Corso di Linguaggi di Programmazione

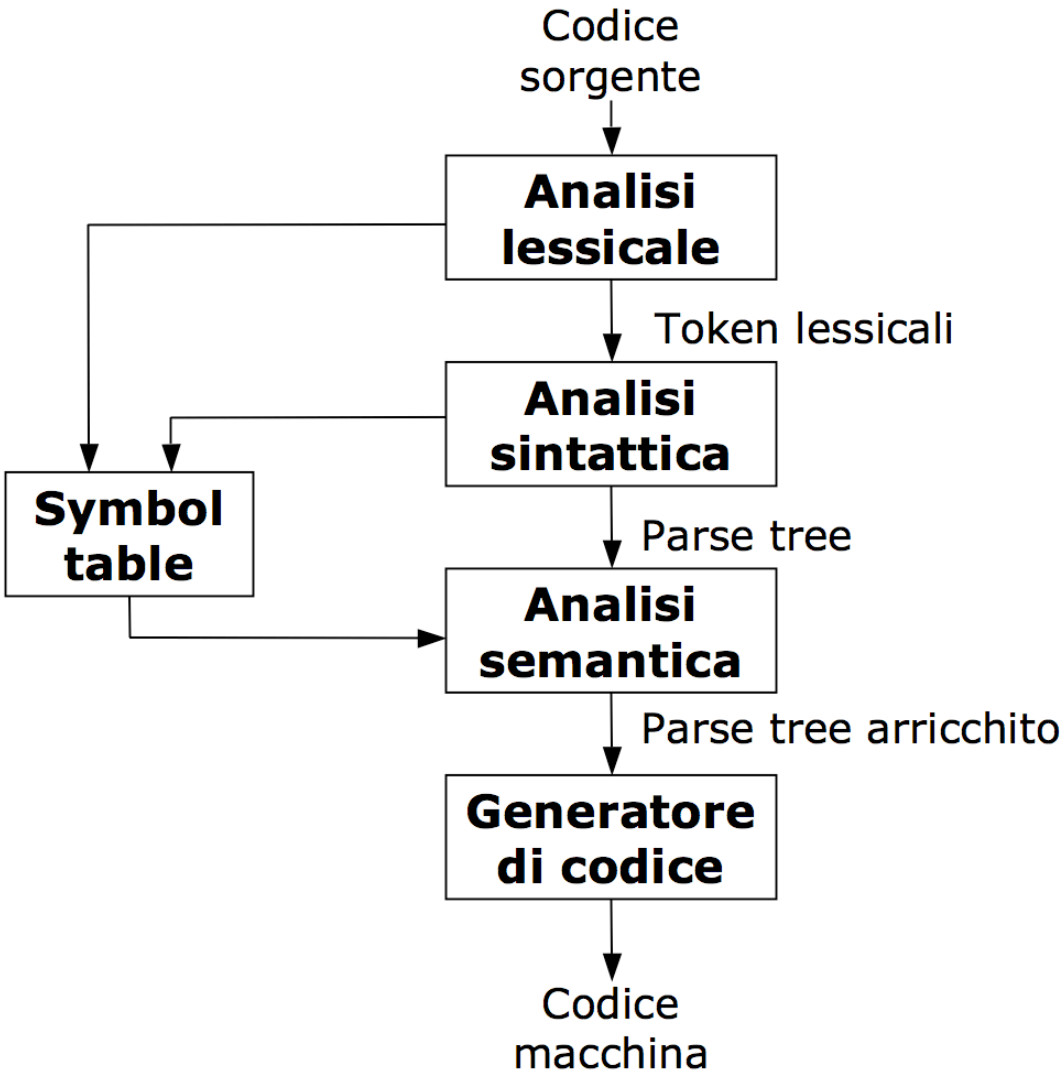
Lezione 14

Alberto Ceselli
alberto.ceselli@unimi.it

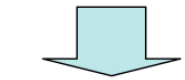
Dipartimento di Informatica
Università degli Studi di Milano

23 Aprile 2013

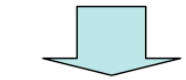
Cosa servono i tipi durante l'analisi semantica?

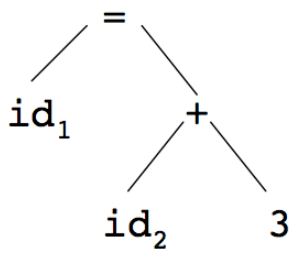


sum=a+3



$id_1 = id_2 + 3$





1	sum	...
2	a	...
3	...	...

Obiettivi dell'analisi semantica

- Controllo di tipo
- Controllare che gli identificatori siano stati dichiarati prima di essere usati
  - come supporto a questa fase viene creata una tabella dei simboli (symbol table) che contiene informazioni su tutti gli elementi simbolici incontrati quali nome, scope, tipo (se presente) etc.
  - il risultato di questa fase è l'albero di sintassi astratta (AST).

Descrittore di Tipo

Un **descrittore di tipo** è una struttura dati che contiene tutte le informazioni necessarie a definire il tipo:

- se il tipo della variabile è determinato all'inizio dell'esecuzione del programma (statico), serve al compilatore per controlli di tipo.
- se il tipo della variabile è determinato o può cambiare in fase di esecuzione (dinamico), una sua parte può servire per i controlli durante l'esecuzione (es. controllo sugli indici di un vettore dinamico).

Controllo sui Tipi

- Determina e controlla la compatibilità dei tipi degli oggetti: assegnazioni, dichiarazioni, uso parametri, conversioni, ...
- Il controllo avviene rispetto alle regole di tipo definite dal particolare linguaggio
- Tipo di controllo:
  - **statico**: avviene in fase di compilazione
    - segue l'albero sintattico (bottom-up)
    - determina e trasmette le informazioni sui tipi degli oggetti coinvolti
  - **dinamico**: modulo di supporto a run-time
    - pesante dal punto di vista prestazionale
    - tipico dei linguaggi interpretati

Tipo di dato

Definition

In un linguaggio di programmazione, un **tipo di dato** (o semplicemente **tipo**) è un *nome* che indica l'insieme di valori omogenei che una variabile, o il risultato di un'espressione, possono assumere, e le operazioni che manipolano tali valori.

- L'insieme dei valori ha una cardinalità *finita*.
- Un tipo è caratterizzato da un *nome* e dalla sintassi della *dichiarazione di tipo*.
  - Es.: in C, la definizione `int x;` indica che `x` può assumere come valori solo numeri interi (appartenenti ad un certo intervallo) e che su tali valori sono ammesse solo certe operazioni.

Classificazioni del processo di tipizzazione (1)

- **Statico** vs **dinamico**
  - Fa riferimento al momento in cui alla variabile è associato il suo tipo (binding tipo-variabile).
    - la tipizzazione è *statica* quando a una variabile viene associato rigidamente in *fase di compilazione* un tipo che rimane lo stesso durante tutta l'esecuzione.
    - la tipizzazione è *dinamica* quando una variabile può cambiare tipo durante l'*esecuzione* a seconda del tipo dell'espressione che le viene assegnata.
- **Esplicito** vs **implicito**
  - Fa riferimento al modo in cui le annotazioni o dichiarazioni di tipo vengono generate, se *specificate esplicitamente* dal programmatore, o *dedotte automaticamente* dal compilatore.

Svantaggi della tipizzazione dinamica

- Permette la cattura di *errori di tipo* solo durante l'esecuzione del programma.
  - Es.: (in pseudo-codice, L# indica la linea)

    ```
    L1 var x = 5;
    L2 var y = "test";
    L3 var z = x + y;
    ```

    In un linguaggio con tipizzazione dinamica, solo in fase di esecuzione viene segnalato un errore quando si cerca di eseguire l'istruzione alla linea L3.
- L'implementazione è più costosa:
  - maggiore *tempo di esecuzione* perché il controllo dei tipi deve venire fatto durante l'esecuzione del programma
  - maggior utilizzo della *memoria* perché si deve tenere memoria del tipo delle variabili durante l'esecuzione del programma.

Inferenza di Tipo

- Il tipo di un'espressione è dedotto senza richiedere al programmatore di specificare il tipo delle variabili.
- Quando non si può determinare subito il tipo, si utilizza una *variabile di tipo*.
- Es.: data la seguente istruzione ML

  `fun f(n) {return n+1;}`

  si può dedurre che `f` è di tipo `int -> int`

Conversione di Tipo

- Conversione **implicita** (*coercion*)
  - operata in fase di compilazione del programma
  - indebolisce il meccanismo di verifica dei tipi
  - in quasi tutti i linguaggi capita se si esegue un'operazione tra tipi numerici diversi.
  - Es `v` e `q` reali, `p` intero: `v = p + q`
- Conversione **esplicita** (*cast*)
  - indicata mediante costrutti linguistici nel sorgente del programma.
  - Es `v` e `q` interi, `p` reale: `v = (int) p + q`

Classificazioni del processo di tipizzazione (2)

Dal punto di vista *qualitativo*:

- **Forte** vs **Debole** Fa riferimento alle regole imposte dal linguaggio sull'utilizzo dei tipi:
  - possono essere rigide (*forti*) ed impedire **qualsiasi** uso dei dati incoerente con il tipo specificato in fase di dichiarazione (**ogni** errore di tipo è rilevato **dal compilatore**)
  - in caso contrario (**qualche** errore di tipo può sfuggire **al compilatore**) la tipizzazione è *debole*.

Classificazioni del processo di tipizzazione (3)

Dal punto di vista *qualitativo*:

- **Safe** vs **Unsafe**
  - Un linguaggio *type safe* garantisce che **tutti** gli errori di tipo siano catturati, ma **non necessariamente** durante la compilazione (un programma potrebbe fallire durante l'esecuzione a causa di violazioni di tipo).
  - Linguaggi che permettono aritmetica dei puntatori **non** possono essere type safe.
  - Un linguaggio con tipizzazione debole **non** è necessariamente unsafe (es. il controllo ritardato in fase di esecuzione – Perl).

Linguaggi fortemente tipati

In genere, un **linguaggio** di programmazione si considera **fortemente tipato** (strongly typed) se:

- Utilizza *tipizzazione statica*
  - i tipi delle variabili sono fissati in fase di compilazione e non sono modificabili.
  - non ci sono tipi di dato varianti (es. *union* in C)
- Impedisce qualsiasi *uso dei dati incoerente* con il tipo specificato in fase di dichiarazione (i.e., impossibilità di conversione di tipi)
  - a volte la conversione esplicita (*casting*) è permessa, mentre quella implicita (*coercion*) no.
- Impedisce di evadere il sistema di tipi.

Questo permette di avere una *garanzia* piuttosto forte sul comportamento del programma in fase di esecuzione.

Forte e Debole, Safe e Unsafe

Linguaggio	Statico / Dinamico	Strong / Weak	Safety
Ada	statico	strong	safe
C	statico	weak	unsafe
C++	statico		
Java	statico		
Pascal	statico		
Lisp	dinamico	strong	
ML	dinamico		
JavaScript	dinamico		
Perl	dinamico		
PHP	dinamico		
Python	dinamico		
Ruby	dinamico		

Esercizio: riempire la tabella.

Tipizzazione
Esempi nei vari linguaggi (1)

- Linguaggi a tipizzazione **statica**:
  - Ada, C, C++, Java, Fortran: tipizzazione **esplicita**
  - Perl, JavaScript, ML: tipizzazione **implicita**
    - in Perl, ogni nome che inizia con $ può contenere una stringa o un valore numerico
    - in ML, la definizione `fun square(x): real = x * x;` è corretta.
  - di norma vengono compilati (i tipi sono importanti anche in *fase di ottimizzazione* del codice).

Tipizzazione
Esempi nei vari linguaggi (1)

- Linguaggi a tipizzazione **statica**:
  - Ada, C, C++, Java, Fortran: tipizzazione **esplicita**
  - Perl, JavaScript, ML: tipizzazione **implicita**
    - in Perl, ogni nome che inizia con $ può contenere una stringa o un valore numerico
    - in ML, la definizione `fun square(x): real = x * x;` è corretta.
  - di norma vengono compilati (i tipi sono importanti anche in *fase di ottimizzazione* del codice).
- Linguaggi a tipizzazione **dinamica**:
  - Lisp, Scheme, Python, Perl, Ruby, JavaScript, PHP
  - la maggior parte dei *linguaggi di scripting* o *di sviluppo rapido* sono di solito a tipizzazione dinamica
  - di norma vengono interpretati.

Tipizzazione
Esempi nei vari linguaggi (2)

- Linguaggio **debolmente tipato**: Perl
  - `print ‘‘2’’ + 4;` in Perl stampa a video 6 poiché viene fatta una conversione implicita

Tipizzazione
Esempi nei vari linguaggi (2)

- Linguaggio **debolmente tipato**: Perl
  - `print ‘‘2’’ + 4;` in Perl stampa a video 6 poiché viene fatta una conversione implicita
- Linguaggio **debolmente tipato** e **unsafe**: C vs C++
  - `char* buf = malloc(20);` viene compilata in C senza problemi, ma genera un errore in C++ perché `malloc` ritorna un valore di tipo `void*` e il C++ richiede una conversione esplicita (cast).

Tipizzazione
Esempi nei vari linguaggi (2)

- Linguaggio **debolmente tipato**: Perl
  - `print ‘‘2’’ + 4;` in Perl stampa a video 6 poiché viene fatta una conversione implicita
- Linguaggio **debolmente tipato** e **unsafe**: C vs C++
  - `char* buf = malloc(20);` viene compilata in C senza problemi, ma genera un errore in C++ perché `malloc` ritorna un valore di tipo `void*` e il C++ richiede una conversione esplicita (cast).
- Linguaggio **fortemente tipato** ma **unsafe**: C++
  - Quando viene eseguito il seguente programma (che passa senza problemi la fase di compilazione), va in crash.

```
int* p = 0;
*p = 5;
```

Tipizzazione
Esempi nei vari linguaggi (3)

Classifica di linguaggi **fortemente tipati**:

- OCaml, Haskell:
  - non permettono alcuna conversione di tipo implicita (coercion)
- Ada:
  - non permette alcuna conversione di tipo implicita, ma ha la possibilità di sospendere temporaneamente il controllo di tipo.
- Java, C#
  - permettono alcuni tipi di casting, e coercion nel caso di operatori aritmetici
  - Es.: l'istruzione $x = 3 * 1.5$; è valida, in quanto l'espressione intera viene convertita in floating-point.

Sistema di tipo di un linguaggio di programmazione

Complesso delle informazioni e delle regole che governano i tipi di un linguaggio:

- Insieme dei *tipi predefiniti*
- *Costrutti* per definire nuovi tipi
- Meccanismi per il controllo dei tipi:
  - Regole di *equivalenza*: due tipi formalmente diversi possono essere equivalenti a livello semantico;
  - Regole di *compatibilità*: un valore di un tipo diverso da quello atteso può comunque essere utilizzato;
  - Regole di *inferenza* attribuzione di un tipo ad un'espressione complessa.
- Lista (possibilmente vuota) di vincoli da controllare staticamente o dinamicamente.

Equivalenza di Tipi

Stabilire quando due tipi, formalmente diversi, sono intercambiabili, ossia non distinguibili nel loro uso.

- **Equivalenza per nome**
  - Due tipi sono equivalenti se e solo se hanno lo stesso nome (sono lo stesso tipo)
  - Es.: `T1 = int` e `T2 = int` sono diversi
  - Facile da implementare ma molto restrittiva
  - Indebolita in alcuni linguaggi: la *ridenominazione* (`T3` e `T4`) genera alias e non nuovi tipi
- **Equivalenza strutturale**
  - Due tipi sono equivalenti se e solo se hanno la stessa struttura (dichiarazioni equivalenti)
  - Più flessibile, ma più complicata da implementare (specialmente per tipi definiti ricorsivamente).

Equivalenza strutturale

Problemi:

- nel caso di strutture, si confronta solo il tipo dei campi o anche il loro nome? E l'ordine in cui compaiono?
- se due tipi derivano da uno stesso tipo base, ma identificano entità diverse (es. `typedef celsius float; typedef fahrenheit float`)?

Equivalenza di Tipi
Esempi

- C, C++ utilizzano la versione indebolita dell'equivalenza per nome nel caso di un unico file, mentre l'equivalenza strutturale nel caso di file compilati separatemente
  - `struct` e `union` creano sempre un nuovo tipo che non è equivalente con nessun altro tipo.
- Es.: (in pseudo-codice)

```
type T1 = int;
type T2 = char;
type T3 = struct { T1 a; T2 b; }
type T4 = struct { T1 a; T2 b; }
```

Compatibilità di Tipi

Definition

Il tipo T è **compatibile** con il tipo S sse un valore del tipo T è ammesso in qualsiasi contesto in cui sarebbe richiesto un valore di tipo S.

- Ogni linguaggio definisce le regole di compatibilità tra oggetti di tipo differente, che vengono adottate nella disciplina dell'assegnazione, delle operazioni aritmetiche e del passaggio dei parametri.
- Compatibilità più debole dell'equivalenza:
  - Due tipi equivalenti sono sempre compatibili, ma non viceversa.
  - Relazione riflessiva, transitiva, ma non simmetrica.

Classificazione di Tipi

- A seconda della loro *struttura*:
  - **semplici** o **primitivi**: tipi che non possono essere decomposti;
  - **strutturati**: tipi ottenuti dai tipi semplici mediante operatori forniti dal linguaggio.
- A secondo di *chi li definisce*:
  - **predefiniti**: insieme di tipi predefiniti dal linguaggio;
  - **definiti dall'utente**: definiti mediante l'utilizzo di operatori forniti dal linguaggio.

Classificazione di Tipi

- A seconda della loro *struttura*:
  - **semplici** o **primitivi**: tipi che non possono essere decomposti;
  - **strutturati**: tipi ottenuti dai tipi semplici mediante operatori forniti dal linguaggio.
- A secondo di *chi li definisce*:
  - **predefiniti**: insieme di tipi predefiniti dal linguaggio;
  - **definiti dall'utente**: definiti mediante l'utilizzo di operatori forniti dal linguaggio.

Nota bene

Le due classificazioni **non** sono perfettamente sovrapponibili: non tutti i tipi predefiniti dal linguaggio sono primitivi.
Es.: le strinche in C sono tipi derivati ma predefiniti dal linguaggio.

Tipi e Linguaggi

Ogni linguaggio definisce accuratamente i tipi che permette di gestire, associando agli stessi:

- un insieme di operazioni ammesse
  - l'insieme di operazioni ammesse dipende dal linguaggio (vedi operazioni su stringhe)
- un ordinamento totale (facoltativo)
  - scalare
- una codifica, spesso dipendente dalla macchina
  - decisa dall'implementatore e a seconda della macchina
  - insieme di regole che permette di tradurre ogni valore in una stringa di bit (complemento a 1, complemento a 2, floating, ...)
  - viene ignorata dal programmatore

Esempio: array

Elementi sintattici:

- Forma di accesso ai dati: parentesi (Fortran, COBOL – unici elementi stampabili su schede perforate), caratteri [ ]
- indice di partenza: 0, 1
- tipo richiesto per gli indici (Ada permette anche booleani, caratteri o enum)
- controllo del corretto uso degli indici (Java, ML, C#)

Esempio: array

Tipo di allocazione:

- static array: allocazione di memoria statica e dimensione fissata
- fixed-stack dynamic array: dimensione fissata, allocazione nello stack quando viene incontrata la dichiarazione
- stack-dynamic array: dimensione non fissata, allocazione nello stack quando viene incontrata la dichiarazione
- fixed-heap dynamic array: dimensione fissata, allocazione nello heap quando viene incontrata la dichiarazione
- heap-dynamic array: dimensione non fissata, allocazione nello heap quando viene incontrata la dichiarazione

Esempio: array

Possibilità di accedere a slices:

- In Fortran 95: `Integer Cube(1:3,1:3,1:4)`,
- `Cube(1:3,1:3,2)` è una *porzione* dell'array Cube
- comune in linguaggi di programmazione per calcolo scientifico (Matlab)