

# Compilatore di Functional and Object Oriented Language

# Functional and Object Oriented Language

---

- Sviluppo di un **compilatore** per il **Functional and Object Oriented Language (FOOL)**.
- La sintassi del linguaggio è definita nel file **FOOL.g4** allegato a queste slides.
- Il linguaggio è una **estensione** della versione di base sviluppata in laboratorio con:
  - operatori aggiuntivi "**<=**", "**>=**", "**||**", "**&&**", "**/**", "**-**" e "**!**", con stesso significato che hanno in C/Java;
  - passaggio (higher-order) di funzioni;
  - gestione dell'object orientation.

# Il compilatore

---

- Descritto in modo dettagliato come *estensione del compilatore sviluppato in laboratorio* per la versione di base del linguaggio.
- Il compilatore quindi *produce codice per la Stack Virtual Machine (SVM)* sviluppata a lezione: *senza bisogno di modificarla!*

**Estensione Higher Order**

# Elementi Sintattici Nuovi

---

- Tipi in dichiarazioni (non-terminale "hotype") ora includono:
  - oltre a tipi di base (non-terminle "type"): bool e int
  - anche tipi funzionali (non-terminale "arrow"):  
(hotype<sub>1</sub>,...,hotype<sub>n</sub>)->type
    - cioè un "type" come tipo di ritorno e tipi (possibilmente funzionali) "hotype" per i parametri
- Esempio di dichiarazione con tipo funzionale "arrow" di una variabile (o parametro) "x":  
x: (int,int)->bool  
cioè x contiene una funzione che ritorna un bool ed ha due parametri, entrambi int

# Layouts

---

- layout **AR** (amb globali/funzioni) invariato
  - ma ora qualsiasi ID con tipo funzionale (vero ID di funzione oppure ID di variabile o parametro di tipo funzionale) occupa un **offset doppio**:

[a offset messo in symbol table ] indir (fp) di **AR dichiarazione funzione**

[a offset messo in symbol table-1] **indir funzione** (per invocazione suo codice)

# Estensione Higher Order Funzionamento Parser

# Abstract Syntax Tree

---

- Dichiarazioni
  - interfaccia `DecNode` con metodo `getSymType()` che implementano tutte: `VarNode`, `FunNode`, `ParNode`
  - `getSymType()` su un `DecNode` deve essere implementato in modo che ritorni il tipo messo in `Symbol Table`
    - per `FunNode` prevedere un campo `symType` dove memorizzarlo
- Tipi
  - per i tipi "arrow" creare degli `ArrowTypeNode` (che risulteranno possibilmente annidati nei parametri)



# Symbol table

---

- **STentry**: invariata
  - ora, oltre agli ID di funzione, anche gli ID di variabili/parametri potranno avere tipo funzionale ArrowTypeNode
- Durante il parsing dei parametri e delle dichiarazioni
  - incremento/decremento offset deve tener conto che gli ID di tipo funzionale occupano offset doppio

# Estensione Higher Order Type Checking

# Subtyping

---

- `isSubtype()` in FOOLlib ora deve gestire (oltre a "bool" sottotipo di "int") **tipi funzionali**  
**ArrowTypeNode**
  - entrambi devono essere **ArrowTypeNode** con **stesso numero di paramteri** e deve valere:
    - relazione di **co-varianza** sul tipo di ritorno
    - relazione di **contro-varianza** sul tipo dei parametri

# Espressioni (a lato si indica elemento sintattico)

---

- **IdNode**            **ID**
  - ora ammettere **anche un ID con tipo funzionale!**  
(nome di funzione o var/par di tipo funzionale)
- **EqualNode**         **$exp_1 == exp_2$** 
  - **non consentire l'uso di espressioni  $exp_i$  con tipi funzionali** (dovrei confrontare coppie di valori)
- **CallNode**            **ID()**
  - **check invariato: il tipo dell'ID deve essere funzionale**  
(nome di funzione o var/par di tipo funzionale)
- **Dichiarazioni** **invariate**

# Estensione Higher Order Code Generation

# Dichiarazioni

---

- FunNode
  - codice ritornato: due cose sono messe nello stack, nell'ordine
    1. indir (fp) a questo AR (in reg \$fp)
    2. (finisce a offset-1) indir della funzione (etichetta generata)
  - codice della funzione:
    - in caso tra i parametri o le dichiarazioni vi siano ID di tipo funzionale (usare getSymType() su DecNode) si devono deallocare due cose dallo stack (con due "pop")
- VarNode invariato
  - ritorna codice generato da sua inizializzazione che metterà due cose in stack se var è di tipo funzionale
    - si vedano a riguardo espressioni nel seguito (IdNode)

# Espressioni

---

- **IdNode**      **ID**
  - se il tipo non è funzionale, ritorna codice invariato
  - se lo è, due cose sono messe nello stack, recuperandole come valori dall'AR dove è dichiarato l'**ID**, con meccanismo usuale di risalita catena statica; nell'ordine
    1. indir (fp) ad AR dichiaraz. funzione (recuperato a offset **ID**)
    2. indir funzione (recuperato a offset **ID - 1**)

# Espressioni

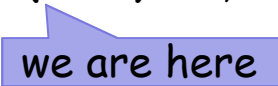
---

- CallNode ID()
  - codice ritornato modificato: due cose recuperate come valori dall'AR dove è dichiarato l'ID con meccanismo usuale di risalita catena statica
    - indir (fp) ad AR dichiaraz. funzione (recuperato a offset ID)
      - usato per settare nuovo Access Link (AL)
    - indir funzione (recuperato a offset ID - 1)
      - usato per saltare a codice funzione



# Esempio linsum

**let**

```
fun g:int(x:(int,int)->int)
  x(5,7);
fun f:int(c:int)
  let
    fun linsum:int(a:int,b:int)
      (a+b)*c;
  in  we are here
    g(linsum);
```

**in**

```
print(f(2));
```

Rappresentazione **semplificata**:  
solo **contenuto** (e non posizione)  
di **parametri/dichiarazioni** e **AL**

## CONTENUTO STACK

**AR1: ambiente globale**

**g=(g\_entry,AR1)**

**f=(f\_entry,AR1)**

**AR2: f call**

**AL=AR1**

**c=2**

**linsum=(linsum\_entry,AR2)**

**AR3: g call**

**AL=AR1**

**x=(linsum\_entry,AR2)**

**AR4: linsum call**

**AL=AR2**

**a=5**

**b=7**

**TOP DELLO STACK**

**Estensione Object Oriented**

## Commenti Preliminari su Estensione OO

---

- Descritta come **estensione della versione base del linguaggio fatta in laboratorio**
  - in seguito vedremo come si combina con higher order
- Si usa lo **heap** per allocare gli **oggetti** e le **dispatch tables** (spiegate a lezione)
- Lo heap viene allocato da **indirizzi bassi verso indirizzi alti** (registro **\$hp** inizialmente è 0)
- Per semplicità **non implementeremo deallocazione oggetti** (es. garbage collector)

# Elementi Sintattici Nuovi

---

- dichiarazioni (solo ambiente globale e all'inizio)  
`class ID1 [extends ID2] (..campi dichiarati come parametri..) {`  
    ..`metodi dichiarati come funzioni`..  
`}`  
dove `extends ID2` è opzionale e `ID2` è ID di una classe
- espressioni
  - `ID1.ID2(..)`
  - `new ID(..)`      dove `ID` è ID di una classe
  - `null`
- tipi (tipo dei riferimenti)
  - `ID`      dove `ID` è ID di una classe

# Esempio: Dichiarazione Classe

---

**let**

```
class A (a:int, b:bool) {  
  fun n:int(...) ... ;  
  fun m:bool(...) ... ;  
}
```

**in**

**... ;**

- **Campi** "a" e "b" dichiarati con **sintassi di parametri**
- Esempio **creazione oggetto** di classe A: **new A(5,true)**
  - **costruisce** un oggetto che ha campi **a=5** e **b=true**
  - **restituisce** il riferimento (di tipo A) **all'oggetto**

- **Oggetti**, una volta creati, sono **immutabili**
  - **campi non modificabili**
- **Campi accessibili** (leggibili) **solo da dentro la classe A** (o da dentro una **classe che eredita da A**)
  - tramite il **nome**, es. **a+5**
- **Metodi** invocabili da **dentro** classe (che eredita da) A
  - come funzioni, es. **n(...)**o anche dall'**esterno**
  - con notazione **x.n(...)** dove "x" contiene riferimento di tipo A

# Esempio: Ereditarietà

---

```
let
  class A (a:int, b:bool) {
    fun n:int(...) ... ;
    fun m:bool(...) ... ;
  }
  class B extends A (c:int) {
    fun l:int(...) ... ;
  }
in
  ... ;
```

- Esempio **creazione oggetto** di classe B: `new B(5,true,7)`
  - **costruisce** un oggetto che ha campi `a=5`, `b=true` e `c=7`
  - **restituisce il riferimento** (di tipo B) **all'oggetto**

# Esempio: Ereditarietà e Overriding di Campi

---

```
let
  class A (a:int, b:bool) {
    fun n:int(...) ... ;
    fun m:bool(...) ... ;
  }
  class B extends A (c:int, a:bool/* overriding */) {
    fun l:int(...) ... ;
  }
in
  ... ;
```

- L'overriding di campi **modifica il tipo di un campo** ma **non estende** l'elenco dei campi
- Es. **creazione oggetto** di classe B: `new B(false,true,7)`
  - **costruisce** un oggetto che ha campi `a=false`, `b=true` e `c=7`

# Layouts

---

- layout **oggetti** in HEAP:

[PRIMA POSIZIONE LIBERA HEAP]	<- \$hp subito dopo allocazione oggetto
<b>dispatch pointer</b>	[offset 0] <- object pointer
valore primo campo dichiarato	[offset -1]
.	
.	
valore ultimo (n-esimo) campo	[offset -n]



# Layouts

---

- layout **dispatch tables** in HEAP:

```
[PRIMA POSIZIONE LIBERA HEAP] <- $hp subito dopo allocazione tabella  
addr ultimo (m-esimo) metodo      [offset m-1]  
.  
.  
addr primo metodo dichiarato      [offset 0] <- dispatch pointer
```

# Layouts

---

- layout degli **AR** (amb. globale/funzioni/metodi)
  - invariato
    - dichiarazioni classi in ambiente globale occupano lo spazio di un indirizzo: il dispatch pointer della classe
    - sono insieme alle altre dichiarazioni dell'ambiente globale (variabili e funzioni): in nostro layout offset iniziale è -2
- Nota: in caso di AR di un metodo
  - il suo Access Link (AL) contiene l'object pointer dell'oggetto ("this" in C/Java) su cui lo si ha invocato (ambiente delle dichiarazioni nel corpo della classe)

# Estensione Object Oriented Funzionamento Parser

# Abstract Syntax Tree

---

- Dichiarazioni
  - **ClassNode**
    - mettere i figli campi in campo "fields" e i figli metodi in campo "methods", in ordine di apparizione
  - **FieldNode**
  - **MethodNode**
- Nuove e vecchie (VarNode, FunNode, ParNode) implementano **interfaccia DecNode**
  - contiene **getSymType()**, da implementare in modo che ritorni il **tipo messo in Symbol Table**
    - per FunNode, MethodNode e ClassNode prevedere un **campo symType** dove memorizzarlo

# Abstract Syntax Tree

---

- Espressioni (a lato si indica elemento sintattico)
  - IdNode ID
  - CallNode ID()
  - ClassCallNode ID.ID()
  - NewNode new ID()
  - EmptyNode null
- Tipi (in AST/restituiti da type checking)
  - RefTypeNode ID
    - contiene l'ID della classe come campo
  - EmptyTypeNode (tipo di null)
    - non in AST ma restituito da typeCheck() di EmptyNode

# Symbol Table: struttura STentry

---

- In aggiunta a **nesting level**, **offset** e **tipo**:
  - booleano **isMethod**
    - per distinguere ID di funzioni da ID di metodi, che richiedono uso **dispatch table** quando vengono invocati
- Nota. Offset calcolato diversamente per:
  - classi/funzioni/variabili
  - parametri
  - campi
  - metodiin base al rispettivo layout

# Symbol Table: STentry per i nomi delle Classi

---

- **Nesting level** è 0 (ambiente globale)
- **Offset**: da -2 decrementando ogni volta che si incontra una nuova **dichiarazione di classe**
  - in base alla sintassi, **dichiarazioni di funzioni/variabili appaiono in seguito nell'ambiente globale**
    - quindi **a nesting level 0** offset di funzioni/variabili si calcolano decrementando **a partire dall'offset raggiunto dalle classi**
- **Tipo**:
  - **ClassTypeNode** che ha come **campi**:
    - `ArrayList<Node> allFields`  
(tipi dei campi, inclusi quelli ereditati, in ordine di apparizione)
    - `ArrayList<Node> allMethods`  
(tipi funzionali metodi, inclusi ereditati, in ordine apparizione)<sup>31</sup>

## Symbol Table: dentro classi diviene Virtual

---

- Mentre il parser è dentro una classe, la Symbol Table per il livello corrispondente (livello 1 da noi) deve includere anche le
  - STentry per i simboli (metodi e campi) ereditati su cui non è stato fatto overriding
- Per questo motivo tale tabella viene chiamata Virtual Table

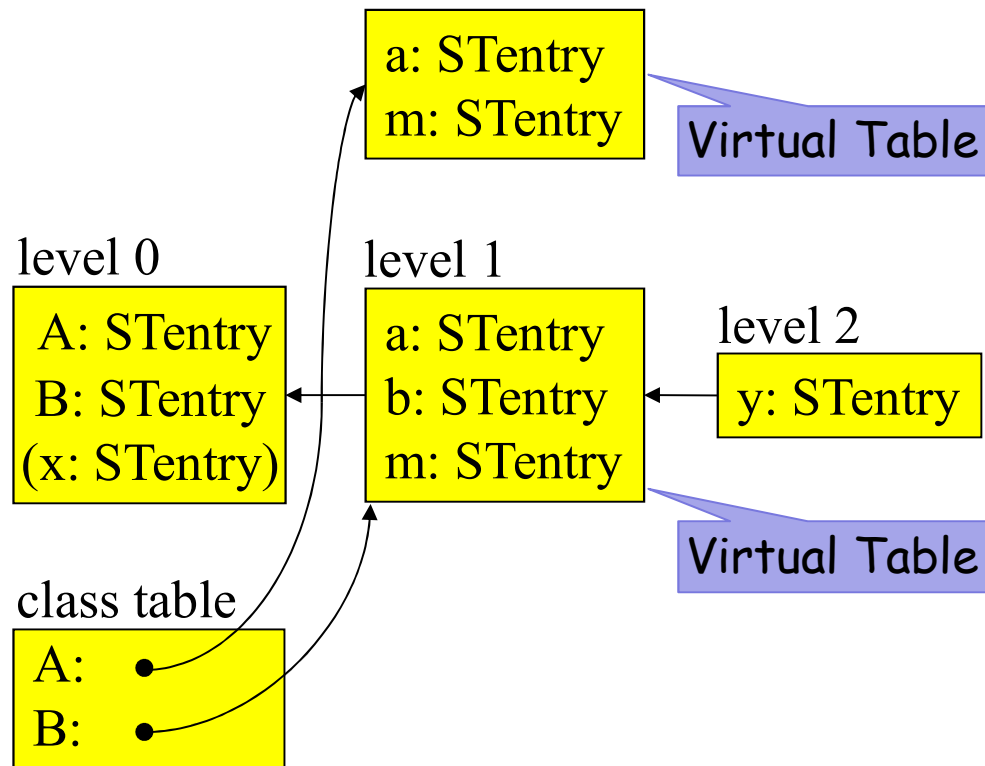


## Symbol Table: aggiunta della Class Table

---

- In aggiunta a Symbol Table multilivello uso anche una Class Table
  - mappa ogni nome di classe nella propria Virtual Table
    - `HashMap<String, HashMap<String, STentry>>` classTable
  - serve per preservare le dichiarazioni interne ad una classe (campi e metodi) una volta che il parser ha concluso la dichiarazione di una classe
    - le rende accessibili anche in seguito tramite il nome della classe, es.
      - uso di un metodo tramite notazione `ID1.ID2(..)`
      - calcolo Virtual Table di classe che eredita

# Symbol Table: esempio



```
let
  class A (a: int) {
    fun m:int() ... ;
  }
  class B extends A (b: int) {
    fun m:bool()
      let
        var y:bool = true;
      in ... ;
  }
  var x:int = 5;
in
  ... ;
```

we are here

# Symbol Table: gestione Class e Virtual Tables

---

- Quando si incontra una **dichiarazione di classe**:
  - nella **Symbol Table (livello 0)** viene aggiunto il **nome della classe** mappato ad una **nuova STentry**
    - se non si eredita, il tipo è un nuovo oggetto **ClassTypeNode** con una lista **inizialmente vuota** in **allFields** e **allMethods**
    - altrimenti, il tipo viene creato **copiando il tipo della classe da cui si eredita** (si deve creare copia di **tutto il contenuto** dell'oggetto **ClassTypeNode** e non copiare il solo riferimento)
  - nella **Class Table** viene aggiunto il **nome della classe** mappato ad una **nuova Virtual Table**
    - se non si eredita, essa viene **creata vuota**
    - altrimenti, viene creata **copiando la Virtual Table della classe da cui si eredita** (si deve creare copia di **tutto il contenuto** della **Virtual Table** e non copiare il solo riferimento)

# Symbol Table: gestione Class e Virtual Tables

---

- All'entrata dentro la dichiarazione della classe:
  - viene creato un nuovo livello per la Symbol Table
    - ma anziché creato vuoto, viene posto essere la nuova Virtual Table creata (ogni livello è un riferimento!)
- All'interno della dichiarazione della classe:
  - Virtual Table e oggetto ClassTypeNode (contenuto dentro la STentry del nome della classe) vengono aggiornati tutte le volte che si incontra
    - la dichiarazione di un campo (parametro della classe)
    - la dichiarazione di un metodo
- All'uscita dalla dichiarazione della classe
  - inalterato: rimosso livello corrente Symbol Table

# Symbol Table: dichiarazioni campi e metodi

---

## 1. Aggiornamento Virtual Table

- come inserimento in livello corrente Symbol Table di dichiarazioni di variabili e funzioni, a parte
  - se nome di campo/metodo è già presente, non lo considero errore, ma **overriding**: sostituisco nuova STentry alla vecchia **preservando l'offset** che era nella vecchia STentry
    - non consentire overriding di un campo con un metodo o viceversa
  - altrimenti, invariato: uso **contatore di offset** e lo decremento/incremento

# Symbol Table: dichiarazioni campi e metodi

---

## 2. Aggiornamento `ClassTypeNode`

- considero `tipo` e `offset` del campo/metodo dichiarato che ho messo dentro la sua `STentry` al punto 1.
  - per i `campi` aggiorno array `allFields` settando la posizione `-offset-1` al `tipo` (in nostro layout `offset` primo campo è -1)
  - per i `metodi` aggiorno array `allMethods` settando la posizione `offset` al `tipo` (in nostro layout `offset` primo metodo è 0)

# Symbol Table: dichiarazioni campi e metodi

---

- Contatore di offset per campi/metodi
  - se non si eredita, settato inizialmente in base a layouts di oggetti e dispatch tables
    - in nostri layouts: -1 per campi e 0 per metodi
  - altrimenti, settato in base a `ClassTypeNode` in `STentry` della classe da cui si eredita: primo offset libero in base a lunghezza di `allFields` e di `allMethods`
    - nostri layouts: -lunghezza-1 per campi e lunghezza per metodi

## Symbol Table: decorazione nodi AST (usi di ID)

---

- IdNode e CallNode ID e ID()
  - invariati: STentry di ID in campo "entry"
- ClassCallNode ID1.ID2()
  - STentry di ID1 in campo "entry"
    - cercata come per ID in IdNode e CallNode (discesa livelli)
  - STentry di ID2 in campo "methodEntry"
    - cercata nella Virtual Table (raggiunta tramite la Class Table) della classe del tipo RefTypeNode di ID1



# Symbol Table: decorazione nodi AST (usi di ID)

---

- **SingleNode**                      `new ID()`
  - **STentry** della **classe ID** in **campo "entry"**
    - ID deve essere in Class Table e STentry presa direttamente da livello 0 della Symbol Table
- **ClassNode**                      `class ID1 [extends ID2]....`
  - **STentry** della **classe ID2** in **campo "superEntry"**
    - ID2 deve essere in Class Table e STentry presa direttamente da livello 0 della Symbol Table

# Estensione Object Oriented Type Checking

# Struttura Super Type

---

- Campo statico `superType` di FOOLlib che mappa ID di classe in ID di sua classe super
  - `HashMap<String,String> superType`
    - struttura che definisce la gerarchia dei tipi riferimento da costruire durante il parsing in aggiunta a quanto già visto

# Subtyping

---

- `isSubtype()` in FOOLlib estesa considerando:
  - un tipo riferimento `RefTypeNode` sottotipo di un altro in base alla funzione `superType`
    - raggiungibilità applicandola multiple volte
  - un tipo `EmptyTypeNode` sottotipo di un qualsiasi tipo riferimento `RefTypeNode`
  - un tipo funzionale `ArrowTypeNode` sottotipo di un altro (come per estensione Higher Order, ma qui necessario per overriding tra metodi) in base alla:
    - relazione di co-varianza sul tipo di ritorno
    - relazione di contro-varianza sul tipo dei parametri

# Dichiarazioni

---

- FieldNode
  - non usato (come ParNode)
- MethodNode
  - identico a FunNode
- ClassNode
  - si richiama sui figli che sono metodi
  - confronta suo tipo `ClassTypeNode` in campo "symType" con quello del genitore in campo "superEntry" per controllare che eventuali **overriding** siano **corretti**
    - scorre posizioni array `allFields/allMethods` del genitore e controlla che il **tipo** alla stessa posizione **nel proprio array** `allFields/allMethods` sia **sottotipo** del tipo in tale posizione

# Espressioni

---

- **IdNode** **ID**
  - invariato (ID non deve essere di tipo funzionale) con aggiunta **controllo che ID non sia il nome di una classe** (di tipo **ClassTypeNode**)
- **ClassCallNode** **ID1.ID2()**
  - **come CallNode**
    - parser ha **già controllato che ID1 sia un RefTypeNode**

# Espressioni

---

- `NewNode` `new ID()`
  - controlla `parametri` come `CallNode`, e torna un `RefTypeNode`
    - recupera i `tipi dei parametri` tramite `allFields` del `ClassTypeNode` in campo "entry"
- `EmptyNode` `null`
  - ritorna tipo `EmptyTypeNode`

# Estensione Object Oriented Code Generation



# Dispatch Tables

---

- Quando si genera il codice per la **dichiarazione di una classe** viene creata la **sua Dispatch Table** (seguendo le regole spiegate a lezione)
- Il codice generato la **alloca nello heap** e mette il relativo **dispatch pointer in AR dell'amb. globale**
  - sarà reperibile all'**offset della classe**
- E' quindi comodo **accedere direttamente ad indirizzo (fp)** dell'AR dell'ambiente globale
  - tale indirizzo in base a nostro layout dell'ambiente globale è **costante MEMSIZE** (valore iniziale di \$fp)
    - spostare tale costante in **FOOLlib** per poterla leggere

# Struttura Dati per Dispatch Tables

---

- Per ogni classe si costruisce la relativa **Dispatch Table** (un ArrayList di String)
  - **etichette** (indirizzi) di tutti i **metodi**, anche ereditati, **ordinati in base ai loro offset**
    - cioè stesso ordine di allMethods nel ClassTypeNode
- Le Dispatch Table di **tutte le classi** vengono **create staticamente** dal compilatore
  - in campo statico **dispatchTables** di FOOLlib
    - `ArrayList< ArrayList<String> > dispatchTables`
  - in **ordine di dichiarazione classi** nell'ambiente globale

# Dichiarazioni

---

- FieldNode
  - non usato (come ParNode)
- MethodNode
  - genera un'etichetta nuova per il suo indirizzo e la mette nel suo campo "label" (aggiungere tale campo)
  - genera il codice del metodo (invariato rispetto a funzioni) e lo inserisce in FOOLlib con putCode()
  - ritorna codice vuoto

# Dichiarazioni

---

- **ClassNode**
  - ritorna codice che alloca su heap la dispatch table della classe e lascia il dispatch pointer sullo stack,
  - ciò viene fatto come descritto in seguito
    - necessita di recuperare l'etichetta e l'offset per ogni suo figlio metodo
      - aggiungere campo "offset" a **MethodNode** e, durante il parsing, settarlo a offset messo in Symbol Table

# Dichiarazione Classe: costruzione Dispatch Table

---

1. aggiungo una nuova Dispatch Table a dispatchTables
  - se non si eredita, essa viene inizialmente creata vuota
  - altrimenti, viene creata copiando la Dispatch Table della classe da cui si eredita (si deve creare copia di **tutto il contenuto** della Dispatch Table e non copiare il solo riferimento)
    - la individuo in base a **offset classe da cui eredito** in "superEntry"; per layout ambiente globale: posizione **-offset-2** di dispatchTables
2. considero in ordine di apparizione **i miei figli metodi** (in campo **methods**) e, per ciascuno di essi,
  - invoco la **sua codeGeneration()**
  - **leggo l'etichetta** a cui è stato posto il suo codice dal suo campo "label" ed il suo **offset** dal suo campo "offset"
  - **aggiorno la Dispatch Table** creata **settando la posizione** data dall'offset **del metodo alla sua etichetta**

## Dichiarazione Classe: codice ritornato

---

1. metto valore di \$hp sullo stack: sarà il dispatch pointer da ritornare alla fine
2. creo sullo heap la Dispatch Table che ho costruito: la scorro dall'inizio alla fine e, per ciascuna etichetta,
  - la memorizzo a indirizzo in \$hp ed incremento \$hp

# Espressioni: codice ritornato

---

- EmptyNode null
  - mette sullo stack il valore -1
    - sicuramente diverso da object pointer di ogni oggetto creato
- IdNode ID
  - invariato
    - indipendentemente che, risalendo la catena statica, giunga ad AR in stack o ad oggetto in heap comunque prendo il valore che c'è all'offset della STentry

# Espressioni: codice ritornato

---

- CallNode ID()
  - controllo se ID è un metodo ("isMethod" in STentry)
    - se non lo è, invariato
    - se lo è, modificato: quando si recupera indirizzo a cui saltare aggiungere 1 alla differenza di nesting level in modo che, risalendo la catena statica, si raggiunga la dispatch table



# Espressioni: codice ritornato

---

- ClassCallNode ID1.ID2()
  - recupera **valore dell'ID1 (object pointer)** dall'**AR dove è dichiarato** con meccanismo usuale di risalita catena statica (come per IdNode) e lo usa:
    - per **settare** a tale valore **l'Access Link** nell'AR del metodo ID2 invocato e
    - per recuperare (usando **l'offset di ID2** nella **dispatch table** riferita dal dispatch pointer dell'oggetto) **l'indirizzo del metodo a cui saltare**

# Espressioni: codice ritornato

---

- NewNode      new ID()
  - **prima:**
    - si richiama su **tutti i parametri** in ordine di apparizione (che mettono ciascuno il **loro valore calcolato sullo stack**)
  - **poi:**
    - prende i **valori dei parametri**, uno alla volta, dallo stack e li mette **nello heap**, incrementando \$hp dopo ogni singola copia
    - **scrive** a indirizzo \$hp il **dispatch pointer** recuperandolo da contenuto indirizzo MEMSIZE + offset classe ID
    - **carica sullo stack** il valore di \$hp (indirizzo **object pointer** da ritornare) e incrementa \$hp
  - **nota:** anche se la **classe ID non ha campi l'oggetto allocato contiene comunque il dispatch pointer**
    - **==** tra **object pointer** ottenuti da due new è **sempre falso!**

# Estensione Object Oriented

## Ottimizzazioni

# Ridefinizione Erronea di Campi e Metodi

---

- Rende possibile rilevare la ridefinizione (erronea) di campi e metodi con stesso nome effettuata all'interno della stessa classe
  - la trattavamo come fosse un overriding
- In parsing, mentre si scorrono le dichiarazioni di campi e metodi dentro una classe,
  - usare una variabile locale contenente un oggetto `HashSet<String>` creato vuoto all'entrata nella classe
  - ad ogni dichiarazione di campo o metodo:
    - controllare se il suo nome è già presente nella `HashSet`
    - se lo è notificare l'errore, altrimenti aggiungerlo alla `HashSet` e gestire la dichiarazione come in precedenza

# Type Checking Più Efficiente per ClassNode

---

- Migliora l'efficienza nel type checking della dichiarazione delle classi
  - effettua il controllo di correttezza (subtyping) solo per i campi/metodi su cui è stato fatto overriding
- Nuovo funzionamento type checking descritto in slide successiva:
  - richiede di recuperare l'offset ed il tipo per ogni suo figlio campo o metodo
    - aggiungere campo "offset" a FieldNode (come già fatto per MethodNode) e, durante il parsing, settarlo a offset messo in Symbol Table

# Type Checking Più Efficiente per ClassNode

---

- Si richiama sui figli che sono metodi (invariato)
- In caso di ereditarietà controlla che l'overriding sia corretto
  - legge il ClassType in "superEntry" e, per ogni proprio figlio campo/metodo:
    - calcola la **posizione** che, in allFields/allMethods di tale ClassType, corrisponde al **suo offset**
      - in nostri layouts: **-offset-1** per campi e **offset** per metodi
    - se la posizione è **inferiore a lunghezza** di allFields/allMethods (overriding) controlla che il **tipo** del figlio sia **sottotipo** del tipo in allFields/allMethods in tale posizione

# Type Checking con Lowest Common Ancestor

---

- Rende possibile utilizzare nei rami **then** ed **else** di un "if-then-else" due espressioni
  - non solo quando sono una sottotipo dell'altra,
  - ma anche quando hanno un lowest common ancestor
- Type checking di **IfNode**
  - chiama **lowestCommonAncestor** (nuovo metodo statico da aggiungere a FOOLlib) sui **tipi** ottenuti per le espressioni nel **then** e nell'**else**:
    - se ritorna null il typechecking fallisce, altrimenti restituisce il tipo ritornato

# Type Checking con Lowest Common Ancestor

---

metodo "Node lowestCommonAncestor (Node a, Node b)"

- per a e b tipi riferimento (o EmptyTypeNode)
  - se uno tra "a" e "b" è EmptyTypeNode torna l'altro; altrimenti
  - all'inizio considera la classe di "a" e risale, poi, le sue superclassi (tramite la funzione "superType") controllando, ogni volta, se "b" sia sottotipo (metodo "isSubtype") della classe considerata:
    - torna un RefTypeNode a tale classe qualora il controllo abbia, prima o poi, successo, null altrimenti
- per a e b tipi bool/int
  - torna int se almeno uno è int, bool altrimenti
- in ogni altro caso torna null



**Estensione sia Object Oriented  
che Higher Order  
(linguaggio FOOL completo)**

# Funzionalità OO e Higher Order non Mischiate

---

- Le funzionalità Object Oriented ed Higher Order possono essere combinate in uno stesso linguaggio con cambiamenti minimi
  - evitando di assegnare/passare ID di metodi (come invece si fa per le funzioni in Higher Order)
  - cioè no offset doppio per metodi
- Basta combinare le specifiche delle due estensioni descritte in precedenza
  - in quanto riguardano parti diverse del linguaggio
- Unici accorgimenti aggiuntivi da considerare in versione combinata sono descritti nel seguito

# Type Checking

---

- IdNode ID
  - ID può essere di tipo funzionale ma non deve essere un metodo (controllare "isMethod" in STentry); né deve essere il nome di una classe
- IfNode (in versione OO con ottimizzazioni)
  - metodo lowestCommonAncestor di FOOLlib esteso a tipi funzionali come descritto nella slide successiva

# Type Checking

---

metodo "Node lowestCommonAncestor (Node a, Node b)"

- per a e b tipi funzionali con stesso numero di parametri
  - controlla se esiste **lowest common ancestor** dei tipi di ritorno di a e b (si chiama ricorsivamente) e se, per ogni i, i tipi parametro i-esimi sono **uno sottotipo dell'altro** (metodo "isSubtype"):
    - torna **null** se il controllo non ha successo; altrimenti
    - torna **un tipo funzionale** che ha come **tipo di ritorno** il risultato della **chiamata ricorsiva** (covarianza) e come **tipo di parametro** i-esimo il tipo che è **sottotipo** dell'altro (controvarianza)

# Code Generation

---

- CallNode ID()
  - controllo se ID è un metodo ("isMethod" in STentry)
    - se non lo è, ritorno codice di estensione Higher Order
    - se lo è: ritorno codice di estensione Object Oriented