Corso di Laurea Magistrale in Ingegneria Informatica
A.A. 2013-2014

Linguaggi Formali e Compilatori

Compilatori

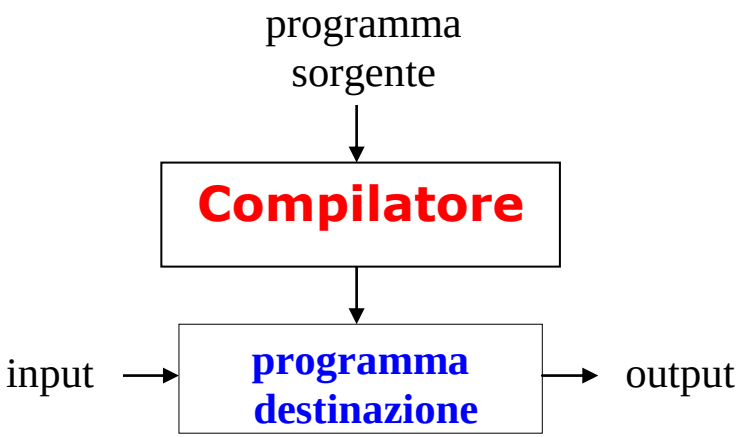
Giacomo PISCITELLI

I traduttori

Genericamente, i programmi responsabili della traduzione da un linguaggio ad alto livello a programmi scritti in linguaggio di macchina sono detti **traduttori**.

Essi normalmente si distinguono in:

- **Compilatori**;
- **Interpreti**;
- **Traduttori ibridi**.



I compilatori

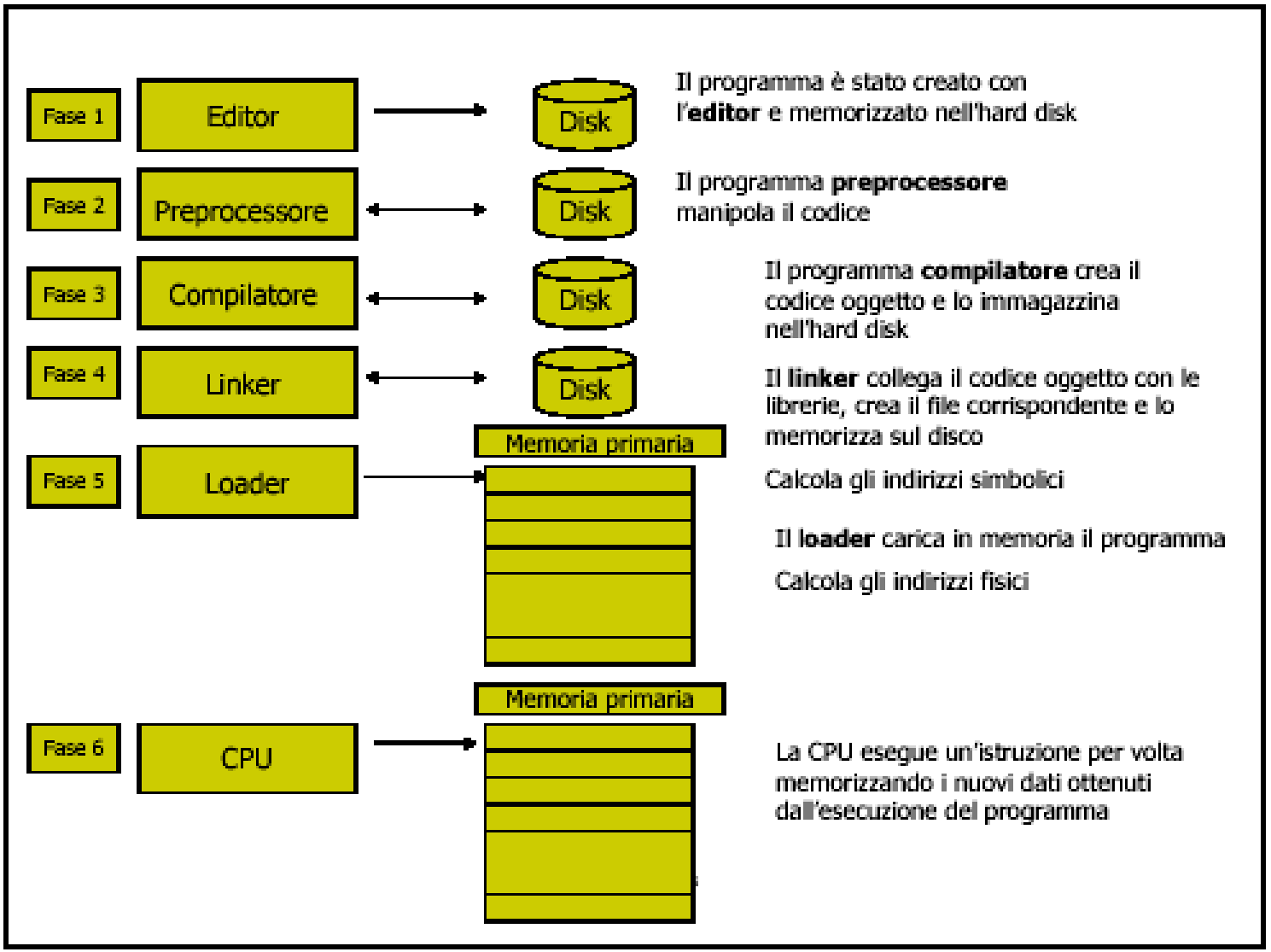
I **Compilatori** traducono i programmi, codificati in un linguaggio di programmazione (*modulo sorgente*), in codice macchina specifico per una determinata architettura hardware (*modulo oggetto o destinazione*). Potendo il modulo sorgente essere suddiviso in diverse parti, potrebbe essere necessario disporre di un preprocessore, capace anche di espandere opportune abbreviazioni, dette *macro*, in istruzioni del linguaggio sorgente.

Il *modulo oggetto* deve essere successivamente fuso con altri moduli oggetto o funzioni di libreria tramite un linkage editor al fine di produrre il *modulo eseguibile*, che, una volta caricato in memoria da un loader, può essere fruito direttamente da un processore.

Non è però scontato che il linguaggio di destinazione sia il linguaggio macchina della macchina ospite, in quanto esistono anche i cross-compilatori, il cui scopo è generare codice eseguibile su altre macchine, le cui risorse (processore, memoria, dischi) potrebbero essere troppo limitate per ospitare l'ambiente di sviluppo nel quale viene scritto il programma.

Quindi i programmi vengono eseguiti in un emulatore (in sostanza una macchina virtuale) e poi il modulo eseguibile creato viene portato sulla macchina di destinazione.

L'ambiente di programmazione di un linguaggio



Gli Interpreti

Gli **Interpreti** traducono di volta in volta la singola istruzione nella/e equivalente/i istruzione/i in linguaggio di macchina e poi eseguono tale/i istruzione/i. Devono perciò essere sempre attivi durante l'esecuzione del programma principale.

Anzi, per essere più precisi, si può affermare che l'unico programma in esecuzione è l'interprete, mentre il modulo sorgente contiene i "dati" per l'interprete.

Gli Interpreti

Ci sono due tipi diversi di interpreti che supportano l'esecuzione di programmi: interpreti di macchina e interpreti di linguaggio.

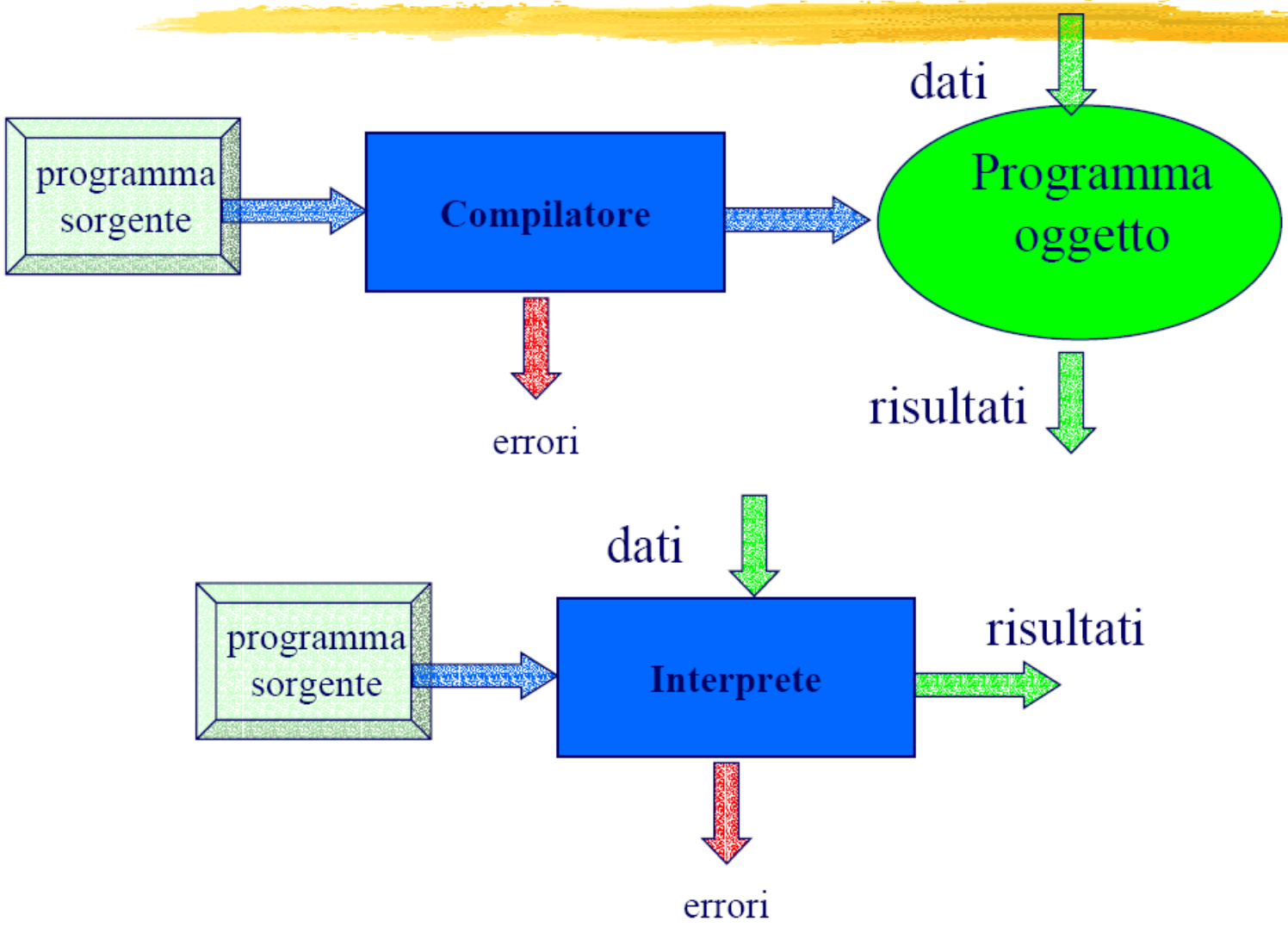
Machine Interpreters

Gli interpreti di macchina simulano l'esecuzione di un programma compilato per una particolare architettura: il programma "compilato" viene interpretato da un *bytecode interpreter* (scritto in linguaggio C o altro); in questo caso il programma "compilato" è il codice di macchina per la macchina virtuale, che non è implementata in hardware, ma nel *bytecode interpreter*. Java usa un interprete *bytecode per simulare l'effetto del programma compilato per la Java Virtual Machine* (JVM).

Language Interpreters

Gli interpreti di linguaggio simulano l'effetto dell'esecuzione di un programma (senza compilarlo) scritto per un particolare *instruction set* (reale o virtuale). Una forma di *Intermediate Representation* (IR, per esempio un *Abstract Syntax Tree*, AST) viene utilizzata per l'esecuzione.

Compilatore vs. interprete



Gli Interpreti

Vantaggi

È possibile eseguire modifiche ed aggiunte al codice durante l'esecuzione, consentendo un *debug* interattivo e una migliore diagnostica. A seconda della struttura del linguaggio, le modifiche del programma possono richiedere *reparsing* o ripetizione dell'analisi semantica.

Gli interpreti supportano l'indipendenza dalla macchina.

Svantaggi (*large overheads*)

Non è possibile effettuare ottimizzazione del codice.

Durante l'esecuzione il testo del programma è continuamente riesaminato: i *bindings*, i tipi e le operazioni sono ricomputate anche ad ogni uso.

Per linguaggi molto dinamici questo rappresenta un *overhead* di 100:1 (o peggiore) nella velocità di esecuzione rispetto a quella del codice compilato. Per linguaggi più statici (C o Java), il degrado della velocità è dell'ordine di 10:1.

Il tempo di startup per piccoli programmi è lungo, dato che deve essere caricato l'interprete e il programma deve essere parzialmente ricompilato prima dell'esecuzione.

Si può verificare un sostanziale overhead di spazio: infatti l'interprete e tutte le routine di supporto di solito devono essere tenuti a disposizione.

Il programma sorgente spesso non è compatto come se fosse compilato. Ciò può causare una limitazione nella dimensione dei programmi.

Naturalmente, molti linguaggi (tra cui, C, C + + e Java) hanno gli interpreti (per il debug e lo sviluppo del programma) e i compilatori (per la produzione dell'applicazione).

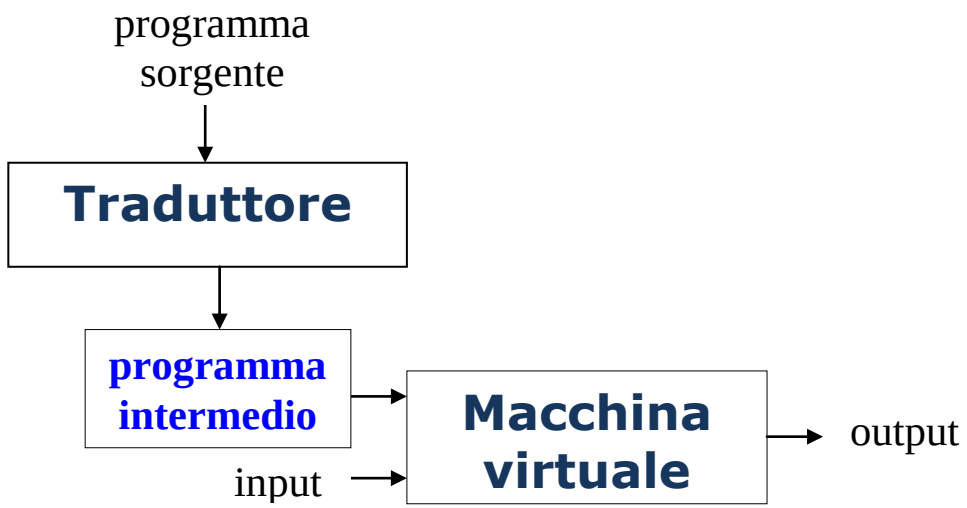
Alcuni linguaggi presentano traduzioni ibride.

I traduttori ibridi

Esistono anche **traduttori ibridi** come Java, che creano file con contenuto bytecode.

Tale contenuto non è fruibile direttamente dal processore, ma deve essere interpretato dalla Java Virtual Machine (JVM), che in sostanza è un ulteriore strato di software che si interpone tra la macchina reale ed il programma.

Questo rende quindi il programma indipendente dal processore; una applicazione pratica è quella dei programmi per telefoni cellulari, condivisibili anche su apparecchi di marche o serie diverse, purché dotate di una JVM compatibile (cioè di una versione contenente le caratteristiche necessarie al funzionamento del programma).



Compilatori vs Interpreti in sintesi

Interpretazione

- Un ***Interprete*** è un programma che legge il programma sorgente e, per ogni istruzione
  1. Verifica la correttezza sintattica
  2. Effettua la traduzione nella corrispondente sequenza di istruzioni in linguaggio macchina
  3. Esegue direttamente la sequenza di istruzioni in linguaggio macchina

- SVANTAGGIO: Istruzioni eseguite più volte (es. ciclo), vengono verificate e tradotte più volte
- VANTAGGIO: Facile sviluppo e correzione dei programmi

Compilatori vs Interpreti in sintesi

Compilazione

- Un ***Compilatore*** è un programma che legge il programma sorgente e lo **traduce interamente** in un programma scritto in linguaggio macchina (programma oggetto)
  - Verifica la correttezza sintattica di ciascuna istruzione
  - Il programma oggetto è generato solo se non ci sono errori sintattici
  - La correttezza semantica è effettuata solo in fase di esecuzione

Compilatori vs Interpreti (efficienza e produttività)

Interpretazione vs Compilazione

- Velocità di esecuzione:
  - Bassa per i linguaggi interpretati
  - Alta per i linguaggi compilati
- Facilità di messa a punto dei programmi:
  - Alta per i linguaggi interpretati
  - Bassa per i linguaggi compilati

Storia dei compilatori

Il termine compilatori fu coniato da Gracev Murray Hopper nel 1950.

La traduzione fu vista come una "compilazione" di una sequenza di sottoprogrammi in linguaggi macchina selezionati da una libreria.

Nel 1957 il team FORTRAN presso l'IBM, guidato da John Backus, fu accreditato come primo inventore di un compilatore completo. Ci erano voluti 18 anni di un nutrito staff di progettisti/realizzatori.

Il COBOL fu uno dei primi linguaggi nel 1960 ad essere compilato su più architetture.

Un compilatore è esso stesso un programma scritto in un qualche linguaggio. I primi compilatori venivano scritti in Assembler.

Il primo compilatore auto-compilato, capace cioè di compilare il suo stesso codice, fu creato per il linguaggio Lisp da Hart e Levin presso il MIT nel 1962.

L'uso di un linguaggio ad alto livello per scrivere i compilatori ebbe una spinta nei primi anni '70, quando i compilatori Pascal e C furono scritti negli stessi linguaggi.

Creare un compilatore autocompilante introduce un problema di *bootstrapping*: il primo compilatore di quel linguaggio deve essere per forza scritto in un altro linguaggio o compilato facendo operare il compilatore come un interprete (come fecero Hart e Levin con il loro compilatore Lisp).

Ambiziose "ottimizzazioni" furono utilizzate per produrre codice macchina efficiente, che fu vitale per i primi computer con capacità piuttosto limitate.

L'uso efficiente delle risorse è comunque ancora essenziale per i moderni compilatori.

Obiettivi dell'implementazione di un compilatore

Correttezza

I compilatori consentono ai programmatori di scoprire gli errori ortografici e logici.

Le tecniche di compilazione consentono di migliorare la sicurezza: per esempio Java Bytecode Verifier.

Protezione della proprietà intellettuale.

Efficienza

Ridurre il tempo di esecuzione, lo spazio per i dati e per il codice a *compile & run-time*.

Supportare l'espressività del linguaggio

Offrire un "ambiente di programmazione".

Realizzare un *fast turn-around time*.

Consentire compilazioni separate.

Permettere il *debugging* del codice sorgente.

"compilatore" solo come traduttore di LdP ?

I compilatori, al di là della funzione di traduttori di linguaggi di programmazione evoluti in linguaggio macchina, hanno anche altri utilizzi:

- TeX e LaTeX usano compilatori per tradurre testo e comandi di formattazione.
- Postscript (generato da LaTeX, Word, etc) può essere considerato un linguaggio di programmazione tradotto ed eseguito da una stampante o da un visualizzatore postscript per produrre una forma leggibile di un documento.
- Un linguaggio per la rappresentazione standardizzata dei documenti permette il loro interscambio, indipendentemente da come e dove essi sono stati creati e visualizzati.
- Mathematica è un sistema interattivo che miscela programmi con matematica. Il sistema utilizza tecniche di compilazione per gestire le specifiche del problema, la sua rappresentazione interna e la sua soluzione.
- Verilog e VHDL supportano la creazione di circuiti VLSI. Un *silicon compiler* specifica il *layout* e la composizione della maschere del circuito VLSI utilizzando celle standard: il silicon compiler, come un compilatore "tradizionale", comprende e applica le regole di progettazione che determinano la fattibilità di un circuito.
- Strumenti interattivi spesso hanno bisogno di un linguaggio di programmazione per sostenere l'analisi automatica e la modifica di un sistema.

Cos'è un compilatore

E' un programma che legge una frase di un linguaggio e la traduce in frase/i di un altro linguaggio

Ruolo dei Compilatori come traduttori dei LdP

Consentire l'uso di linguaggi di programmazione ad alto livello

- Incremento della produttività dei programmatori
- Facilità di *maintenance* del codice
- Maggiore portabilità del codice

Sfruttare le opportunità offerte dai dettagli architetturali di basso livello

- Scelta delle istruzioni
- Metodi di indirizzamento
- *Pipelines*
- Uso della *cache*
- Parallelismo a livello di istruzione

I compilatori sono necessari per coprire il gap tra i linguaggi high-level e low-level

Cambi nell'architettura → cambi nei compilatori.

Significative differenze nelle performance.

Lo sviluppo di compilatori

I compilatori sono sistemi software di grandi dimensioni e di enorme complessità.

Lo sviluppo di compilatori richiede, infatti, conoscenze su:

- Strumenti di programmazione (compilers, debuggers)
- Strumenti per la generazione di programma (lex-yacc, flex-bison)
- Librerie software (sets collections)
- Simulatori

La conoscenza della struttura di un compilatore migliora perciò le qualità del progettista del software

Cosa richiedere ad un compilatore?

→ Il codice prodotto sia corretto

→ Il codice prodotto sia efficiente (*output runs fast*)

→ Il compilatore sia efficiente (*Compiler runs fast*)

→ Il tempo di compilazione sia proporzionale alla dimensione del codice

→ Sia prevista la compilazione separata

→ La diagnosi degli errori sintattici sia puntuale

→ Che funzioni adeguatamente con il debugger

→ La diagnosi delle anomalie sia accurata

→ Siano possibili *Cross language calls*

→ L'ottimizzazione sia predicibile

Struttura di un compilatore

Ci sono due fasi fondamentali in un compilatore: **Analisi** e **Sintesi**

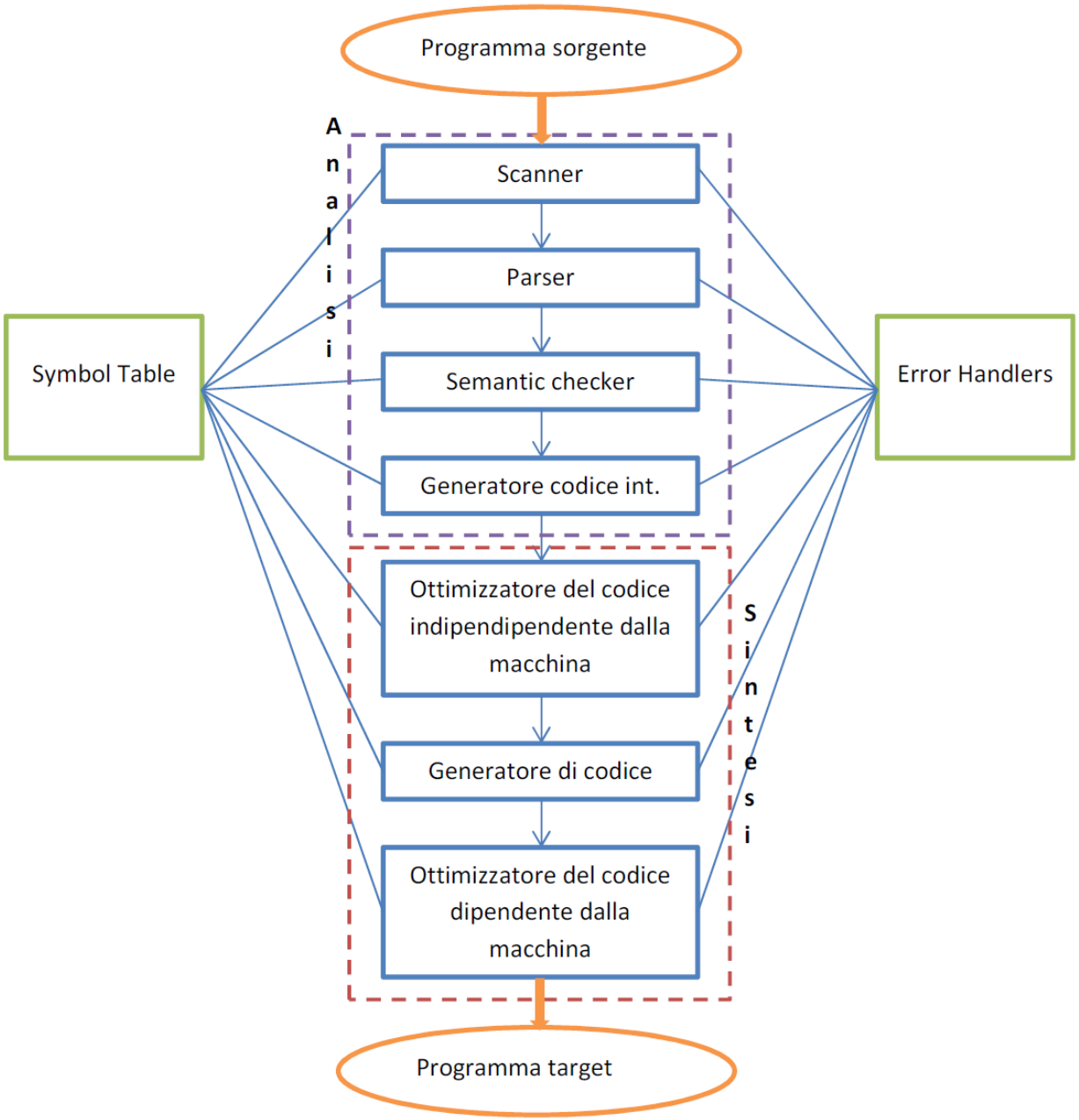
Nella fase di **analisi**, viene creata una rappresentazione intermedia del programma sorgente.

Sono parti di questa fase l'Analizzatore lessicale, l'Analizzatore Sintattico, l'Analizzatore Semantico e il Generatore di codice intermedio.

Nella fase di **sintesi**, viene creato, partendo dalla rappresentazione intermedia, il programma target equivalente.

Sono parti di questa fase il Generatore di Codice e l'Ottimizzatore.

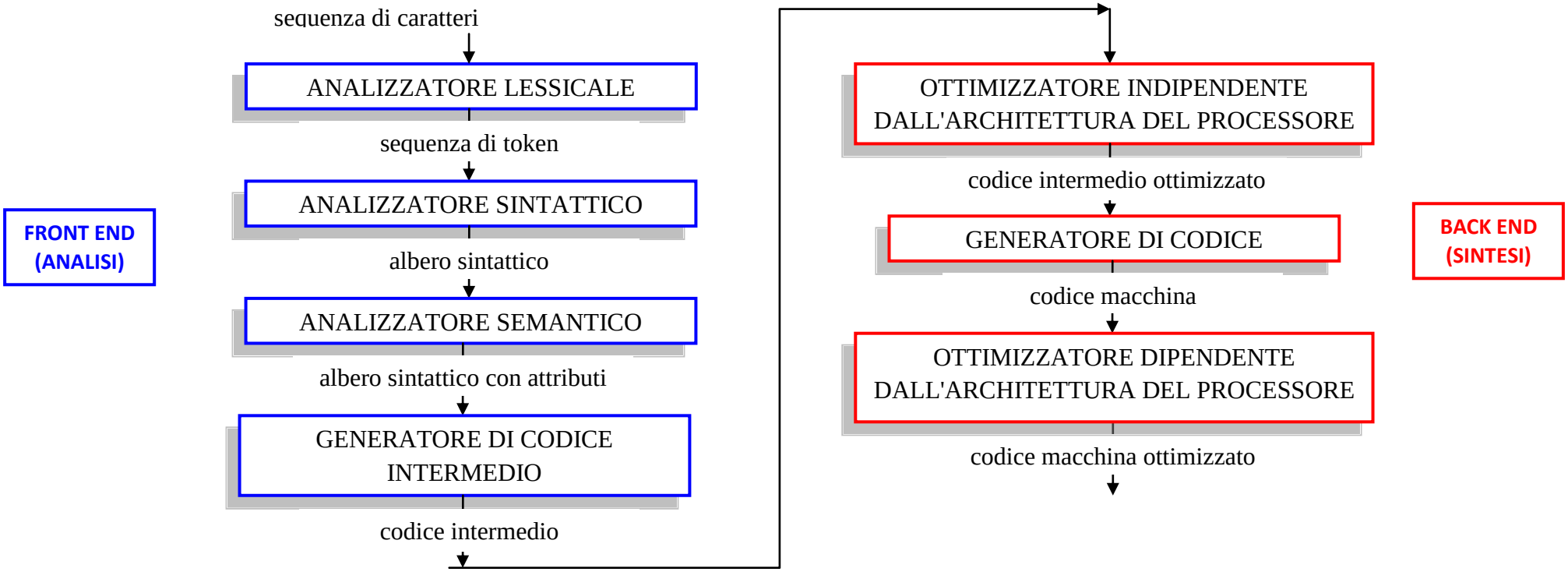
Fasi di un compilatore



Fasi di un compilatore

- Ogni fase trasforma il programma sorgente da una rappresentazione in un'altra
- Durante queste fasi vengono utilizzate: Error handlers e Symbol table.

La tabella dei simboli contiene informazioni su tutti gli elementi simbolici incontrati quali nome, *scope*, tipo (se presente) etc.



FASI DELLA COMPILAZIONE

Criteri di classificazione dei compilatori (1/2)

Numero di passi

- *compilatori a singolo passo - compilatori a più passi*
  con numero di passi si intende il numero di volte che viene letto il codice sorgente durante la compilazione

Ottimizzazione

- *nessuna ottimizzazione*
- *ottimizzazione in spazio*
- *ottimizzazione in tempo*
- *ottimizzazione in potenza dissipata*

Criteri di classificazione dei compilatori (2/2)

Linguaggio oggetto prodotto

- *pure machine code*
  I compilatori generano codice per un particolare set di istruzioni macchina non assumendo l'esistenza di alcun sistema operativo o libreria di funzioni. Questo approccio è raro ed è utilizzato per compilatori di linguaggi per l'implementazione di sistemi.
- *augmented machine code*
  I compilatori generano codice per un particolare set di istruzioni macchina arricchito con routine di sistema operativo o di supporto: per eseguire un tale codice oggetto sulla macchina devono esistere un sistema operativo e una collezione di routine di supporto (I/O, allocazione di memoria) che vanno fuse con il codice oggetto. Il grado di corrispondenza fra codice virtuale e hardware può variare moltissimo.
- *virtual machine code*
  I compilatori generano codice composto esclusivamente da codice virtuale. Questo approccio è attraente perché permette di generare codice eseguibile indipendente dall'hardware. Se la macchina virtuale viene mantenuta semplice, il suo interprete può essere facile da scrivere. Questo approccio penalizza la velocità di esecuzione di un fattore da 3:1 a 10:1. A "Just in Time" (JIT) può tradurre porzioni di virtual code in codice nativo per velocizzare l'esecuzione.

formato target prodotto

- *assembly language*,
- *relocatable binary*,
- *memory-image*.

Vantaggi del "virtual machine code"

L'uso di *virtual machine code* può servire per una varietà di scopi:

- ✓ semplificare un compilatore fornendo le primitive adatte (ad esempio come chiamate di metodo, *string manipulation*, e così via);
- ✓ trasportabilità del compilatore;
- ✓ si può diminuire la dimensione del codice generato dato che le istruzioni sono progettate per un particolare linguaggio di programmazione (per esempio codice JVM per Java).
- ✓ in quasi tutti i compilatori, in maggiore o minor misura, per generare codice per una macchina virtuale, è necessario interpretare alcune delle operazioni.

Formato target prodotto (1/2)

Assembly Language (Symbolic) Format

Viene prodotto un file di testo contenente il codice sorgente assembler: un certo numero di decisioni nella generazione del codice (target delle istruzioni di salto, forma degli indirizzi, e così via) sono lasciate all'assemblatore.

Questo è un approccio buono per i progetti didattici.

Supporta la generazione di codice assembler per "cross-compilation" (la compilazione viene effettuata su un computer diverso da quello dove dovrà essere eseguito).

La generazione di codice assembly semplifica il debug e la comprensione di un compilatore (in quanto si può vedere il codice generato).

Piuttosto che uno specifico linguaggio assembly, si utilizza C come linguaggio "assembly universale". Il C è "*machine independent* " molto più di qualsiasi particolare linguaggio assembly. Tuttavia, alcuni aspetti di un programma (come la rappresentazione a run-time dei dati) non sono accessibili utilizzando il codice C, ma facilmente accessibile in linguaggio assembly.

Formato target prodotto (2/2)

Relocatable Binary Format

Il codice può essere generato in un formato binario con riferimenti esterni e a istruzioni locali e con gli indirizzi dei dati non ancora "vincolati".

Gli indirizzi vengono assegnati relativamente all'inizio del modulo o rispetto ad una unità denominata simbolicamente.

Un passo di link (*linkage step*) consente di aggiungere le librerie di supporto e altre routine compilate separatamente e produce un formato binario assoluto eseguibile del programma.

Memory-Image (Absolute Binary) Format

Il codice compilato può essere caricato in memoria ed immediatamente eseguito.

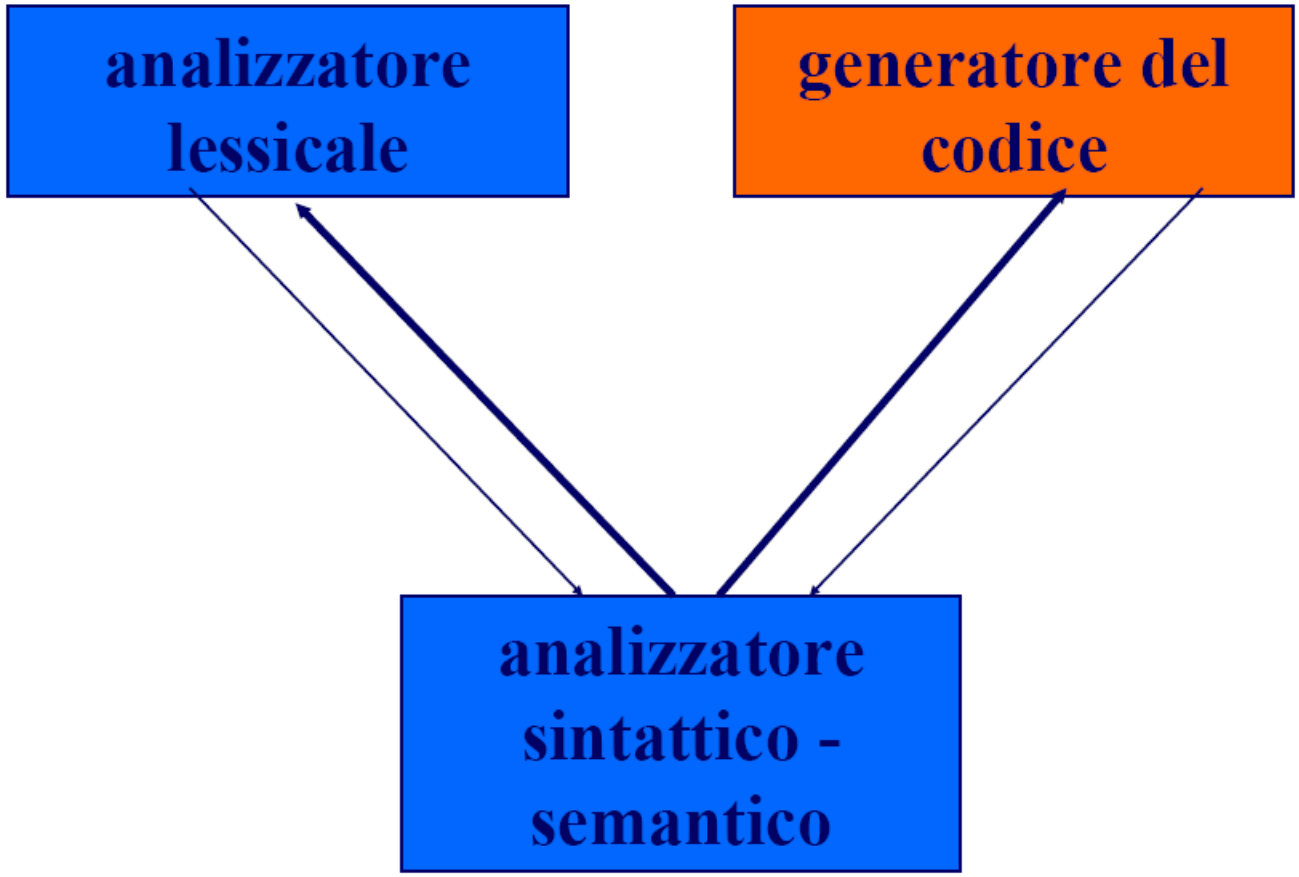
Questo è il metodo più veloce.

La possibilità di utilizzo di librerie può essere limitata.

Il programma deve essere ricompilato per ogni esecuzione.

I compilatori Memory-image sono utili per gli studenti (per i quali il debug e i frequenti cambiamenti sono la norma e i costi di compilazione superano di gran lunga i costi di esecuzione.

Compilatori a singolo passo

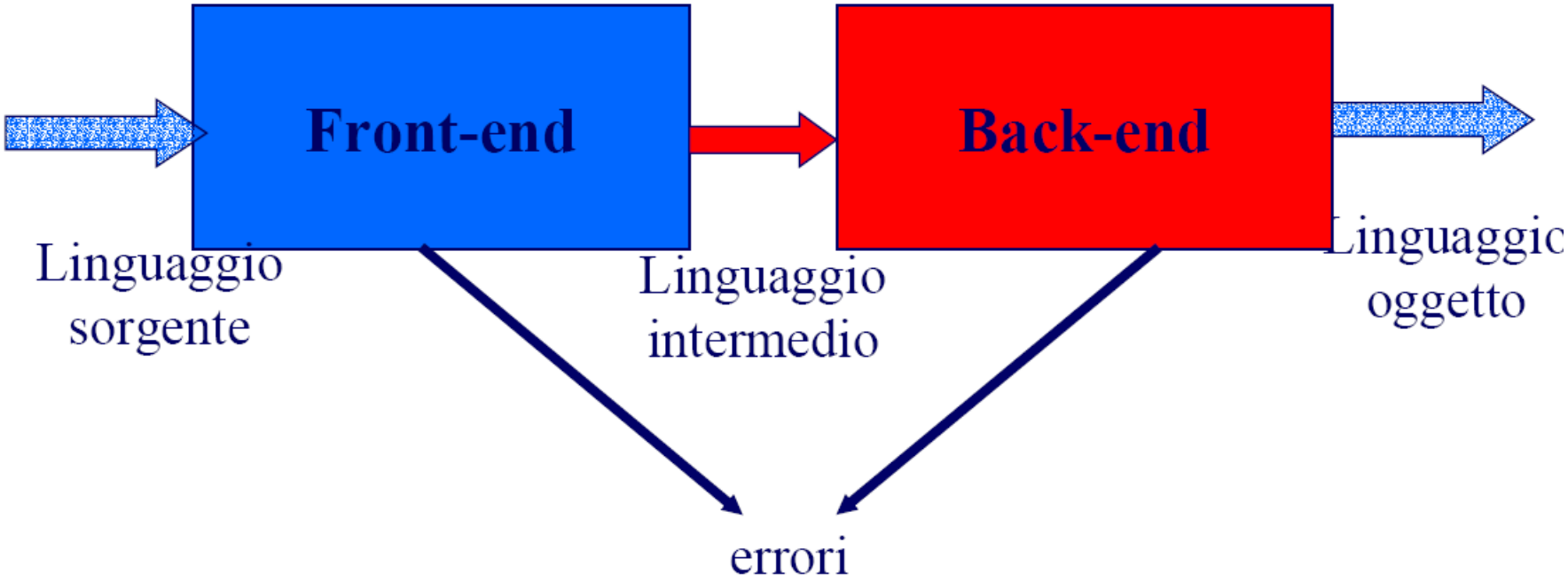


Compilatori a due passi

I compilatori attuali dividono l'operazione di compilazione in due stadi principali, ciascuno dei quali richiede la lettura del codice sorgente: il **front end** e il **back end**.

Nello stadio di front end il compilatore traduce il sorgente in un linguaggio intermedio (di solito interno al compilatore). Il Front end dipende perciò dal linguaggio sorgente, ma è indipendente dalla macchina target.

Nello stadio di back end alcune volte viene preliminarmente effettuata l'ottimizzazione del codice intermedio, quindi si procede con la generazione del codice oggetto e l'ottimizzazione. Il Back end è quindi indipendente dal linguaggio sorgente, ma dipende dalla macchina target.



Compilatori a due passi

I passi o "passate"

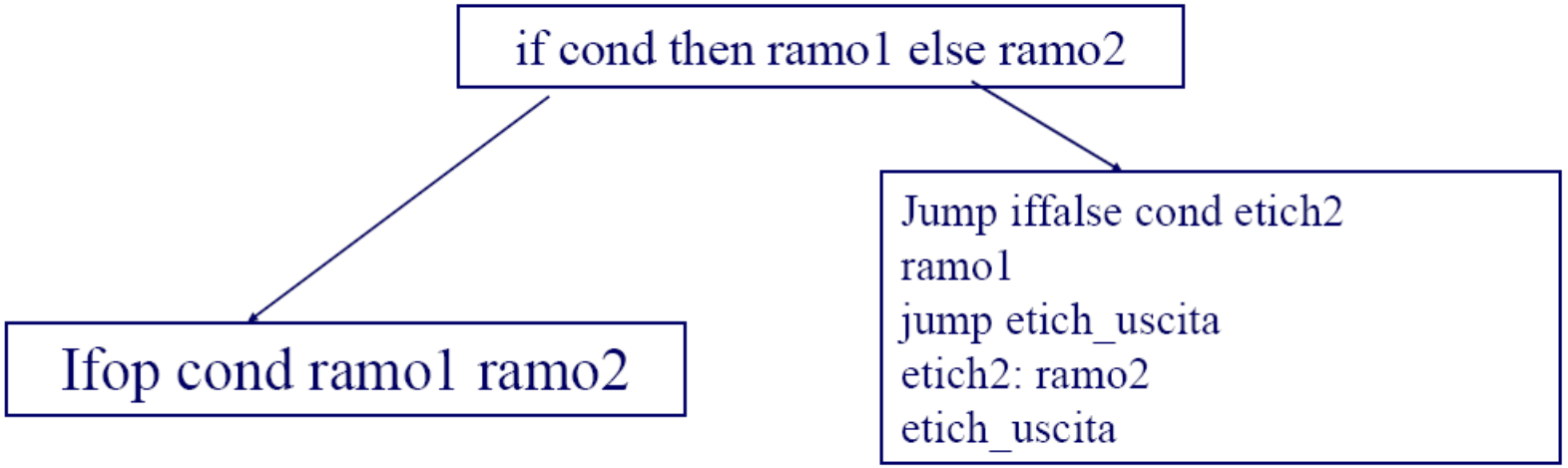
Una singola passata può bastare per più fasi (interfogliate durante la passata): ad esempio, analisi e generazione codice intermedio possono essere svolte in una sola passata "guidata" dall'analizzatore sintattico.

Ridurre il numero di passate

Diminuisce il tempo, ma aumenta la memoria richiesta.

Il Linguaggio intermedio può essere:

- **di alto livello**, il che implica che gli operatori del linguaggio sorgente siano ancora presenti nel linguaggio;
- **di basso livello**, il che comporta che gli operatori sorgenti siano tradotti in altri più semplici o specializzati.



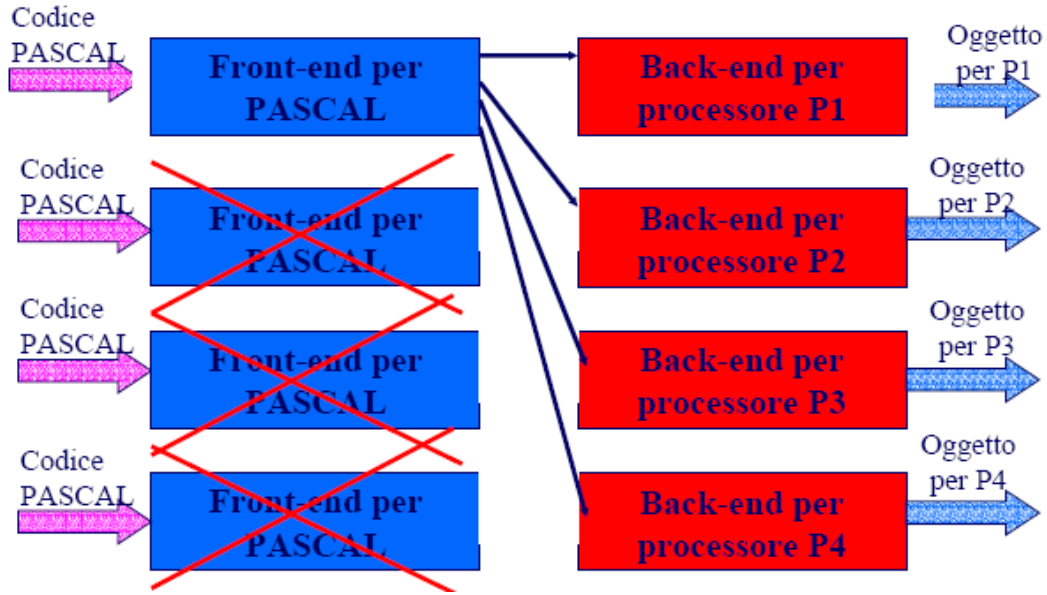
Compilatori a due passi

Implicazione della compilazione a due passi

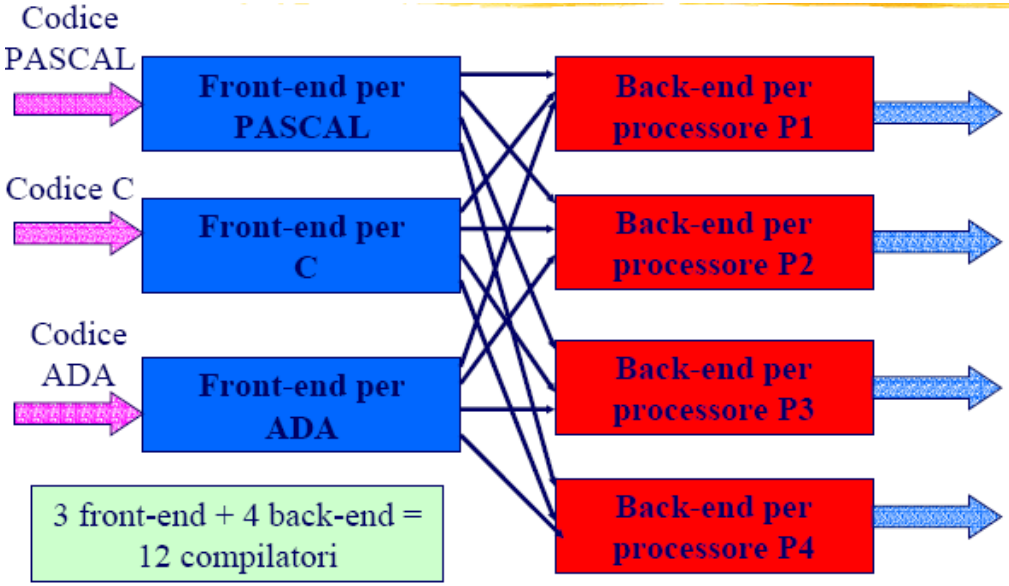
- ✓ rende più semplice la costruzione di un compilatore per un nuovo processore (retargeting)
- ✓ consente di progettare compilatori con multipli front-end

Front end e Back end si possono riutilizzare separatamente per nuovi compilatori

Compilatori a due passi



Retargeting



Multiple Front-end

L'analisi lessicale: Lexical Analyzer o scanner

- ✓ trasforma il codice sorgente in una forma compatta e uniforme (tokens=elementi lessicali)
- ✓ elimina informazioni non necessarie presenti nel codice sorgente (commenti)
- ✓ processa le direttive di compilazione (**`include`**, **`define`**, etc)
- ✓ scopre eventuali errori nel lessico
- ✓ consente di descrivere i token in modo efficace mediante la notazione delle espressioni regolari.

Un *token* descrive un insieme di caratteri che hanno lo stesso significato (come identificatori, operatori, *keywords*, numeri, delimitatori, etc).

I token sono gli elementi minimi (non ulteriormente divisibili) di un linguaggio, ad esempio parole chiave (**`for`**, **`while`**), nomi di variabili (**`pippo`**), operatori (**`+`**, **`-`**, **`<<`**).

L'analisi lessicale: Lexical Analyzer o scanner

Un **analizzatore lessicale** legge il programma sorgente carattere per carattere e ritorna *token* del programma sorgente.

`newval := oldval + 12` => token: `newval` *identifier*

`:=` *assignment operator*

`oldval` *identifier*

`+` *add operator*

`12` *a number*

Le informazioni sugli identificatori sono memorizzate nella tabella dei simboli (**symbol table**).

Le espressioni regolari sono utilizzate per descrivere i token.

Un Automa a Stati Finiti Deterministico può essere utilizzato per implementare un analizzatore lessicale.

L'analisi sintattica: Syntax Analyzer o parser

L'analisi sintattica è il procedimento di costruzione della derivazione di una frase rispetto ad una data grammatica. Un ***Analizzatore Sintattico*** è, perciò, un algoritmo che opera su una stringa e, se tale stringa appartiene al linguaggio associato alla data grammatica, ne produce una derivazione, altrimenti l'algoritmo si ferma indicando:

- il punto della stringa dove si è presentato l'errore;
- il tipo di errore (diagnosi).

L'analisi sintattica prende in ingresso la sequenza di token generata nella fase precedente di analisi lessicale ed esegue il controllo sintattico. Il controllo sintattico è effettuato attraverso una grammatica. Il risultato di questa fase è un **albero sintattico**.

L'Analisi sintattica, quindi:

- cerca errori sintattici
- raggruppa i token in frasi grammaticali.

Syntax Analyzer (CFG)

La sintassi di un linguaggio è specificata per mezzo di una grammatica non contestuale (**Context Free Grammar** CFG).

Le regole di una CFG sono ricorsive.

Spesso per specificare una CFG viene utilizzata una BNF (*Backus Naur Form*)

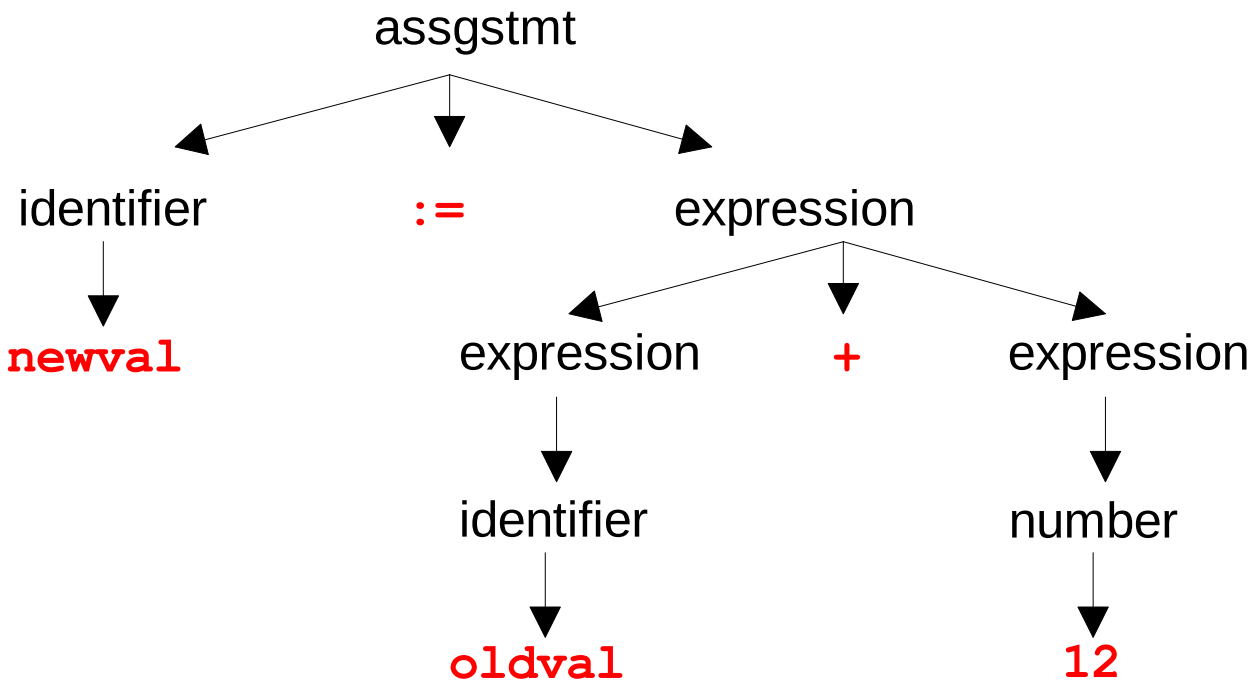
```
assgstmt     -> identifier := expression
expression   -> identifier
expression   -> number
expression   -> expression + expression
```

Un analizzatore sintattico verifica se ciascuna frase di un programma soddisfa le regole derivate dalla CFG.

Se le regole sono soddisfatte l'analizzatore sintattico crea un *albero sintattico* per ciascuna frase del programma.

L'analisi sintattica: Syntax Analyzer o parser

Un Analizzatore Sintattico crea la struttura sintattica generalmente sotto forma di un albero chiamato ***parse tree*** o albero sintattico di ciascuno specifico costrutto di un LdP. Di seguito viene riportato il parse tree di uno statement di assegnazione; un analogo albero viene prodotto dal parser per un'istruzione di ciclo, come si vedrà in seguito.



In un parse tree, tutti i terminali sono foglie. Tutti gli altri nodi sono non terminali.

Tecniche di Parsing

In relazione a come viene creato il *parse tree*, ci sono differenti tecniche, che possono essere divise in due gruppi:

Top-Down Parsing: *analisi sintattica discendente o "predittiva"*

- La costruzione del *parse tree* inizia dalla radice e procede verso le foglie.
- Efficienti parser top-down possono facilmente essere costruiti a mano.
- *Recursive Predictive Parsing*, *Non-Recursive Predictive Parsing* (LL Parsing: Left to right, Left most).

Bottom-Up Parsing: *analisi sintattica ascendente o "a spostamento e riduzione"*

- La costruzione del *parse tree* inizia dalle foglie e procede verso la radice.
- Di norma efficienti parser bottom-up sono creati con l'ausilio di tool software
- Il parser Bottom-up è anche conosciuto con il nome di *shift-reduce parsing*.
- *Operator-Precedence Parsing* – facili da implementare
- LR Parsing: Left to right, Right most.

Analisi semantica

L'analisi semantica si occupa di controllare il significato delle istruzioni presenti nel codice in ingresso. Controlli tipici di questa fase sono il *type checking*, ovvero controllare che gli identificatori siano stati dichiarati prima di essere usati e così via. Il risultato di questa fase è l'albero sintattico astratto (***AST - Abstract Syntax Tree***).

L'Analisi semantica, quindi:

- cerca errori semantici
- raccoglie informazioni sui tipi effettuando il type checking.

Esempio:

```
newval := oldval + 12
```

Il tipo dell'identificatore **`newval`** deve corrispondere (*match*) con il tipo dell'espressione (**`oldval+12`**)

Un **Semantic Analyzer** cerca nel programma sorgente errori semantici e raccoglie informazioni necessarie per la successiva generazione del codice e l'ottimizzazione.

Analizzatore Semantico

- Di norma le informazioni semantiche non possono essere rappresentate con un semplice *parse tree*. È necessario associare ad un costrutto (o, meglio, ai simboli terminali e non terminali della grammatica) specifici ***attributi*** (ad es. i tipi di dato delle espressioni, i loro valori, ecc.).
- Le produzioni di una CFG devono essere integrate annotandole con regole o con frammenti di codice (azioni o ***regole semantiche***). Tali frammenti sono eseguiti quando la produzione viene utilizzata durante l'analisi sintattica. Le regole definiscono come calcolare i valori degli attributi dei nodi coinvolti nella produzione.
- L'esecuzione dei frammenti, nell'ordine determinato dall'analisi sintattica, produce come risultato la ***traduzione syntax-directed*** delle istruzioni del programma.

Type-checking è un'importante parte dell'analisi semantica: controlla la semantica statica di ogni nodo del parse tree; verifica che il costrutto sia legale (che tutti gli identificatori coinvolti siano dichiarati, che i tipi siano corretti, e così via); se il costrutto è semanticamente corretto, il *type checker* "decora" il nodo, aggiungendo informazioni relative al tipo o alla parte della tabella di simboli ad esso dedicata e permettendo la produzione dell'AST relativo al costrutto; se viene scoperto un errore semantico, viene notificato un opportuno messaggio d'errore.

Il *type checking* è puramente dipendente dalle regole semantiche del linguaggio sorgente. Esso, cioè, è indipendente dal linguaggio target.

Syntax Analyzer versus Lexical Analyzer

Quali costrutti di un programma devono essere riconosciuti da un *lexical analyzer*, e quali da un *syntax analyzer*?

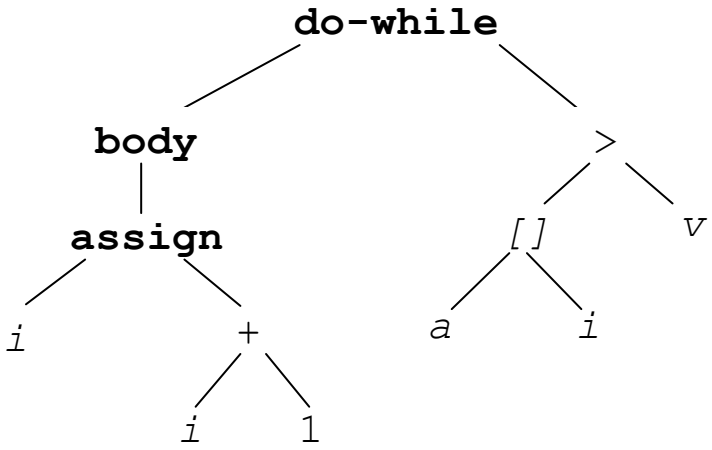
- Ambedue fanno le stesse cose; ma il lexical analyzer tratta solo i costrutti del linguaggio non-ricorsivi.
- Il syntax analyzer tratta i costrutti del linguaggio ricorsivi.
- Il lexical analyzer semplifica il lavoro del syntax analyzer.
- Il lexical analyzer riconosce i componenti più piccoli (token) del programma sorgente.
- Il syntax analyzer lavora sui token riconoscendo le strutture del linguaggio.

GENERATORE DI CODICE INTERMEDIO

Quella dell'**albero sintattico astratto** rappresenta la prima forma di rappresentazione intermedia (IR) del programma sorgente.

Da essa è possibile effettuare esplicitamente una rappresentazione intermedia di basso livello, simile al codice macchina, che si può considerare come una sorta di programma per una macchina virtuale.

Tale codice intermedio, semplice da generare e semplice da tradurre nel codice di macchina del calcolatore di destinazione, è noto come ***codice a tre indirizzi*** (*three-address code*) sia per le espressioni aritmetiche che per le istruzioni.



L'albero sintattico astratto

```
1: i = i + 1
2: t1 = a[i]
3: if  t1 < v goto 1
```

Il *three-addess code*

Rappresentazioni intermedie dello statement	`do i = i + 1; while ( a[i] < v );`

GENERATORE DI CODICE INTERMEDIO

Se un elemento è semanticamente corretto può essere tradotto.

La traduzione richiede che venga catturato il "significato" a run-time di un costrutto.

Per esempio, l'AST di un ciclo **`while`** contiene due sottoalberi, uno per controllare l'espressione di controllo, e l'altro per il corpo del ciclo.

Nulla nell'AST dimostra che un ciclo **`while`** "cicla". Questo "senso" è catturato quando un AST di un ciclo **`while`** viene tradotto. La traduzione è dettata dalla semantica del linguaggio sorgente.

Nella "decorazione"/annotazione dell'albero sintattico effettuata dalle azioni/regole semantiche, la nozione di verificare il valore dell'espressione di controllo, e l'eventuale esecuzione del corpo del ciclo, diventa esplicita.

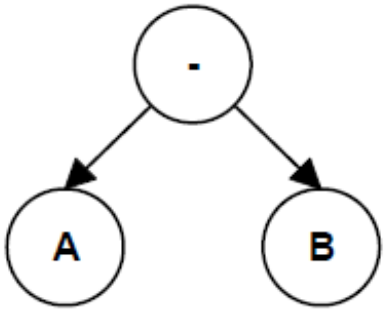
In altri termini, una CFG, oltre a a definire la sintassi di un LdP, con la tecnica della *syntax directed translation* può essere usata come supporto alla traduzione dei programmi.

Per far ciò, come sarà chiaro quando si tratterà in dettaglio il tema della traduzione guidata dalla sintassi, bisognerà trasformare l'AST (e quindi le espressioni e/o le istruzioni di un LdP) dalla notazione infissa alla notazione postfissa, la notazione, cioè, in cui gli operatori appaiono dopo i relativi operandi.

Per semplicità esplicativa considereremo, nel seguito, la traduzione da notazione infissa a postfissa di espressioni aritmetiche.

TRADUZIONE DA NOTAZIONE INFISSA A POSTFISSA

Una generica espressione aritmetica può essere descritta graficamente mediante un albero binario che ricorsivamente riporti nella radice un operatore aritmetico e nei due figli il primo ed il secondo operando rispettivamente. Nella figura l'albero rappresenta, secondo la cosiddetta "***notazione infissa*** sinistra (destra)", l'espressione (A – B) oppure (B – A), a seconda che l'ordine di visita dell'albero preveda prima la visita del figlio sinistro o quella del figlio destro.



Applicando la "***notazione prefissa*** sinistra (destra)", tale rappresentazione grafica corrisponde alla stringa di simboli –AB oppure –BA.

Applicando, infine, la "***notazione postfissa*** sinistra (destra)", tale rappresentazione grafica corrisponde alla stringa di simboli AB– oppure BA–.

Le notazioni pre-fissa e post-fissa si prestano alla memorizzazione ed esecuzione, in forma non ambigua, delle espressioni aritmetiche. La notazione in-fissa, invece, soffre di ambiguità interpretativa e, pertanto, richiedendo ulteriori specifiche, viene meno frequentemente adoperata.

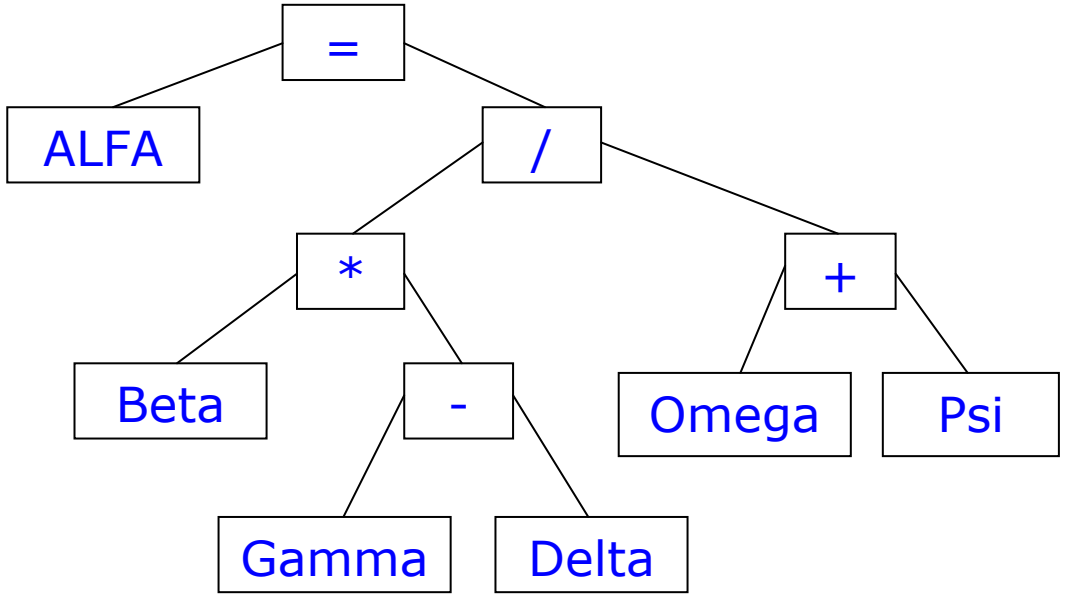
La notazione post-fissa può consentire la esecuzione dell'espressione aritmetica a cui equivale se, percorrendo la stringa da sinistra verso destra, ogni operatore incontrato viene ricorsivamente applicato, rispettivamente, ai due operandi che immediatamente lo precedono. Tali due operandi sono da intendersi il sinistro e poi il destro se la notazione è sinistra, il destro e poi il sinistro se la notazione è destra.

TRADUZIONE DI UN'ESPRESSIONE IN NOTAZIONE POSTFISSA

Considerata l'espressione sorgente

ALFA = Beta * (Gamma - Delta) / (Omega + Psi)

l'analizzatore sintattico ne riconoscerà l'equivalente parse tree e poi il relativo AST,



che potrà essere trasformato nella corrispondente notazione postfissa sinistra

ALFA Beta Gamma Delta - * Omega Psi + / =

agevolmente traducibile in una sequenza di istruzioni a tre indirizzi.

CODICE A TRE INDIRIZZI

Un compilatore può quindi, attraverso la trasformazione dell'AST in istruzioni a tre indirizzi, produrre una rappresentazione intermedia esplicita del codice intermedio.

Questo codice intermedio è generalmente indipendente dall'architettura e deve il suo nome alle istruzioni che lo costituiscono, che hanno la forma ***x = y op z***, in cui **op** è un operatore binario, ***y*** e ***z*** sono degli operandi e ***x*** è l'indirizzo del risultato dell'operazione.

Un'istruzione a tre operandi è in grado di svolgere una sola operazione: tipicamente un calcolo, un confronto oppure un salto.

Il livello del codice intermedio è generalmente vicino a quello del codice macchina.

esempio 1

ALFA Beta Gamma Delta - * Omega Psi + / =

```
minus   Delta,  , t1
add     Gamma, t1 , t2
mult    Beta, t2 , t3
add     Omega, Psi, t1
div     t3 , t1 , t2
mov     t2 ,  , ALFA
```

esempio 2

```
newval := oldval * fact + 1
id1 := id2 * id3 + 1

MULT  id2,id3,temp1
ADD   temp1,#1,temp2
MOV   temp2,,id1
```

Ottimizzatore

L'Ottimizzazione cerca di migliorare il codice per limitare il tempo di esecuzione e la memoria necessaria (questa fase può essere molto complessa)

Esempio di Ottimizzatore di codice (per codice intermedio):

```
MULT id2,id3,temp1
ADD temp1,#1,id1
```

GENERATORE DI CODICE

(Program Synthesis)

Poco della natura del macchina di destinazione deve essere reso evidente durante la fase di generazione ed ottimizzazione del codice intermedio.

Dettagliate informazioni sulla natura della macchina di destinazione (operazioni disponibili, caratteristiche, ecc) sono riservate per la fase finale di generazione del codice di macchina di destinazione.

In semplici compilatori non ottimizzati il traduttore genera il codice di riferimento direttamente, senza utilizzare una IR.

In compilatori più complessi prima si genera una IR di alto livello (*source oriented*) e successivamente il codice viene tradotto in una IR a basso livello (*target oriented*).

Questo approccio consente una chiara separazione fra dipendenze derivanti dal sorgente e dal target.

GENERATORE DI CODICE

(Program Synthesis)

ll codice IR generato dal traduttore è analizzato e trasformato in codice IR equivalente ottimizzato per una specifica architettura.

Il termine ottimizzazione è in realtà ambiguo: non sempre il codice prodotto è la migliore traduzione possibile del codice IR; infatti, alcune ottimizzazioni sono impossibili da eseguire in certe circostanze, perché sono un problema "indecidibile". L'eliminazione del codice "morto" è in generale un problema impossibile da risolvere.

Altre ottimizzazioni sono troppo costose in tutti i casi. Questo comporta problemi NP-complessi, che si ritiene essere intrinsecamente esponenziali .

L'assegnazione di registri ad una variabile è un esempio di un problema NP-complesso.

L'ottimizzazione può essere complessa, può coinvolgere numerose fasi, che potrebbero doversi eseguire più volte.

L'ottimizzazione può determinare la scarsa velocità di traduzione. Tuttavia, un ottimizzatore ben progettato può incrementare in modo significativo la velocità di esecuzione del programma spostando o eliminando operazioni non necessarie.

Generazione del codice

Il codice IR prodotto dal traduttore è mappato nel codice macchina target dal generatore di codice, che produce il linguaggio target per una specifica architettura.

Il programma target è normalmente un codice oggetto rilocabile contenente codice macchina.

Esempio (supponendo di avere un'architettura per cui al più un operando è un registro):

```
MOVE    id2,R1
MULT    id3,R1
ADD     #1,R1
MOVE    R1,id1
```

Questa fase fa uso di informazioni dettagliate sulla macchina target e include ottimizzazione legate alla specifica macchina come *register allocation e code scheduling*.

Il generatore di codice può essere piuttosto complesso, poiché per produrre buon codice target bisogna considerare molti casi particolari.

E' possibile utilizzare i generatori di codice in modo automatico.

L'impostazione di base è quella di definire dei *template* che mettano in corrispondenza le istruzioni low-level IR con le istruzioni target.

Un noto compilatore che utilizza tecniche di generazione automatica di codice è il compilatore GNU C, un compilatore fortemente ottimizzato attraverso l'utilizzo di un file di descrizione contenente la macchina per più di dieci architetture PC, e di almeno due linguaggi (C and C++).

Symbol Table(s)

Una o più tabelle dei simboli sono strutture di dati utilizzate dal compilatore per mantenere informazioni relative agli identificatori e ai costrutti del linguaggio sorgente. Tali informazioni sono raccolte durante le fasi di analisi e sono condivise dalle diverse fasi di compilazione.

Ogni volta che un identificatore viene utilizzato, la tabella di simboli consente di accedere alle informazioni raccolte circa l'identificatore quando la sua dichiarazione è stata elaborata.

Pur apparendo naturale che sia lo scanner, che per primo individua un lessema, ad associare ad esso un elemento della tabella dei simboli, in realtà spesso è il parser a farlo, soprattutto in quanto esso è capace di distinguere le diverse dichiarazioni di un identificatore.

Frequentemente ad ogni scope (nel C, ad esempio, ad ogni blocco di un programma) è opportuno associare una specifica symbol table.

Tool per il progetto di compilatori

- ✓ Tool per lo sviluppo di software general-purpose + tool specializzati

- ✓ Tool per il progetto automatico di parti di compilatori
  - **parser generators**: producono analizzatori sintattici
  - **scanner generators**: producono analizzatori lessicali
  - **syntax-directed translator engines**: producono collezioni di routine che visitano il parse tree e generano il codice intermedio ...
  - **automatic code generators**: prendono regole per tradurre da linguaggio intermedio in linguaggio macchina attraverso soluzioni alternative gestite con "template matching"
  - **data-flow engines:** analisi del data-flow per ottimizzare il codice attraverso informazioni su come si propagano i dati da una parte all'altra del programma

Formalismi di interesse per il progetto di compilatori

Analisi lessicale

- ✓ **Regular Grammars**
- ✓ **Finite State Automata**
- ✓ **Regular Expressions**

Analisi sintattica

- ✓ **Grammatiche non contestuali (Context-Free Grammars)**, anche se non ogni elemento linguistico in un linguaggio di programmazione può essere trattato come grammatica non contestuale
- ✓ **Parser CFG** aventi una complessità $O(n^3)$

Analisi semantica

- ✓ Traduzione diretta alla sintassi **(Syntax-directed translation)**
- ✓ Grammatiche ad attributi **(Attribute grammars)**
- ✓ Altri approcci più sofisticati

Code generation

- ✓ **Pattern matching**
- ✓ Euristica
- ✓ Soluzioni ad-hoc

Tabella dei simboli