UNIVERSITÀ DEGLI STUDI DI BERGAMO
Dipartimento di Ingegneria Gestionale, dell'Informazione e della Produzione

INGEGNERIA INFORMATICA

01 - Computability and Fundamentals

Anno di corso: 1

Anno accademico di offerta: 2023/2024

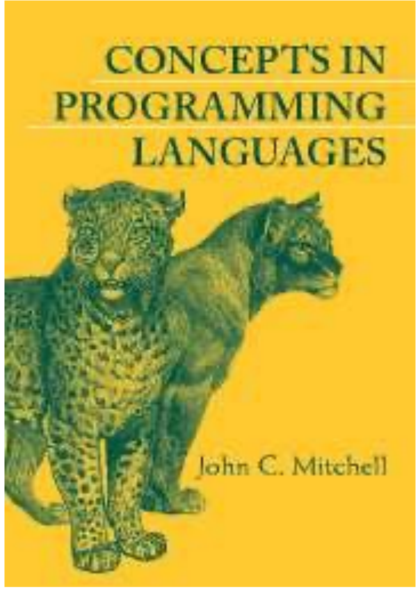
Crediti: 6

Ore di didattica frontale: 48

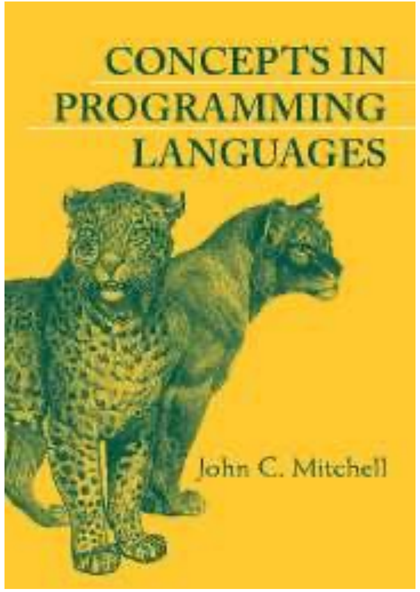
Prof. Claudio MENGHI

Dalmine

18 Settembre 2024



Capitolo 2: computability
Capitolo 4: fundamentals



Capitolo 2: computability

Sommario

- funzioni “recursively defined” (definite ricorsivamente) e funzioni parziali
- funzioni computabili
- Turing completeness
- Indecidibilità e Halting problem

2.1 Funzioni Parziali e Computabilità

- Da un punto di vista matematico un programma è una funzione
- L’output di un programma dipende da (a) lo stato della macchina prima che il programma venga eseguito e (b) dall’input fornito al programma
- Tuttavia, un programma può implementare solo funzioni computabili

2.1.1 Espressioni, Errori e Non Terminazione

- In **matematica**, le espressioni possono avere un valore o meno
  - 3+2 assume valore 5
  - 3/0 non è definito

2.1.1 Espressioni, Errori e Non Terminazione

- Nella **computazione** eseguita per mezzo di calcolatori ci sono varie ragioni per cui un'espressione può non ritornare un valore

  - **Errore**: si verifica quando c'è un problema (per esempio la valutazione di un'espressione su due operandi non compatibili)
    Esempio: divisione per zero

  - **Non terminazione**: si verifica quando la computazione procede all'infinito senza mai produrre un risultato
    Esempio: f(x:int) = if x = 0 then 0 else x + f(x-2)
    f(4) termina
    f(5) non termina mai
    [Hmmm davvero? **1pt bonus**]

2.1.1 Espressioni, Errori e Non Terminazione

- [-2147483648 to 2147483647] 32 bits
    - -2147483643
    - -2147483645
    - -2147483647
    - 2147483647
    - 2147483645
    - 2147483643

2.1.1 Espressioni, Errori e Non Terminazione

- java.lang.StackOverflowError
- -Xss515m

2.1.2 Funzioni Parziali

- Una funzione partiziale è definita per certi argomenti ma non per altri ovvero può ritornare un risultato per qualche input ma non terminare per altri.

- Una funzione f: A→B da un insieme A a un insieme B è una *regola* che associa un unico valore y=f(x) appartenente all'insieme B per ogni input x di A.
  - A è il dominio di f
  - B è il codominio di f

2.1.2 Funzioni Parziali

- Una funzione f: $A \rightarrow B$ è un insieme di coppie $f \subseteq A \times B$ che soddisfano le seguenti condizioni
  - Se $<x,y> \in f$ e $<x,z> \in f$, allora $y=z$
  - Per ogni $x \in A$, esiste un $y \in B$ con $<x,y> \in f$

- Una funzione parziale f: $A \rightarrow B$ è un insieme di coppie $f \subseteq A \times B$ che soddisfano la seguente condizione
  - Se $<x,y> \in f$ e $<x,z> \in f$, allora $y=z$

Esempio: f(x:int) = if x = 0 then 0 else x + f(x-2) è una funzione parziale [Termina solo se x è pari]

2.1.3 Computabilità

- Una funzione è computabile se c'è un programma che la computa, ovvero
- Una funzione $f: A \rightarrow B$ è computabile se esiste un algoritmo che, dato in input un qualsiasi input $x \in A$ termina e ritorna $y=f(x)$ come output

- È possibile che l'implementazione di tale algoritmo sia possibile in un linguaggio di programmazione ma non in un altro.

2.1.3 Computabilità

- La classe di funzioni sui numeri naturali che sono computabili in principio è la classe della funzioni parziali ricorsive
    - La ricorsione è essenziale per la computazione
    - Le funzioni sono parziali in generale

2.1.3 Computabilità

- **A function on the natural numbers can be calculated by an effective method if and only if it is computable by a Turing machine.**

Ci sono tre dimostrazioni

- Alonso Church (Funzioni)

- Lamda Calculus

- Alan Turing

- **Tutti i linguaggi di programmazione sono Turing complete**

2.1.3 Computabilità

- La macchina di Turing ha
  - Un nastro infinito sul quale è possibile leggere e scrivere e un controllore (a stati finiti)
  - Il nastro è diviso in un insieme di celle diviso
  - Il controllore può decidere se leggere o scrivere dal nastro o muoversi di una cella a sinistra o a destra

2.1.3 Computabilità

- Halting problem: dato un (generico) programma P che riceve una stringa x come input, determinare se il programma P termina quando riceve la stringa x

- Posssiamo associare l'halting problem con una funzione f_{halt} tale che
  - f_{halt}(P,x)=halt se il programma P termina per l'input x
  - f_{halt}(P,x)=not halt se il programma P non termina per l'input x

L'halting problem è indecidibile: la funzione f_{halt} non è computabile (in generale)

2.1.3 Computabilità

- **HP:** Supponiamo che esista un programma Q che risolva l'halting problem
  - $Q(P,x) = \begin{cases} halt & se\ P(x)\ termina \\ not\ halt & se\ P(x)\ non\ termina \end{cases}$
- Utilizzando Q creiamo un programma D che a volte non termina
  - D(P)= se «$Q(P,P)$=halt» allora run forever altrimenti halt

2.1.3 Computabilità

- $D(P) = \begin{cases} halt & se\ P(P)\ non\ termina \\ not\ halt & se\ P(P)\ \ termina. \end{cases}$

- Consideriamo il comportamento di D(D)

  - $D(D)$ =halt se D(D) non termina

  - $D(D)$ = $not\ halt$ se D(D) termina

2.1.3 Computabilità

- Halting problem: dato un (generico) programma P che riceve una stringa x come input, determinare se il programma P termina quando riceve la stringa x

- Posssiamo associare l'halting problem con una funzione f_{halt} tale che
    - $f_{halt}(P,x)$=halt se il programma P termina per l'input x
    - $f_{halt}(P,x)$=not halt se il programma P non termina per l'input x

L'halting problem è indecidible: la funzione P non è computabile (in generale)

Davvero??? [1pt bonus]

2.1.3 Computabilità

- Computabili “in principio”: sono le funzioni che sono computabili
- Computabili “in pratica”: alcune delle funzioni computabili in principio richiedono moltissimo tempo. Se una funzione non ritornerà un valore in un quantitativo di tempo pari alla durata della storia dell’universo all’ora non è computabile in pratica

2.1.3 Computabilità

```
i=0;
while(i!=f(i)) i=g(i);
printf(…i…..);
```

Può il compilatore capire se il programma termina?

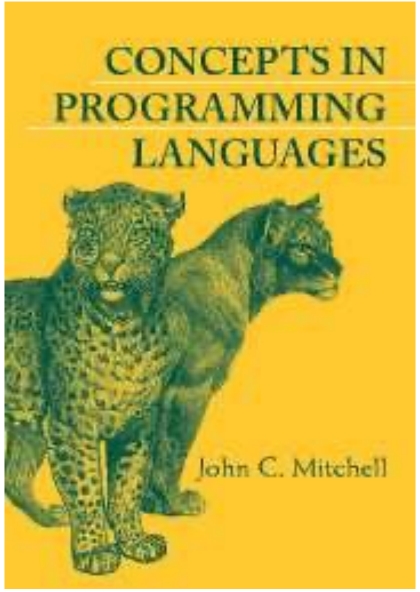
Sommario

Partiality: Recursively defined functions may be partial functions. They are not always total functions. A function may be partial because a basic operation is not defined on some argument or because a computation does not terminate.

Computability: Some functions are computable and others are not. Programming languages can be used to define computable functions; we cannot write programs for functions that are not computable in principle.

Turing completeness: All standard general-purpose programming languages give us the same class of computable functions.

Undecidability: Many important properties of programs cannot be determined by any computable function. In particular, the halting problem is undecidable.



Capitolo 4: fundamentals

Capitolo 4

- Descrizione di un compilatore e parser
- Lambda calculus
- Denotational semantics
- Linguaggi imperativi e funzionali

4.1.1 Structure of a Simple Compiler

- Sintassi: Il testo di un programma
- Semantica: quello che il programma significa [quello che fa]

4.1.1 Structure of a Simple Compiler

```java
public class Test {

    public static int counter=0;

    superpublic static void main(String[] args) {
        check(-2147483641);

    }

    public static int check(int arg) {
        if(counter<20) {
        System.out.println(arg);
        counter++;
        }
        if(arg==0) return 0;
        else return check(arg-2);
    }

}
```

```java
public class Test {

    public static int counter=0;

    public static void main(String[] args) {
        check(-2147483641);

    }

    public static int check(int arg) {
        if(counter<20) {
        System.out.println(arg);
        counter++;
        }
        if(arg==0) return 0;
        else return check(arg-2);
    }

}
```

roblems @ Javadoc Declaration Console

```
inated> Test (1) [Java Application] /Users/admin/.p2/pool/plugins/org.eclipse.justj.openjdk.hotspot.jre.full.macosx.x86_64_17.0.6.v2023020
ception in thread "main" java.lang.StackOverflowError
	at Test/test.Test.check(Test.java:18)
	at Test/test.Test.check(Test.java:18)
```

4.1.1 Structure of a Simple Compiler

- **Compilatore**: traduce il programma in un insieme di istruzioni che possono essere eseguite dalla macchina

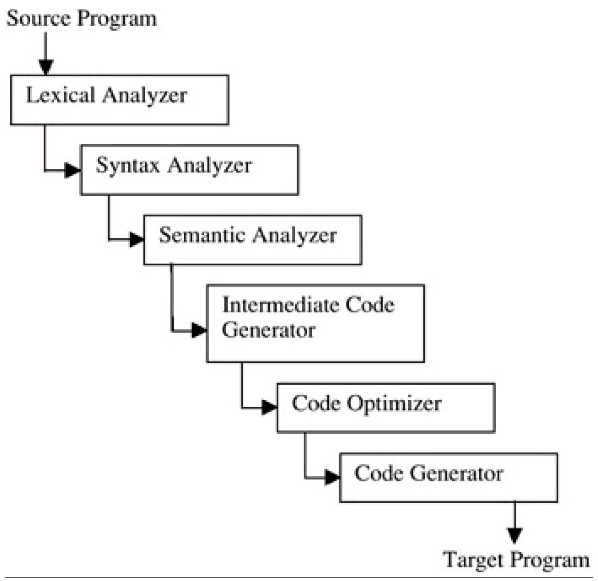
- **Interprete**: combina la traduzione con l’esecuzione del programma

4.1.1 Structure of a Simple Compiler

- **Compilatore**: traduce il programma in un insieme di istruzioni che possono essere eseguite dalla macchina

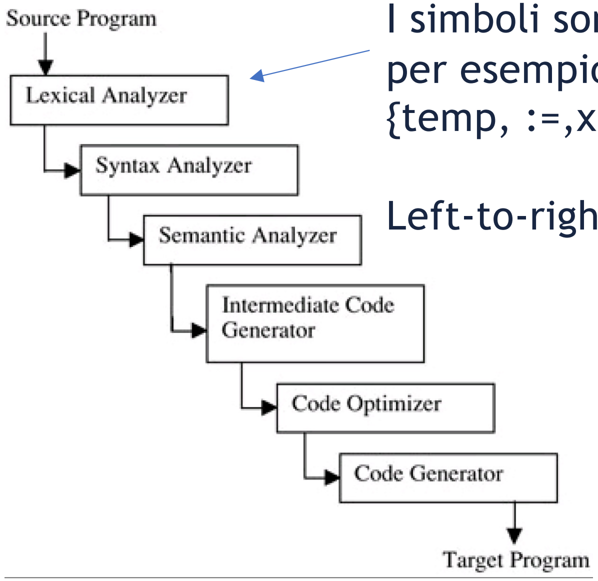
4.1.1 Structure of a Simple Compiler

- **Compilatore**: traduce il programma in un insieme di istruzioni che possono essere eseguite dalla macchina



4.1.1 Structure of a Simple Compiler

- **Compilatore**: traduce il programma in un insieme di istruzioni che possono essere eseguite dalla macchina



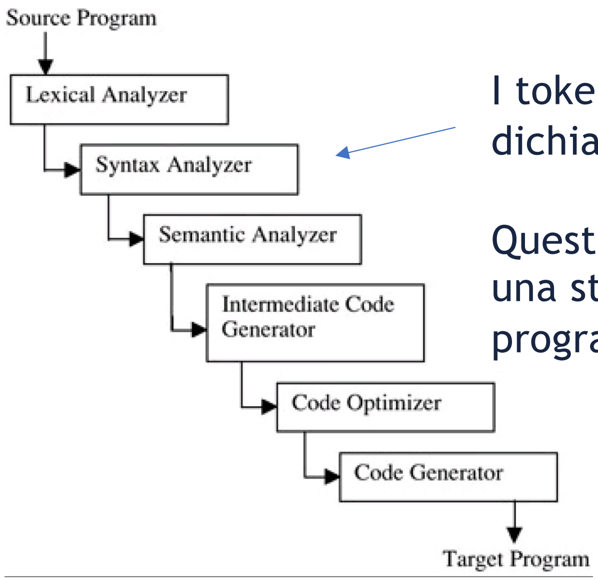
I simboli sono scanditi e raggruppati in tokens
per esempio temp:=x+1 è costituito da 5 tokens:
{temp, :=,x,+,1}

Left-to-right and top-to-bottom

UNIVERSITÀ DEGLI STUDI DI BERGAMO
Dipartimento di Ingegneria Gestionale, dell'Informazione e della Produzione

4.1.1 Structure of a Simple Compiler

- **Compilatore**: traduce il programma in un insieme di istruzioni che possono essere eseguite dalla macchina

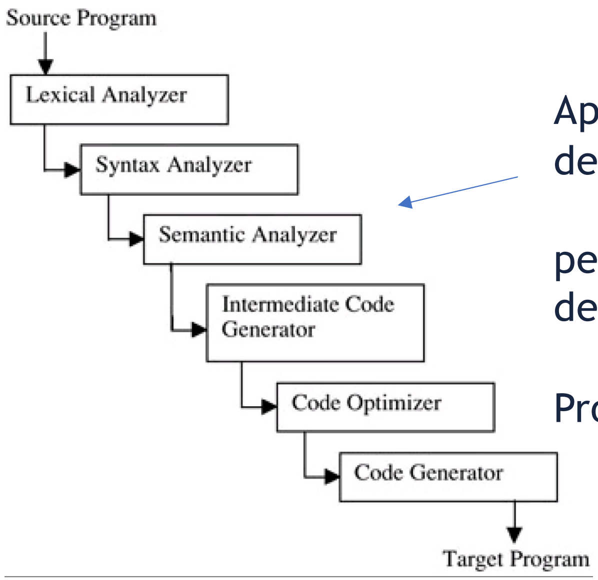


I token sono raggruppati in espressioni, statements e dichiarazioni in base alle regole della grammatica

Questa attività è eseguita dal parser. L'obiettivo è creare una struttura chiamata «parse tree» che rappresenta il programma

4.1.1 Structure of a Simple Compiler

- **Compilatore**: traduce il programma in un insieme di istruzioni che possono essere eseguite dalla macchina



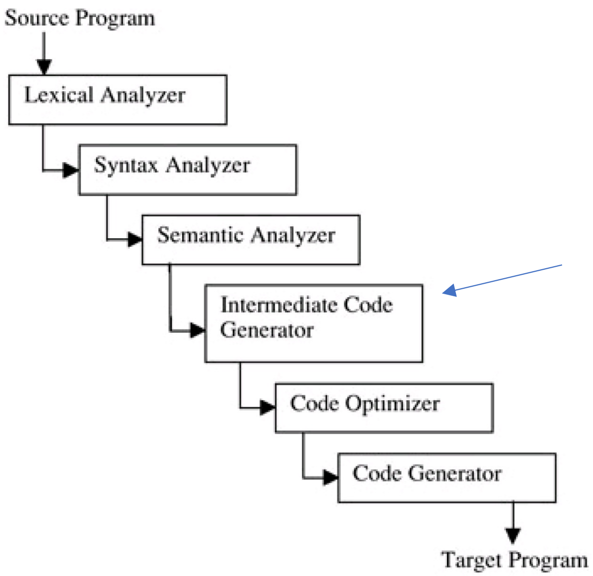
Applica regole e procedure che dipendono dal contesto delle espressioni

per esempio controllare che i tipi all'interno dell'espressione temp:=x+1 siano consistenti

Produce un «augmented parse-tree»

4.1.1 Structure of a Simple Compiler

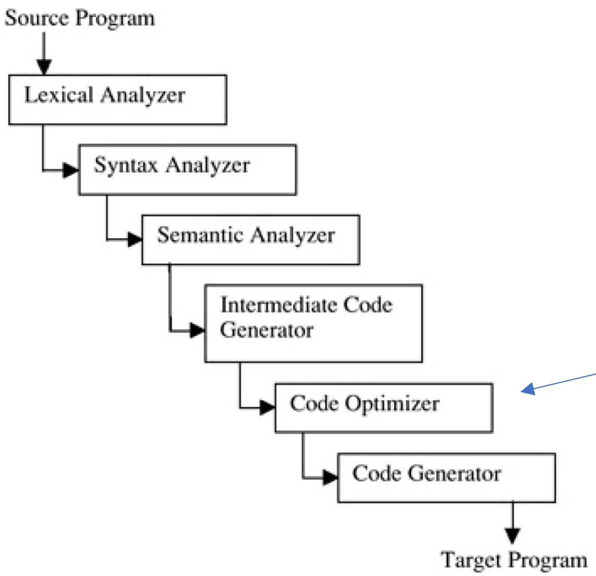
- **Compilatore**: traduce il programma in un insieme di istruzioni che possono essere eseguite dalla macchina



Producono una versione intermedia del codice per poi procedere a delle ottimizzazioni

4.1.1 Structure of a Simple Compiler

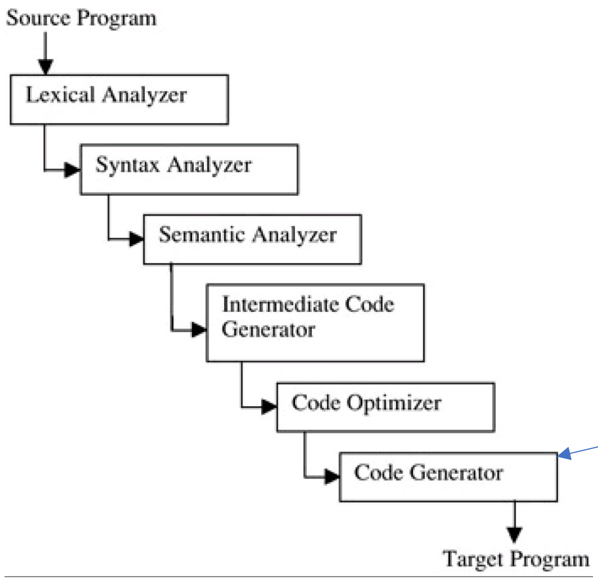
- **Compilatore**: traduce il programma in un insieme di istruzioni che possono essere eseguite dalla macchina



Applica un insieme di tecniche per ottimizzare il codice
- elimina sottoespressioni (se la stessa espressione è computata + di una volta)
- x=y sostituisce y a x.
- elimina codice morto
- cerca di rimuovere istruzioni dai loop
- rimpiazza una funzione con il corrispettivo codice

4.1.1 Structure of a Simple Compiler

- **Compilatore**: traduce il programma in un insieme di istruzioni che possono essere eseguite dalla macchina



Converte il codice intermedio nel linguaggio del target program

4.1.2 Grammatiche e Parse Trees

- Grammatiche: forniscono un metodo per definire un insieme (infinito) di espressioni. Le grammatiche sono composte da
  - Un simbolo iniziale
  - Un insieme di non-terminali
  - Un insieme di terminali
  - Un insieme di regole di produzione

I non terminali sono simboli utilizzati per scrivere la grammatica. I terminali sono simboli che appariranno nel linguaggio

4.1.2 Grammatiche e Parse Trees

- Un esempio di grammatica in Backus–Naur or Backus normal form (BNF)

Start symbol

e ::= n | e+e | e-e

n ::= d | nd

d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

4.1.2 Grammatiche e Parse Trees

- Un esempio di grammatica in Backus–Naur or Backus normal form (BNF)

Non terminali

e ::= n | e+e | e-e

n ::= d | nd

d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

4.1.2 Grammatiche e Parse Trees

- Un esempio di grammatica in Backus–Naur or Backus normal form (BNF)

$$
\begin{aligned}
e &::= n \mid e\boxed{+}e \mid e\boxed{-}e \\
n &::= d \mid nd \\
d &::= \boxed{0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9}
\end{aligned}
$$

Terminali

4.1.2 Grammatiche e Parse Trees

e ::= n | e+e | e-e

n ::= d | nd

d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Nel linguaggio

0, 1+3+5, 2+4 - 6 - 8

Non nel linguaggio

e, e+e, e+6 - e

4.1.2 Grammatiche e Parse Trees

e ::= n | e+e | e-e

n ::= d | nd

d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- Derivazione: sequenza di step di “rimpiazzo” che porta a una stringa di terminali

$$e \to n \to nd \to dd \to 2d \to 25$$

$$e \to e - e \to e - e{+}e \to \ldots \to n{-}n{+}n \to \ldots\ \ldots\ 10{-}15{+}12$$

UNIVERSITÀ DEGLI STUDI DI BERGAMO
Dipartimento di Ingegneria Gestionale, dell'Informazione e della Produzione

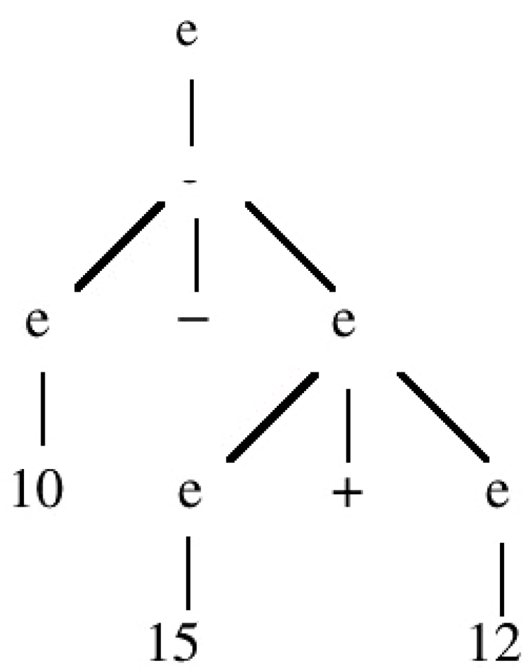
4.1.2 Grammatiche e Parse Trees - Ambiguità

e ::= n | e+e | e-e

n ::= d | nd

d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

10—15+12



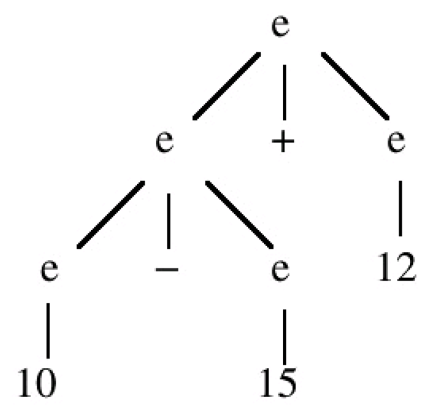
4.1.2 Grammatiche e Parse Trees - Ambiguità

e ::= n | e+e | e-e

n ::= d | nd

d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

10−15+12



4.1.2 Grammatiche e Parse Trees - Ambiguità

- Una grammatica è ambigua se la stessa espressione ha più di un «parse tree»

4.1.2 Grammatiche e Parse Trees - Precedenza

- Parsing: procedura di costruzione di un parse tree da una sequenza di simboli

- Parsing algorithm: un algoritmo che capisce quando una stringa appartiene a un linguaggio (e costruisce il corrispondente parse tree) è chiamato parsing algorithm.

4.2 Lamda Calculus (λ)

4.2 Lamda Calculus (λ)

Notazione per descrivere la computazione

Composta da tre parti

- notazione per descrivere le funzioni
- meccanismo di prova per descrivere equazioni tra espressioni
- set di regole di calcolo chiamate riduzioni

4.2 Lamda Calculus (λ)

- una funzione è una regola per determinare un valore da degli argomenti

  $f(x)=x^2+3$

- $h(x)=f(g(x))$
  - la funzione h è definita dall'applicazione della funzione f alla funzione g.

4.2 Lamda Calculus (λ)

I due concetti principali del lambda calculus sono:

- lambda abstractions. If M è un espression, λx.M è la funzione che otteniamo trattando M come una funzione della variable x

- per esempio λx.x è una astrazione che funzione la funzione di identità dato un x ritorna il suo valore oppure alternaviamente
- I(x)=x

- application. Per applicare una funzione ad un'altra, possiamo mettere l'espressione davanti all'altra

- per esempio, possiamo applicare la funzione di identità all'espressione M scrivendo (λx.x)M (ovvero, dato un M ritorna se stesso)
  - (λx.x)M=??
  - (λx.x)M=M

4.2 Lamda Calculus (λ): Espression

Dato un insieme di variabili V con $x \in V$ una lambda espression è definita come:

λ term $\longrightarrow$ M::= x | M M | λx.M

M_1 M_2 corrisponde all'applicazione di M_1 ad M_2

λx.M è la λ abstraction che dato un argomento x ritorna il valore M

4.2 Lamda Calculus (λ):

Linguaggio di programmazione = applied λ-calculus = pure λ-calculus+additional data types

4.2 Lamda Calculus (λ):

Variable binding

- free variable: variabile che non è “dichiarata nell’espressione” (opposto bounded variable)
    - esempio x+3
    - esempio λx.x+3
    - esempio $\int$ f(x) dx

4.2 Lamda Calculus (λ):

Starting with **0** *not applying the function at all, proceed with* **1** *applying the function once,* ***2*** applying the function twice, **3** *applying the function three times, etc.*:

Number	Function definition	Lambda expression
0	$0\ f\ x = x$	$0 = \lambda f.\lambda x.x$
1	$1\ f\ x = f\ x$	$1 = \lambda f.\lambda x.f\ x$
2	$2\ f\ x = f\ (f\ x)$	$2 = \lambda f.\lambda x.f\ (f\ x)$
3	$3\ f\ x = f\ (f\ (f\ x))$	$3 = \lambda f.\lambda x.f\ (f\ (f\ x))$
$\vdots$	$\vdots$	$\vdots$
n	$n\ f\ x = f^n\ x$	$n = \lambda f.\lambda x.f^{\circ n}\ x$

4.2 C vs Lamda Calculus (λ):

- in C an assignment statement has side effects

- in lamda calculus gli assignment sono puramente funzionali

4.4 Functional and Imperative Languages

- natural language (linguaggi naturali): linguaggi utilizzati dagli umani

- Ambiguita

- frasi imperative “prendi il pesce”
- frasi dichiarative “a Claudia piacciono le mele”
- frasi interrogative e quesiti

4.4 Functional and Imperative Languages

```
{ int x=1;               /* declares new x */
    x = x+1;             /* assignment to existing x */
    { int y = x+1;       /* declares new y */
        { int x = y+1;   /* declares new x */
}}}
```

Imperative

4.3 Denotational Semantics

- Nella semantica denotazionale un programma è una funzione matematica da stato a stato.
- Lo stato è una funzione matematica che rappresenta i valori della memoria in un determinato stato dell'esecuzione di un programma
    - x := 0; y:=0; while x $\leq$ z do y:=y+x;x:=x+1

4.3.2 Denotational Semantics of Binary Numbers

Grammatica

e ::= n | e+e | e-e

n ::= b | nb

b ::= 0 | 1

4.3.2 Denotational Semantics of Binary Numbers

Grammatica

e ::= n | e+e | e-e

n ::= b | nb

b ::= 0 | 1

Semantica

E[[0]] = 0

E[[1]] = 1

E[[nb]] = E[[n]] * 2 + E[[b]]

E[[e_1+e_2]] = E[[e_1]] + E[[e_2]]

E[[e_1−e_2]] = E[[e_1]]-E[[e_2]]

Esercizio

P1: Ho implementato una funzione f che genera tutte le possibili stringhe composte da “a e b” di lunghezza inferiore di ≤ 5?

Vero of Falso? Mi fido?

Esercizio

P2: Ho implementato una funzione f che dato un programma p (passato come parametro) trova **tutte** le istanze di codice morto?

Vero of Falso? Mi fido?

UNIVERSITÀ DEGLI STUDI DI BERGAMO

Dipartimento di Ingegneria Gestionale, dell'Informazione e della Produzione

Domande?