Sintassi e Semantica

- Sintassi
  - Forma di un linguaggio
  - Definisce come una frase può essere formata da una sequenza di sinboli
- Semantica
  - Indica il significato di una frase corretta
- Regole lessicali
  - Insieme di caratteri (alfabeto) + regole di combinazione per formare un simbolo valido
  - – Caratteri minuscoli/maiuscoli: identici in Pascal, distinti in C, Ada
  - – <> operatore valido in Pascal, rappresentato da != in C, /= in Ada

EBNF (1)

(a) Regole Sitattiche

<program> ::= { <statement>* }
<statement> ::= <assignment> | <conditional> | <loop>
<assignment> ::= <identifier> = <expr> ;
<conditional> ::= **if** <expr> { <statement> + } |
if <expr> { <statement> + } **else** { <statement> + }
<loop> ::= **while** <expr> { <statement> + }
<expr> ::= <identifier> | <number> | (<expr>) |
<expr> <operator> <expr>

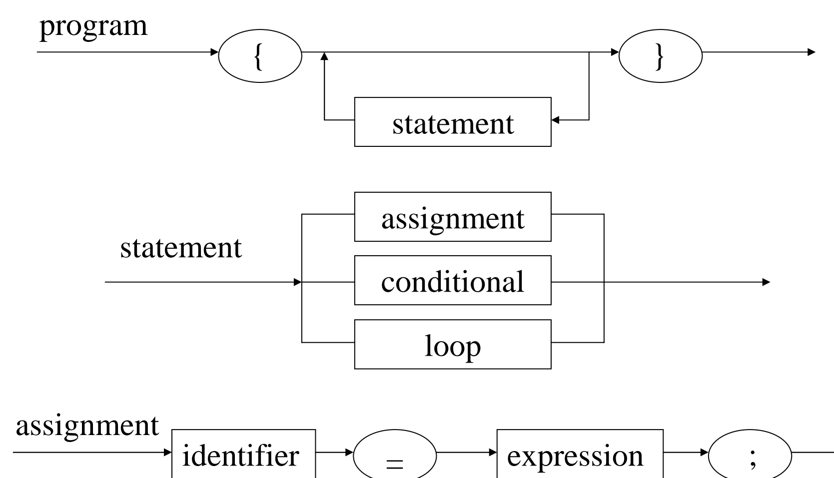
EBNF (2)

(b) Regole Lessicali

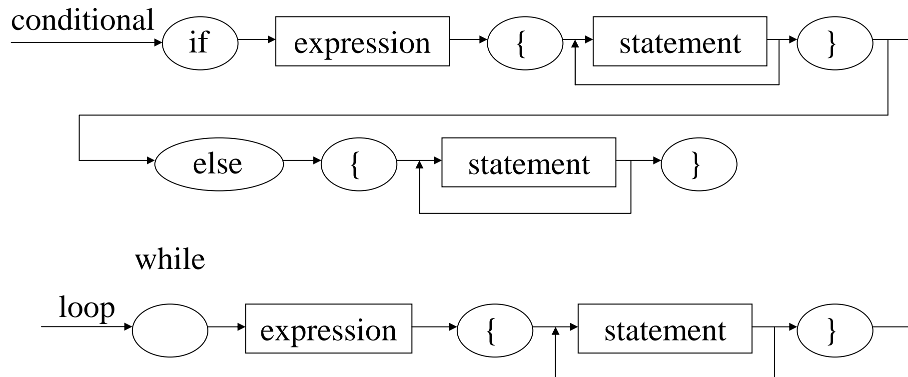
<operator>	::= + | - | * | / | = | /= | < | > | <= | >=
<identifier>	::= <letter> <ld>*
<ld>	::= <letter> | <digit>
<number>	::= <digit>+
<letter>	::= a | b | c | . . . | z
<digit>	::= 0 | 1 | . . . | 9

Nota: * rappresenta sia un simbolo che un metasimbolo

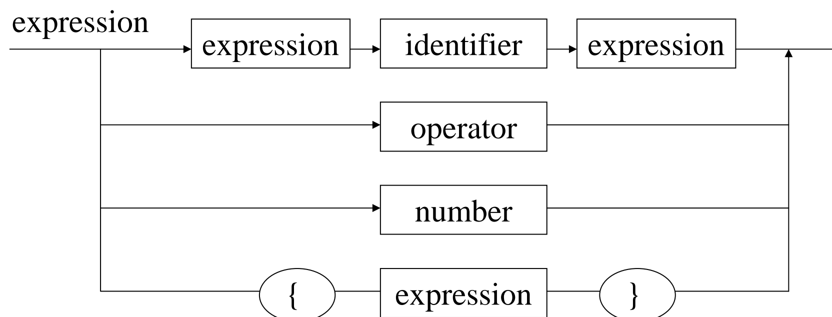
Diagrammi Sintattici (1)



Diagrammi Sintattici (2)



Diagrammi Sintattici (3)



Sintassi astratta

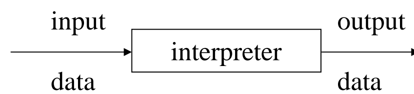
```
while (x != y) {
        …
}
```

```
while x <> y do
begin
        …
end
```

Stessa sintassi astratta, differente sintassi concreta

Interpretazione e traduzione

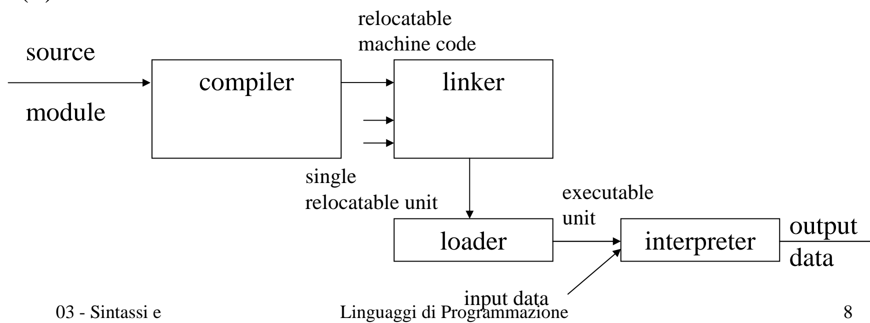
(a) Interpretazione



Modello di esecuzione (interprete)

- ottiene la nuova istruzione
- determina l'azione da eseguire
- compie l'azione

(b) Traduzione



Il concetto di legame (binding)

- ***Entità*** variabili, routines, dichiarazioni, ...
- ***Attributi***
  variabili: nome, tipo, area di memoria
  routine: nome, tipi di parametri, modalità passaggio di parametri
- ***Legame (Binding)*** specifica l'attributo per le entità
- ***Descrittore*** contiene le informazioni sugli attributi

- I linguaggi di programmazione differiscono per
  - numero di entità
  - numero di attributi legati alle entità
  - istante in cui avviene il legame

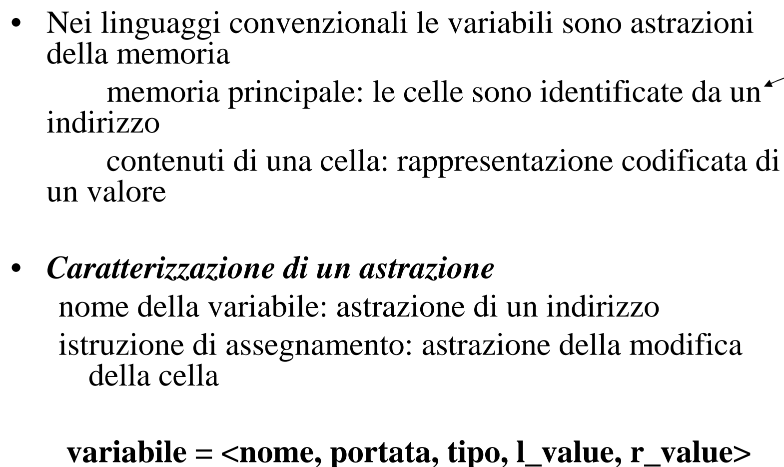
stabilità: può un legame stabilito essere modificato?
legame statico: non può essere modificato (dinamico altrimenti)

Esempi

- FORTRAN, Ada, C, C++ integer legato all'insieme dei valori nella definizione / implementazione
  - static binding
- Pascal integer legato alla traduzione del linguaggio
  - dynamic binding

Generalmente un legame statico è stabilito prima dell'esecuzione

Variabili



- Nei linguaggi convenzionali le variabili sono astrazioni della memoria
  - memoria principale: le celle sono identificate da un indirizzo
  - contenuti di una cella: rappresentazione codificata di un valore

- ***Caratterizzazione di un astrazione***
  - nome della variabile: astrazione di un indirizzo
  - istruzione di assegnamento: astrazione della modifica della cella

variabile = <nome, portata, tipo, l_value, r_value>

Nome e portata (scope)

- **nome** (generalmente) introdotto con una dichiarazione
- **scope** insieme di istruzioni in cui la variabile e conosciuta
  la variabile è visibile con il suo nome nel suo scope

- legame di visibilità S*tatico* (adottato da molti linguaggi):
  lo scope è definito in termini di struttura lessicale

- legame di visibilità *Dinamico* (APL, most old LISPs, SNOBOL4):
  lo scope è definito in termini di esecuzione del programma
  la dichiarazione della variabile estende il suo effetto su tutte le istruzioni successive fino ad una nuova dichiarazione con lo stesso nome

Legame statico vs legame dinamico

- Le regole di scope dinamico sono semplici e spesso facili da implementare
- Svantaggi in termini di disciplina di programmazione e efficienza di implementazione
- I programmi sono difficili da leggere

Tipo

- tipo = insieme di valori + op. per creare, accedere, e modificare il valore
- *protegge le variabili da operazioni insensate*
- variabile = instanze del tipo
- Un linguaggio può essere
  - non tipizzato
  - tipizzato dinamicamente
  - tipizzato staticamente

Controllo del Tipo

- Verifica il corretto uso delle variabili
- Linguaggi staticamente tipizzati
  - Le variabili sono legate staticamente al tipo prima dell'esecuzione
- Linguaggi dinamicamente tipizzati
  - Le variabili sono polimorfiche
- Il controllo del tipo può essere statico
  - Per i linguaggi tipizzati staticamente
  - Per certe categorie di linguaggi tipizzati dinamicamente (vedremo questo caso con i linguaggi OO)

l_value

- Area di memoria legata alla variabile
- ***durata***: periodo di tempo nel quale tale legame esiste
- l_value necessario per mantenere un r_value
- (dati) oggetto <l_value, r_value>.
- allocazione di memoria: ottiene lo spazio + lega l-value alla variabile
- la durata si estende dall'allocazione alla deallocazione della memoria

allocazione statica vs. dinamica
automatico vs. esplicito

r_value

- codifica del valore della variabile memorizzato nella locazione (l_value)
- interpretato in accordo con il tipo della variabile
- instruzioni
  - accesso alle variabili attraverso l_value — *lato sinistro dell'assegnazione*
  - modifica del loro r_value — *lato destro dell'assegnazione*
- legame tra la variabile e il valore: generalmente dinamico
- costante simbolica: legame statico

const float pi = 3.1416;
circumference= 2 * pi * radius;

Inizializzazione

- Qual è r_value subito dopo che la variabile è stata creata?

C	***Ada***
int i = 0, j = 0;	i, j: INTEGER := 0;

- Qual è se non c'è inizializzazione?
- Soluzioni differenti con languaggi/implementazioni differenti
- *ignoriamo*: il valore iniziale è la stringa di bit contenuta nell'area di memoria
- *inizializzazione definita dal sistema*: e.g., int = zero, char = blank
- *valore speciale "indefinito"* per inizializzare la variabile

Variabili anonime e riferimenti

- possiamo accedere alle variabili attraverso r_value di un'altra variabile
- chiamato *riferimento* (*puntatore*) alla variabile

  C/C++

  ```
  int x = 5;
  int* px;
  px = &x;
  ```

  genera un oggetto intero il cui r_value è 5
  - direttamente accessibile attraverso x
  - indirettamente accessibile attraverso px

Moduli

Unità di composizione di un programma sviluppabili separatamente

Es: sottoprogrammi in assembler; subroutines in FORTRAN; procedure e funzioni in Pascal e Ada; funzioni in C

Le *funzioni* restituiscono un valore; le *procedure* no

```
/* somma dei primi n interi positivi */
int sum (int n)  // intestazione
{       // corpo
        int i, s;
        s = 0;
        for (i = 1; i <= n ; ++i)
                s+= i;
        return s;
}
```

Routine objects

- I moduli hanno un nome, scope, tipo, l_value, e r_value
- L'attivazione è ottenuta con una *chiamata* al modulo
- Possono accedere ad item locali, non locali, e globali
- Hanno una **intestazione** e un **corpo**
- L'intestazione definisce il tipo del modulo
- La chiamata deve essere conforme con il corrispondente tipo di modulo

l_value e r_value

- l_value
  posizione di memorizzazione del corpo del modulo
- r_value
  corpo correntemente legato al modulo
  •• generalemente il legame è statico, stabilito nella fase di traduzione
  •• alcuni linguaggi supportano variabili di tipo modulo
  ==> un valore modulo può essere assegnato
  variabili di tipo "puntatore a modulo"

```
int(*ps)(int);      // ps è un puntatore ad un modulo
ps = & sum;         // sum è definito precedentemente
int i = (*ps)(5);   // invoca il modulo sum
```

Dichiarazione e definizione

Alcuni linguaggi (es. Pascal, Ada, C, C++) distinguono tra ***dichiarazione*** e ***definizione***

- La *dichiarazione* introduce l'intestazione del modulo senza specificare il corpo; viene specificato lo scope
- La *definizione* specifica sia l'intestazione che il corpo

  la distinzione è fatta per supportare la mutua ricorsione

  (traduzione one-pass)

```
int A (int x, int y);         // dichiarazione
float B (int z) {             // definizione
        int w, u;
        w = A (z, u);         // A è visibile a questo punto
        …
};
```


Rappresentazione di un modulo durante l'esecuzione

- **segmento di codice** – istruzioni dell'unità – (di solito) contenuto fisso
- **record di attivazione** – contenuto modificabile
  - informazioni sull'esecuzione del modulo (***stato***)
- **ambiente di riferimento** di un istanza U
  - variabili locali ad U (ambiente locale)
  - variabili non-locali ad U (ambiente non-locale)
    - modifica: *side-effect*

Moduli ricorsivi

- Tutte le istanze della stessa unità sono formate da:
  - stesso segmento di codice
  - differente record di attivazione
- Il legame tra il record di attivazione e il suo segmento di codice è *dinamico*

Parametri

I parametri supportano il flusso di informazioni inter-unità
Lo stesso effetto può essere ottenuto utilizzando le variabili non-locali
l'uso dei parametri ==> migliora la leggibilità e la modificabilità
Distinzione tra parametri ***formali*** e ***attuali***
Corrispondenza attraverso
- ***metodo posizionale***
- ***associazione nominale***

metodo posizionale
routine S (F1,F2, . . . Fn);
call S (A1, A2,... An

associazione nominale
procedure Example (A: T1; B: T2 := B1;
C: T3);
Example (X, Y, Z);
Example (X, C => Z)
Example (C => Z, A => X, B => Y);

Overloading

```
int i, j, k;
float a, b, c;
...
i = j + k;
a = b + c;
```

+ sovrapposto: somma di interi e somma di floating-point

OVERLOADING

- ***un nome, in un dato istante, è legato a più di un'entità***
- ***l'occorrenza del nome fornisce sufficienti informazioni per rendere non ambiguo il legame***

ESEMPI

`a = b + c + b ( );`

b indica sia una variabile che una routine

`a = b ( ) + c + b (i);`

chiamate a due distinte routine

Aliasing

Opposto dell'overloading

- due nomi indicano la stessa entità nello stesso punto del programma
- loro rendono disponibile lo stesso oggetto nello stesso ambiente di riferimento
- modifiche fatte attraverso name1, affetti visibili su name2

May lead to error prone and difficult to read programs.

```
int i;
int fun (int& a);
{ . . .
a = a + 1;
printf (i);
. . .
}
main ( )
{ . . .
x = fun (i);
}
```

i and a denote the same object

```
int x = 0;
int* i = &x;
int* j = &x;
```

**i, *j and x are aliases*