Linguaggi di Programmazione

Luca Tagliavini

25 febbraio 2022

Indice

Capitolo 1

Introduzione

1.1 Storia dei linguaggi

Nei primi anni 40-50 i linguaggi erano molto vicini alla macchina, in quanto l'obbiettivo era ottimizzare il numero di operazioni svolte dal calcolatore e non agevolare il lavoro dello sviluppatore.

Successivamente si sono imposti alcuni linguaggi che hanno portato idee che troviamo ancora oggi nei linguaggi moderni:

1. **FORTRAN**(Bakus): primo linguaggio che contiene espressioni aritmetiche.
2. **ALGOL**(Naur): primo linguaggio generico, pensato per girare su più architetture e scrivere codice astratto come pseudocodice.
3. **LISP**(McCarty): primo prototipo (involontario) di linguaggio funzionale, che sfrutta ampiamente espressioni (S-Expression).
4. **COBOL** e **ALGOL 68**: linguaggio specifici per memorizzazione di dati e il primo linguaggio con strutture a blocchi.
5. **SIMULA**(Nygaard): primo linguaggio con classi e oggetti.
6. **Pascal**(Wirth): linguaggio volto all'insegnamento ed il primo che si è posto di risolvere veramente il problema della portabiltà usando un bytecode itermedio (simile a Java).
7. **C**(Ritchie e Kernigan): linguaggio favorito ancora oggi per la programmazione di sistemi e inizialmente pensato per UNIX.
8. **Prolog**: primo linguaggio logico.
9. **SmallTalk**: primo linguaggio ad usare classi e oggetti.
10. **PostScript**: linguaggio per descrivere documenti (PDF).

11. **ADA**: linguaggio che punta alla scrittura di software *real-time* pensato per grandi team di sviluppatori.
12. **C++**: ew.
13. **Java**: primo linguaggio altamente portatile ed il primo ad essere usato ampiamente per lo sviluppo web.
14. **Perl**: linguaggio di scripting di successo su piattaforme UNIX.
15. **HTML** e **XML**: per descrivere dati strutturati.
16. **Post-2000**: linguaggi sviluppati per descrivere servizi web.

1.2 Vari paradigmi

Nella letteratura classica esistono quattro tipi fondamentali di pradigmi per linguaggi di programmazione:

- **Iperativi**: sono basati sulla nozione di stato, assegnando variabili, nomi, a celle di memoria e alterandone i valori tramite istruzioni. Esempi celebri sono C, Pascal, FORTRAN.
- **Dichiarativa**: le istruzioni sono dichiarazioni di nuovi valori costanti o ottenuti tramite la combinazione di altri valori. Esempi:
- **Funzionale**: il risultato del programma è il valore restituito da una funzione in base ai parametri di input. Viene ampiamente usata la ricorsione.
- **Logici**: sono basati su relazioni. Si descrivono fatti noti, che possono essere pesati come assiomi, e si compone il risultato finale tramite le relazioni (implicazioni) che legano i valori.
- **Ad oggetti**: sono basati sul paradigma imperativo ma offrono strutture in cui incapsulare funzioni strettamente connesse con relativi dati. Esempi famosi sono: Java, C++ e Smalltalk.

1.2.1 Esempio imperativo (C)

```
int fact(int n) {
  int res = n;
  while(n--)
    res *= --n;
  return res;
}
```

1.2.2 Esempio funzionale (SCHEME)

```
(define (fact n)
  (cond [(= n 0) 1]
  [else (* n (fact (- n 1)))]
  )
)
```

1.3 Confronti generali tra linguaggi

Esistono una serie di caratteristiche intrinseche ovvero dipendenti dalla sua struttura:

1. **Espressività**: è touring completo o no? può risolvere ogni tipo di problema?
2. **Didattica**: quanto è semplice la sintassi per un novizio.
3. **Leggibilità**: quanto è semplice da leggere e comprendere l'intento dello scrittore.
4. **Robustezza**: quanto può prevenire errori classici.

Capitolo 2

Macchine astratte

2.1 Macchine astratte e linguaggi

Una macchina astratta di *Von Neumann* è una semplice macchina di Touring che è composta da un processore, una memoria interna ed esterna e una interfaccia alle periferiche. Tutti questi componenti comunicano tramite un *bus*. Lo scopo della macchina è eseguire un ciclo ripetitivo (Fetch-Decode-Execute) che sfrutta le operazioni native dell'hardware per il quale è stata progettata.

Una macchina fisica esiste per eseguire il **suo** linguaggio. Ad ogni macchina corrisponde un linguaggio, *ma* uno stesso linguaggio può essere supportato da più architetture.

In una macchina astratta invece si ha un solo **interprete**, che svolge lo stesso ruolo del ciclo *Fetch-Decode-Execute* e si occupa essenzialmente di eseguire i programmi, che possono essere scritti in un linguaggio astratto dall'implementazione hardware specifica.

2.1.1 Interprete

L'interprete è in grado di eseguire operazioni su dati primitivi, controlli di sequenza come salti e incrementi del Program Counter, operazioni per il trasferimento dei dati tra le procedure/funzioni ed infine metodologie per la scrittura in memoria principale/secondaria.

2.1.2 Definizione formale

Sia M una qualunque macchina astratta, che comprende ed esegue il *linguaggio macchina* L_M. Il linguaggio L_M può essere rappresentato in plain text sotto linguaggio astratto di programmazione, oppure come celle di memoria in cui vengono contenuti i valori binari corrispondenti alle istruzioni del programma.

La macchina hardware come Macchina Astratta

Una CPU e i suoi relativi componenti possono essere visti come una macchina astratta: Le *operazioni primitive* sono quelle offerte dal set di istruzioni (CISC o RISC), i *controlli di sequenza* tipicamente sono i salti e le chiamate di subroutine, il *controllo dati* è tipicamente svolto tramite accessi diretti in memoria o con una struttura a stack e ciò varia il tipo di *gestione di memoria*.

Spesso accade per strutture più complesse come la RISC che la macchina hardware stessa sia microprogrammata. Ossia la Macchina Hardware offre un limtato set di operazioni base, che vengono sfruttate da una macchina astratta che implementa tutte le operazioni possibili per la specifica dell'architettura. Il ciclo FDE non è dunque realizzato in hardware, il quale offre solo soluzioni di bassissimo livello che vengono sfruttate da un μ interprete.

2.1.3 Nesting

Si possono dunque innestare più macchine astratte (viene fatto nella pratica) in modo da offrire sempre più funzionalità appoggiandosi sempre sulla macchina astratta del livello sottostante. I sistemi operativi spesso implementano loro stessi una macchina astratta che è in grado di eseguire il linguaggio specifico per gli esegubili di quel SO.

Notazioni matematiche

Una funzione $f : A \to B$ mappa un qualunque elemento $a \in A$ ad un solo $b \in B$. Una funzione $f : A - o \to B$ può essere non definita su un qualche $a \in A$.

2.2 Implementare un Linguaggio

Viene dato un linguaggio $\mathcal{L}$ da implementare su una macchina astratta $Mo_{\mathcal{L}o}$ che esegue un suo linguaggio $\mathcal{L}o$. Si vuole implementare $\mathcal{L}$ su $Mo_{\mathcal{L}o}$.

$P_r^{\mathcal{L}}$ indica un programma scritto nel linguaggio $\mathcal{L}$. A $P_r^{\mathcal{L}}$ viene associata una funzione *parziale* $P^{\mathcal{L}}$ che rappresenta la semantica di $P_r^{\mathcal{L}}$

2.2.1 Esempio di semantica di un programma sequenziale

```
read(x)
if(x == 1)
  return x
else
  while true
  done
```

Ha una funzione semantica $P^{\mathcal{L}}(x) = \begin{cases} 1 & \text{se } x = 1 \\ \text{indefinito} & \text{altrimenti} \end{cases}$

2.2.2 Approccio dell'interprete

La nostra macchina astratta $M_{\mathcal{L}}$ può essere un *interprete* per $\mathcal{L}$ su $Mo_{\mathcal{L}o}$ che dunque traduce il programma $\mathcal{L}$ in una funzione semantica $P^{\mathcal{L}}$ che calcola l'esatto risultato del programma dato un qualunque input. In altre parole l'interprete *calcola la corretta semantica.*

2.2.3 Approccio del compilatore

I programmi scritti in $\mathcal{L}$ sono invece *tradotti* dalla macchina in programmi nel linguaggio $\mathcal{L}o$. La macchina prende notazione $C^{\mathcal{L}a}_{\mathcal{L},\mathcal{L}o}$.

2.2.4 Deifnizioni formali

Un interprete di un linguaggio $\mathcal{L}$ scritto in $\mathcal{L}o$ è un programma che realizza una funzione parziale:

$$\mathcal{I}^{\mathcal{L}o}_{\mathcal{L}}(Prog^{\mathcal{L}} \times D) \to D \text{ tale che } \mathcal{I}^{\mathcal{L}o}_{\mathcal{L}}(P^{\mathcal{L}}_r, Input) = P^{\mathcal{L}}(Input)$$

Un compilatore da $\mathcal{L}$ a $\mathcal{L}o$ è un programma che realizza una funzione:

$$\mathcal{C}_{\mathcal{L},\mathcal{L}o} : Prog^{\mathcal{L}} \to Prog^{\mathcal{L}o}$$

tale che, dato un programma $P^{\mathcal{L}}_r$ se

$$C_{\mathcal{L},\mathcal{L}o}(P^{\mathcal{L}}_r) = P_r c^{\mathcal{L}o}$$

allora per ogni $Input \in \mathcal{D}$:

$$P^{\mathcal{L}}(Input) = Pc^{\mathcal{L}o}(Input)$$

2.2.5 Compilatori vs Interpreti

Gli interpreti sono estremamente più versatili al costo delle performance. Infatti essi devono reinterpretare ogni istruzione ogni volta, mentre il compilatore fa il processo una sola volta al *compile time*. Compilando non si ha dunque il costo della decodifica ad ogni istruzione al tempo dell'esecuzione, tuttavia si ha una scarsa tracciabilità del codice tradotto. Risalire all'istruzione del codice sorgente è molto arduo: infatti molti debugger per linguaggi compilati sono effettivamente degli interpeti per quelli stessi linguaggi (in un interprete è triviale capire a quale riga avviene un errore).

2.2.6 Compilazione & Interpretazione

Nel caso reale moltissimi linguaggi **non** sono *puramente* compilati o interpretati, bensì si usa un'approccio ibrido. Ad esempio alcune istruzioni come quelle di input/output sono quasi sempre *simulate* (interpretate) o si preferisce sfruttare una macchina astratta intermedia che esegue un codice intermedio tra quello ad alto livello $\mathcal{L}$ e quello della macchina ospite $\mathcal{L}o$.

La scelta di cosa va compilato e cosa va interpretato segue il seguente criterio: Tradurre le istruzioni che sono più simili a quelle offerte nativamente dalla macchina Mo. Un approccio maggiormente interpretativo favorirà la flessibilità e la portabilità mentre una compilativa favorisce l'ottimizzazione.

Capitolo 3

Linguaggi

3.1 Parti di un linguaggio

La descrizione di un linguaggio è suddivisa in tre parti:

1. **Sintassi**: regole di formulazione, quando una frase è corretta.
2. **Semantica**: attribuzione di un significato alle frasi.
3. **Pragmatica**: in quale modo le frasi corrette sono usate (i.e. vari tipi di indirizzamento)
4. **Implementazione** (solo per linguaggi di programmazione eseguibili)

3.1.1 Sintassi

La sintassi definisce l'insieme di parole (*keyword*) che possono essere usate (aspetto lessicale) ed anche quando un insieme di parole corrette è sintaticamente corretto, nel senso che rispettano tutte le regole grammaticali definite (i.e. il `.` va usato solo e soltanto dopo un identifier).

3.1.2 Semantica

Per assegnare un significato a frasi sintatticamente corrette abbiamo bisogno di un contesto (ad esempio quale linguaggio stiamo analizzando, latino o italiano). È importante basarsi su un lingiaggio/concetti noti, per dare un senso a ciò che è scritto.

3.1.3 Pragmatica

Buone pratiche, ottimizzazioni da applicare nel linguaggio. Ad esempio usare il "lei" e non il "tu" o non sfruttare il `goto`.

3.1.4 Implementazione

Esegue una frase sintatticamente corretta rispettandone la semantica.

3.2 Linguaggio Formale

1. Un **alfabeto** A è un insieme finito i cui elementi sono detti *simboli*.
2. Una **parola** su un alfabeto A è una sequenza finita di simboli di A.
3. Un **linguaggio formale** L su A è un insieme di parole su A.

Si può dire che un linguaggio formale $L \subseteq A^*$ dove

$$\begin{aligned} A^* &= \cup_{n\geq 0} A^n \text{ dove } A^0 = \{\varepsilon\} \\ A^{n+1} &= A \cdot A^n \quad n \geq 0 \quad \text{dove} \\ A \cdot A^n &= \{aw \mid a \in A, w \in A^n\} \quad \text{(concatenamento)} \end{aligned}$$

Si può osservare che A^* è un insieme *finito contabile*.

3.2.1 Notazioni e definizioni

La *lunghezza* di una parola è denotata come:

$$|\varepsilon| = 0 \qquad |a \cdot w| = 1 + |w| \quad \forall a \in A$$

La stringa x *concatenata* a y (xy o $x \cdot y$) è ottenuta giustapponendo le due stringhe:

$$w = xy \iff \begin{cases} |w| = |x| + |y| & \\ w[j] = x[j] & \text{se } 1 \geq j \geq |x| \\ w[|x| + j] = y[j] & \text{se } 1 \geq j \geq |y| \end{cases}$$

Una stringa v si dice *sottostringa* di w sse:

$$\exists x, y \in A^*. \quad w = x \cdot v \cdot y$$

v è un suffisso di w sse $\exists x \in A^*.w = xv$.
v è un prefisso di w sse $\exists x \in A^*.w = vx$.
La *potenza n-esima* di una stringa w è definita come:

$$w^0 = \varepsilon \qquad w^{n+1} = w \cdot w^n \qquad \forall n \geq 0$$

3.2.2 Linguagi su alfabeti & definizioni

Un *linguaggio* L su un alfabeto A è un oggetto del tipo $L \subseteq A^*$, dove si possono imporre ulteriori regole logiche tramite l'assioma di separazione.

Un complemento di un linguaggio è definito come:

$$\overline{L} = \{w \in A^* \mid w \notin L\} = A^* \setminus L$$

L'unione e l'intersezione sono ovvie:

$$L_1 \cup L_2 = \{w \mid w \in L_1 \vee w \in L_2\}$$
$$L_1 \cap L_2 = \{w \mid w \in L_1 \wedge w \in L_2\}$$

Infine si ha la concatenazione:

$$L_1 \cdot L_2 = \{w_1 w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}$$

Una *potenza* di un lingiaggio è definita da:

$$L^0 = \{\varepsilon\} \quad L^{n+1} = L \cdot L^n \quad \forall n \geq 0$$

La chiusura/stella di Kleene/ripetizione di un linguaggio è definita come:

$$L^* = \cup_{n \geq 0} L^n$$
$$L^+ = \cup_{n \geq 1} L^n \quad (chiusurapositiva)$$

3.2.3 Memorizzazione dei linguaggi

I linguaggi fino ad ora definiti sono possibilmente insiemi *infiniti* contabili che non possono dunque essere salvati in nessuna memoria. Per poterli descrivere e memorizzare si usano dunque gli stessi approcci matematici (indiretti) con cui si generano ad esempio i numeri naturali: ad esempio secondo Peano i $\mathbb{N}$ sono 0 per definizione e ogni successore di un numero naturale $x \in N \Rightarrow s(x) \in N$.

Ci sono due modi per generare/identificare un valido linguaggio:

1. generativo: si generano tutte le possibili parole esistenti nel linguaggio partendo da una serie di regole finite.
2. riconoscitivo: viene memorizzata l'insieme delle stringhe da riconoscere da una struttura detta *automa*.

Bisogna osservare che non tutti i linguaggi possono essere generati da grammatiche o riconosciuti da automi.

3.3 Grammatiche

Esistono svariate tipologie di grammatiche: regolari, libere dal contesto, dipendenti dal contesto, monotone, generali, etc. Tutte seguono lo stesso pattern differenziandosi solo per come sono caratterizzate le *produzioni*. Le grammatiche più utili sono quelle "libere".

3.3.1 Grammatica libera da contesto

Definizione: una *grammatica libera dal contesto* è una quadrupla (NT, T, R, S) dove:

- NT è un insieme finito di simboli non terminali.
- T è un insieme finito di simboli terminali.
- $S \in NT$ è detto simbolo iniziale.
- $R \in NT$ è un insieme *finito* di produzioni o regole di forma. Esempio: $V \to w$ dove $V \in NT$ e $w \in (T \cup NT)^*$.

3.3.2 Derivazioni

Data $G = (NT, T, R, S)$ libera da contesto, diciamo che da v si deriva immediatamante w (o che v si descrive in w) e lo denotiamo come $V \Rightarrow w$, se:

$$\forall x, y, z \in (T \cup NT)^*. \quad v = xAy \land (A \to w) \in R \land w = xzy$$

Diciamo che da v si deriva w (o anche v si riscrive in w) e lo denotiamo con $v \Rightarrow^* w$ se esiste una sequenza finita (o vuota) di derivazioni immediate:

$$v \Rightarrow w_0 \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n \Rightarrow w$$

3.3.3 Linguaggio generato

Il linguaggio generato da una grammatica $G = (NT, T, R, S)$ è l'insieme

$$L(G) = \{w \in T^* \mid s \Rightarrow^* w\}$$

Si noti che w può solo essere un simbolo terminale.

Data la grammatica G come faccio a determinare $L(G)$? E a verificare se una stringa $w \in L(G)$?

Capitolo 4

Alberi sintattici & ambiguità

4.1 Derivazioni ad alberi

Una derivazione è una espansione di una grammatica per un linguaggio, che può essere analizzata in due modi:

1. **Leftmost**: analizza i caratteri a partire dalla parte più a *sinistra* della definizione.
2. **Rightmost**: analizza i caratteri a partire dalla parte più a *sinistra* della definizione.

Esiste una corrispondenza biunivoca tra le derivazioni leftmost e rightmost e gli alberi da essi generati. Le due tecniche generano lo stesso albero ma con le foglie ordinate in modo diverso.

L'albero di derivazione fornisce oltretutto informazioni semantiche: "quali operandi per quali operatori": Infatti vediamo che i nodi terminali che sono opeartori contendono a destra e sinistra gli operandi che possono prendere in input. Oltretutto da un albero di derivazione è molto facile generare un *albero sintattico*.

4.1.1 Albero di derivazione

Definizione: data una grammatica libera $G = (NT, T, S, R)$ un albero di derivazione (o di parsing) è un albero ordinato in cui:

1. Ogni nodo è etichettato con un simbolo in $NT \cup \{\varepsilon\} \cup T$.
2. La radice è etichettata con S.
3. Ogni nodo interno è etichettato con un simbolo NT.

4. Se il nodo ha etichetta $A \in NT$ e i suoi figli sono nell'ordine $n_1, \ldots, n_k$ con etichetta $x_1, \ldots, x_k$ (operandi) allora $A \Rightarrow x_1, \ldots, x_k$ è una produzione in R.

5. Se il nodo ha etichetta ε allora è una foglia e detto A suo padre, $A \Rightarrow \varepsilon$ è una produzione di R.

6. Se inoltre ogni nodo foglia è etichettato su $T \cap \{\varepsilon\}$ allora l'alfabeto corrisponde ad una derivazione completa.

Teorema: una stringa $w \in T^*$ appartiene a $L(G)$ se e solo se ammette un albero di derivazione completo (le cui foglie lette da sinistra a destra diano la stringa w).

4.2 Ambiguità

Definizione: una grammatica libera G è ambigua se $\exists w \in L(G)$ che ammette più alberi di derivazione.
Definizione: un linguaggio L è ambiguo se $\forall G.L(G) = L$ sono ambigue.

Spesso è possibile alterare la definizione della grammatica affinchè non generi più ambiguità.

4.2.1 Risolvere le ambiguità

Esempi di ambiguità possono essere dare precedenza al $\cdot$ rispetto al $+$ in una grammatica per opearzioni matematiche, in modo che esista un solo ed univoco modo per interpretare una eventuale stirnga appartenente a quel linguaggio.
Ad esempio la grammatica ambigua

$$S \to a \mid b \mid c \mid S + S \mid S \cdot S$$

può essere riscritta (complicandola) in

$$\begin{aligned} S &\to S + T \mid T \\ T &\to A \cdot T \mid A \\ A &\to a \mid b \mid c \end{aligned}$$

Si noti che in questa grammatica ho anche espresso la precedenza del $\cdot$ rispetto al $+$ posizionandolo sempre più in basso nell'albero di derivazione.

Tuttavia questa grammatica è meno espressiva, infatti non si possono più esprimere alcuni alberi di derivazione. L'unica alternativa è dunque esprimere una grammatica più ampia dotata anche di terminali, effetivamente inutili, (e) per ottenere una grammatica non ambigua. Questi non terminali *non espressivi* sono chiamati formalmente *zucchero sintattico*. Il linguaggio diventa dunque:

$$\begin{aligned} S &\to S + T \mid T \\ T &\to A \cdot T \mid A \\ A &\to a \mid b \mid c \mid (E) \end{aligned}$$

4.2.2 Costruzione dell'albero astratto

Una volta ottenuto un albero derivato da una grammatica non ambigua si svolge un ulteriore passo volto ad astrarre, ovvero eliminare il superfluo, dove vengono rimossi tutti i terminali di zuccero sintattico e le derivazioni intermedie inutili fino ad ottenere un albero che potrebbe essere modellato dalla grammatica semplice ed intuitiva iniziale. Questo albero finale prende nome di *albero sintattico astratto* (AST).

Capitolo 5

Grammatiche Contestuali

5.1 Vincoli sintattici Contestuali (e semantica statica)

Esistono vincoli sulla grammatica che non possono essere descritti dalle grammatiche libere. Sono detti *semantica statica* poichè possono essere interpretati al compile-time, tuttavia appartengono più alla sintassi che alla semantica. Alcuni esempi sono:

- Una variabile in uso deve prima essere dichiarata
- Compatibilità di un assegnamento, nella verifica dei tipi coerenti.

Questi vincoli sintattici non sono esprimibili con grammatiche libere come BNF, ed esistono due soluzioni a questa mancanza:

1. Usare strumenti più potenti come le *grammatiche dipendenti dal contesto* la cui interpretazione avrebbe tuttavia costo esponenziale.
2. Controlli *ad-hoc* svolti spesso dal compilatore, come type checking, dichiarazione delle variabili nello scope, etc.

5.1.1 Disambiguazione tra sintassi e semantica

La *semantica statica* è impropriamente chiamata, in quanto concerne elementi tipici della sintassi. Tuttavia per motivi storici si intende:

- **Sintassi**: quello che è descritto dalla BNF.
- **Semantica**: tutto il resto.

I vincoli contestuali (non compresi nelle BNF) sono dunque comprensibili al compile-time (dunque statici) ma nel dominio della semantica.

5.1.2 Semantica dinamica

La semantica dinamica fornisce un modello matematico che descrive indipendentemente dall'architettura sottostante il "comportamento" del programma. L'implementazione deve rispettare la specifica descritta sopra.

Un tipico modello funziona tramite un grafo, dove ogni nodo indica uno stato e i rami uscenti indicano le operazion successive, che vanno a modificare questo stato.

Questo modello astratto e puro è utile a varie figure, per varie motivi:

1. Al programmatore: deve sapere esattamente come si comporta il proprio linguaggio, e poter dimostrare matematicamente che il programma è corretto.

2. Al progettista: è la migliore specifica per un qualunque linguaggio e consente di provarne alcune proprietà come la touring-completezza.

3. All'implementatore: è un importante riferimento per chi implementa compilatori/interpreti che possono verificare la correttezza della loro macchina astratta.

5.1.3 Come definire la semantica

Esistono due tecniche per definire la semantica di un linguaggio:

1. **Operazionale**: si costruisce una specie di automa che passo per passo mostra l'esecuzione delle varie istruzioni di cui il programma è composto.

2. **Denotazionale**: si associa ad ogni programma sequenziale una funzione che da input-output e viene verificata la correttezza del risultato.

5.1.4 Altri aspetti di un linguaggio

Oltre alla sintassi e alla semantica esistono altre descrizioni di un linguaggio:

1. **Pragmatica**: un insieme di regole/buone pratiche da seguire quando si utilizza un dato linguagigo. Alcuni esempi possono essere l'evitare il `goto`, non alterare il valore del contatore in un ciclo `for`, e allocare solo la memoria strettamente necessaria.

2. **Implementazione**: ovvero scrivere un compilatore (o interprete) per una macchina ospite già esistente. Per provare che questo tool è corretto bisogna verificare che rispetti la semantica dinamica.

Capitolo 6

Struttura del compilatore

6.1 Le fasi

Il compilatore è un programma che prende in input un programma sorgente e ne genera un codice eseguibile sulla macchina astratta esistente. Questo strumento è suddiviso in una serie di componenti:

1. **Analisi lessicale**: legge il programma parola per parola e popola una tabella di simboli, controllando che i simboli usati nel linguaggio siano keyword corrette. Restituisce una *lista di token*.
2. **Analisi sintattica**: consulta la tabella dei simboli per aggiungerne attributi riguardanti ogni token, e genera un *albero di derivazione* che restituisce.
3. **Analisi semantica**: viene stabilito il significato di ogni istruzione e l'albero ricevuto in input viene arricchito di queste informazioni per formare un *albero di derivazione aumentato*. Questo step esegue anche il controllo della validità dei tipi.
4. **Generatore della forma intermedia**: genera un codice generico (non specifico per la macchina) molto ridondante che rispetta perfettamente la semantica. Questo codice è detto *codice intermedio*.
5. **Ottimizzazione**: il codice intermedio viene ripulito di informazioni inutili e ottimizzato per la migliore esecuzione sulla macchina ospite. Questo step restituisce infine un *codice intermedio ottimizzato*.
6. **Generazione del codice**: il codice intermedio viene tradotto infine in codice nativo che può essere eseguito dalla macchina padre.

6.1.1 Analisi lessicale (scanner)

Spezza il programma nei componenti sintattici primitivi detti *token*. Nel mentre controlla che il lessico sia corretto e riempie la tabella dei simboli. Per fare ciò

vengono usate le *grammatiche regolari*, *espressioni regolari* e infine gli *automi a stati finiti* (NFA, DFA).

6.1.2 Analisi sintattica (parser)

Organizza i token in un albero di derivazione riconoscendo se esso è sintatticamente corretto. Per il funzionamento useremo le *grammatiche libere dal contesto* e *automi a pila* (DPDA).

6.1.3 Analisi semantica

Arricchisce l'albero generato con infromazioni sui tipi, che vengono poi verificati ed eventualmente vengono lanciati errori. Vengono anche controllati i parametri formali, le dichiarazioni, etc.

Capitolo 7

Semantica Operazionale Strutturata

7.1 Linguaggio

Questo linguaggio sarà definito tramite una sintassi astratta e semplice ma ambigua. Una stringa viene sempre accoppiata ad un albero sintattico.

I dati di base sono i valori:

1. Booleani: $\{tt, ff\}$ che corrispondono ai metabooleani $true$ e $false$.
2. Naturali: $\{0, 1, \ldots\}$ che corrispondono a $m, n, \ldots \in \mathbb{N}$.
3. Variabili: $\{a, b, c, \ldots\}$ che corrispondono alle metavariabili $v \in V$.

Una sintassi BNF che descrive questo linguaggio (ambigua) può essere quella che segue:

- Espressioni Aritmetiche: $e \in Exp$
  $e \to m \mid v \mid e + e \mid e - e \mid e \cdot e$
- Espressioni Booleane: $b \in Bexp$
  $b \to tt \mid e = e \mid b \vee b \mid \neg b$
- Comandi: $c \in Com$
  $c \to ski[\mid v := e \mid c; c \mid whilebdoc \mid ifbthencelsec$

7.2 Semantica

Dobbiamo dunque dare una semantica agli alberi generati da questa grammatica. Useremo un *sistema di transizione*, ovvero una tripla $(\Gamma, T, \to)$ dove

1. Γ è l'insieme possibilmente infinito degli stati

2. $T \subseteq \Gamma$ è l'insieme degli stati terminali

3. $\rightarrow \subseteq \Gamma \times \Gamma$ è la relazione di transizione

Una computazione a partire da uno stato σ_0 è una seguenza $\gamma_0 \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \ldots$ che può essere finita o infinita.

Con $\gamma \rightarrow^* \gamma''$ indichiamo la chiusura riflessiva e transitiva di $\rightarrow$, ovvero quando da $\gamma \rightarrow \gamma'$ e $\gamma' \rightarrow \gamma''$.

Problemi della semantica

1. Γ solitamente è infintamente contabile, dunque si deve trovare un modo di rappresentarla in modo finito tramite una BNF. (ad esempio $\Gamma_e = \{\langle e, \sigma\rangle \mid e \in Expr, \sigma \in Store\}$

2. $\rightarrow \subseteq \Gamma \times \Gamma$ è una relazione costituita di solito da coppie $\gamma \rightarrow \gamma'$ e si deve dunque trovare un modo di rappresentare in modo finito solo le regole non deducibili, ovvero gli assiomi e le regole d'inferenza.

3. Per dare valore alle variabili si ha bisogno di uno **store** $\sigma : Var \rightarrow \mathbb{N}$ come funzione che associa ad ogni variabile un valore.

7.2.1 Osservazioni

Gli assiomi e le regole di un linguaggio aritmetico come quello descritto sono facilmente implementabili in un linguaggio logico come *prolog*. Si potrebbe dunque creare un interprete in modo facile, anche se poco performante.

7.2.2 Transizione deterministica

Vogliamo dimostrare che $\rightarrow_e$ è deterministica, ovvero che se $\gamma \rightarrow \gamma'$ e $\gamma \rightarrow \gamma''$ allora $\gamma' = \gamma''$, $\forall \gamma, \gamma', \gamma''$.

Vedi appunti prof.

7.2.3 Valutazione

Sapendo che $\rightarrow_e$ è deterministica posso allora definire la funzione $eval : Expr \times Store \rightarrow \mathbb{N}$ **parziale** poichè non sempre restituisce un valore (se si incontra un errore). La possiamo implementare come seuge:

$$eval(e, \sigma) = \begin{cases} m & \text{se } \langle e, \sigma\rangle \rightarrow^* \langle m, \sigma\rangle \\ indefinita & \text{altrimenti} \end{cases}$$

Si ricordi che la funzione *eval* è definita rispetto alla chiusura IS (interna sinistra). Si può provare che per la grammatica interpretata la $eval_{IS} = eval_{ID}$. Esistono tuttavia le ED, ES, EP (E = esterna, S = sinistra, D = destra, P = parallela) che sono tuttavia diverse da $ID = IS$

Capitolo 8

Linguaggi regolari

Lo scopo dei linguaggi regolari è quello di riconoscere nella stringa d'ingresso pattern regolari o espressioni ricorrenti.

8.1 Token

Un *token* è solitamente una coppia $(nome, valore)$ dove il nome è il simbolo astratto che rappresenta una categoria sintattica, mentre il valore è una sequenza di simboli del testo in ingresso.

Dato il token (Ide, x_1) diciamo che:

1. **Nome**: Ide è l'informazione che identifica il tipo di token.
2. **Valore**: x_1 è l'informazione che identifica lo specifico token.
3. **Pattern**: è la descrizione generale della forma dei valori di una classe di token. Ad esempio `/(x|y)(x|y|0|1)/`
4. **Lessema**: è una stringa istanza di un pattern. Nell'esempio x_1 è un lessema istanza del pattern `/(x|y)(x|y|0|1)/`.

Normalmente lo scanner associa agli identificatori un valore nella tabella dei simboli, dunque il *valore* sarebbe il puntatore alla riga della tabella in cui quel valore è contenuto.

8.2 Espressioni regolari

Fissato un alfabeto $A = \{a_1, a_2, \ldots, a_n\}$ definiamo le espressioni regolari su A con la seguente BNF:

$$r \to \emptyset \mid \varepsilon \mid a \mid r \cdot r \mid r|r \mid r^* \qquad \forall a \in A$$

Questa sintassi è estremamente ambigua, ma possiamo disambiguarla usando le parentesi e assunzioni di precedenza come che tutti gli operatori associano

a sinistra e che la precedenza tra gli operatori sia $* > \cdot > |$. Si noti che la concatenazione è solitamente omessa.

8.2.1 Denotazione

Dato l'alfabeto A, definiamo la funzione

$$\mathcal{L} : ExprReg \to P(A^*)$$

come segue:

$$\begin{aligned}
\mathcal{L}[\emptyset] &= \emptyset \quad \text{(linguaggio vuoto)} \\
\mathcal{L}[\varepsilon] &= \{\varepsilon\} \quad \text{(linguaggio con solo stringa vuota)} \\
\mathcal{L}[a] &= \{a\} \\
\mathcal{L}[r_1 \cdot r_2] &= \mathcal{L}[r_1] \cdot \mathcal{L}[r_2] \\
\mathcal{L}[r_1 \cdot r_2] &= \mathcal{L}[r_1] \cup \mathcal{L}[r_2] \\
\mathcal{L}[r^*] &= (\mathcal{L}[r])^*
\end{aligned}$$

8.2.2 Linguaggio regolare

Un linguaggio è detto *regolare* sse esiste una espressione regolare r tale che $L = \mathcal{L}[r]$.

Proprietà: ogni linguaggio finito è anche regolare.

8.2.3 Altri operatori

Esistono altri operatori di comodità per semplificare la sintassi:

$$\begin{aligned}
\mathcal{L}[r^+] &= \mathcal{L}[rr^*] \vee \mathcal{L}[r^*r] && \text{ripetizione positiva} \\
\mathcal{L}[r^?] &= \mathcal{L}[r|\varepsilon] && \text{possibilità} \\
\mathcal{L}[[a_1, \ldots, a_n]] &= \mathcal{L}[a_1|a_2|\ldots|a_n] && \text{elenco} \\
\mathcal{L}[[a_i - a_n]] &= \mathcal{L}[a_i|a_{i+1}|\ldots|a_n] \quad i < n && \text{intervallo}
\end{aligned}$$

8.2.4 Definizioni regolari

Una definizione regolare su alfabeto A è costituita da una lista di definizioni

$$\begin{aligned}
d_1 &= r_1 \\
d_2 &= r_2 \\
&\vdots \\
d_k &= r_k
\end{aligned}$$

dove i d_i sono valori nuovi, e possono dunque essere usati per estendere l'alfabeto originale A in $A \cup \{d_1, \ldots, d_k\}$.

Esempio di definizione regolare

$$\begin{aligned} number &\rightarrow segno \cdot cifre(.cifre)? \\ segno &\rightarrow +|- \\ cifra &\rightarrow [0-9] \\ cifre &\rightarrow cifra^{+} \end{aligned}$$

8.3 Automi a Stati Finiti (FSM)

Un automa a stati finiti è una macchina con memoria finita che contiene un dato numero di stati tra cui si può muovere in base a date regole. La macchina ha un solo bit in output (booleano) che indica se una stringa viene riconosciuta o meno.

Inizialmente la macchina viene posizionata sul primo carattere dell'input e viene svolto un controllo iniziale sullo stato q_0. Per il funzionamento si ripete un ciclo che svolge le seguenti azioni:

- Leggi il carattere di input. In base allo stato in cui ci si trova si
  - Si cambia stato a qello successivo
  - Sposta la testina sull'input successivo
- **fino a quando**:
  - Ho finito di leggere l'input ed eventualmente ho riconosciuto la stringa sse sono in uno stato finale.
  - Mi sono bloccato prima perchè la coppia $(stato\_corrente, input\_attuale)$ non è specificato in uno stato successivo.

Gli automi sono rappresentati molto bene dai diagrammi di transizione di stati.

8.3.1 Automi Finiti Non-deterministici (NFA)

Un NFA è una quintupla (Σ, Q, S, q_0, F) dove

- Σ è un alfabeto finito di simboli.
- Q è l'insieme finito di stati.
- $q_0 \in Q$ è lo stato iniziale.
- $F \subseteq Q$ è l'insieme degli stati finali.
- S è la funzione di transizione con tipo:

$$S : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q)$$

8.3.2 Linguaggio Riconosciuto

Un NFA $N = (\Sigma, Q, S, q_0, F)$ accetta una stringa $w = a_1 \dots a_n$ sse nel diagramma di transizione **esiste** un cammino da q_0 ad uno stato $\in F$ nel quale la stringa che si ottiene concatenando le etichette degli archi percorsi è esattamente uguale a w.

Si può mostare che per un dato automa NFA ne esistono infiniti analoghi (basata pensare che aggiungendo transizioni inutili si cambia il tipo di automa, si preserva il linguaggio usato e se ne possono aggiungere quante se ne vuole).

8.3.3 DFA

I NFA sono comodi, facili da definire e intuitivi, tuttavia sono *inefficienti*. Al fine di colmare questa mancanza vengono introdotti i DFA, ovvero *deterministic Finite Automata*. Eccone la definizione formale:

Un automa finito deterministico è una quintupla $(\Sigma, Q, \delta, q_0, F)$ dove Σ, Q, q_0, F sono analoghi agli NFA, mentre la funzione δ ha tipo $\delta : Q \times \Sigma \to Q$.

Osservazione: NFA vs DFA

I DFA sono in realtà un sottoinsieme degli NFA tali per cui:

- $\forall a \in Q.\delta(q, \varepsilon) = \emptyset$ (non ci sono transizioni ε
- $\forall G \in \Sigma.\forall q \in Q.\exists q' \in Q.\delta(q, \sigma) = \{q'\}$ sempre solo una mossa possibile

8.3.4 I DFA sono tanto espressivi quanto gli NFA

Si può provare la proprietà secondo la quale ogni DFA è espressivo tanto quanto ogni altro NFA. Si può provare a mostrarlo in modo intuitivo nel seguente modo:

1. Seguendo **contemporaneamente** tutti i possibili cammini (alternativi o nulli ε) del'NFA.
2. Gli stati del DFA che andrmeo a costruire saranno fortmati da parte degli stati degli NFA.

Questa tencica è la così detta *subset constuctin*.

Teorema: Equivalenza DFA e NFA

Guarda i suoi appunti. Non viene chiesto all'orale.

8.3.5 Teorema: Da espressioni regonali a NFA equivalenti

Data una espresione regolare s, possiamo costruire un NFA $N[s]$ tali che

$$\mathcal{L}[s] = L[N[s]]$$

Dunque gli NFA riconoscono *tutti* i linguaggi regolari (poichè composti da espressioni regolari)

Dimostrazione per induzione sulla struttura della espressione regolare s. Costruiremo $N[s]$ cioè un possibile NFA associato alla espressione regolare s in modo da mantenere i seguenti dure invarianti:

1. lo stato iniziale non ha archi entranti
2. $N[s]$ ha un solo stato finale senza archi uscenti

Ecco i vari casi:

1. $s = \emptyset$: $L[\emptyset] = \emptyset = L[N[s]]$
2. $s = \varepsilon$: $L[\varepsilon] = \{\varepsilon\} = L[N[s]]$
3. $s = a$: $L[a] = \{a\} = L[N[a]]$
4. $s = r \mid l$: $L[r \mid l] = L[r] \cup L[l] = L[N[r]] \cup L[N[s]] = L[N[r \mid l]]$
5. $s = r \cdot l$: $L[r \cdot l] = L[r] \cdot L[l] = L[N[r]] \cdot L[N[s]] = L[N[r \cdot l]]$
6. $s = r^*$: $L[r^*] = (L[r])^* = (L[N[r]])^* = L[N[r^*]]$

□

8.3.6 Teorema: grammtica regolare in NFA

Data una grammatica regolare G si può sempre costruire un NFA N_G equivalente. **Dimostrazione**: Sia $G = (NT, T, R, S)$ allora $N_G = (T, Q, \delta, S, \{\varepsilon\})$ è definito come segue:

1. $Q = NT \cup \{\varepsilon\}$
2. δ è definita come segue: (copia)

Capitolo 9

Proprietà algoritmiche dei linguaggi

9.1 Grammatiche libere non regolari

Non abbiamo ancora mostrato che esistono grammatiche libere ma non regolari. Sarà questo lo scopo di questa sezione. Ad esempio il linguaggio

$$L = \{a^n b^n \mid n \geq 0\}$$

è sicuramente libero ma possiamo dimostrare non sia regolare. Infatti un linguaggio simile avrebbe bisogno di una quantità *infinita* di stati, per fare in modo che dopo una serie di k `a` ci sia una serie sempre lunga k di `b`. Poichè servono almeon $2k$ stati e k può essere grande a piacere servirebbero $2 \cdot \infty$ stati, mentre gli automi studiati sono *finiti*.

9.1.1 Pumping lemma

Se L è un linguaggio regolare allora $\exists N > 0$ tale che $\forall z \in L$ con $|Z| \geq N$ allora $\exists u, v, w$ tali che

1. $z = uvw$
2. $|uv| \leq N$
3. $|v| \geq 1$
4. $\forall k \geq 0 \quad uv^k w \in L$

Inoltre N è minore o uguale del numero di stati del DFA minimo che accetta L.

Dimostrazione (spesso all'orale)

Sia $N = |Q_M|$ dove M è il DFA minimo che accetta L. Sia $z = a_1 a_2 \dots a_m \in L$ con $m \geq N$. Quindi:

$$q_0 \to_{a_1} q_1 \to_{a_2} q_2 \to_{a_3} \dots \to_{a_m} q_m \in F$$

Ora $0 - m$ è dato da $m + 1$ stati con $m + 1 > N$ quindi $\exists i, j. \quad (i \neq j)$ tali che $q_i = q_j$. Esiste dunque per forza un ciclo:

$$\begin{array}{rr} q_0 \to_{a_1} q_1 \to_{a_i} q_i = & q_j \to_{a_{j+1}} q_{j+1} \to_{a_{j+2}} \dots \to_m q_m \in F \\ \uparrow_{a_j} & \downarrow_{a_{i+1}} \\ q_j \leftarrow_{\dots} & q_{i+1} \end{array}$$

Poichè $i \neq j$ allora $v = a_{i+1} \dots a_j$ è tale che $|v| \geq 1$.
La condizione $|uv| \leq N$ mi dice che prendo il primo ciclo (quando ce ne sono molteplici).

Si noti che

$$\begin{aligned} uv^0w &= uw \in L \\ uv^1w &= uvw \in L \\ uv^2w &= uvvw \in L \\ uv^2w &= uvvw \in L \\ &\vdots \\ uv^kw &\in L \end{aligned}$$

e dunque $\forall k \geq 0$ si ha che $uv^kw \in L$ poichè il ciclo v può essere percorso un numero arbitrario di volte. □

Osservazioni

Se L è finito allora non esiste nessuna $z \in L$ con $|z| \geq N$ e qiundi l'implicazione è vera poichè è falsa la premessa.

Se $\exists z \in L$ con $|Z| \geq N$ allora M riconosce un linguaggio infinito.

Prova di non regolarità

Ora come possiamo usare il *pumping lemma* per provare che un linguaggio non è regolare? Se il linguaggio L è regolare allora valgono le proprietà P del lemma. Useremo l'implicazione opposta:

$$\neg P \Rightarrow L \text{ non è regolare}$$

9.1.2 Negazione del pumping lemma

Se $\forall N > 0. \exists z \in L$ con $|z| \geq N$, se $\forall u, v, w$ vale

1. $z = uvw$
2. $|uv| \leq N$
3. $|v| \geq 1$

allora $\exists k \geq 0. \quad uv^k w \notin L$ allora L non è regolare.

9.1.3 Chiusura dei linguaggi regolari

I linguaggi regolari sono chiusi rispetto alle seguenti operazioni:

1. unione
2. concatenazione
3. stella di Kleene
4. complementazione
5. intersezione

Dimostrazione

(1), (2), (3) sono ovvie, poichè usate per costruire i linguaggi regolari stessi (i quali ovviamente appartengono ai linguaggi regolari).

(4) È vera poichè dato un DFA $M = (\Sigma, Q, \delta, q_0, F)$ tale che $L = L[M]$ possiamo costruire un altro DFA $\overline{M} = (\Sigma, Q, \delta, q_0, Q \setminus F)$ tale che $\overline{L} = L[\overline{M}]$. Infatti $w \in L[M] \iff w \notin L[\overline{M}]$ e quindi $L[\overline{M}] = \Sigma^* \setminus L[M]$ (ovvero il linguaggio complementare).

(5) Deriva trivialmente da De Morgan:

$$L(M_1) \cap L(M_2) = \overline{\overline{L(M_1)} \cup \overline{L(M_2)}}$$

Prova alternativa disponibile sulle slide.

9.1.4 Utilizzo di queste proprietà

La chiusura per intersezione può essere utile per dimostrare che un linguaggio non è regolare. Se ho un linguaggio L da identificare, posso intersecarlo con un linguaggio che **so** essere regolare e se il risultante non lo è allora ho provato che L *non è regolare*. Vedi esepmio sulle slide.

9.1.5 Proprietà algoritmiche dei linguaggi regolari

Le seguenti proprietà di un linguaggio regolare possono essere verificate tramite algoritmi (M_* sono tutti DFA):

- $w \in L[M]$: basta leggere w e vedere se finisce su uno stato finale nel DFA.
- $L[M] = \emptyset$: esiste *almeno* un cammino aciclico dallo stato iniziale ad un finale?
- $L[M] = A^* \iff L[\overline{M}] = \emptyset$
- $L[M_1] \subseteq L[M_2] \iff L[\overline{M_2}] \cap L[M_1] = \emptyset$
- $L[M_1] = L[M_2] \iff L[M_1] \subseteq L[M_2] \wedge L[M_1] \supseteq L[M_2]$
- $L[M]$ è infinito? $\exists z \in L[M]$ tale che $n \leq z \leq 2n$ dove $n = |Q|$?