Ogni nodo implementa concretamente questo metodo

Il risultato è un `Integer` o un `Boolean` per le espressioni, `null` per gli statement

Se c'è un errore, lanciamo eccezioni come `TypeErrorException`, `NotDefinedException` ecc...

Demo 4

Eseguire un programma Demo

Ricapitolando...

- Abbiamo costruito un linguaggio ed il suo interprete!
  - Incrementalmente: espressioni, poi statement, poi...
- Partendo dalla grammatica
  - Con l'aiuto di ANTLR 4
    - Generatore di lexer e parser
  - Ricordando alcune regole di parsing LL
- Alla grammatica abbiamo aggiunto una semantica
  - Structural operational semantics
- Dalla semantica abbiamo capito come scrivere l'interprete
- Per chi si fosse perso qualcosa, il codice sorgente di questo progetto è su Github: https://github.com/Annopaolo/demo

E adesso?

Estensioni a Demo

- Procedure: funzioni di arietà (numero di parametri) 0

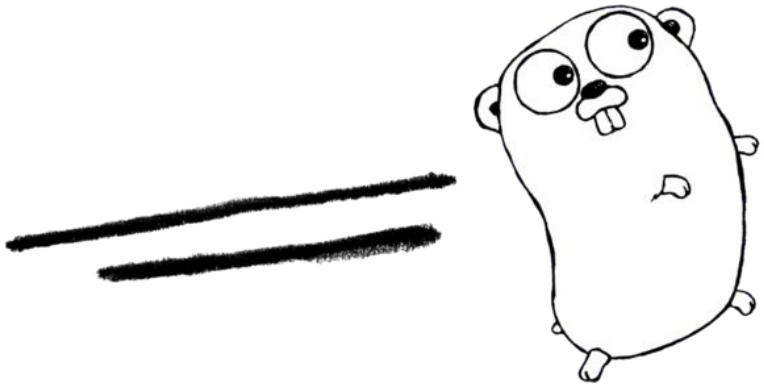
$$\text{(Proc-declare)}\ \frac{}{\langle \texttt{proc}\ p\ \{s\},\ \sigma\rangle \to \sigma \cup \{p \mapsto s\}}\quad \text{se } p \notin \text{dom}(\sigma)$$

$$\text{(Proc-call)}\ \frac{\langle s,\ \sigma\rangle \Downarrow \sigma' \qquad \sigma(p) = s}{\langle p,\ \sigma\rangle \Downarrow \sigma'}$$

- Esecuzione concorrente di procedure: go p
  - Semantica formale non banale: bisogna modellare un runtime (thread, main...)
  - C'è anche un altro problema: quale?

Demo 5

Procedure ed esecuzione concorrente



Fine

Grazie, e buon proseguimento!