Corso di Laurea Magistrale in Ingegneria Informatica
A.A. 2010-2011

Linguaggi Formali e Compilatori

Generazione del codice intermedio
Symbol Table

Giacomo PISCITELLI

Le rappresentazioni intermedie

Il front-end di un compilatore costruisce una rappresentazione intermedia del programma sorgente.

Come rappresentazione intermedia possono essere utilizzati diversi linguaggi:

- Il parse tree (che chiameremo albero sintattico) o l'albero sintattico astratto (AST)
- Una notazione postfissa (*postfix notation*)
- Un *three-address code* (*4-uple o quadruple*)

Talvolta si utilizzano linguaggi intermedi più evoluti:

- java – java virtual machine
- prolog – warren abstract machine

Più frequentemente vengono usati l'AST e le Quadruple.

AST e Quadruple

Durante l'analisi sintattica vengono creati, per rappresentare i costrutti di programmazione significativi, i nodi di un albero sintattico. Via via che l'analisi procede, ai nodi viene aggiunta informazione, sotto forma di attributi, diversi a seconda del tipo di traduzione che s'intende effettuare, creando così un **AST**.

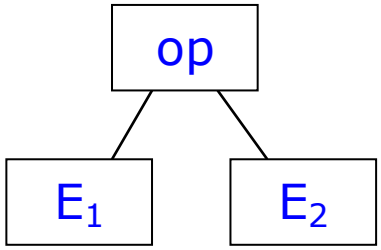
Una **quadrupla**, a sua volta, è una sequenza di istruzioni elementari, quale ad esempio l'addizione di due valori. In questo caso non c'è alcuna struttura gerarchica.

Prima di passare ad esaminare i 2 tipi di rappresentazione, si osservi che il front-end controlla che le istruzioni del *source program* rispettino le regole sintattiche e semantiche che governano il linguaggio di programmazione del source program. Questo controllo (detto controllo statico, perché svolto durante la compilazione) assicura che gli errori (sintattici e semantici) siano rilevati e segnalati nel corso della compilazione.

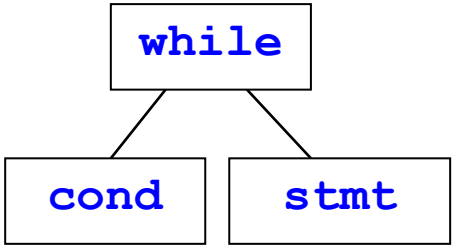
In realtà è possibile che un compilatore, mentre costruisce l'albero sintattico, produca contemporaneamente le istruzioni elementari. La produzione delle quadruple, però, avviene nel corso della costruzione dell'albero sintattico, senza che questo sia ancora completato. Infatti il compilatore, e in particolare il parser, memorizza i nodi dell'albero sintattico e i loro attributi (necessari per la successiva fase di analisi semantica o altro) mano a mano che si definisce la struttura dell'albero. Ciò comporta che le parti dell'albero necessarie per costruire le quadruple siano disponibili quando necessarie, ma scompaiano quando non più necessarie.

Schema di traduzione basato sulla costruzione dell'AST

Un generico nodo non terminale di un AST rappresenta un'espressione formata applicando un operatore **op** a due operandi E_1 ed E_2 (costituiti da 2 sotto-espressioni).



Un AST può essere costruito per qualunque costrutto di una grammatica. Ciascun costrutto può essere rappresentato da un nodo dotato di nodi figli, che rappresentano i componenti semanticamente significativi del costrutto. Per esempio, il seguente nodo dell'albero sintattico per uno statement **while (cond) stmt** ha un operatore **while** e due figli



che sono l'albero sintattico per il componente **cond** e quello per **stmt**.

Costruzione dell'albero sintattico

A ogni nonterminale dell'albero sintattico che si diparte dal nodo **`while`** è associato un nodo. Quando, perciò, il parser riconosce la regola corrispondente a un nonterminale, viene creato un nodo corrispondente alla parte sinistra della regola con figli corrispondenti alla parte destra.

Mentre per i nodi corrispondenti a costrutti che iniziano con una parola chiave si può usare quest'ultima come codice del nodo, i nodi corrispondenti ad espressioni condizionali (cfr. **`cond`**) possono essere gestiti definendo 2 operatori **`ifelse`** e **`if`** per statement if rispettivamente con e senza la parte else.

La costruzione dell'albero sintattico procede poi come indicato alle sect. 2.8.2 e 5.3.1 del testo, in maniera analoga sia per gli statement che per le espressioni. Si raccomanda di tener conto di quanto riportato alle sect. 5.5e 5.5.4 del testo, stante il tipo di parsing bottom-up con tecnica LALR che Yacc/Bison effettuano.

La costruzione dell'AST si conclude con gli eventuali ulteriori controlli sintattici statici e con i controlli di coerenza di tipo (*type checking*).

La rappresentazione intermedia (IR)

Terminate le operazioni associate alla costruzione dell'AST, passiamo a vedere come processare l'AST e come emettere il codice nella rappresentazione intermedia scelta.

Un compilatore può usare una sequenza di differenti IR

- → le IR di alto livello preservano la struttura di alto livello del programma es., classi, cicli, statement, espressioni
- → le IR di basso livello supportano espressioni esplicite e ottimizzazione di dettagli realizzativi

La selezione della IR dipende dall'obiettivo di ogni passo

- → una traduzione source-to-source: quando si mira ad un linguaggio simile a quello sorgente, allora la IR più adatta è il parse tree o l'AST
- → una traduzione source-to-code: tenderà a far uso di un linguaggio vicino al machine code, e quindi il *linear three-address code* sarà il più opportuno

Il linguaggio delle quadruple

Per avvicinarsi al linguaggio macchina si utilizzerà un linguaggio pozzo costituito da quadruple (4-uple), dette anche codice a 3 indirizzi (three address code).

Il linguaggio delle quadruple è molto simile al linguaggio macchina ed è costituito da una sequenza di istruzioni della forma

```
x = y op z
```

ove `x`, `y` e `z` sono identificatori, costanti o variabili temporanee generate dal compilatore; e `op` rappresenta un operatore.

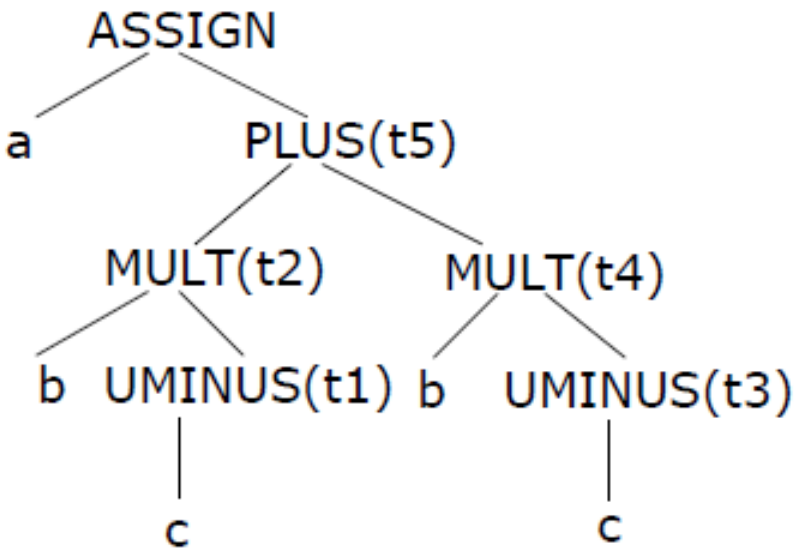
Gli array possono essere gestiti usando le seguenti 2 varianti di istruzioni:

```
x[y] = z
x = y[z]
```

Example: translating expressions

Input: `a := b* -c + b * -c`

Abstract syntax tree:



Three-address code:

```
t1 := - c
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t4
a := t5
```

Storing three-address code

Per memorizzare tutte le istruzioni in una tabella di quadruple

- ogni istruzione ha 4 campi: op, arg1, arg2, result
- ogni istruzione ha un numero nella tabella ➔ indice dell'istruzione

Three-address code

```
t1 := - c
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t4
a := t5
```

Quadruple entries

	op	arg1	arg2	result
(0)	Uminux	c		t1
(1)	Mult	b	t1	t2
(2)	Uminus	c		t3
(3)	Mult	b	t3	t4
(4)	Plus	t2	t4	t5
(5)	Assign	t5		a

Le quadruple

Le quadruple vengono eseguite in rigorosa sequenza, a meno che un salto condizionato o incondizionato forzi altrimenti il flusso di esecuzione (control flow).

Esempio di istruzioni di salto condizionato

```
jmpt y,,L1   // jump (goto) to the instruction labelled L1 if y == true
jmpf y,,L1   // jump (goto) to the instruction labelled L1 if y == false
```

Esempio di istruzione di salto incondizionato

```
jmp  ,,L     // jump (goto) to the instruction labelled L
```

Nel seguito utilizzeremo la seguente notazione per una quadrupla:

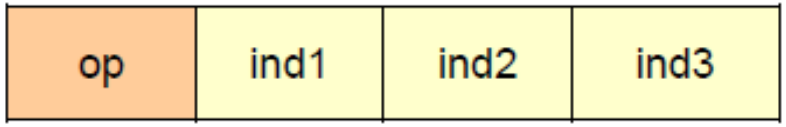
```
op y,z,x
```

con il significato applica l'operatore *op* a **y** e **z**
e restituisci il risultato in **x**

Rappresentazione del codice a quadruple: tipicamente

∀ istruzione → record di campi
codice completo → array o lista a puntatori dei record

Record: tipicamente



Le quadruple

Operatori binari: `op y,z,result`

dove `op` è un operatore binario o logico.

Esempio:

```
add a,b,c
gt a,b,c
addr a,b,c
addi a,b,c
```

Operatori unari: `op y,,result`

dove `op` è un operatore binario o logico.

Esempio:

```
uminus a,,c
not a,,c
inttoreal a,,c
```

Move Operator: `mov y,,result`

il contenuto di `y` è copiato in `result`.

Esempio:

```
mov a,,c
movi a,,c
movr a,,c
```

Le quadruple

Unconditional Jumps: **`jmp ,,L`**

salto all'istruzione di etichetta **`L`**

Esempio: **`jmp ,,L1`** // salto a L1
`jmp ,,7` // salto all'istruzione 7

Conditional Jumps: **`jmp`***`relop`* **`y,z,L`**

salta all'istruzione con etichetta **`L`** se il risultato di **`y relop z`** è vero.
Se il risultato è false continua con l'istruzione seguente

Esempio: **`jmpgt y,z,L1`** // jump to L1 if y>z
`jmpgte y,z,L1` // jump to L1 if y>=z
`jmpe y,z,L1` // jump to L1 if y==z
`jmpne y,z,L1` // jump to L1 if y!=z

oppure

`jmpnz y,,L1` // jump to L1 se y == 0
`jmpz y,,L1` // jump to L1 if y != 0
`jmpt y,,L1` // jump to L1 if y == true
`jmpf y,,L1` // jump to L1 if y == false

Codice intermedio: quadruple per le altre istruzioni

Parametri di Procedure: `param x,,`

Chiamata di Procedure : `call p,n,`

dove **x** è un parametro "attuale"

Esempio:

```
param x1,,
param x2,,
        ➔ p(x1,...,xn)
param xn,,
call p,n,
```

f(x+1,y) ➔

```
add x,1,t1
param t1,,
param y,,
call f,2,
```

Indexed Assignments:

`move y[i],,x` con il significato x = y[i]

`move x,,y[i]` con il significato y[i] = x

Address and Pointer Assignments:

`moveaddr y,,x` con il significato x = &y

`movecont y,,x` con il significato x = *y

TAVOLA DEI SIMBOLI

Structure of a Compiler

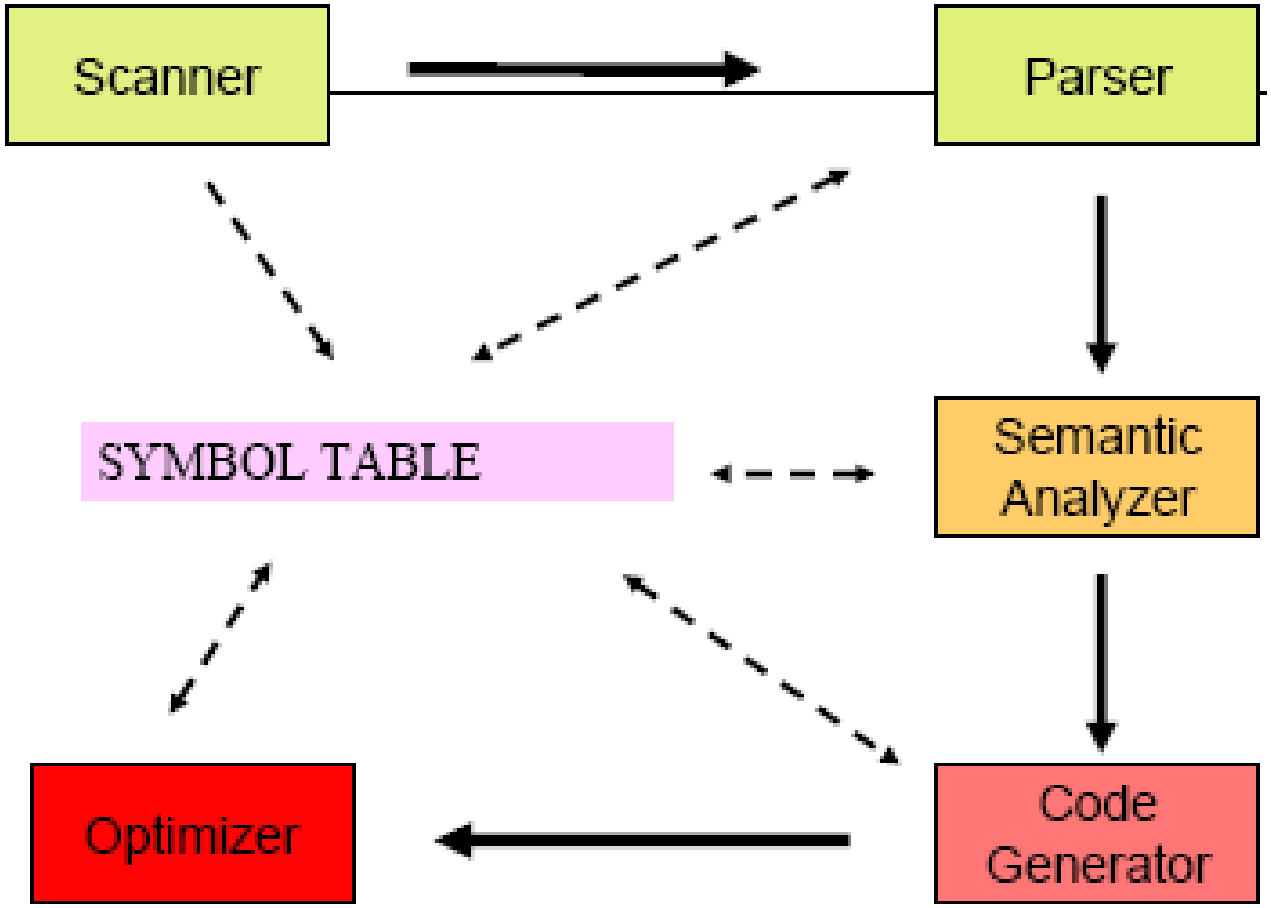


TAVOLA DEI SIMBOLI: funzione

Ruolo della Symbol Table (ST)

La **symbol table** è una struttura di dati, usata da un compilatore per tener traccia dei costrutti del source program e, in particolare, della semantica degli identificatori; essa contiene un record per ogni identificatore, con campi per i suoi vari attributi, ad es. la stringa di caratteri (lessema) che lo identifica, il tipo, lo *scope* o porzione di programma in cui si applica la dichiarazione di un identificatore, l'indirizzo di memoria (se viene memorizzato), il valore (se viene calcolato), ecc.
È frequente l'uso di una symbol table separata per ogni scope di programma.

Le tipiche **operazioni** su una ST sono:

- Find: per verificare se un id è già presente nella ST.
- Insert: per aggiungere un id nella ST.
- Delete: per cancellare un id dalla ST, quando la sua dichiarazione non è più valida, i.e. il suo *scope* si è concluso.

I record di una ST vengono creati nell'ambito del front-end e l'informazione in essi contenuta viene raccolta incrementalmente nelle varie fasi.

TAVOLA DEI SIMBOLI: creazione e uso

Lo scanner può creare un entry nella ST appena riconosciuto un lessema. Più spesso lo scanner ritorna al parser solo un token, per es. **`id`**, assieme ad un pointer al relative lessema: infatti solo il parser può decidere se far uso di un entry precedente oppure creare un nuovo entry per l'**`id`**.

Tutte le parti del compilatore (FE e BE) fanno riferimento ai nomi contenuti nella ST mediante i puntatori ai corrispondenti elementi della ST. Ciò significa che si fa riferimento alla ST fino alla fine della compilazione e che il contenuto della ST può essere modificato anche durante la fase di ottimizzazione.

TAVOLA DEI SIMBOLI: tipi di realizzazione

La ST può essere realizzata mediante:

- ✓ **Una lista non ordinata:**
  - Quando gli identificatori sono in numero ridotto;
  - Il codice per gestire una tale ST è semplice, ma la performance è cattiva se il numero di identificatori cresce.
- ✓ **Una lista lineare ordinata:**
  - Si può usare la ricerca binaria;
  - Inserimento e cancellazione sono "dispendiose";
  - Il codice di gestione è relativamente semplice.
- ✓ **Un albero binario di ricerca:**
  - Il tempo per operazione (ricerca, inserimento o cancellazione) per n elementi nella ST varia secondo O(log n);
  - Il codice di gestione è relativamente difficile.
- ✓ **Hash table:**
  - È la struttura più comunemente usata per la ST;
  - È molto efficiente, se lo spazio di memoria disponibile è adeguatamente più grande di quello richiesto dagli elementi della ST;
  - La performance può essere cattiva se si è "sfortunati" o la tabella si riempie;
  - Il codice di gestione non è difficile.

TAVOLA DEI SIMBOLI: insieme limitato di elementi

Se l'universo delle chiavi è piccolo allora è sufficiente utilizzare una *tabella ad indirizzamento diretto*.

Una tabella ad indirizzamento diretto corrisponde al concetto di array:

- ✓ ad ogni chiave possibile corrisponde una posizione, o slot o bucket, nella tabella
- ✓ una tabella restituisce in tempo O(1) il dato memorizzato (costituito dalla chiave e da altri dati ad essa associati e detti dati satellite) nello slot di posizione indicato tramite la chiave

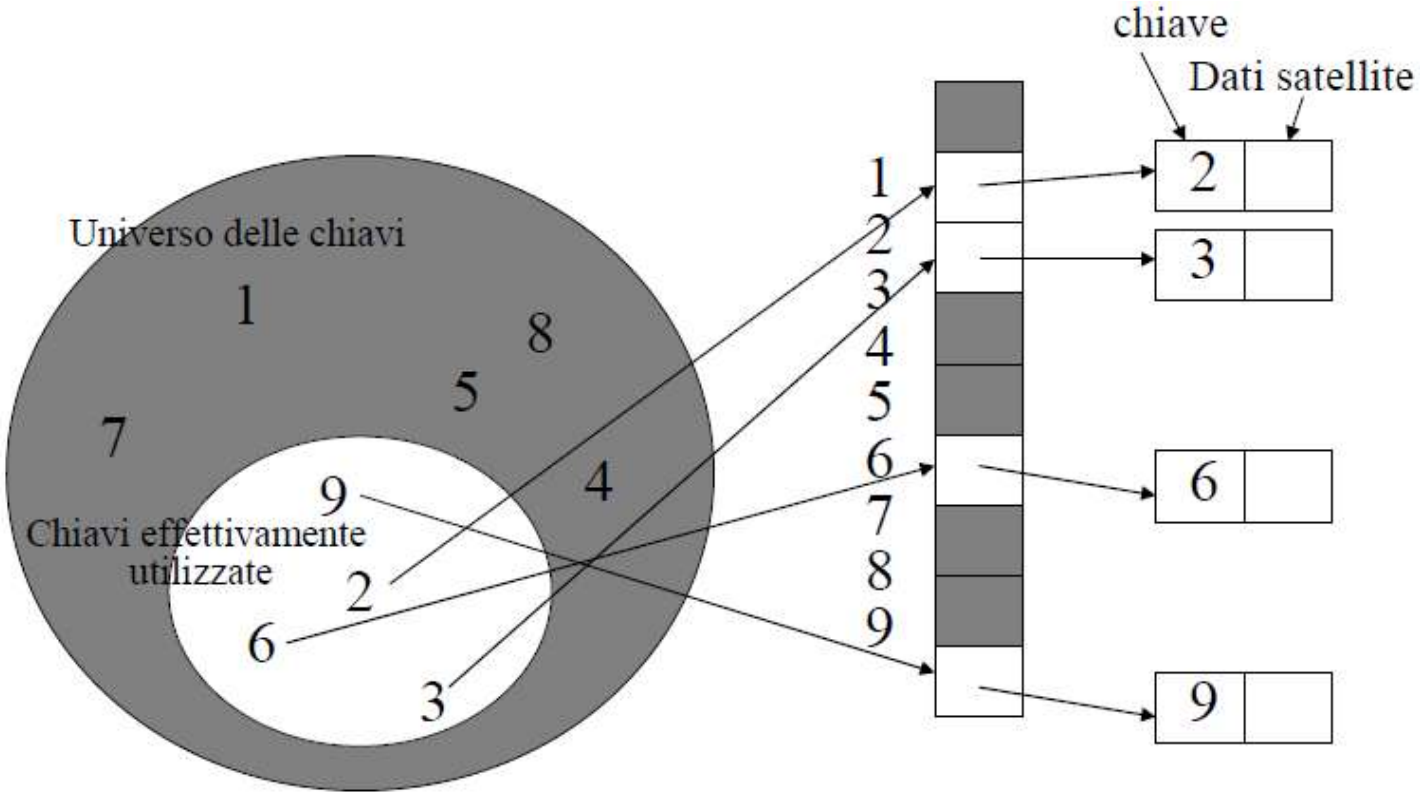


TAVOLA DEI SIMBOLI: insieme esteso di elementi

Se l'universo delle possibili chiavi è molto grande, non è possibile o conveniente utilizzare il metodo delle tabelle ad indirizzamento diretto

- ✓ può non essere possibile a causa della limitatezza delle risorse di memoria
- ✓ può non essere conveniente perché se il numero di chiavi effettivamente utilizzato è piccolo si hanno tabelle quasi vuote e viene quindi allocato spazio inutilizzato

Con il metodo di indirizzamento diretto un elemento con chiave k viene memorizzato nella tabella in posizione **`k`**

Con il metodo hash un elemento con chiave k viene memorizzato nella tabella in posizione **`h(k)`**

La funzione **`h(.)`** è detta *funzione hash*

Lo scopo della funzione hash è di definire una corrispondenza tra l'universo **`U`** delle chiavi e le posizioni di una tabella hash **`T[0..m-1]`**

$$h : U \rightarrow \{0,1,\ldots,m-1\}$$

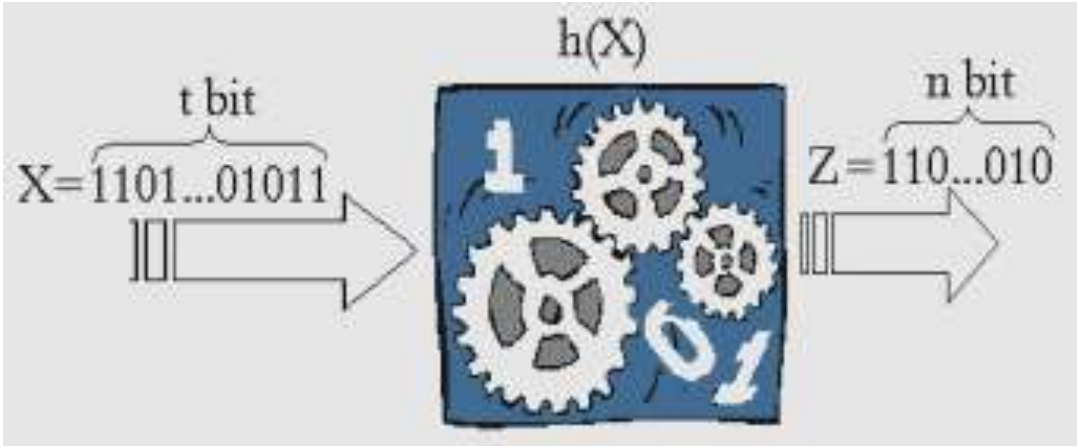
TAVOLA DEI SIMBOLI: Hash function

La ST realizzata mediante una tabella hash si basa sull'uso di una funzione hash.

Talvolta si richiede che su un insieme dinamico di elementi (ad es. record) si possano effettuare solamente operazioni di inserzione, cancellazione e ricerca senza, ad esempio, che sia necessario dover ordinare l'insieme degli elementi o restituire l'elemento massimo, o il successore.

Questi tipi di strutture dati prendono il nome di *dizionari.*

Un'**hash-function**, da un punto di vista matematico, è una funzione non invertibile che trasforma un testo di lunghezza arbitraria in una stringa di lunghezza fissa.
Le funzioni hash stanno hanno una notevole importanza sia per questioni di sicurezza informatica che per altri scopi. Possiamo applicare una funzione hash ad un qualsiasi oggetto digitale: ad un testo ma anche ad un qualsiasi file, come ad esempio uno script di un sistema web oppure una foto.



Funzione hash come macchina che comprime sequenze di bit (immagine di Alfredo De Santis)

TAVOLA DEI SIMBOLI: Hash function

Un elemento con chiave **k** ha posizione pari al *valore hash di* **k** denotato con **h(k)**

Tramite l'uso di funzioni hash il range di variabilità degli indici passa da |**U**| a ***m***

Utilizzando delle dimensioni *m* comparabili con il numero di dati effettivi da gestire si riduce la dimensione della struttura dati garantendo al contempo tempi di esecuzione dell'ordine di O(1)

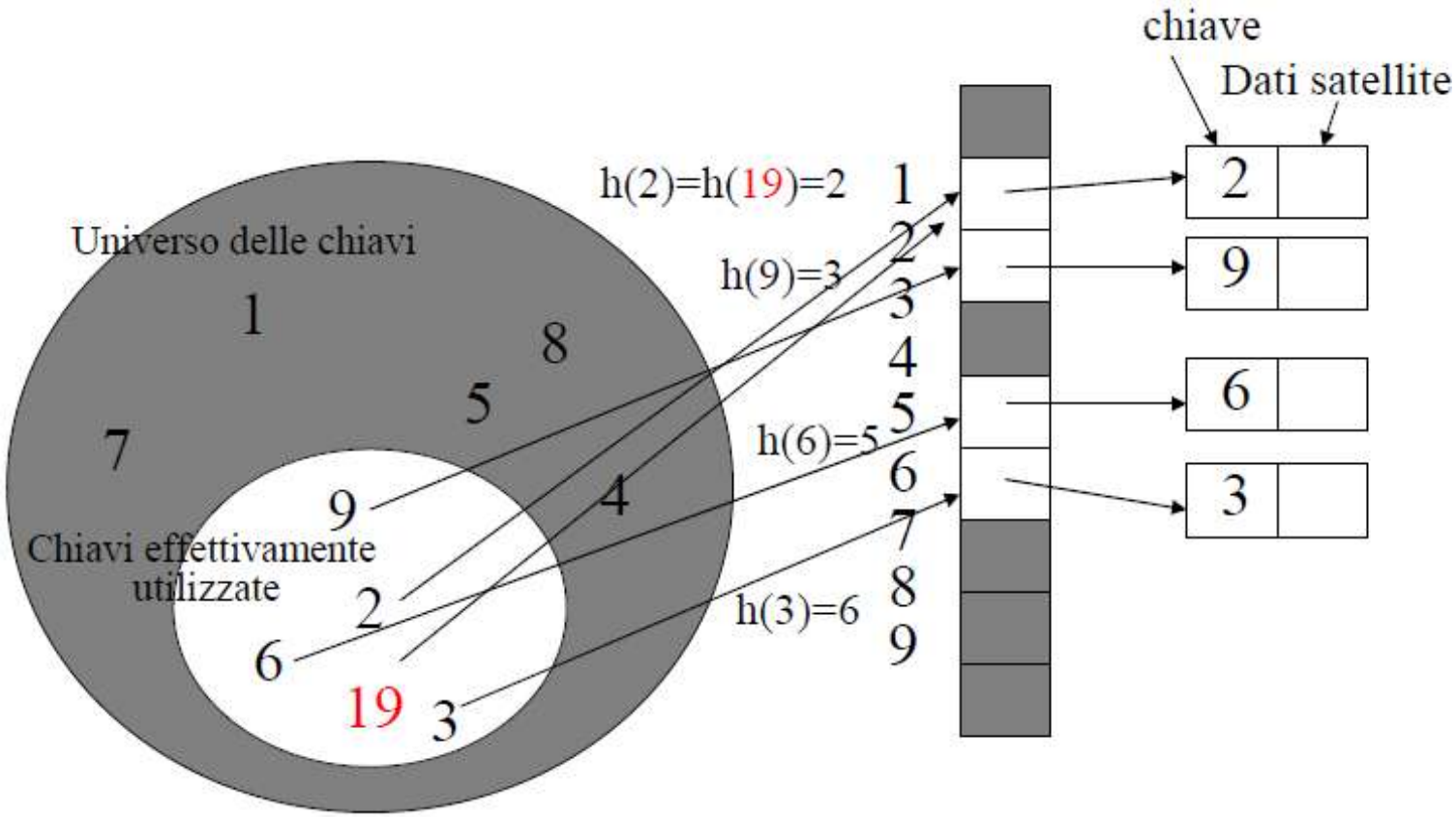
Memorizzazione dei dati satellite

- E' possibile memorizzare i dati satellite direttamente nella tabella
- .. oppure memorizzare solo puntatori agli oggetti veri e propri
- Si deve distinguere l'assenza di un oggetto (oggetto NIL) dal caso particolare di un valore dell'oggetto stesso
  Ad esempio se il dato è un intero positivo è possibile assegnare il codice -1 per indicare NIL

Hash function e collisioni

Necessariamente la funzione hash non può essere bigettiva o biunivoca, ovvero *due chiavi distinte possono produrre lo stesso valore hash*

Quando questo accade si dice che si ha una ***Collisione***



Hash function

Quali sono le caratteristiche di una funzione hash?

Criterio di uniformità semplice: il valore hash di una chiave **k** è uno dei valori **0..m-1** in modo equiprobabile

Formalmente: se si estrae in modo indipendente una chiave **k** dall'universo **U** con probabilità **P(k)** allora:

$$\sum_{k:h(k)=j} P(k) = 1/m \text{ per } j=0,1,\ldots,m-1$$

Cioè se si partiziona l'universo **U** in **m** sottoinsiemi tali per cui nello stesso sottoinsieme consideriamo tutte le chiavi che sono mappate dalla funzione **h** in **j**, allora vi è la stessa probabilità di prendere un elemento da uno qualsiasi di questi sottoinsiemi.

Tuttavia non sempre si conosce la distribuzione di probabilità delle chiavi **P**

Esempio:

Se si ipotizza che le chiavi **k** siano numeri reali distribuiti in modo indipendente ed uniforme nell'intervallo [0,1], cioè $k \in [0,1]$, allora la funzione

$$h(k) = \lfloor k \; m \rfloor$$

soddisfa il criterio di uniformità semplice.

Hash function

Un altro requisito è che una "buona" funzione hash dovrebbe utilizzare tutte le cifre della chiave per produrre un valore hash.

In questo modo valgono le ipotesi sulla distribuzione dei valori delle chiavi nella loro interezza, altrimenti dovremmo considerare la distribuzione solo della parte di chiave utilizzata.

In genere le funzioni hash assumono che l'universo delle chiavi sia un sottoinsieme dei numeri naturali
Quando questo non è verificato si procede convertendo le chiavi in un numero naturale (anche se grande).

Un metodo molto usato è quello di stabilire la conversione fra sequenze di simboli interpretati come numeri in sistemi di numerazione in base diversa

Convertire le chiavi in numeri naturali

Per convertire una stringa in un numero naturale si considera la stringa come un numero in base 128.

Esistono cioè 128 simboli diversi per ogni cifra di una stringa.

E' possibile stabilire una conversione fra ogni simbolo e i numeri naturali (codifica ASCII ad esempio).

La conversione viene fatta considerando il numero espresso in un sistema posizionale in base 128.

Es: per convertire la stringa "casa" si ha:

$c\ a\ s\ a_{(128)} =$

$c_3\ a_2\ s_1\ a_{0\ (128)} =$

$'c' * 128^3 + 'a' * 128^2 + 's' * 128^1 + 'a' * 128^0 =$

$99 * 128^3 + 97 * 128^2 + 115 * 128^1 + 97 * 128^0 =$

$99 * 2097152 + 97 * 16384 + 115 * 128 + 97 * 1 = 209222113_{(10)}$

Hash function

Come realizzare una funzione hash?

- Metodo della divisione
- Metodo della moltiplicazione
- Metodo della funzione hash universale (non lo vedremo)

Hash function

Metodo della divisione

La funzione hash è del tipo:

$$h(k) = k \bmod m$$

Cioè il valore hash è il resto della divisione di k per m.

Caratteristiche:

- il metodo è veloce, ma si deve fare attenzione ai valori di m;
- m deve essere diverso da 2^p per un qualche p, altrimenti fare il modulo in base m corrisponderebbe a considerare solo i p bit meno significativi della chiave.

Dovremmo quindi garantire che la distribuzione dei p bit meno significativi sia uniforme. Analoghe considerazioni per m pari a potenze di 10.

Buoni valori sono numeri primi non troppo vicini a potenze del due.

Hash function

Esempio di metodo della divisione

Attenzione! Se si usasse m=100, allora:

k	**h(k)**
123	23
2323	23
128723	23

Se si usasse m=23, allora:

k	**h(k)**
10110101	101
111101	101
100011101	101

Hash function

Metodo della Moltiplicazione

Il metodo della moltiplicazione per definire le funzioni hash opera in due passi:

- si moltiplica la chiave **k** per una costante **A** in [**0**,**1**] e si estrae la parte frazionaria del risultato;
- si moltiplica questo valore per **m** e si prende la parte intera.

Analiticamente si ha:

$$h(k) = \lfloor m(k\ A \bmod 1) \rfloor$$

Un vantaggio del metodo è che non ha valori critici di **m**.

Hash function

Esempio di metodo della moltiplicazione

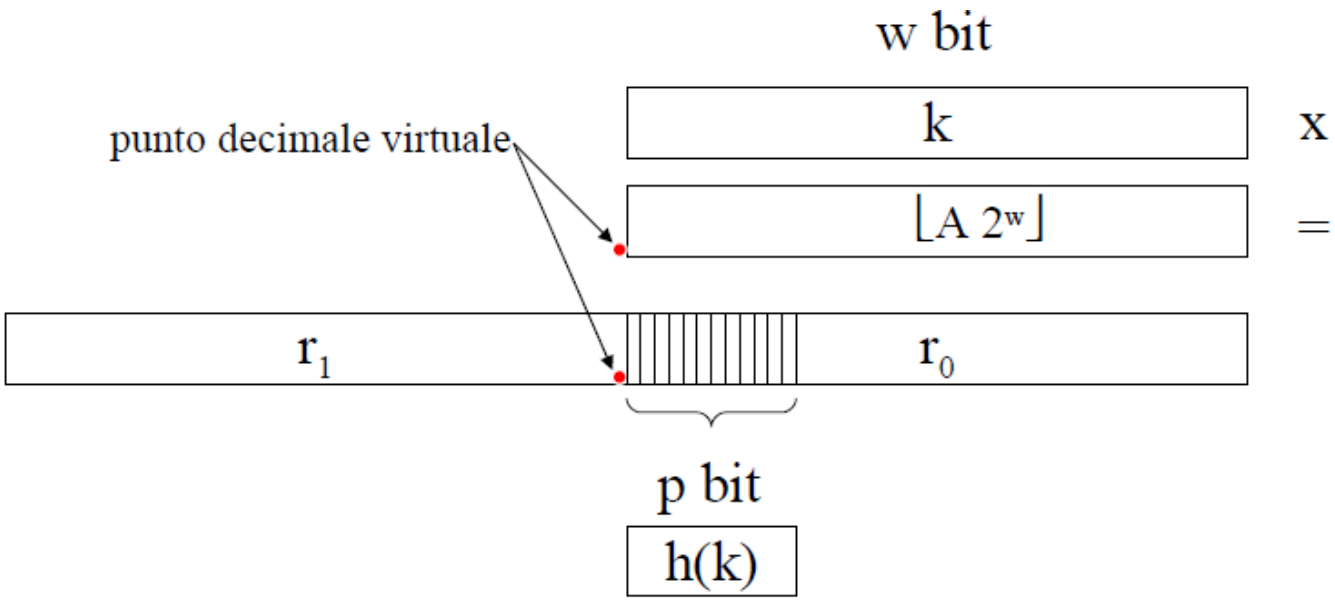
- A=(√5 -1)/2=0.61803…
- sia k=123456 e m=10000
- allora: h(k)= ⌊10000(123456*0.61803.. mod 1)⌋

  = ⌊10000(76300.0041151.. mod 1)⌋ = ⌊10000(0.0041151..)⌋ = ⌊41.151.. ⌋

  = 41

Spesso si sceglie **m**=2^p per un qualche **p** in modo da semplificare i calcoli.
Una implementazione veloce di una **h** per moltiplicazione è la seguente:

- supponiamo che la chiave **k** sia un numero codificabile entro **w** bit dove **w** è la dimensione di una *parola* del calcolatore
- si consideri l'intero anch'esso di **w** bit ⌊A 2^w⌋
- il prodotto **k*** ⌊A 2^w⌋ sarà un numero intero di al più 2w bit
- consideriamo tale numero come $r_1 2^w + r_0$
- r_1 è la parola *più significativa* del risultato e r_0 quella *meno significativa*
- il valore hash cercato consiste nei p bit più significativi di r_0


w bit
punto decimale virtuale
k
x
⌊A 2^w⌋
=
r1
r0
p bit
h(k)

Hash function

Metodi per la risoluzione delle collisioni

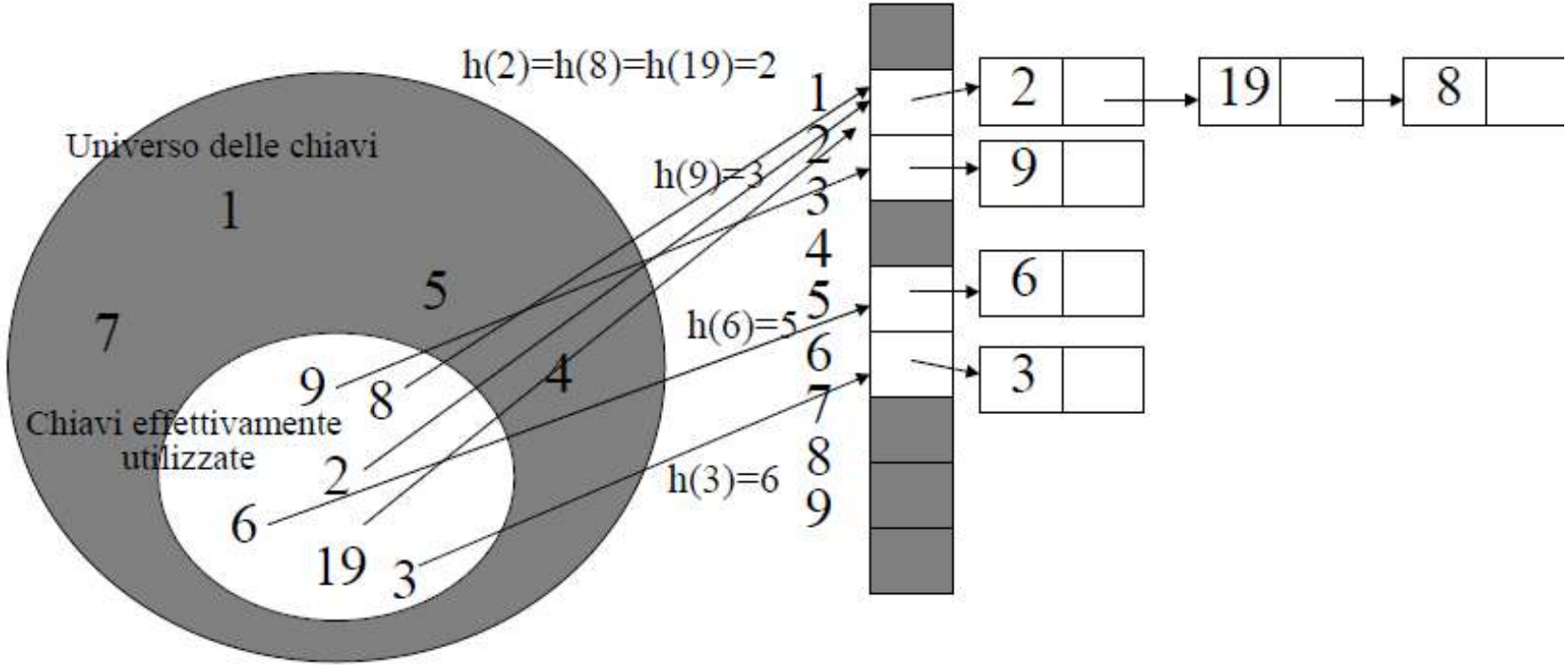
Per risolvere il problema delle collisioni si impiegano principalmente due strategie:

- metodo di concatenazione
- metodo di indirizzamento aperto

Hash function: Metodo di concatenazione

L'idea è di mettere tutti gli elementi che collidono in una lista concatenata.

La tabella contiene in posizione j un puntatore alla testa della j-esima lista oppure un puntatore nullo se non ci sono elementi.

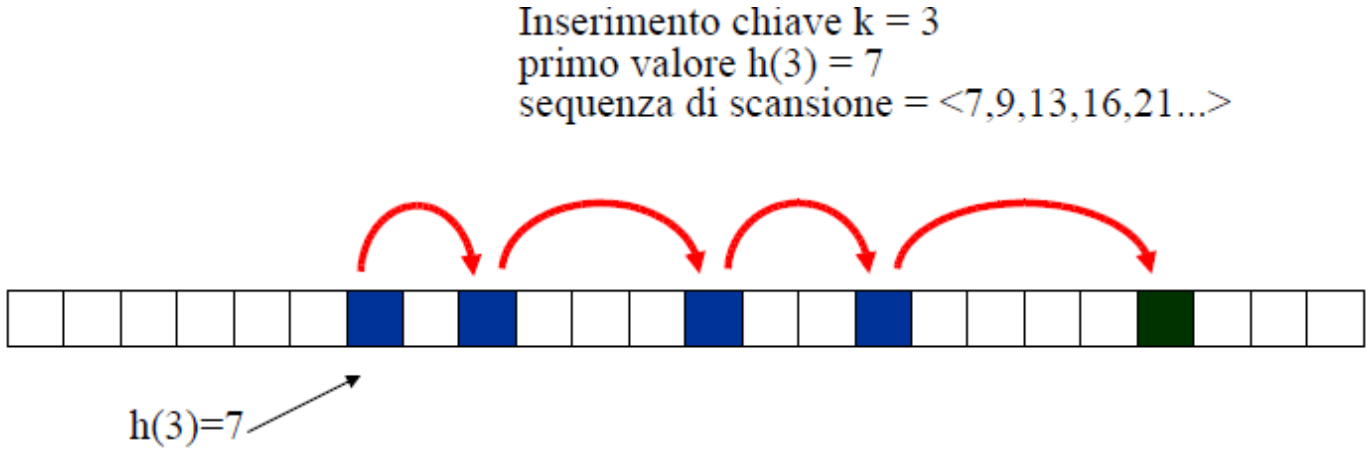


Hash function: Metodo di indirizzamento aperto

- ✓ Scansione lineare
- ✓ Scansione quadratica
- ✓ Hashing doppio

L'idea è di memorizzare tutti gli elementi nella tabella stessa.
In caso di collisione si memorizza l'elemento nella *posizione successiva*.
Per l'operazione di ricerca si esaminano tutte le posizioni ammesse per la data chiave in sequenza.
Non vi sono liste né elementi memorizzati fuori dalla tabella.

Per eseguire l'inserzione si genera un valore hash data la chiave e si esamina una *successione di posizioni della tabella* (*scansione*) a partire dal valore hash fino a trovare una posizione vuota dove inserire l'elemento.



Hash function: performance

Problemi di prestazioni

a. La dimensione dell'hash table deve essere adeguatamente più grande del numero massimo di possibili elementi.

b. Gli elementi usati frequentemente dovrebbero essere distinti.
- parole chiave o riservate.
- nomi "corti di variabili, es., i, j e k.
- identificatori usati frequentemente, es., main.

c. Gli elementi in una ST dovrebbero essere uniformemente distribuiti.

TAVOLA DEI SIMBOLI: contenuto

Una o più Symbol Table conservano informazione su:

- ✓ *identificatori*
- ✓ *label*
- ✓ valori numerici (*constanti*)
- ✓ *stringhe di caratteri*
- ✓ *variabili temporanee* (variabili compiler-generated)

Risulta efficiente conservare separatamente valori numerici e stringhe di caratteri.

Potrebbe convenire, a volte, avere una tabella separata per le label.

Le ST possono essere costruite una per ogni procedura o blocco.

Può esserci una tabella separata per i dati globali.

TAVOLA DEI SIMBOLI: contenuto

Vi è tutta una ***varietà di specifici tipi di identificatori***: variabili scalari, vettori e strutture (record), procedure e funzioni, costanti definite, variabili temporanee.

Alcune tipiche informazioni associate con gli identificatori:
Il nome o valore
Il tipo di dato
Dimensione e numero di elementi (per i vettori)
Altra informazione di validità o memorizzazione
Tipi di risultati, parametri (formali), prototipi

L'informazione salvata dipende dal tipo di elementi contenuti nella ST e quindi potrebbe essere necessario un formato flessibile di ogni entry.

Usage of symbol table with YACC/Bison

- ✓ Define symbol table routines:
- *Find_in_S_ T*(*name,scope*): check whether a name within a particular scope is currently in the symbol table or not.
  - » Return "not found" or
  - » an entry in the symbol table;
- *Insert into S T*(*name,scope*)
  - » Return the newly created entry.
- *Delete from S T*(*name,scope*)

YACC coding: declaration

- *D* → *L V*
  - » {use *Find_in_S_T* to check whether `$2.name` has been declared;
  - » use *Insert_into_S_T* to insert `$2.name` with the type `$1.type`;
  - » allocate `sizeof($1.type)` bytes;
  - » record the storage address in the symbol table entry;
  - » `$$.type = $1.type;`}
- *L* → *L V* ,
  - » {use *Find_in_S_T* to check whether `$2.name` has been declared;
  - » use *Insert_into_S_T* to insert `$2.name` with the type `$1.type`;
  - » allocate `sizeof($1.type)` bytes;
  - » record the storage address in the symbol table entry;
  - » `$$.type = $1.type;`}

  | T
  - » `{$$.type = $1.type;}`
- *T* → *int*
  - » `{$$.type = $1.type;}`
- *V* → *id*
  - » {save `yytext` into `$$.name`;}

YACC coding: declaration

- $D \rightarrow T\ L$
  - » {append each name in **`$2.namelist`** into symbol table, i.e., use *Find_in_S_T* to check for possible duplicated names;
  - » use *Insert_into_S_T* to insert each name in the list with the type **`$1.type`**;
  - » allocate **`sizeof($1.type)`** bytes;
  - » record the storage address in the symbol table entry;}
- $T \rightarrow int$
  - » {**`$$.type = int;`**}
- $L \rightarrow L\ ,\ V$
  - » {insert the new name **`$3.name`** into **`$1.namelist`**;
  - » return **`$$.namelist`** as **`$1.namelist`**;}

  | V
  - » {the variable name is in **`$1.name`**;
  - » create a list of one name, i.e., **`$1.name`**, **`$$.namelist`**;}
- $V \rightarrow id$
  - » {save **`yytext`** into **`$$.name`**;}

YACC coding: Usage of variables

- *Assign_S* → *L var* := *Expression*;
  - » {**`$1.addr`** is the address of the variable to be stored;
  - » **`$3.value`** is the value of the expression;
  - » generate code for storing **`$3.value`** into **`$1.addr`**;}
- *L var* → *id*
  - » { use *Find_in_S_T* to check whether **`yytext`** is already declared;
  - » **`$$.addr`** = storage address;}
- *Expression* → *Expression* + *Expression*
  - » {**`$$.value = $1.value + $3.value;`**}
  - | *Expression* – *Expression*
  - » {**`$$.value = $1.value - $3.value;`**}
  - » **`. . .`**
  - | *id*
  - » { use Find in S T to check whether **`yytext`** is already declared;
  - » if no, error · · ·
  - » if not, **`$$.value`** = the value of the variable **`yytext`**}