Parte 3

Architettura di un linguaggio dinamico

Compilatori ed interpreti

- **Gli strumenti per la generazione di codice eseguibile sono classificabili in due categorie distinte:**
  - ***Compilatori***
  - ***Interpreti***
- **Compilatore: traduce un codice (sorgente) scritto in un linguaggio di programmazione in un altro linguaggio (codice oggetto) di più basso livello**
  - ***Caso "esemplare": C → codice oggetto in linguaggio macchina (codice binario)***
- **Interprete: considera porzioni limitate (statement) di codice (sorgente o intermedio), le traduce in codice macchina e le esegue direttamente**

Ricordiamo: non è una caratteristica del linguaggio

Linguaggi interpretati vs. compilati

- ***Svantaggi* di un linguaggio interpretato:**
  - **Più lento nell'esecuzione (ogni esecuzione implica una traduzione a run time)**
  - **Richiede più memoria**
  - **Richiede la presenza del software interprete sul calcolatore**
- ***Svantaggi* di un linguaggio compilato:**
  - **Minore portabilità**
  - **Minore flessibilità (es. non permette meccanismi di type checking dinamico)**
  - **Supporto per debugging a run time meno flessibile e potente**

Linguaggi Dinamici

- **Obiettivi dei linguaggi dinamici:**
  - ***Portabilità e rapidità di prototipazione***
  - ***Velocità maggiore di un linguaggio completamente interpretato (es. Shell)***
- **Approccio ibrido**
  - **Compilazione + interpretazione**
  - **Formato di rappresentazione intermedia del codice indipendente dall'architettura**
- **A seconda del tipo di formato intermedio, si distinguono due diversi modelli di esecuzione:**
  - **Modello "Abstract Syntax Tree" (AST)**
  - **Modello "Bytecode"**

Modello “Abstract Syntax Tree”

- Modello Abstract Syntax Tree: prevede utilizzo di un compilatore e di un interprete
- Compilatore:
  - Traduce il codice sorgente in una rappresentazione ad albero sintattico (Abstract Syntax Tree)
  - In questa fase (a seconda del linguaggio) è possibile prevedere l’interpretazione o l’esecuzione di speciali sezioni di codice (ad esempio, inizializzazioni)
  - Blocchi BEGIN/END (ES. Perl)

Modello “Abstract Syntax Tree”

- **Interprete:**
  - Attraverso *algoritmi di visita* dell'AST, considera “porzioni di albero”, le traduce in istruzioni corrispondenti e le esegue - statement di codice
  - Possibilità di inserire durante questa fase una fase di compilazione di codice
    - Es. Perl - Comando eval(espressione)
    - Usato tipicamente in caso di Metaprogramming per la generazione di nuovo codice a runtime (richiede presenza di un compilatore a run time)

Modello “Bytecode”

- Il modello bytecode prevede l'utilizzo di un compilatore e di una macchina virtuale (interprete)
- Compilatore:
  - Traduce il codice sorgente in una linguaggio intermedio di basso livello, portabile
    - Bytecode (o p-code – portable code)
    - Rappresenta un passo successivo (potenzialmente più ottimizzato) rispetto all'AST
  - Il bytecode viene installato sui sistemi di destinazione, dove viene tradotto in codice macchina ed eseguito

Modello “Bytecode”

- **Macchina virtuale (interprete):**
  - **Fornisce una astrazione di un sistema operativo**
    - **Il Bytecode può essere considerato come il 'codice macchina' della macchina virtuale**
  - **Traduce le istruzioni e le richieste di sistema espresse in bytecode nelle richieste concrete al sistema operativo reale**
  - **Le funzionalità di base della macchina virtuale sono contenute nella *libreria di funzioni* detta runtime library**
  - **Runtime library + MV = Runtime Environment**

Compilazione Just-In-Time (JIT)

- … o *traduzione dinamica* (*dynamic translation*)
- Ne viene fatto uso in linguaggi che adottano un modello a bytecode (Java, Python, Ruby, PHP,…)
- Utilizzata da quasi tutte le recenti implementazioni di macchine virtuali
- L'ambiente di esecuzione (macchina virtuale) incorpora un compilatore interno *just-in-time* che, a run time:
  - compila porzioni di codice bytecode traducendole nel linguaggio macchina nativo del computer ospite
  - le memorizza per il riuso
- Chiaramente impiegato per ragioni di efficienza (esempio: ottimizzazioni su porzioni di codice eseguite frequentemente)

Modelli a confronto

Forniremo gli elementi essenziali per un confronto fra:

- il modello di esecuzione del C
  - completamente compilato
- il modello di esecuzione del Perl
  - Modello Abstract Syntax Tree (AST)

Compilazione completa

Operazioni principali di un compilatore C-like

1) **Analisi (lessicale, sintattica e semantica):** il testo viene diviso in unità di base e analizzato e con diversi tipi di controlli; genera in output una *rappresentazione intermedia del codice detta Abstract Syntax Tree (AST)*
2) **Generazione del codice:** la rappresentazione intermedia viene tradotta nel *formato finale*
3) **Ottimizzazione del codice:** il codice risultante viene *ottimizzato secondo un qualche criterio specifico* (velocità, uso memoria, consumo di energia)

Compilatore: analisi lessicale

- **Obiettivo: dividere il flusso di caratteri in ingresso (codice sorgente) in tante unità chiamate token**
  - **Token: elementi minimi (non ulteriormente divisibili) di un programma**
    - **Es.: keywords (for, while), nomi di entità (pippo), operatori (+, -, <<), ...**
- **La fase di tokenization è eseguita da un analizzatore lessicale detto scanner (o lexer)**
- **Scanner: macchina a stati finiti che riconosce possibili token definiti mediante espressioni regolari**
  - **Es.: un numero intero è un (eventuale) carattere +/- seguito da una sequenza di cifre**

Compilatore: analisi lessicale

- Ad ogni token identificato è associata una categoria (significato nel lessico del linguaggio) assegnata in base all'espressione regolare che ha riconosciuto il token
  - *Esempio*: tabella dei token di sum = 3 + 2

Token	Categoria
sum	IDENTIFIER
=	ASSIGN_OP
3	INT_NUMBER
+	ADD_OPERATOR
2	INT_NUMBER

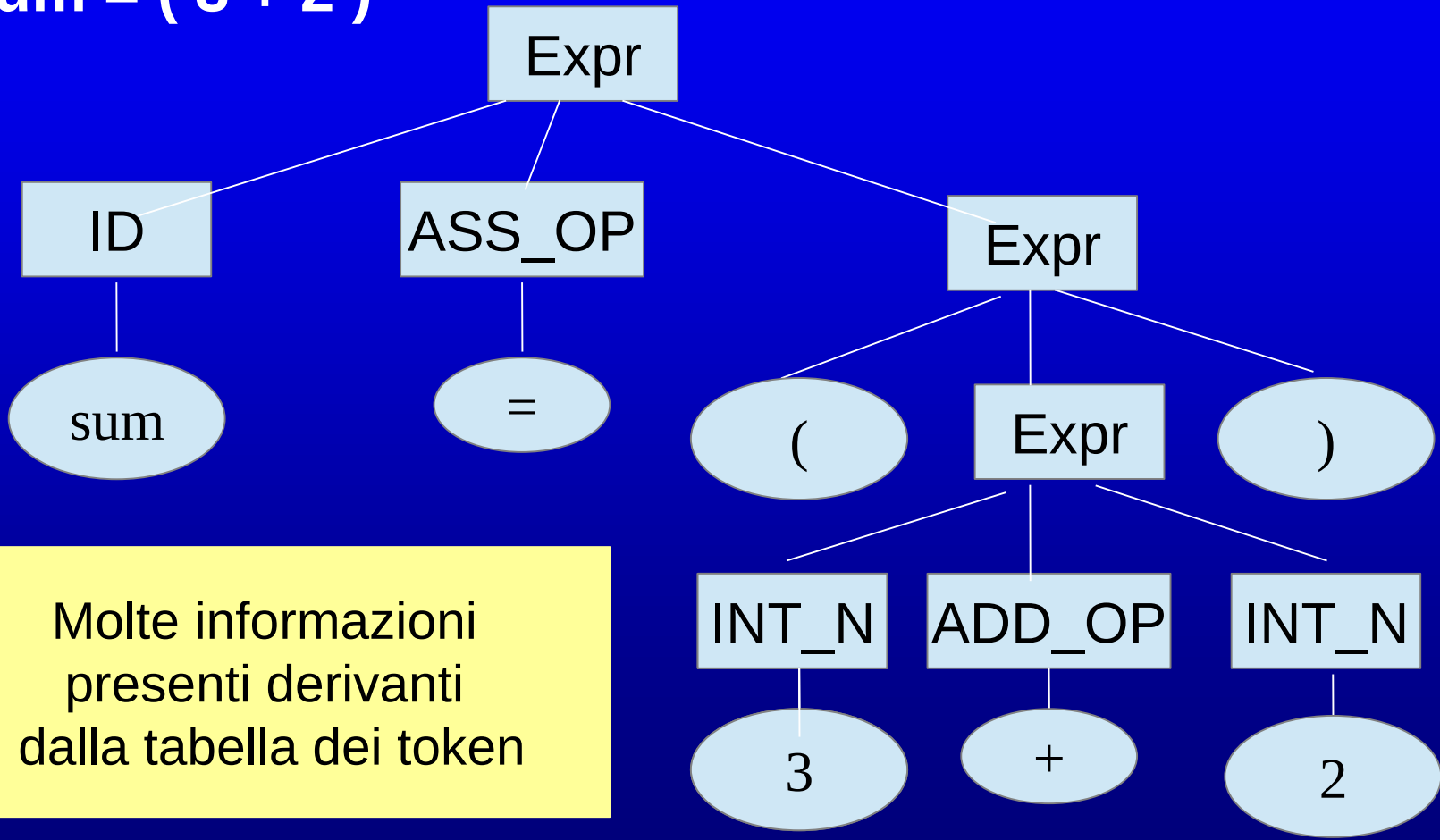
Compilatore: analisi sintattica

- Prende in ingresso la sequenza di token ed esegue il controllo sintattico: verificare che i token formino un'espressione valida
  - ES. L'espressione “a = + = b” è riconosciuta non valida solo a questo livello: prima lo scanner avrebbe targato ogni operatore come identifier, assign o add (visione locale)
- Eseguita da analizzatore sintattico o parser
- Il risultato è un albero di sintassi o parse tree:
  - Albero che rappresenta la struttura sintattica di una espressione in accordo alla grammatica usata
  - I token sono tutti rappresentati da nodi foglia

Compilatore: analisi sintattica

Esempio di Parse Tree generato per:

sum = (3 + 2)



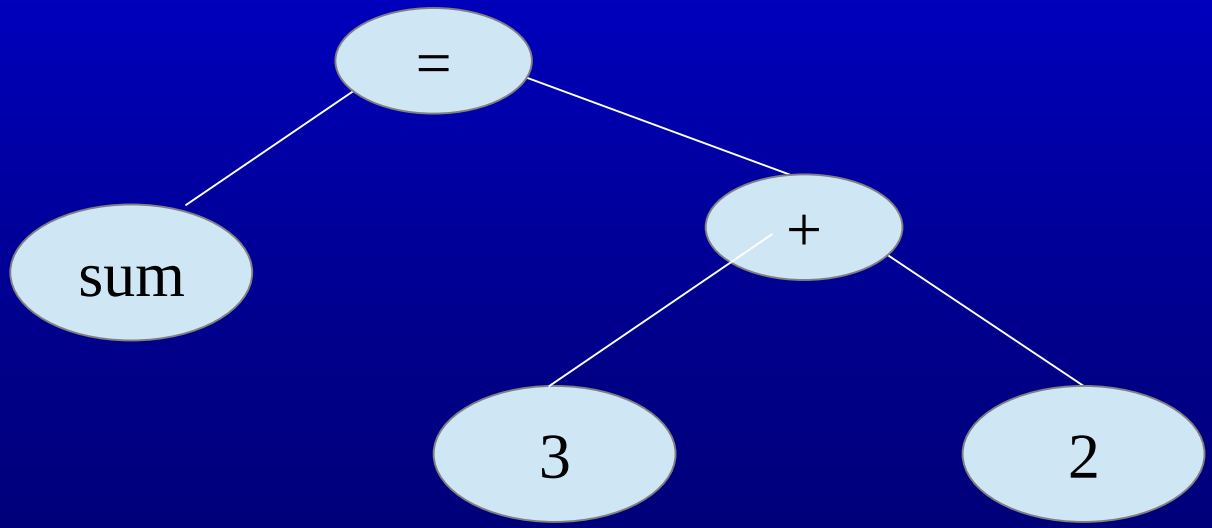
Molte informazioni presenti derivanti dalla tabella dei token

Compilatore: analisi semantica

- **Obiettivo: rilevare (sequenze di) istruzioni non corrette, ovvero senza un significato valido a livello semantico**
- ***Controlli tipici di questa fase:***
  - **Controllo che gli identificatori siano stati dichiarati e inizializzati**
  - **Type checking (controllo di tipo) – operazioni possibili e assegnamenti di valori corretti**
- **Durante questa fase viene creata una tabella dei simboli con informazioni su ciascun simbolo (nome, scope, tipo, …)**

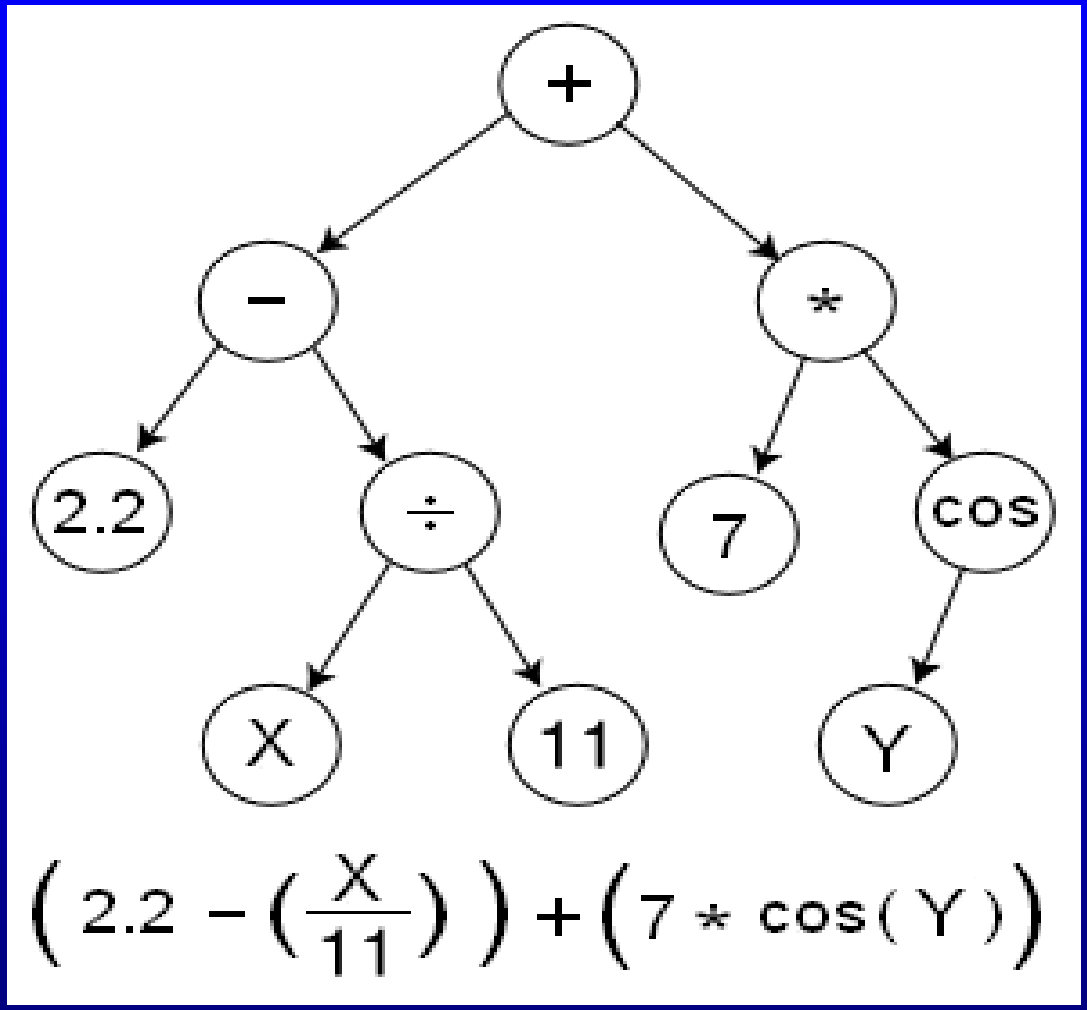
Compilatore: analisi semantica

- **Il risultato dell'analisi semantica è un albero detto Abstract Syntax Tree (AST)**
  - **Operatori e keyword NON sono nodi foglia**
- ***Astratto*: non rappresenta *esplicitamente* tutti i dettagli rappresentati nella sintassi, che vengono però rappresentati in modo implicito (es. parentesi)**



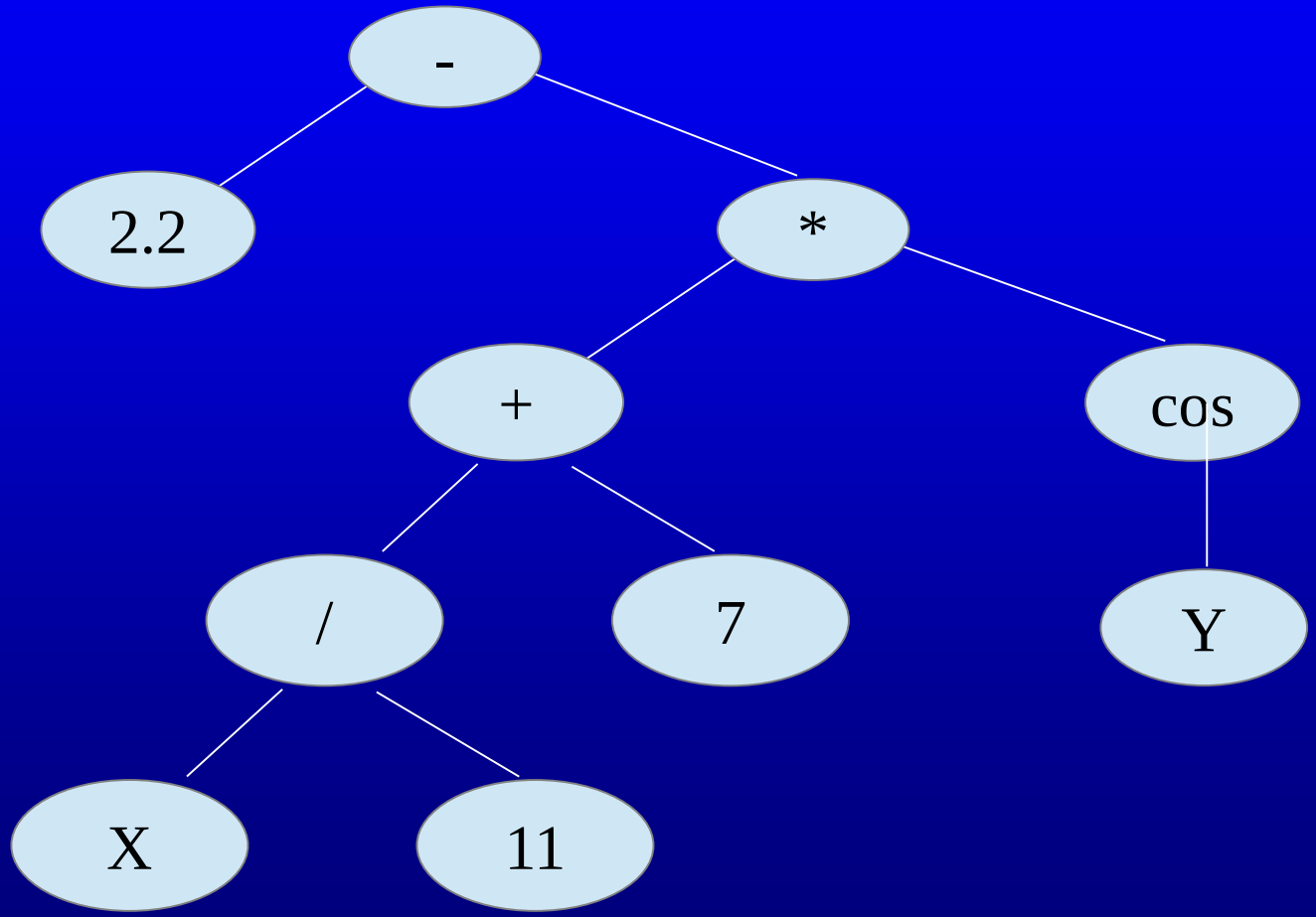
sum = (3 + 2)

Esempio AST


+
−
*
2.2
÷
7
cos
X
11
Y
$\left(2.2 - \left(\frac{X}{11} \right) \right) + \left(7 * \cos(Y) \right)$

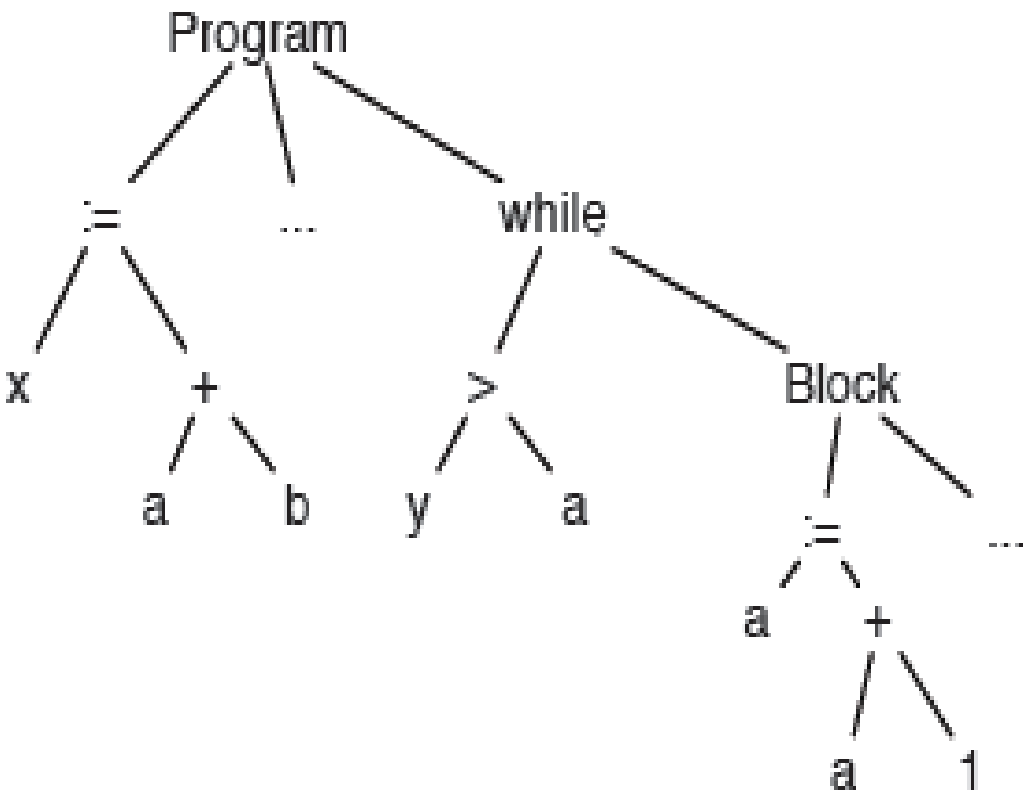
Esempio AST

2.2 – ((X / 11) + 7) * cos(Y)



Esempio AST

```
x := a + b;
y := a * b;
while (y > a) {
    a := a + 1;
    x := a + b
}
```



Compilatore: generazione del codice

- **Il modulo di generazione di codice:**
  - **prende in input l'Abstract Syntax Tree prodotto dall'analisi semantica**
  - **converte la struttura ad albero in una sequenza di codice/istruzioni**
  - **Spesso integra alcune tecniche di ottimizzazione**
  - **La sequenza prodotta può ancora essere intermedia se è prevista una fase separata di ottimizzazione, seguita da generazione del codice target/macchina**
  - **ES. Linguaggio intermedio: assembly o linguaggio intermedio interno al compilatore**

Compilatore: generazione del codice

La generazione del codice avviene attraverso tre processi: instruction selection, instruction scheduling e register allocation

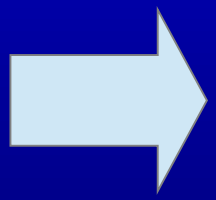
1) Instruction selection: vengono definite le istruzioni che rappresenteranno la conversione dalla rappresentazione ad albero

- ***Approccio:*** **si spezza la rappresentazione ad albero nel numero più basso possibile di tile**
- ***tile*** **= porzione di albero che può essere implementato con una singola istruzione nel codice generato**

Compilatore: generazione del codice

- **Ottimizzazione integrata nella fase di instruction selection**
- **Es. Una traduzione naive può generare codice inefficiente: necessità di eliminare accessi ridondanti alla memoria, riordinando e fondendo istruzioni, e sfruttando l'uso dei registri**

```
t1 = a
t2 = b
t3 = t1 + t2
a = t3
b = t1
```



```
MOV EAX, a
XCHG EAX, b
ADD a, EAX
```

Compilatore: generazione del codice

2) Instruction scheduling: si sceglie la sequenza in cui sistemare le istruzioni

- ***Obiettivo*: migliorare il parallelismo a livello di istruzioni (maggior throughput di istruzioni)**
- **Tecnica di ottimizzazione tipica: instruction pipeline (classica pipeline su architetture RISC - Reduced Instruction Set Computer – applicazioni embedded)**
  - **IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back**
- **Tentativo di utilizzare in modo parallelo accessi a livelli diversi di memoria e risorse di calcolo**

Compilatore: generazione del codice

3) Register allocation: si decide come allocare le variabili dentro ai registri della CPU

- ***Obiettivo*: cercare di assegnare quante più variabili possibili ai registri della CPU per velocizzare l'accesso alle variabili**
  - **Accesso più veloce rispetto alla RAM**
  - **Spesso le variabili contemporaneamente in uso sono in numero maggiore dei registri disponibili**
- ***Ottimizzazione integrata*: scelta delle variabili che vengono accedute più di frequente (scrittura vs lettura)**

Schema delle fasi della compilazione