

Informatica e Computazione

Interprete per Linguaggio LISP-like

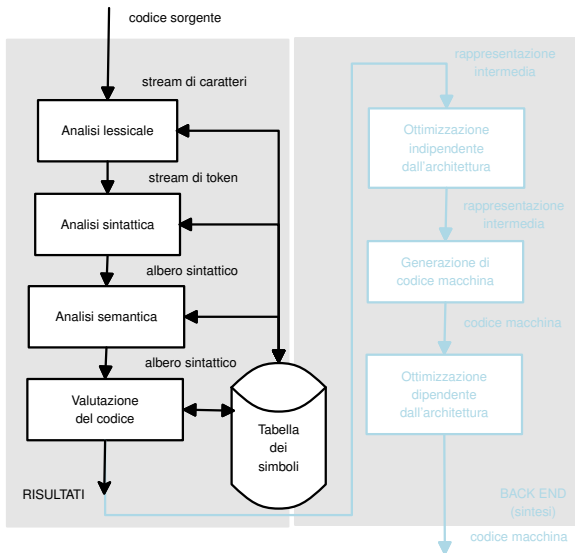
Armando Tacchella

Corso di Laurea in Ingegneria Informatica
Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi

Sommario

- 1 Introduzione
- 2 Formato di input
- 3 Semantica
- 4 Suggerimenti per il progetto e l'implementazione
- 5 Altre specifiche di progetto

Compilatori e interpreti: cosa?



Interprete di un linguaggio LISP-like: perché?

- Un compito **semplice**:
 - ▶ solo costrutti elementari: I/O, variabili, espressioni, scelte condizionali e cicli;
 - ▶ soltanto due tipi di dato: interi e booleani
 - ▶ nessuna gestione di tipi aggiuntivi, funzioni, classi, ...
 - ▶ nessuna gestione di ambiti, visibilità, ...
- Parti costituenti di **tutti** i compilatori ed interpreti per utilizzo professionale

[A. V. Aho, M. Lam, R. Sethi, J.D. Ullman. Compilers - Principles, Techniques and Tools. Addison-Wesley, 2nd edition, 2006]

Interpretazione di espressioni: come?

- Specifica del linguaggio in input e del formato di output
 - ▶ **Grammatica libera dal contesto** per l'input
 - ▶ **Descrizione informale** dei risultati attesi per l'output
 - ▶ **Vettori di test** per verificare la soluzione
- Due approcci possibili:
 - ▶ Approccio **automatizzato**: scrittura della grammatica di input e delle regole di interpretazione nel formato di un generatore di traduttori (ad esempio, ANTLR)
 - ▶ Approccio **manuale**: sviluppo e implementazione degli algoritmi a partire dalle specifiche
- Per l'esercitazione è prevista la soluzione manuale.

Per approfondimenti sulla generazione automatica: www.antlr.org

Notazione di base

Data una grammatica libera dal contesto $G = (V_N, V_T, P, S)$

- la produzione $A \rightarrow \alpha \mid \beta$ **abbrevia** le produzioni

$$A \rightarrow \alpha$$

$$A \rightarrow \beta$$

- i **caratteri corsivi** definiscono i simboli **non terminali** V_N , ad esempio *program*, *statement*, ecc.
- i **caratteri standard** sono riservati ai simboli **terminali** V_T , ad esempio SET, MUL, (,), ecc.
- Come consueto, il simbolo ϵ denota la stringa vuota.

Grammatica di input (1)

$program \rightarrow stmt_block$

$stmt_block \rightarrow statement \mid (BLOCK statement_list)$

$statement_list \rightarrow statement statement_list \mid statement$

$statement \rightarrow variable_def \mid$

$io_stmt \mid$

$cond_stmt \mid$

$loop_stmt$

$variable_def \rightarrow (SET variable_id num_expr)$

$io_stmt \rightarrow (PRINT num_expr) \mid (INPUT variable_id)$

$cond_stmt \rightarrow (IF bool_expr stmt_block stmt_block)$

$loop_stmt \rightarrow (WHILE bool_expr stmt_block)$

Grammatica di input (2)

num_expr → (ADD *num_expr* *num_expr*)
 | (SUB *num_expr* *num_expr*)
 | (MUL *num_expr* *num_expr*)
 | (DIV *num_expr* *num_expr*)
 | *number*
 | *variable_id*

bool_expr | (LT *num_expr* *num_expr*)
 | (GT *num_expr* *num_expr*)
 | (EQ *num_expr* *num_expr*)
 | (AND *bool_expr* *bool_expr*)
 | (OR *bool_expr* *bool_expr*)
 | (NOT *bool_expr*)
 | TRUE | FALSE

Grammatica di input (3)

$variable_id \rightarrow alpha_list$

$alpha_list \rightarrow alpha\ alpha_list \mid alpha$

$alpha \rightarrow a \mid b \mid c \mid \dots \mid z \mid A \mid B \mid C \mid \dots \mid Z$

$number \rightarrow -\ posnumber \mid posnumber$

$posnumber \rightarrow 0 \mid sigdigit\ rest$

$sigdigit \rightarrow 1 \mid \dots \mid 9$

$rest \rightarrow digit\ rest \mid \epsilon$

$digit \rightarrow 0 \mid sigdigit$

- La grammatica proposta è **non ambigua**
- **Al più** due simboli non terminali disambiguano qualsiasi produzione:
 - ▶ “(SET”, “(PRINT”, “(INPUT”, ecc. consentono di risolvere l’ambiguità tra le varie istruzioni
 - ▶ “(ADD”, “(SUB”, ecc., consentono di risolvere l’ambiguità tra i vari tipi di espressione

Esempio di file di input: calcolo del fattoriale

```
(BLOCK
  (INPUT n)
  (SET result 1)
  (WHILE (GT n 0)
    (BLOCK
      (SET result (MUL result n))
      (SET n (SUB n 1)))
    )
  (PRINT result))
```

- Spazi, tabulazioni e righe vuote **non fanno parte** della sintassi del programma
- Devono essere considerati come “spazio vuoto” e **ignorati**.
- Le **parole chiave** in questo programma sono BLOCK, INPUT, SET, WHILE, GT, MUL, SUB e PRINT

Interpretazione delle dichiarazioni di variabili

Il risultato corrispondente ad una istruzione

(SET *variable_id* *num_expr*)

deve essere:

- la **creazione** della variable *variable_id*, se non era stata già definita in precedenza, e
- l'**assegnazione** alla variable *variable_id* del **valore** dell'espressione *num_expr*.

Possono verificarsi i seguenti errori:

- la dichiarazione non è sintatticamente corretta,
- *variable_id* è una parola chiave, oppure
- *num_expr* non è sintatticamente corretta, oppure contiene un errore semantico (ad e.s., divisione per zero).

Interpretazione dello statement di input

Il risultato corrispondente ad una istruzione

(INPUT *variable_id*)

deve essere:

- la **creazione** della variable *variable_id*, se non era stata già definita in precedenza, e
- l'**assegnazione** alla variable *variable_id* del valore **letto da console**.

Possono verificarsi i seguenti errori:

- l'istruzione non è sintatticamente corretta,
- *variable_id* è una parola chiave, oppure
- viene inserito da console un valore illecito (sono consentiti **solo** numeri interi positivi e negativi).

Interpretazione dello statement di output

Il risultato corrispondente ad una istruzione

(PRINT *num_expr*)

deve essere la **visualizzazione in console** del **valore** dell'espressione *num_expr*.

Possono verificarsi i seguenti errori:

- l'istruzione non è sintatticamente corretta,
- *num_expr* non è sintatticamente corretta, oppure contiene variabili non definite in precedenza con SET o INPUT, oppure contiene un errore semantico (ad e.s., divisione per zero).

Interpretazione dell'istruzione di scelta condizionata

Il risultato corrispondente ad una istruzione

$(\text{IF } \text{bool_expr } \text{stmt_block}_1 \text{ stmt_block}_2)$

deve essere:

- l'esecuzione di stmt_block_1 nel caso in cui bool_expr valuti a vero, oppure
- l'esecuzione di stmt_block_2 nel caso in cui bool_expr valuti a falso.

Possono verificarsi i seguenti errori:

- la dichiarazione non è sintatticamente corretta,
- bool_expr non è sintatticamente corretta o contiene variabili non definite in precedenza, oppure
- uno dei due stmt_block non è sintatticamente corretto, oppure, se eseguito, contiene un errore semantico (ad e.s., divisione per zero).

Interpretazione dell'istruzione di iterazione

Il risultato corrispondente ad una istruzione

(WHILE *bool_expr stmt_block*)

deve essere l'esecuzione di *stmt_block* fintanto che l'espressione *bool_expr* valuta a vero; l'esecuzione non avviene se *bool_expr* è falsa in partenza.

Possono verificarsi i seguenti errori:

- la dichiarazione non è sintatticamente corretta,
- *bool_expr* non è sintatticamente corretta o contiene variabili non definite in precedenza, oppure
- lo *stmt_block* non è sintatticamente corretto, oppure, se eseguito, contiene un errore semantico (ad e.s., divisione per zero).

Interpretazione di un blocco di istruzioni

Il risultato corrispondente ad una istruzione

(BLOCK *statement*₁ ... *statement*_n)

deve essere l'esecuzione **in sequenza** dei diversi statement.

Possono verificarsi i seguenti errori:

- la dichiarazione non è sintatticamente corretta,
- per qualche *i*, *statement*_{*i*} non è sintatticamente corretto, contiene variabili non definite in precedenza, oppure contiene un errore semantico (ad e.s., divisione per zero).

Interpretazione di espressioni

- Le espressioni numeriche vengono interpretate con l'ovvia semantica per valori ed operatori: somma per ADD, sottrazione per SUB, ecc.
- Nelle espressioni booleane, gli operatori relazionali hanno la seguente semantica:
 - ▶ GT sta per “greater than”, quindi $(GT\ a\ b)$ è vero se il valore di a è strettamente maggiore di quello di b ;
 - ▶ Analogamente LT sta per “lesser than” e EQ sta per “equal” con il relativo significato inteso.
- Gli operatori booleani AND, OR, NOT hanno anch'essi l'ovvia semantica; AND e OR sono **cortocircuitati**, ossia
 - ▶ $(a\ AND\ b)$ nel caso in cui a valuti a falso è immediatamente valutata a falso (b non viene valutato);
 - ▶ $(a\ OR\ b)$ nel caso in cui a valuti a vero è immediatamente valutata a vero (b non viene valutato).

Osservazioni aggiuntive

- La grammatica **non permette** la definizione di variabili che non siano di tipo numerico: SET richiede una *num_expr*.
- Nel caso di INPUT viene imposto il vincolo che il valore letto sia un intero.
- Esiste un unico ambito globale per le variabili.
- Le variabili sono definite dinamicamente (non è necessario dichiararle prima di utilizzarle per la prima volta come in C/C++/Java).

Esempio di esecuzione

Ad esempio, in corrispondenza del programma visto prima, l'interazione con l'utente sarebbe il seguente:

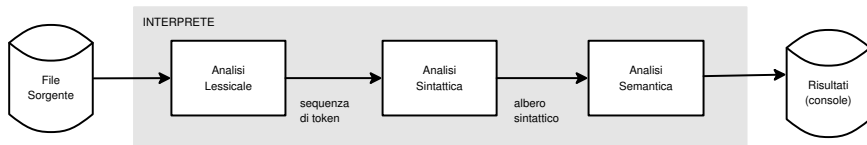
> 5

120

Nota: Come ulteriore semplificazione, si assuma che gli interi forniti in ingresso siano sempre rappresentabili in registri a 64 bit (gli `int64_t` del C++), ossia per qualsiasi numero n utilizzato si ha che $-2^{63} \leq n < 2^{63}$.

Nota: Non è necessario gestire condizioni di overflow o underflow nelle operazioni aritmetiche, mentre è necessario gestire la divisione per zero.

Architettura funzionale



Analisi lessicale (1)

- Lettura del sorgente fornendo in uscita la sequenza degli **elementi lessicali** (**parole** o *token*) del programma.
- Formalmente, per la grammatica dei file sorgente, un elemento lessicale corrisponde ad uno dei seguenti **gruppi**:
 - ▶ una **parola chiave**: BLOCK, INPUT, PRINT, SET, WHILE, IF, GT, LT, EQ, AND, OR, NOT, TRUE, FALSE, ADD, SUB, MUL, DIV;
 - ▶ una **parentesi** aperta o chiusa;
 - ▶ un **numero** (definito con le regole *number*);
 - ▶ un **variabile** (definita con le regole *variable_id*);
- I token corrispondono ai **simboli terminali** di una versione “astratta” della grammatica che descrive le espressioni.

Grammatica astratta

L'utilizzo dei token corrisponde a utilizzare per l'analisi sintattica le seguenti regole (in grassetto i simboli terminali):

$$\text{program} \rightarrow \text{stmt_block}$$
$$\text{stmt_block} \rightarrow \text{statement} \mid (\text{ **BLOCK** statement_list })$$
$$\text{statement_list} \rightarrow \text{statement statement_list} \mid \text{statement}$$
$$\text{statement} \rightarrow \text{variable_def} \mid$$
$$\text{io_stmt} \mid$$
$$\text{cond_stmt} \mid$$
$$\text{loop_stmt}$$
$$\text{variable_def} \rightarrow (\text{ **SET** variable_id num_expr })$$
$$\text{io_stmt} \rightarrow (\text{ **PRINT** num_expr }) \mid (\text{ **INPUT** variable_id })$$
$$\text{cond_stmt} \rightarrow (\text{ **IF** bool_expr stmt_block stmt_block })$$
$$\text{loop_stmt} \rightarrow (\text{ **WHILE** bool_expr stmt_block })$$

Grammatica astratta (segue)

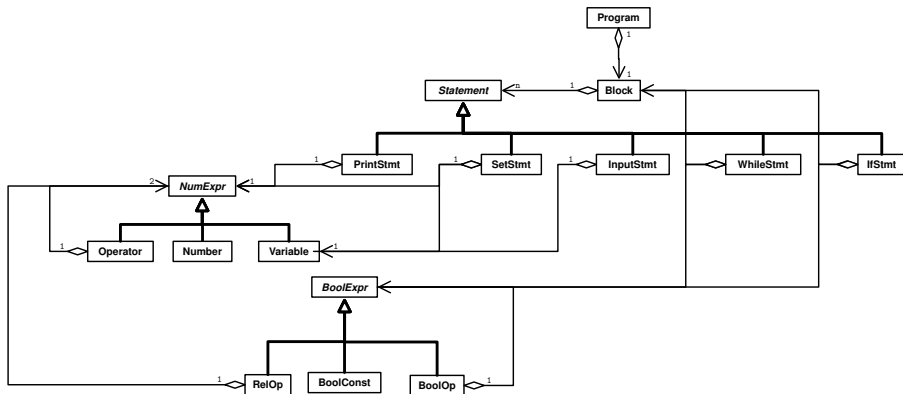
num_expr → (**ADD** *num_expr* *num_expr*)
 | (**SUB** *num_expr* *num_expr*)
 | (**MUL** *num_expr* *num_expr*)
 | (**DIV** *num_expr* *num_expr*)
 | **number**
 | **variable_id**

bool_expr | (**LT** *num_expr* *num_expr*)
 | (**GT** *num_expr* *num_expr*)
 | (**EQ** *num_expr* *num_expr*)
 | (**AND** *bool_expr* *bool_expr*)
 | (**OR** *bool_expr* *bool_expr*)
 | (**NOT** *bool_expr*)
 | **TRUE** | **FALSE**

Analisi sintattica

- Legge la **sequenza di token** fornita dall'analizzatore lessicale e ne controlla la **correttezza sintattica**.
- Fornisce in uscita una descrizione sotto forma di albero del programma in ingresso (c.d. **albero sintattico**)

Classi di supporto all'analisi sintattica



Analisi semantica

- Prende in input l'albero sintattico
- Crea una tabella dei simboli per memorizzare le variabili e i valori ad esse associati
- Crea degli opportuni registri per la valutazione delle espressioni numeriche e booleane (Suggerimento: considerare un'architettura a stack)
- Valuta l'albero sintattico (interpreta il programma) utilizzando una visita.
- Per eseguire la visita si può utilizzare un oggetto "Visitor" con opportuni metodi che consentano l'analisi dei vari elementi degli oggetti "Interpreter".

- Queste slide
- Esempi di programmi di input e relativi output
- Codice C++ di esempio per costruzione e visite di alberi

Specifiche di input/output

- L'interprete prende in input **un parametro da linea di comando** che corrisponde al nome (completo di percorso) del file contenente il programma da interpretare
- Il programma richiede eventuali input e presenta eventuali output in console

Specifiche di consegna

- Standard C++17
- Evitare estensioni specifiche di una IDE o di un sistema operativo
- Consegna dei sorgenti (header `.h` e file `.cpp`) e del diagramma UML dell'applicazione su Aulaweb (singolo file `.zip`)
- Il file `.cpp` contenente il `main` deve chiamarsi `lispInterpreter`
- Utilizzare le guardie di inclusione invece delle direttive `pragma`; ad esempio, il file `Header.h` va strutturato come:

```
#ifndef HEADER_H  
#define HEADER_H
```

```
<corpo del file Header.h>
```

```
#endif
```

- Il progetto fornisce un punteggio massimo di 10 punti così ripartiti:
 - ▶ Correttezza sintattica 4/10: il programma compila senza errori, prende in input il file da linea di comando, genera output su console, non va in loop o in crash
 - ▶ Correttezza su test 4/10: il programma esegue correttamente i test forniti (2/10) e una serie di test ulteriori predisposti in fase di correzione (2/10)
 - ▶ Struttura e documentazione 2/10: il diagramma UML è aderente all'implementazione, il codice sorgente è organizzato e commentato chiaramente
- Il progetto è **individuale**: l'eccessiva similitudine tra diversi progetti darà luogo a penalizzazioni sul punteggio finale.