11. Rappresentazioni Intermedie

Come già detto, il **front-end** di un compilatore produce una rappresentazione intermedia (IR), che è una forma astratta del programma in fase di compilazione. L'IR codifica la conoscenza del compilatore riguardo al programma, raccogliendo le informazioni rilevanti e semplificando i dettagli del linguaggio di programmazione di partenza.

Il passaggio successivo è il **middle end**, che ha il compito di trasformare questa rappresentazione intermedia in una equivalente che possa essere eseguita in modo più efficiente.
Questo processo di trasformazione viene solitamente suddiviso in più passaggi, detti *passaggi di ottimizzazione*. Ogni passaggio si concentra su un aspetto specifico delle prestazioni del programma, migliorandone l'efficienza e riducendo il consumo di risorse.

Infine, il **back end** converte l'IR ottimizzata in codice nativo, adattandolo all'architettura hardware di destinazione. In questa fase, il compilatore si occupa anche di assegnare i registri, generare il codice macchina e applicare eventuali ottimizzazioni specifiche per l'hardware.

Le decisioni prese nella progettazione della rappresentazione intermedia (IR) influenzano direttamente la velocità e l'efficienza del compilatore. Alcune proprietà fondamentali di una buona IR includono:

- La **facilità di generazione**, che permette al front-end di creare rapidamente la rappresentazione intermedia
- La **facilità di manipolazione**, che consente al middle end di ottimizzare e trasformare agevolmente il codice
- La **dimensione delle procedure**, che deve essere gestibile per evitare complicazioni nella fase di ottimizzazione
- La **libertà di espressione**, che riguarda la capacità dell'IR di rappresentare in modo flessibile una varietà di costrutti del linguaggio
- Il **livello di astrazione**, ovvero quanto l'IR si avvicina al linguaggio di programmazione di partenza o al codice macchina finale, è un altro aspetto cruciale che influenza la compatibilità e la potenza espressiva dell'IR stessa.

L'importanza di queste proprietà varia a seconda del compilatore, poiché alcuni sono progettati per favorire la velocità di compilazione, mentre altri si concentrano sull'ottimizzazione delle prestazioni del codice finale.

Tipi di rappresentazioni intermedie

Le rappresentazioni intermedie (IR) possono essere suddivise in tre grandi categorie.

La prima è la **strutturale**, che è graficamente orientata e fortemente utilizzata nei traduttori da sorgente a sorgente. Queste IR sono tipicamente grandi e utilizzano strutture grafiche per rappresentare la logica del programma, come alberi sintattici o grafi di dipendenza. Questo approccio è particolarmente utile per comprendere e trasformare la struttura del programma, ma tende a generare rappresentazioni di dimensioni significative.

La seconda categoria è quella **lineare**, che rappresenta una sorta di pseudo-codice per una macchina astratta. Il livello di astrazione può variare in base al tipo di compilatore, ma in generale queste IR sono caratterizzate da strutture dati semplici e compatte. Le IR lineari sono più facili da riorganizzare rispetto a quelle strutturali, poiché le loro strutture sono più vicine al codice macchina, rendendole più flessibili in fase di ottimizzazione.

Infine, esiste una categoria **ibrida**, che combina elementi di grafi e codice lineare. Un esempio comune di IR ibrida è il **control-flow graph** (grafo di flusso di controllo), che rappresenta le relazioni di esecuzione tra le diverse istruzioni di un programma sotto forma di grafo, ma mantiene una rappresentazione lineare per le singole istruzioni. Questo approccio bilancia la flessibilità e l'efficienza, offrendo un compromesso tra le strutture complesse della IR strutturale e la compattezza della IR lineare.

Per quanto riguarda il livello di astrazione, le rappresentazioni di tipo strutturale sono generalmente (non sempre) di **alto livello**, mentre quelle lineari sono più spesso di basso livello

Albero di sintassi astratta

Un **albero di sintassi astratta** (AST) è una versione semplificata dell'albero di analisi di una procedura, in cui la maggior parte dei nodi non terminali viene rimossa. Questa struttura rappresenta la sintassi di un programma in una forma più compatta, che riflette la struttura logica delle operazioni senza dettagli superflui.

Ad esempio, per l'espressione $x - 2 * y$, l'AST può essere linearizzato in forme che facilitano la manipolazione rispetto ai puntatori tradizionali. Nella forma **postfissa** (notazione polacca inversa), l'espressione diventa:

$x\ 2\ y\ *\ -$

Nella forma **prefissa** (notazione polacca), invece, l'espressione diventa:

$-\ *\ 2\ y\ x$

Queste forme linearizzate sono particolarmente utili nelle fasi successive di un compilatore, poiché sono più facili da manipolare rispetto alla struttura ad albero puntatore tipica.

Inoltre, le **S-espressioni** utilizzate in linguaggi come **Scheme** e **Lisp** sono essenzialmente una rappresentazione degli AST. Le S-espressioni rappresentano i dati e le operazioni in una forma gerarchica e simbolica, che rispecchia molto da vicino il concetto di AST, rendendole una rappresentazione naturale per linguaggi basati su espressioni.

Grafo aciclico diretto

Un **grafo aciclico diretto** (DAG) è una forma avanzata di albero di sintassi astratta (AST) in cui ogni valore o sottoespressione ha un unico nodo. Questa caratteristica rende esplicita la condivisione delle sottoespressioni e codifica eventuali ridondanze presenti nel programma.

Se lo stesso valore o sottoespressione appare in più punti all'interno del programma, il DAG consente al compilatore di rappresentare tali ripetizioni con un solo nodo. In questo modo, il compilatore può ottimizzare il codice evitando di ricalcolare più volte la stessa espressione. Se ci sono due copie della stessa espressione, il DAG permette al compilatore di riorganizzare il codice in modo che venga valutata una sola volta, migliorando così l'efficienza.

In sintesi, l'uso di un DAG aiuta a eliminare ridondanze e a rendere più chiara la condivisione di sottoespressioni, permettendo ottimizzazioni che riducono il numero di operazioni ripetitive nel codice finale.

Stack Machine Code

Originariamente utilizzata per i computer basati su stack, questa rappresentazione è ora impiegata in linguaggi come **Java**.

Esempio. Un esempio tipico potrebbe essere l'espressione $x - 2 * y$, che viene trasformata in una sequenza di istruzioni lineare facilmente eseguibile da una macchina virtuale basata su stack, come

```
push x
push 2
push y
multiply
subtract
```

Questa forma ha diversi **vantaggi**. In primo luogo, è compatta: le variabili introdotte non richiedono nomi espliciti, ma sono implicite, il che significa che non occupano spazio nella rappresentazione. Al contrario, i nomi espliciti avrebbero bisogno di memoria, ma i nomi impliciti vengono gestiti direttamente nel contesto dell'esecuzione.

Inoltre, questa rappresentazione è **semplice da generare ed eseguire**. I compilatori che producono codice per macchine basate su stack, come la JVM di Java, possono generare questa forma con facilità, e la macchina esegue il codice usando operazioni push e pop su uno stack.

Questa rappresentazione è particolarmente utile quando il codice deve essere trasmesso attraverso **connessioni di comunicazione lente**, come nel caso di reti. La compattezza della rappresentazione riduce il tempo di trasmissione, rendendola efficiente per l'invio di codice su Internet o attraverso altre reti a bassa velocità.

Three Address Code

Il **three-address code** (TAC) è una forma intermedia comunemente utilizzata nei compilatori, in cui ogni istruzione ha la forma generale:

$$x \leftarrow y \operatorname{op} z$$

Qui, x, y, e z rappresentano nomi di variabili o valori temporanei, mentre "op" è un operatore che può essere aritmetico, logico o di altro tipo. Questa struttura permette di rappresentare operazioni che coinvolgono fino a tre operandi, rendendo più chiara e lineare l'esecuzione delle istruzioni.

Esempio. L'espressione $z \leftarrow x - 2 * y$ potrebbe essere suddivisa in più istruzioni di three-address code come segue:

$t_1 \leftarrow 2 * y$
$z \leftarrow x - t_1$

I **vantaggi** del three-address code includono il fatto che **somiglia molto al funzionamento di molte macchine reali**, in cui le operazioni binarie coinvolgono al massimo tre registri o locazioni di memoria (due operandi e una destinazione). Inoltre, il TAC introduce una nuova serie di nomi temporanei (come t_1) che facilitano la gestione delle sottoespressioni e la memorizzazione temporanea dei risultati intermedi.

Questa forma è anche **compatta** e facilmente manipolabile, poiché permette di scomporre espressioni complesse in passi semplici e sequenziali, agevolando così il processo di ottimizzazione e generazione di codice.

Three Address Code: quadruple

Il **Three Address Code (TAC)** può essere rappresentato tramite le **quadruple**, una rappresentazione semplice e diretta che utilizza una tabella in cui ogni istruzione è memorizzata come un record di quattro campi. Questa rappresentazione ingenua del TAC organizza ogni operazione in una struttura composta da quattro elementi, corrispondenti a:

1. L'operatore (op)
2. Il primo operando (arg1)
3. Il secondo operando (arg2)
4. La destinazione (result)

Esempio. Per l'istruzione $z \leftarrow x - 2 * y$, le quadruple potrebbero apparire come:

Operatore	Arg1	Arg2	Risultato
*	2	y	t1
-	x	t1	z

Questa rappresentazione in quadruple è particolarmente utile per le sue **caratteristiche**:

1. **Tabella di $k \times 4$ piccoli interi**: Ogni istruzione occupa una riga della tabella, rendendo facile la gestione e il tracciamento delle operazioni
2. **Struttura semplice dei record**: L'uso di una struttura di record fissa, con quattro campi per ogni istruzione, rende facile sia la generazione del codice sia il processo di ottimizzazione
3. **Facilità di riordino**: Poiché le istruzioni sono organizzate in forma tabellare, è relativamente semplice riordinarle o modificarle durante il processo di ottimizzazione senza dover gestire complessi puntatori o collegamenti
4. **Nomi espliciti**: Le quadruple usano nomi espliciti per le variabili e i risultati intermedi, il che aiuta a rendere la rappresentazione più chiara e a evitare ambiguità durante l'ottimizzazione e la generazione di codice

Three Address Code: triple

Il Three Address Code può essere rappresentato anche tramite le **triple**, una variante delle quadruple che utilizza un **indice implicito** invece di nomi espliciti per i risultati intermedi. In questa rappresentazione, ogni operazione è memorizzata in una tabella con tre campi: l'operatore e i due operandi. Il risultato della valutazione di ogni operazione non ha un nome esplicito; invece, il suo **indice nella tabella** diventa il riferimento implicito per le operazioni successive.

Esempio. L'espressione $z \leftarrow x - 2 * y$ potrebbe essere rappresentata come:

Indice	Operatore	Arg1	Arg2
0	*	2	y
1	-	x	(0)

Qui, (0) si riferisce al risultato dell'operazione indicizzata con 0, ovvero $2 * y$.

I vantaggi dell'utilizzo delle triple sono i seguenti:

1. **Meno spazio**: Poiché non è necessario memorizzare nomi espliciti per i risultati intermedi, le triple utilizzano circa **il 25% di spazio in meno** rispetto alle quadruple. Questo risparmio di spazio è particolarmente significativo in ambienti con risorse limitate, come i primi computer, quando 640KB di RAM era considerato un quantitativo abbondante.
2. **Efficienza della memoria**: La rappresentazione implicita dei risultati riduce il numero di nomi o variabili temporanee, risparmiando memoria.

Viceversa, gli svantaggi sono:

1. **Difficoltà di riordino**: A differenza delle quadruple, dove i nomi espliciti permettono un riordino più semplice delle istruzioni, nelle triple il riordino delle operazioni è molto più difficile. Questo perché i riferimenti sono fatti tramite indici, e modificare l'ordine delle istruzioni richiede l'aggiornamento manuale dei riferimenti in tutta la tabella.

Three Address Code: triple indirette

Le **triple indirette** sono un'altra variante del **Three Address Code** che bilanciano l'efficienza spaziale delle triple con la facilità di manipolazione delle quadruple. Nelle triple indirette, ogni istruzione non memorizza direttamente i dettagli di operatore e operandi, ma fa riferimento a una lista separata di triple. Questo approccio mantiene un **namespace implicito** e permette di elencare la prima triple di ogni istruzione, creando una struttura più flessibile per la manipolazione e il riordino delle operazioni.

Esempio. Per l'espressione $z \leftarrow x - 2 * y$, il processo potrebbe funzionare così:

Indice	Tripla indiretta
0	Riferimento a tripla 1
1	Riferimento a tripla 2

E la lista delle triple potrebbe apparire così:

Operatore	Arg1	Arg2
*	2	y
-	x	(0)

I vantaggi dell'utilizzo delle triple indirette sono i seguenti:

1. **Facilità di riordino**: A differenza delle triple standard, le triple indirette sono più facili da riordinare perché ogni istruzione fa riferimento a una lista separata di triple, permettendo un aggiornamento meno complesso dei riferimenti.
2. **Namespace implicito**: Non sono necessari nomi espliciti per i risultati intermedi, simile alle triple, ma senza il problema di dover gestire indici manualmente.

Viceversa, gli svantaggi sono:

1. **Maggiore uso di spazio** rispetto alle triple standard, perché è necessario memorizzare sia le istruzioni principali sia i riferimenti alle triple. Tuttavia, l'aumento dello spazio è bilanciato dalla facilità di riordino e gestione delle istruzioni.

In particolare, il **trade-off principale** tra le **quadruple** e le **triple** sta nel compromesso tra la **compattezza** delle triple e la **facilità di manipolazione** delle quadruple. In passato, quando lo spazio di memoria disponibile era una risorsa critica, la compattezza era una priorità assoluta, e le triple erano preferite. Oggi, con risorse di memoria molto più abbondanti, la **velocità** e la **facilità di manipolazione** sono diventate fattori chiave, rendendo

rappresentazioni come le quadruple e le triple indirette più comuni, poiché permettono un'ottimizzazione del codice più semplice e veloce.

Two address code

Il **two address code** permette istruzioni della forma $x \leftarrow x \operatorname{op} y$, dove c'è un operatore (op) e, al massimo, due nomi x e y.

Esempio. L'istruzione $z \leftarrow x - 2 * y$ diventa

```
t₁ ← 2
t₂ ← load y
t₂ ← t₂ * t₁
z  ← load x
z  ← z - t₂
```

1

Questo tipo di codice può essere molto efficiente, ma presenta alcuni problemi: le macchine non si basano più su operazioni distruttive, lo spazio dei nomi diventa complesso e le operazioni distruttive rendono difficile il riutilizzo delle variabili.

Control-flow graph

Il **Control-flow Graph** (grafico del flusso di controllo) modella il trasferimento del controllo all'interno di una procedura. I nodi del grafo rappresentano i blocchi basilari, ossia sequenze di istruzioni che vengono eseguite in modo lineare senza interruzioni. Questo grafo può essere rappresentato utilizzando quad (quaterne) o qualsiasi altra rappresentazione lineare. Gli archi del grafo, invece, rappresentano il flusso di controllo tra i vari blocchi, indicando come il controllo del programma si sposta da un blocco all'altro.

Static single assignment form

La **Static Single Assignment Form** (forma a singola assegnazione statica) si basa sull'idea che ogni variabile venga definita esattamente una volta nel codice. Per far funzionare questo approccio, vengono introdotte le funzioni ϕ

Le **funzioni** ϕ servono per risolvere ambiguità nelle variabili quando esistono percorsi multipli di esecuzione in un programma. In pratica, quando una variabile può assumere valori da percorsi di controllo diversi, la funzione ϕ viene utilizzata per unire questi valori in un unico punto del programma

Nota. Se abbiamo due percorsi distinti in un programma che assegnano valori diversi alla stessa variabile, la funzione ϕ viene utilizzata nel punto in cui i percorsi si congiungono per selezionare quale valore utilizzare

Formalmente, la funzione ϕ può essere vista come una funzione che "sceglie" il valore appropriato della variabile a seconda del percorso che ha portato al punto di convergenza.

Esempio. Immaginiamo il seguente flusso di controllo

$\text{if } (\text{condizione}) \; x_1 \leftarrow 10$
$\text{else } x_2 \leftarrow 20$

Quando i due rami si congiungono, possiamo usare una funzione ϕ per scegliere il valore corretto di (x) in base al percorso eseguito:

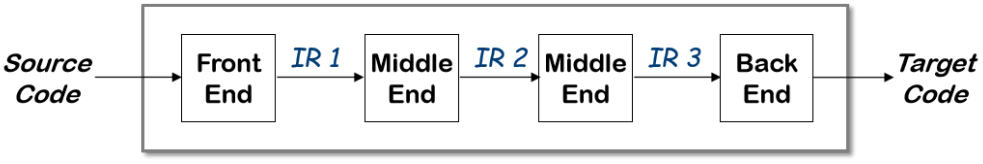
$x_3 \leftarrow \phi(x_1, x_2)$

In questo modo, (x_3) assumerà il valore di (x_1) se la condizione era vera, oppure il valore di (x_2) se la condizione era falsa.

Le funzioni ϕ permettono quindi di mantenere la proprietà della singola assegnazione per ogni variabile, facilitando l'analisi e l'ottimizzazione del codice.

I punti di forza della SSA-form risiedono nella possibilità di fare un'analisi più precisa del programma. Le funzioni ϕ forniscono indicazioni importanti sulla posizione delle variabili nel codice, facilitando l'ottimizzazione. Inoltre, in alcuni casi, gli algoritmi risultano più veloci grazie a questa rappresentazione.

Utilizzo di rappresentazioni multiple



L'uso di **rappresentazioni multiple** consiste nell'abbassare progressivamente il livello di astrazione della rappresentazione intermedia del programma durante il processo di compilazione. Ogni rappresentazione intermedia (IR) è ottimizzata per un determinato tipo di analisi o ottimizzazione, permettendo di applicare trasformazioni specifiche con maggiore efficienza.

Esempio. Un esempio pratico è il compilatore **Open64**, che utilizza un formato intermedio chiamato **WHIRL**. Questo formato intermedio è suddiviso in cinque livelli distinti, ciascuno progressivamente più dettagliato e meno astratto rispetto al precedente. I vari livelli di **WHIRL** permettono al compilatore di applicare diverse ottimizzazioni e analisi in base al grado di dettaglio richiesto:

1. Nei livelli più alti, le rappresentazioni sono più astratte, adatte per ottimizzazioni che coinvolgono strutture di alto livello, come loop o chiamate di funzioni.
2. Nei livelli più bassi, la rappresentazione è più vicina al codice macchina, ideale per ottimizzazioni più precise e legate all'architettura specifica della macchina target.

Questo approccio modulare permette una maggiore flessibilità e precisione nel processo di ottimizzazione del codice, sfruttando le caratteristiche di ciascun livello di rappresentazione intermedia per migliorare le prestazioni del codice finale.

Modelli di memoria

I **modelli di memoria** principali utilizzati durante la compilazione e l'esecuzione di un programma sono due: il **modello registro-a-registro** e il **modello memoria-a-memoria**.

Nel **modello registro-a-registro**, l'idea di base è quella di mantenere tutti i valori che possono essere legalmente conservati in un registro, all'interno dei registri stessi. Questo modello ignora i limiti fisici del numero di registri disponibili sul processore. Di conseguenza, spetta al back-end del compilatore inserire esplicitamente le istruzioni di caricamento (load) e memorizzazione (store) per trasferire i dati tra memoria e registri quando necessario. Questo modello è tipicamente usato nei compilatori per macchine **RISC** (Reduced Instruction Set Computer), dove si cerca di riflettere il modello di programmazione della macchina, rendendo più semplice determinare quando e come vengono utilizzati i registri.

Nel **modello memoria-a-memoria**, invece, tutti i valori sono mantenuti in memoria, e vengono promossi nei registri solo immediatamente prima del loro utilizzo. Questo approccio richiede al back-end del compilatore di ottimizzare e rimuovere caricamenti e

memorizzazioni non necessari per migliorare l'efficienza. In questo modello, l'uso della memoria è più predominante e diretto rispetto al modello registro-a-registro.

In sintesi, il **modello registro-a-registro** è preferibile per architetture **RISC**, in quanto semplifica la gestione dei registri, mentre il **modello memoria-a-memoria** tende ad essere più comune su macchine con un'architettura diversa o quando la gestione della memoria è meno vincolata al numero di registri disponibili.

Rappresentare il codice è solo una parte della **rappresentazione intermedia** (IR) in un compilatore. Per gestire correttamente l'analisi e l'ottimizzazione del programma, sono necessari altri componenti fondamentali che forniscono informazioni aggiuntive e dettagliate sul programma. Tra questi componenti ci sono:

- **Symbol table** (tabella dei simboli): contiene informazioni sui nomi delle variabili, delle funzioni, e degli altri identificatori usati nel programma. La tabella include dati come la rappresentazione del simbolo, il suo tipo, la sua classe di memorizzazione (ad esempio variabili locali o globali), e il suo offset in memoria o nei registri.
- **Constant table** (tabella delle costanti): memorizza le costanti utilizzate nel programma, insieme alla loro rappresentazione e tipo. Questa tabella permette al compilatore di gestire e ottimizzare l'uso delle costanti durante la generazione del codice.
- **Storage class e offset**: queste informazioni descrivono la posizione e la modalità di accesso ai vari simboli. La **classe di memorizzazione** specifica se un simbolo è, ad esempio, una variabile globale, locale o statica. L'**offset** indica la posizione del simbolo all'interno della memoria o nei registri.
- **Storage map** (mappa della memoria): fornisce una panoramica del layout generale della memoria, inclusi dettagli sull'allocazione della memoria per le variabili e i dati.
- **Overlap information** (informazioni su sovrapposizioni): tiene traccia di possibili sovrapposizioni di memoria, come quando due variabili condividono la stessa posizione di memoria (ad esempio, attraverso l'uso di unione o aliasing).
- **Virtual register assignments** (assegnazioni dei registri virtuali): in fase di compilazione, il compilatore può utilizzare registri virtuali prima di effettuare l'assegnazione finale ai registri fisici della macchina. Questo aiuta il compilatore a gestire in modo efficiente l'uso dei registri e a massimizzare le ottimizzazioni prima di considerare le limitazioni fisiche del numero di registri.

Questi componenti aggiuntivi sono essenziali per garantire che il compilatore possa analizzare e trasformare il programma in modo efficiente, ottimizzando l'uso della memoria e dei registri, e generando codice che rispetti le regole dell'architettura target.

Tabelle dei simboli

L'approccio classico alla costruzione di una **tabella dei simboli** utilizza l'**hashing** per gestire l'accesso efficiente alle informazioni sui simboli nel programma. Tuttavia, un approccio che può risultare più efficiente è un sistema a **due tabelle**.

Questo metodo prevede l'uso di:

1. **Indice sparso**: una tabella più piccola progettata per ridurre la probabilità di collisioni durante l'hashing. L'indice sparso contiene riferimenti a una seconda tabella, utilizzata per conservare i dati effettivi sui simboli.
2. **Tabella densa**: una struttura che contiene i dati veri e propri dei simboli (nomi, tipi, classi di memorizzazione, offset, ecc.). Questa tabella è più densa e compatta, facilitando l'espansione, la traversata (ovvero la scansione) e la lettura/scrittura da e verso file.

In questo sistema, quando si verifica una collisione, ossia quando due simboli hashano alla stessa posizione nell'indice sparso, si usano delle **catene** nell'indice per gestire tali collisioni. Ciò significa che gli elementi che condividono la stessa posizione nell'indice sono collegati tra loro in una lista, consentendo di risolvere la collisione tramite scansione lineare della lista.

I vantaggi principali di questo schema a due tabelle includono:

- **Facilità di espansione**: è più semplice ridimensionare la tabella densa quando vengono aggiunti nuovi simboli, senza dover ridimensionare contemporaneamente l'indice sparso.
- **Efficienza nel traversare i dati**: la struttura densa rende la scansione dei simboli più efficiente, poiché le informazioni sono conservate in una forma più compatta e organizzata.
- **Facilità di lettura e scrittura**: il formato della tabella densa si presta meglio alla scrittura su file o alla lettura da file per scopi come il salvataggio dello stato della compilazione o l'esportazione dei dati.

In sintesi, questo approccio offre una soluzione flessibile ed efficiente per la gestione delle collisioni e la manipolazione dei dati nella tabella dei simboli.

Tabelle dei simboli senza hashing

Le **tabelle dei simboli senza hashing** rappresentano un'alternativa all'approccio classico che utilizza una funzione di hash per memorizzare e cercare i simboli. Sebbene l'hashing sia molto efficiente nella media, esistono preoccupazioni legate al **comportamento nel peggior caso**, in particolare quando si verificano collisioni nella funzione di hash, che possono portare a ricerche lineari all'interno delle catene.

Per mitigare questi problemi, alcuni autori suggeriscono l'uso di un **"hash perfetto"** per la ricerca di parole chiave. Un hash perfetto è una funzione che non genera collisioni per un insieme di input specifico (ad esempio, un insieme predefinito di parole chiave in un linguaggio di programmazione), garantendo che la ricerca avvenga sempre in tempo costante.

Tuttavia, un'alternativa interessante è quella di evitare completamente le problematiche associate all'hashing, applicando tecniche derivanti dalla **teoria degli automi**. Gli automi, in particolare gli **automi deterministici a stati finiti (DFA)**, possono essere utilizzati per riconoscere parole chiave o identificatori senza rischiare di incorrere in comportamenti lineari nel peggior caso. In questo contesto, ogni simbolo è associato a uno stato specifico nell'automa, e il riconoscimento del simbolo avviene semplicemente percorrendo una sequenza di transizioni attraverso gli stati.

Un metodo alternativo all'uso delle tabelle hash per gestire le tabelle dei simboli è il sistema di **discriminazione di multinsiemi** proposto da **Paige & Cai**. Questo metodo organizza lo spazio dei nomi **in anticipo**, prima che inizi la ricerca durante la compilazione, assegnando un **indice** a ciascun nome. Così, quando il compilatore processa l'input, i nomi vengono sostituiti dagli indici corrispondenti.

Per migliorare ulteriormente il processo, possiamo usare **automi a stati finiti deterministici** (**DFA**) al posto delle funzioni hash. Questi automi garantiscono un **tempo di ricerca lineare**, evitando problemi come le collisioni. Un DFA costruito da una lista di parole (ovvero i simboli da cercare) è **aciclico**, cioè non presenta loop, quindi ha un comportamento prevedibile.

L'espressione regolare per una serie di simboli potrebbe essere qualcosa come $r_1 \mid r_2 \mid r_3 \mid \ldots \mid r_k$, dove $r_1, r_2, \ldots, r_k$ sono i simboli. Con questa configurazione, il DFA può essere costruito in due fasi: la prima per crearlo e la seconda per usarlo. Tuttavia, con alcune tecniche ottimizzate, si potrebbe **evitare il doppio passaggio**, rendendo il tutto ancora più efficiente.

In sintesi, l'uso di un DFA per gestire le tabelle dei simboli elimina i problemi tipici dell'hashing, garantendo una ricerca in **tempo lineare** e senza rischiare collisioni o altri inconvenienti.

La **costruzione incrementale di un DFA aciclico** permette di aggiungere nuove parole man mano che il programma si evolve. Se si vuole aggiungere una parola che il DFA attuale non riconosce, vengono creati nuovi stati e transizioni per includere questa parola.

Ogni carattere della parola richiede un accesso alla memoria, che diventa un problema solo se il DFA diventa molto grande. Tuttavia, per insiemi di parole relativamente piccoli, come i

nomi all'interno di una procedura, questo non rappresenta un grosso limite.

Ci sono diverse **ottimizzazioni** per ridurre i costi di memoria:

- Rendere lo **stato finale** esplicito su ogni percorso può ridurre significativamente la memoria necessaria, creando lo stato finale solo quando serve estendere un percorso.
- Esiste un **compromesso** tra la complessità della rappresentazione degli stati e il costo in termini di memoria: più dettagliata è la rappresentazione, maggiore sarà il costo, ma si ridurrà la complessità delle transizioni.
- La **capitalizzazione** delle lettere può essere codificata separatamente usando stringhe di bit, riducendo lo spazio necessario per distinguere tra maiuscole e minuscole.

In conclusione, con un approccio incrementale e queste ottimizzazioni, si può costruire un DFA aciclico efficiente, capace di riconoscere nuove parole senza aumentare significativamente i costi di memoria, specialmente per insiemi di parole ridotti, come i nomi in una procedura.