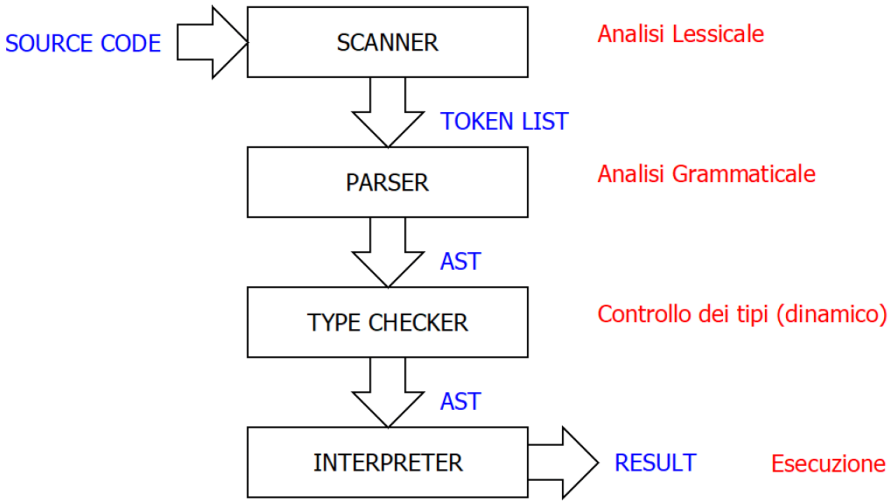
Introduzione

Componenti principali da analizzare



Ciclo di interpretazione

L'interprete esegue le operazioni elementari del programma una dopo l'altra. Le fasi di scanning/parsing possono essere svolte a livello di:

- intero programma
- singole istruzioni / espressioni

> ☝🏻 *L'interprete **JavaScript** (Node.js) prima esegue, analizza la sintassi di tutto il programma e poi interpreta*

```
console.log(1);console.log(2; //manca la parentesi
```

Risultato:

```
console.log(2;
                        ^
```

```
SyntaxError: missing ) after argument list
```

☝🏻 *Il toplevel di **Ocaml** esegue il parsing ed esegue espressione per espressione*

```
let x = 10 ;;lett y = 20 ;; (* let scritto male... *)
```

```
File "[1]", line 2, characters 0-4:
2 | lett y = 20 ;; (* let scritto male... *)
        ^^^^
Error: Unbound value lett
```

Un interprete di espressioni aritimetiche

Sintassi delle espressioni aritmetiche:

```
Exp ::= n | Exp op Exp | (Exp)
op ::= + | - | * | /
```

```
type op = Add | Sub | Mul | Div ;;
type exp =
    | Val of int
    | Op of op*exp*exp;;
```

esempi di espressioni:

$$exp1 = (3 * 7) - 5$$

```
let exp1 = Op (Sub, (Op (Mul, Val 3, Val 7)), Val 5) ;;
```

Scanner

Rappresentiamo ogni simbolo che può apparire in una espressione con un token

```
type token =
    | Tkn_NUM of int (* numero n *)
    | Tkn_OP of string (* operatore + - * / *)
    | Tkn_LPAR (* simbolo ( *)
    | Tkn_RPAR (* simbolo ) *)
    | Tkn_END;; (* fine dell'espressione *)
```

```
type token =
        Tkn_NUM of int
    | Tkn_OP of string
    | Tkn_LPAR
    | Tkn_RPAR
    | Tkn_END
```

Lo scanner trasforma la rappresentazione testuale dell'espressione (stringa) in una lista di token

Implementazione dello scanner in OCaml

```
(* scanner *)
let tokenize s =
        (* funzione che scandisce ricorsivamente s,
             dove pos è la posizione del carattere corrente *)
        let rec tokenize_rec s pos =
                if pos=String.length s then [Tkn_END] (* caso base: fine li
                else
                        let c = String.sub s pos 1 (* estrae il carattere c
                        in
                            (* si richiama ricorsivamente prima di gestire
                            let tokens = tokenize_rec s (pos+1)
                            in
                                (* trasforma il carattere corrente in un to
                                match c with
                                    | " " -> tokens
                                    | "(" -> Tkn_LPAR::tokens
                                    | ")" -> Tkn_RPAR::tokens
                                    | "+" | "-" | "*" | "/" -> (Tkn_OP c)::
                                    | "0" | "1" | "2" | "3" | "4" | "5" | "
```

```
                                        (* accorpa cifre consecutive *)
                                        (match tokens with
                                            | Tkn_NUM n::tokens' ->
                                                        Tkn_NUM (int_of_str
                                            | _ -> Tkn_NUM (int_of_string c
                                        )
                                    | _ -> raise (ParseError ("Tokenizer","
        in
                tokenize_rec s 0 ;;
```

Esempio d'uso:

```
 let t1 = tokenize "(34 + 41) - (2223 * 2)";;
```

```
val t1 : token list =
     [Tkn_LPAR; Tkn_NUM 34; Tkn_OP "+"; Tkn_NUM 41; Tkn_RPAR; Tkn_OP "-";
      Tkn_LPAR; Tkn_NUM 2223; Tkn_OP "*"; Tkn_NUM 2; Tkn_RPAR; Tkn_END]
```

Funzioni di utilità per stampare un token o una lista di token:

```
let string_of_token t =
    match t with
        | Tkn_NUM n -> "Tkn_NUM "^(string_of_int n)
        | Tkn_OP s -> "Tkn_OP "^s
        | Tkn_LPAR -> "Tkn_LPAR"
        | Tkn_RPAR -> "Tkn_RPAR"
        | Tkn_END -> "Tkn_END" ;;

let print_token t =
    print_endline (string_of_token t);;

let print_tokenlist tl =
    List.iter (fun t -> (print_token t)) tl;;
```

```
val string_of_token : token -> string = <fun>
val print_token : token -> unit = <fun>
val print_tokenlist : token list -> unit = <fun>
```

Parser

Il parser controlla che l'espressione sia sintatticamente corretta, appartenga al linguaggio definito dalle regole BNF e generi l'abstract syntax tree (AST)

Come realizzare un parser:

La grammatica deve essere resa non ambigua

- se il linguaggio è molto semplice, si implementa un parser a discesa ricorsiva
- se il linguaggio non è molto semplice, si usa un parser generator (software che genera il codice del parser a partire dalla grammatica)
  - Flex/Bison per generare un parser in C
  - PEG.js, ATNLR4 o Jison per generare un parser in JavaScript
  - ocamllex/ocamlyacc o Menhir per generare un parser in OCaml
  - ATNLR4 o JavaCC per generare un parser in Java

Ridefiniamo la grammatica delle espressioni in modo non ambiguo:

```
Exp ::= Term + Exp | Term - Exp | Term
Term ::= Factor * Term | Factor / Term | Factor
Factor ::= n | (Exp)
```

Implementiamo la funzione `parse` a discesa ricorsiva con:

- una funzione per ogni categoria sintattica (exp, term e factor)
- le funzioni sono mutuamente ricorsive
- le funzioni si richiamano l'una con l'altra e consumano token secondo quanto indicato dalla grammatica
- ogni chiamata di funzione restituisce un nodo dell'AST
  - l'AST viene in questo modo corrisponde all'albero delle chiamate

```
(* parser *)
let parse s =
  (* usiamo un riferimento per scandire la lista dei token ( ottenuta da to
  let tokens = ref (tokenize s) in

  (* restituisce il primo token senza rimuoverlo *)
  let lookahead () =
```

```
  match !tokens with
  | [] -> raise (ParseError ("Parser","lookahead error"))
  | t::_ -> t
in

(* elimina il primo token *)
let consume () =
  match !tokens with
  | [] -> raise (ParseError ("Parser","consume error"))
  | t::tkns -> tokens := tkns
in

(* funzioni mutuamente ricorsive che seguono dalla grammatica *)
(* Exp ::= Term [ + Exp | - Exp ] *)
let rec exp () =
  let t1 = term() in
  match lookahead () with
  | Tkn_OP "+" -> consume(); Op (Add,t1,exp())
  | Tkn_OP "-" -> consume(); Op (Sub,t1,exp())
  | _ -> t1
(* Term ::= Factor [ + Term | - Term ] *)
and term () =
  let f1 = factor() in
  match lookahead() with
  | Tkn_OP "*" -> consume(); Op (Mul,f1,term())
  | Tkn_OP "/" -> consume(); Op (Div,f1,term())
  | _ -> f1
(* Factor ::= n | ( Exp ) *)
and factor () =
  match lookahead() with
  | Tkn_NUM n -> consume(); Val n
  | Tkn_LPAR -> consume(); let e = exp() in
    (match lookahead() with
    | Tkn_RPAR -> consume(); e
    | _ -> raise (ParseError ("Parser","RPAR error"))
    )
  | _ -> raise (ParseError ("Parser","NUM/LPAR error"))
(* Si comincia chiamando exp che fa tutto il lavoro e restituisce la radi
in
let ast = exp() in
(* controlliamo che al termine sia rimasto solo Tkn_END *)
match lookahead() with
```

```
  | Tkn_END -> ast
  | x -> print_tokenlist !tokens; raise (ParseError ("Parser","parse error"
```

Esempi di esecuzione del parser:

```
let ast = parse "32 + 24 * 12 * (3-1) +2" ;;
```

```
val ast : exp =
    Op (Add, Val 32,
     Op (Add, Op (Mul, Val 24, Op (Mul, Val 12, Op (Sub, Val 3, Val 1))),
      Val 2))
```

```
let err = parse "16 + (7 - 2" ;;
```

```
Exception: ParseError ("Parser", "RPAR error").
Raised at file "[13]", line 43, characters 35-71
Called from file "[13]", line 30, characters 17-25
Called from file "[13]", line 22, characters 17-23
Called from file "[13]", line 24, characters 46-51
Called from file "[13]", line 49, characters 18-23
Called from unknown location
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

Interprete

La definizione di una relazione di transizione per descrivere il comportamento dei programmi (o espressioni) scritti in un certo linguaggio segue solitamente l'approccio della ***Structural Operational Semantics*** (**SOS**), o Semantica Strutturale Operazionale.

- Semantics: è una descrizione del "significato" del linguaggio. Nei linguaggi di programmazione il significato è dato dal comportamento dei programmi scritti in quel linguaggio
- Operational: descrive il comportamento dei programmi tramite una relazione di transizione ($\rightarrow$) che cattura le operazioni che vengono svolte passo-passo durante l'esecuzione
- Structural: la relazione di transizione è definita usando regole di inferenza basate sulla struttura sintattica (una o più regole per ogni costurutto definito dalla grammatica del linguaggio)

Esistono due approcci per la semantica **SOS**:

1. *Small-step semantics*:
   - ogni passo della relazione di transizione si esegue una singola operazione
   - una computazione è una sequenza di passi
   - esempio: $((3 + (5 * 2)) - 1) \rightarrow_{ss} ((3 + 10) - 1) \rightarrow_{ss} (13 - 1) \rightarrow_{ss} 12$
2. **Big-step semantics**:
   - la relazione di transizione descrive in un solo passo l'intera computazione
   - le singole operazioni sono descritte nell'albero di derivazione di quella transizione
   - esempio:

$$\dfrac{\dfrac{\vdots}{3 \rightarrow_{bs} 3} \quad \dfrac{\vdots}{(5*2) \rightarrow_{bs} 10} \quad 3+10=13}{(3+(5*2)) \rightarrow_{bs} 13} \quad 1 \rightarrow_{bs} 1 \quad 13 - 1 = 12}{((3 + (5 * 2)) - 1) \rightarrow_{bs} 12}$$

Semantica delle espressioni

semantica small-step

$$n \to_{ss} n \qquad \frac{E_1 \to_{ss} E_1'}{E_1\, op\, E_2 \to_{ss} E_1'\, op\, E_2}$$

$$\frac{E_2 \to_{ss} E_2'}{n\, op\, E_2 \to_{ss} n\, op\, E_2'} \qquad \frac{n_1 \mathrm{op}\, n_2 = n}{n_1\, op\, n_2 \to_{ss} n}$$

semantica big-step

$$n \to_{bs} n \qquad \frac{E_1 \to_{bs} n_1 \quad E_2 \to_{bs} n_2 \quad n_1 \mathrm{op}\, n_2 = n}{E_1\, op\, E_1 \to_{bs} n}$$

Implementazione interprete tramite big-step

```
(* interprete big-step *)
let rec eval e =
  match e with
  | Val n -> Val n
  | Op (op, e1, e2) ->
    (* chiamate ricorsive che calcolano le derivazioni per e1 ed e2 *)
    match (eval e1, eval e2) with
    | (Val n1, Val n2) ->
      (match op with (* calcola n1 op n2 *)
      | Add -> Val (n1 + n2)
      | Sub -> Val (n1 - n2)
      | Mul -> Val (n1 * n2)
      | Div -> Val (n1 / n2)
      )
    (* caso ( inutile ) aggiunto solo per rendere esaustivo il pattern matc
    | _ -> failwith "Errore impossibile che si verifichi" ;;
```

```
 val eval : exp -> exp = <fun>
```

Esempio di utilizzo dell'interprete:

```
(* AST dell'espressione 3+ ( 5 * 2 ) *)
let exp = Op (Add , Val 3, Op ( Mul , Val 5, Val 2)) ;;

eval exp;;
```

```
val exp : exp = Op (Add, Val 3, Op (Mul, Val 5, Val 2))

- : exp = Val 13
```

Implementazione interprete tramite small-step

```
(* interprete small-step *)
let rec eval_ss e =
  match e with
  | Val n -> Val n
  | Op (op, e1, e2) ->
    (match (e1, e2) with
    (* se gli operandi sono entrambi valori, calcola n1 op n2 *)
    | (Val n1, Val n2) ->
      Val (match op with
          | Add -> n1 + n2
          | Sub -> n1 - n2
          | Mul -> n1 * n2
          | Div -> n1 / n2 )
    (* se e1 può fare un passo ( cioè se è un Op e non un valore ) *)
    | (Op (_, _, _), _) -> Op (op, (eval_ss e1), e2)
    (* altrimenti, se e1 è un valore ma e2 può fare un passo *)
    | (Val _, Op (_, _, _)) -> Op (op, e1, (eval_ss e2)) ;;
```

```
val eval_ss : exp -> exp = <fun>
```

Esempio di utilizzo dell'interprete:

```
(* AST dell'espressione 3+ ( 5 * 2 ) *)
let exp = Op (Add , Val 3, Op ( Mul , Val 5, Val 2)) ;;

eval_ss exp;; (* fa un passo *)
eval_ss (eval_ss exp);; (* fa due passi *)
```

```
val exp : exp = Op (Add, Val 3, Op (Mul, Val 5, Val 2))
- : exp = Op (Add, Val 3, Val 10)
- : exp = Val 13
```

Questi due esempi mostrano un approccio sistematico di implementazione:

1. Si usa il pattern matching per considerare i vari tipi di nodo dell'AST
2. Ogni tipo di nodo corrisponde a un costrutto sintattico definito dalla grammatica
3. Per ogni caso del pattern matching (costrutto sintattico) si identificano le regole della semantica relative ad esso (la semantica è syntax-driven)
4. Si verificano le precondizioni delle varie regole, possibilmente richiamando ricorsivamente l'interprete (per le regole che non sono assiomi)
5. Quando si trova una regola le cui precondizioni sono verificate, si calcola risultato della transizione

Scanner + Parser + Interprete

```
(* con inteprete big-step *)
let exec1 s = eval (parse s) ;;
(* con inteprete small-step *)
let exec2 s =
    let rec eval_rec ast = (* chiusura transitiva *)
        match ast with
         | Val n -> Val n
         | _ -> eval_rec ( eval_ss ast )
    in
         eval_rec (parse s) ;;
```

```
val exec1 : string -> exp = <fun>
val exec2 : string -> exp = <fun>
```