



Domain Specific Language

Linguaggi custom per configurazioni e personalizzazioni



LUCA MINUTI DeveLOPer

email

Luca.minuti@gmail.com

GITHUB

[HTTPS://GITHUB.COM/LMINUTI/](https://github.com/Lminuti/)



GITHUB PROJECTS



WIRL

[HTTPS://GITHUB.COM/DELPHI-BLOCKS/WIRL](https://github.com/Delphi-Blocks/WIRL)

GRAPHQL FOR DELPHI

[HTTPS://GITHUB.COM/LMINUTI/GRAPHQL](https://github.com/Lminuti/GraphQL)

DELPHI-OPENSSL

[HTTPS://GITHUB.COM/LMINUTI/DELPHI-OPENSSL](https://github.com/Lminuti/Delphi-OpenSSL)

DELPHI-WKHTMLTOX

[HTTPS://GITHUB.COM/LMINUTI/DELPHI-WKHTMLTOX](https://github.com/Lminuti/Delphi-WKHTMLTOX)





AGENDA

- Cos'è un DSL
- Tipi di DSL
- Casi d'uso
- Struttura
- Personalizzazione



Introduzione

1



COS'È UN DSL?

“A Domain-Specific Language (DSL) is a computer language that's targeted to a particular kind of problem, rather than a general purpose language that's aimed at any kind of software problem. Domain-specific languages have been talked about, and used for almost as long as computing has been done.”

– Martin Fowler



COS'È UN DSL?

- Pensato per essere usato in un particolare dominio
- Contesto applicativo:
 - bancario, assicurativo, sanitario...
- o ambito tecnologico:
 - web, database, grafica...
- Mentre i GPL (general purpose language) sono pensati per adattarsi a qualsiasi ambito



ESEMPI NOTI

- SQL → Database
- HTML → Sviluppo web
- RegExp → Ricerca e manipolazione di testi
- Make → Automatizzazione del processo di build
- MATLAB → Calcolo numerico e analisi statistica



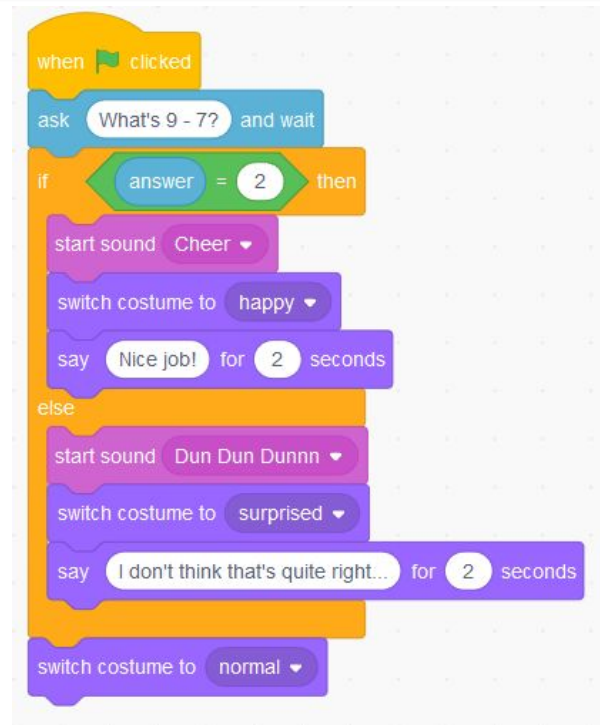
INTERNI O ESTERNI

- Internal DSL (o Embedded DSL)
 - Estendono le funzionalità di un linguaggio esistente
 - Possono usare un preprocessore, plugin del linguaggio o essere integrati nel linguaggio principale
- External DSL
 - Linguaggio indipendente
 - Non ha bisogno di un linguaggio “padre”
 - Ha un proprio parser



TIPI DI DSL

- Testuali
 - Basati su testo come i linguaggi tradizionali
- Grafici
 - Hanno un'interfaccia grafica spesso basata su blocchi
 - Particolarmente interessante se chi li usa non è un programmatore





CASI D'USO

- Personalizzazione dell'applicativo
- Analisi della sintassi di espressioni
- Parser di SQL ad alto livello
- Controllo di hardware (IoC)
- Gestione di flussi di dati (leggi da FTP, salva su DB, invia per email...)



Struttura

2



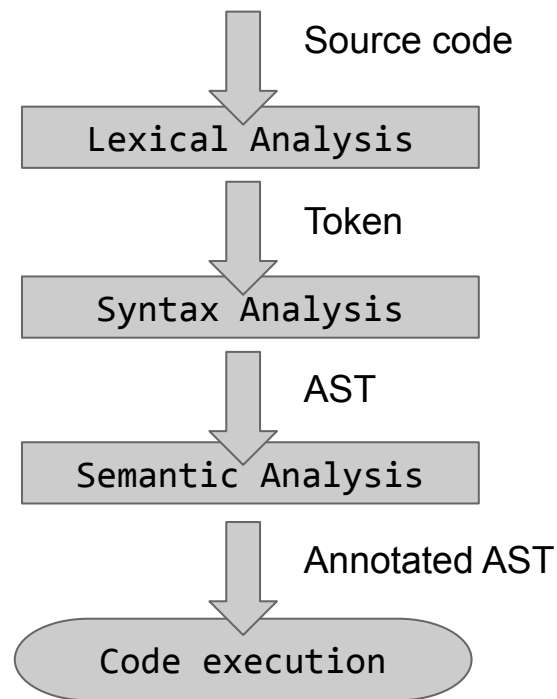
PREREQUISITI

- Definire l'ambito d'uso del linguaggio
- Chiarire l'utente tipo del linguaggio
- Definire la grammatica



FA SI

- Dividere il codice in token (Lexical Analysis)
- Analizzare i token per comprenderne il senso e verificarne la correttezza (Syntax Analysis)
- Eseguire il codice





LEXICAL ANALYSIS

- Divide il codice in token:
- ◆ Operatori (+, -, /, > ...)
 - ◆ Keyword (if, else, then, ...)
 - ◆ Identificatori (nomi di variabili, funzioni, ecc)
 - ◆ Literal (numeri, stringhe, ...)

```
test = 12;  
// One line comment  
if test == 12 then  
    println(test)  
end;
```

```
(Identifier)test  
(Assignment)  
(Integer)12  
(Semicolon)  
(KeywordIf)if  
(Identifier)test  
(Equivalence)  
(Integer)12  
(KeywordThen)then  
(KeywordPrintln)println  
(LeftParenthesis)  
(Identifier)test  
(RightParenthesis)  
(KeywordEnd)end  
(Semicolon)
```



TOKEN

```
TToken = record  
  LineNumber: Integer;  
  ColumnNumber: Integer;  
  Kind: TTokenKind;  
  StringValue: string;  
  FloatValue: Double;  
  IntegerValue: Integer;  
  function ToString: string;  
end;
```

} Posizione all'interno del codice
} Tipo: identificativo, keyword, ...
} Valore per i letterali o nome
dell'identificativo

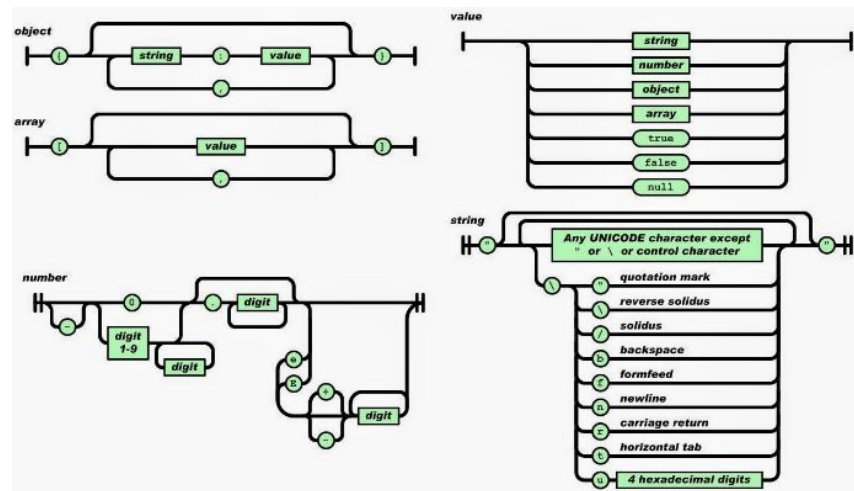
demo time





SYNTAX ANALYSIS

- Si basa su una grammatica
- Verifica se il codice è corretto
- Restituisce una esatta indicazione dell'errore
- Costruisce l'albero sintattico (AST)





GRAMMAR

- Insieme di regole che definiscono la struttura sintattica del linguaggio
- Permettono di individuare in modo non ambiguo se l'insieme dei token in input è corretto
- Si possono usare varie notazioni tra cui EBNF (Extended Backus–Naur form)



GRAMMAR - EBNF

```
program = 'PROGRAM', white space, identifier, white space,  
         'BEGIN', white space,  
         { assignment, ";", white space },  
         'END.' ;  
  
identifier = alphabetic character, { alphabetic character | digit } ;  
number = [ "-" ], digit, { digit } ;  
string = "'", { all characters - "'" }, "'" ;  
assignment = identifier, ":", ( number | identifier | string ) ;  
alphabetic character = "A" | "B" | "C" | "D" | "E" | "F" | "G"  
                      | "H" | "I" | "J" | "K" | "L" | "M" | "N"  
                      | "O" | "P" | "Q" | "R" | "S" | "T" | "U"  
                      | "V" | "W" | "X" | "Y" | "Z" ;  
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;  
white space = ? white space characters ? ;  
all characters = ? all visible characters ? ;
```



GRAMMAR - EBNF

- È un modo per rappresentare testualmente la sintassi di un linguaggio
- È composto da simboli terminali (token trovati dal lexer) e non terminali (derivati)
- Ogni riga è una regola
- Ogni regola associa un simbolo non terminale ad una sequenza di simboli terminali e non terminali



GRAMMAR - EBNF

```
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | ;  
integer = [ "-" ] , { digit } ;  
float = [ "-" ] , { digit } [ "." , { digit } ] ;  
number = integer | float ;  
assignment = identifier , "=" , number ;
```

Usage	Notation
definition	=
concatenation	,
termination	;
alternation	
optional	[...]
repetition	{ ... }
grouping	(...)
terminal string	" ... "
terminal string	' ... '
comment	(* ... *)
special sequence	? ... ?
exception	-



GRAMMAR - EBNF

```
// assignment = identifier , "=" , number ;
```

```
procedure Assignment;
```

```
begin
```

```
    Expect(TTokenKind.Identifier);
```

```
    Expect(TTokenKind.Assignment);
```

```
    Expect(TTokenKind.IntegerLiteral);
```

```
end;
```



GRAMMAR - EBNF

```
// assignment = identifier , "=", expression ;  
// expression = number | identifier ;
```

```
procedure Assignment;  
begin  
    Expect(TTokenKind.Identifier);  
    Expect(TTokenKind.Assignment);  
    Expression;  
end;
```

```
procedure Expression;  
begin  
    if not Token.isIdentifier() and  
        not Token.isNumber() then  
        raise ...;  
    NextToken();  
end;
```



GRAMMAR - EBNF

EBNF	Codice pascal
<code>A B C</code>	<pre>case token of A: A() B: B() C: C() else raise ...</pre>
<code>A { '*' A }</code>	<pre>A(); while token = '*' do begin NextToken(); A(); end;</pre>
<code>[A]</code>	<pre>if token = A then NextToken();</pre>

demo time



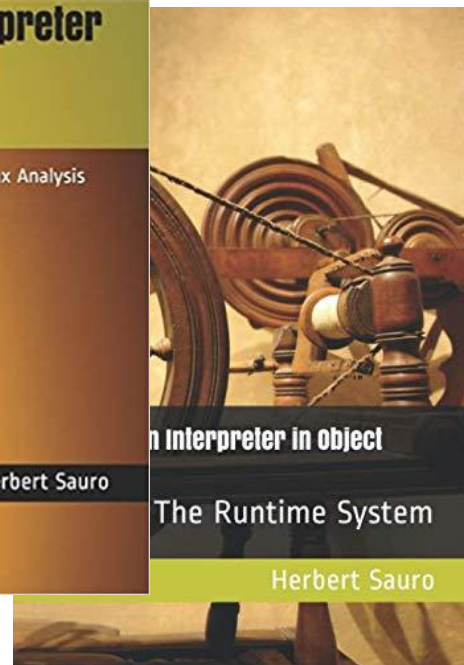
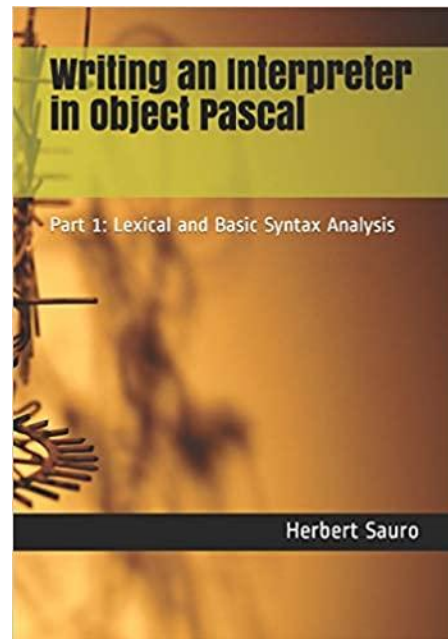


Domande?



APPROFONDIMENTI

- Writing an Interpreter in Object Pascal: Part 1: Lexical and Basic Syntax Analysis
- Writing an Interpreter in Object Pascal: Part II: The Runtime System





THANK YOU