Linguaggi e compilatori

Corso di Laurea in Informatica

Mauro Leoncini

A.A. 2023/2024

1. Generazione di codice intermedio LLVM
   - Cenni su LLVM-IR
   - API LLVM per la generazione della IR
   - Generazione del codice del prototipo
   - Riassunto delle istruzioni usate nella sintesi della IR

Linguaggi e compilatori

Rappresentazione intermedia LLVM

- Per completare il progetto del front-end, dobbiamo ora concentrarci sulla produzione di codice secondo un modello di *rappresentazione intermedia* (*IR*) esistente
- La nostra scelta è ricaduta su IR di *LLVM* (originariamente acronimo di *Low Level Virtual Machine*)
- In realtà LLVM è oggi un insieme di strumenti per la costruzione di compilatori
- Nella seconda parte del corso studierete, in particolare, le ottimizzazioni effettuate dal cosiddetto *middle-end* su IR LLVM
- IR LLVM (da ora in avanti semplicemente IR) è dunque non solo l'output del front-end, ma anche il "testimone" passaggio fra le due parti del corso
- Inizieremo noi a fornirne una descrizione sommaria, che verrà affinata nella seconda parte

Tre differenti forme per la IR di LLVM

- Il codice IR può servire per tre scopi differenti e, di conseguenza, essere presentato in tre diverse forme, tutte equivalenti
  - come rappresentazione intermedia in memoria, manipolata dal compilatore per l'analisi e l'ottimizzazione
  - come rappresentazione bitcode su disco, caricabile per una compilazione Just-In-Time
  - come linguaggio assembly human-readable.
- Nei file su disco, per la versione human-readable si usa l'estensione `.ll` mentre per quella in bit-code si usa `.bc`
- Come brevemente vedremo, il codice IR ha istruzioni assembly ma nel complesso ha una struttura generale relativamente di alto livello

Il modello di calcolo

- Il modello di calcolo di riferimento per IR è una macchina con un numero illimitato di registri (*URM*, *Unlimited Register Machine*).
- A questo sono poi associate due specifiche caratteristiche.
  - I registri sono del tipo *SSA* (*Static Single Assignment*) → Una sola istruzione scrive in quel registro
  - I valori assegnati ad un registro sono tipizzati.
- SSA significa che, "staticamente" (ovvero nel codice scritto), l'assegnamento al registro viene effettuato da una sola istruzione
- Si dice anche che l'istruzione *definisce* il registro
- A tempo di esecuzione (cioè *dinamicamente*), un registro può invece essere riscritto più volte, se l'istruzione che lo definisce viene ri-eseguita

Istruzione di IR LLVM

- IR può essere visto come il set di istruzioni di un'architettura *RISC*
- Ad eccezione delle istruzioni `LOAD` e `STORE`, che operano trasferimenti da e verso la memoria, tutte le altre istruzioni coinvolgono solo registri virtuali
- IR include istruzioni aritmetico/logiche, di salto (condizionato o incondizionato), di aritmetica dei puntatori, di accesso alla memoria e di controllo del flusso

Il grafo di controllo del flusso

- Un programma in IR è composto da uno o più *moduli*, ognuno dei quali è un file (in formato assembly o bitcode, come visto)
- Un modulo è composto dalla definizione di variabili globali e funzioni, a loro volta formate da una serie di "porzioni" di codice dette *blocchi di base* (*Basic Block, BB*)
- Ogni BB inizia con un'etichetta e termina con un'istruzione di salto o una return e non include altre istruzioni che trasferiscono il controllo (con la sola possibilità di chiamate di funzione)
- Ogni blocco può essere visto come nodo di un grafo, detto *grafo di controllo* del flusso di esecuzione (*Control Flow Graph, CFG*), di cui istruzioni di salto ed etichette individuano gli *archi*
- Una specifica istruzione LLVM, detta *istruzione phi*

  `phi tipo [val1, pred1], [val2, pred2], ...`

  consente di "sfruttare" la conoscenza del blocco di provenienza in quanto il suo valore sarà `val`i sse il flusso di esecuzione procede dal blocco `pred`i

Rappresenta un insieme di basic block

Il valore che viene assegnato dipende da dove viene il flusso. Se arriva da x faccio Se arriva di y faccio

Uno sguardo diretto a IR compilando un semplice esempio

- Consideriamo la seguente semplice funzione `fact`, che supponiamo memorizzata nel file `fact.cpp`

```
#include "llvm/IR/LLVMContext.h"

int fact(int n) {
        if (n==0) return 1;
        else return n*fact(n-1);
}
```

Sequenza di istruzioni, chiusa da un istruttore di salto.
Ha un etichettà.
(All'interno si può fare una chiamata di funzione)

- Per compilare il file in IR human-readable format possiamo dare il seguente comando

```
clang++ -S -c -emit-llvm fact.cpp -I/usr/lib/llvm-16/include
```

- N.B. La versione utilizzata `llvm-16` dipende dal sistema utilizzato a fine 2023 su un computer con S.O. Linux Debian 6.1

Il file fact.ll

```
define dso_local noundef i32 @_Z4facti(i32 noundef %0) #0 {
   %2 = alloca i32, align 4
   %3 = alloca i32, align 4
   store i32 %0, i32* %3, align 4
   %4 = load i32, i32* %3, align 4
   %5 = icmp eq i32 %4, 0
   br i1 %5, label %6, label %7

6:                                                ; preds = %1
   store i32 1, i32* %2, align 4
   br label %13
```

Il file fact.ll

```
7:                                                ; preds = %1
  %8 = load i32, i32* %3, align 4
  %9 = load i32, i32* %3, align 4
  %10 = sub nsw i32 %9, 1
  %11 = call noundef i32 @_Z4facti(i32 noundef %10)
  %12 = mul nsw i32 %8, %11
  store i32 %12, i32* %2, align 4
  br label %13
13:                                               ; preds = %7, %6
  %14 = load i32, i32* %2, align 4
  ret i32 %14
}
```

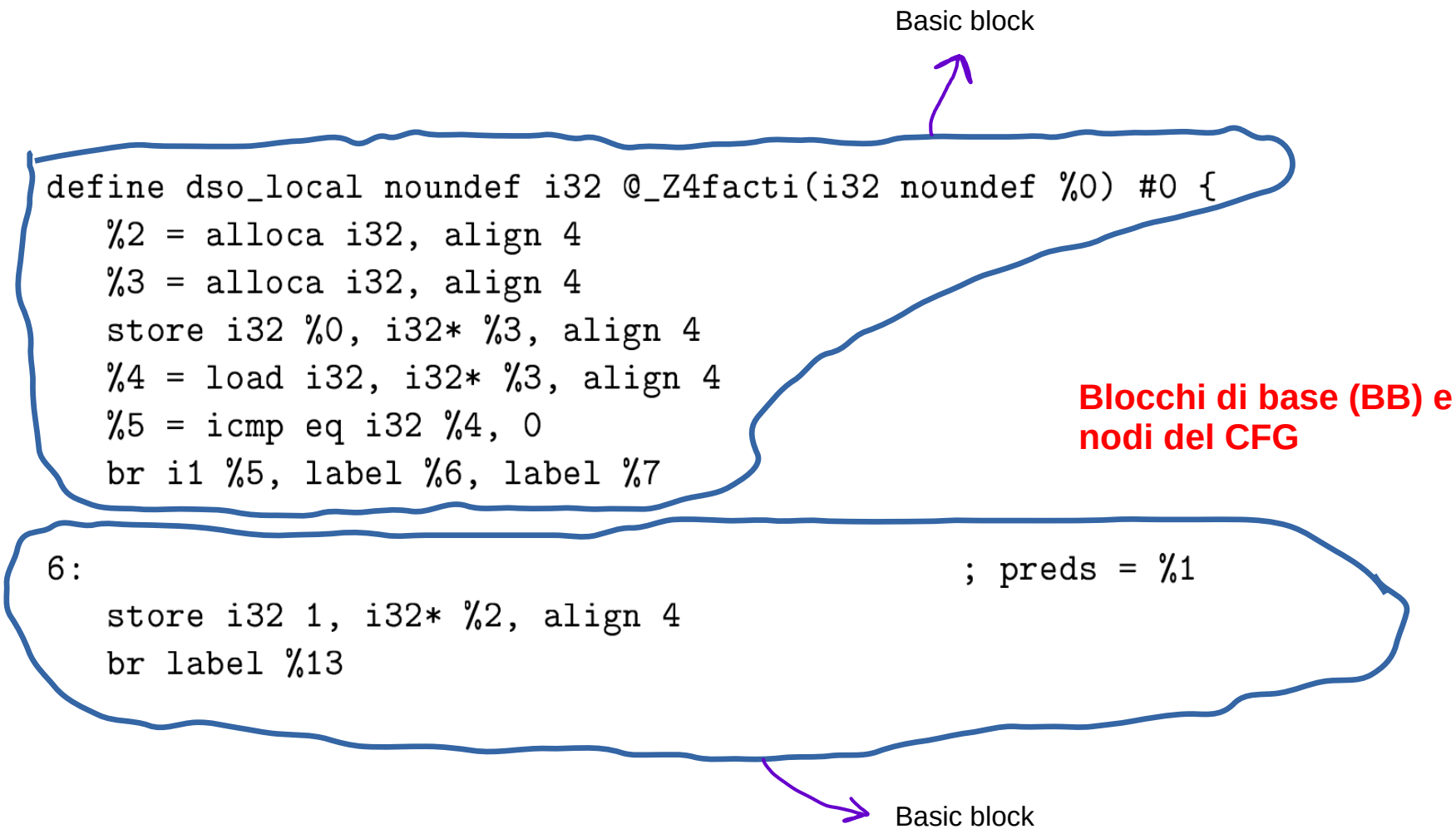
Osservazioni

```
define dso_local noundef i32 @_Z4facti(i32 noundef %0) #0 {
   %2 = alloca i32, align 4
   %3 = alloca i32, align 4
   store i32 %0, i32* %3, align 4
   %4 = load i32, i32* %3, align 4
   %5 = icmp eq i32 %4, 0
   br i1 %5, label %6, label %7

6:                                                ; preds = %1
   store i32 1, i32* %2, align 4
   br label %13
```

Il simbolo @ denota un nome globale (di variabile o funzione)

Osservazioni



Osservazioni

```
define dso_local noundef i32 @_Z4facti(i32 noundef %0) #0 {
   %2 = alloca i32, align 4
   %3 = alloca i32, align 4
   store i32 %0, i32* %3, align 4
   %4 = load i32, i32* %3, align 4
   %5 = icmp eq i32 %4, 0
   br i1 %5, label %6, label %7

6:
   store i32 1, i32* %2, align 4
   br label %13
```

Individua l'arco entrante del CFG

```
; preds = %1
```

Osservazioni

```
7:                                                ; preds = %1
    %8 = load i32, i32* %3, align 4
    %9 = load i32, i32* %3, align 4
    %10 = sub nsw i32 %9, 1
    %11 = call noundef i32 @_Z4facti(i32 noundef %10)
    %12 = mul nsw i32 %8, %11
    store i32 %12, i32* %2, align 4
    br label %13
13:                                               ; preds = %7, %6
    %14 = load i32, i32* %2, align 4
    ret i32 %14
}
```

Etichetta BB

Jump

Etichetta BB

Jump

Due altri BB del programma per il fattoriale

Osservazioni

```
define dso_local noundef i32 @_Z4facti(i32 noundef %0) #0 {
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  %4 = load i32, i32* %3, align 4
  %5 = icmp eq i32 %4, 0
  br i1 %5, label %6, label %7

6:                                                ; preds = %1
  store i32 1, i32* %2, align 4
  br label %13
```

Registri virtuali “anonimi”
La numerazione è automatica da parte di CLANG
Un programmatore può definirli assegnando un nome simbolico
Il simbolo % denota nome locale

Osservazioni

```
define dso_local noundef i32 @_Z4facti(i32 noundef %0) #0 {
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  %4 = load i32, i32* %3, align 4
  %5 = icmp eq i32 %4, 0
  br i1 %5, label %6, label %7

6:                                                ; preds = %1
  store i32 1, i32* %2, align 4
  br label %13
```

Alcune istruzioni assegnano un valore ad un registro virtuale
Si dice che l'istruzione "definisce" il registro
I registri non possono essere definiti più volte
=> un registro non può essere riscritto

Osservazioni

```
define dso_local noundef i32 @_Z4facti(i32 noundef %0) #0 {
   %2 = alloca i32, align 4
   %3 = alloca i32, align 4
   store i32 %0, i32* %3, align 4
   %4 = load i32, i32* %3, align 4
   %5 = icmp eq i32 %4, 0
   br i1 %5, label %6, label %7

6:                                                ; preds = %1
   store i32 1, i32* %2, align 4
   br label %13
```

Registri e operazioni sono tipizzati

Osservazioni

```
define dso_local noundef i32 @_Z4facti(i32 noundef %0) #0 {
   %2 = alloca i32, align 4
   %3 = alloca i32, align 4
   store i32 %0, i32* %3, align 4
   %4 = load i32, i32* %3, align 4
   %5 = icmp eq i32 %4, 0
   br i1 %5, label %6, label %7

6:                                                ; preds = %1
   store i32 1, i32* %2, align 4
   br label %13
```

L'istruzione alloca riserva spazio sullo stack e restituisce un puntatore
A differenza dei registri, la memoria è ovviamente riscrivibile

Osservazioni

```
7:                                                ; preds = %1
  %8 = load i32, i32* %3, align 4
  %9 = load i32, i32* %3, align 4
  %10 = sub nsw i32 %9, 1
  %11 = call noundef i32 @_Z4facti(i32 noundef %10)
  %12 = mul nsw i32 %8, %11
  store i32 %12, i32* %2, align 4
  br label %13
13:                                               ; preds = %7, %6
  %14 = load i32, i32* %2, align 4
  ret i32 %14
}
```

Attributi/flag di dati e operazioni
(NSW = No Signed Wrap)

Osservazioni

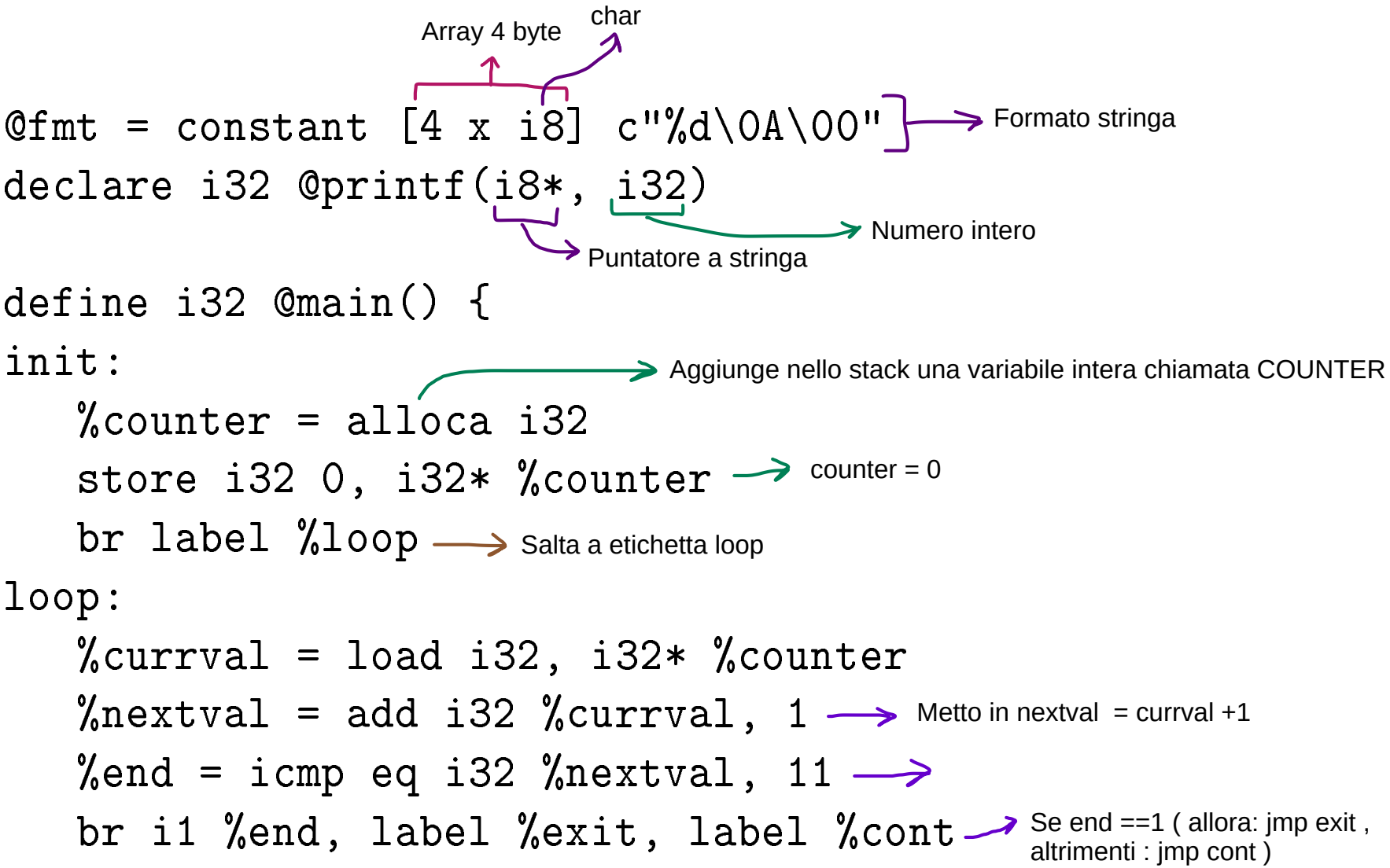
```
7:                                                ; preds = %1
  %8 = load i32, i32* %3, align 4
  %9 = load i32, i32* %3, align 4
  %10 = sub nsw i32 %9, 1
  %11 = call noundef i32 @_Z4facti(i32 noundef %10)
  %12 = mul nsw i32 %8, %11
  store i32 %12, i32* %2, align 4
  br label %13
13:                                               ; preds = %7, %6
  %14 = load i32, i32* %2, align 4
  ret i32 %14
}
```

Si può notare come la chiamata di funzione sia un costrutto di alto livello rispetto ad un linguaggio assembly

Osservazioni sull'esempio: i moduli LLVM

- Il "programma" che stiamo analizzando è composto da più blocchi di base ma da un solo *modulo*
- Un generico programma può però essere composto da più moduli che, compilati separatamente, vengono poi uniti nel processo di *linking*
- Il nostro modulo non è ovviamente eseguibile e deve essere "linkato" ad un modulo dotato di `main` program
- Un modulo contiene tutti i dati che compongono un singolo file di programma
- In generale un modulo contiene dichiariazioni di *variabili globali*, *funzioni* (dichiarazioni per esterne, dichiarazioni e definizioni per interne), *dichiarazioni di tipo*, ...
- Il programma ella seguente slide può risultare utile a chi sia interessato a fare qualche semplice prova

Programma che stampa i numeri da 1 a 10



Programma che stampa i numeri da 1 a 10

```
cont:
    store i32 %nextval, i32* %counter
    call i32(i8*, i32) @printf(i8* getelementptr([4 x i8],
              [4 x i8]* @fmt, i32 0, i32 0), i32 %nextval)
    br label %loop
exit:
    ret i32 0
}
```

counter = nextval

Return 0

Compilazione ed esecuzione

- Il precedente programma (che chiameremo `easyprint.ll`) può essere eseguito in (almeno) due modi differenti
- Il modo più veloce usa l'interprete e compilatore JIT `lli`

```
> lli easyprint.ll
```

- Il secondo modo rende esplicito il processo di compilazione

```
# Da formato testuale human readable a bitcode
> llvm-as easyprint.ll
# Da bitcode ad assembler nativo (produce .s)
> llc easyprint.bc
```

(oppure direttamente: `llc easyprint.ll`)

```
# Generazione del codice/linking/eseguibile
> clang -o easyprint easyprint.s
> ./easyprint
```

Linguaggi e compilatori

L'infrastruttura LLVM

- L'infrastruttura LLVM è scritta in linguaggio C++
- Essa include diverse classi che rendono la generazione della IR un procedimento "sufficientemente" agevole
- La buona notizia è che non è necessario conoscere i dettagli delle istruzioni né avere una chiara "mappa" della loro collocazione fisica nel programma (nel suo insieme)
- In questa seconda parte discuteremo le classi che utilizzeremo nel nostro front-end:
  - `LLVMContext`
  - `Module`
  - `Function`
  - `BasicBlock`
  - `IRBuilder`
  - `Value`

La classe `LLVMContext`

- Viene definita *opaca* perché, nonostante la classe includa strutture dati cruciali per il funzionamento di LLVM, non è in generale necessario avere di essa alcuna conoscenza in ordine alla generazione di IR
- Per i nostri scopi sarà sufficiente creare una singola istanza della classe e "passare" il relativo puntatore alle funzioni che ne fanno uso
- Il contesto è specialmente importante quando si tratta di definire il *tipo* di una variabile
- Si tenga presente che IR è indipendente dal linguaggio sorgente e dunque non possiede un insieme di tipi predefiniti, ad esempio non possiede interi di lunghezze prefissate
- Se viene richiesta l'allocazione di, poniamo, variabili intere di 8, 16 e 96 bit, il contesto ne tiene traccia per evitare duplicazioni nel caso di altre richieste di allocazioni simili

La classe `Module`

- La classe `Module` rappresenta il modulo, come lo abbiamo definito in precedenza, ovvero un insieme di definizioni di variabili globali e di funzioni
- Anche della classe `Module` sarà sufficiente disporre di una sola istanza per ogni file sorgente
- Chiaramente il modulo espone metodi per l'accesso ai componenti
- Un metodo della classe che useremo è `getFunction` che, dato il nome di una funzione, restituisce un puntatore all'oggetto che la rappresenta
- Nell'istanza di `Module` si potrebbero poi inserire le informazioni relative all'architettura, che però noi ignoriamo perché ci fermiamo alla sola produzione dell'IR in formato human-readable

La classe `Function`

- Gli oggetti della classe `Function` sono chiaramente la controparte in IR delle funzioni definite nel programma sorgente
- I metodi della classe che utilizzeremo sono:
  - `getEntryBlock()`, che restituisce un puntatore al BB iniziale della funzione
  - `insert()`, che inserisce un BB in un determinato punto della funzione
  - `end()`, da utilizzare tipicamente con `insert()`, che indica la (attuale) fine del body della funzione
  - `eraseFromParent()`, che cancella una definizione (parziale) non completata con successo

La classe `BasicBlock`

- Gli oggetti della classe `BasicBlock` (BB) sono gli effettivi "contenitori" del codice
- Un BB è dunque sempre un argomento nei metodi che generano IR
- Ad ogni oggetto BB può essere (ed in genere è) attribuita un'etichetta, da utilizzare come riferimento per istruzioni di salto
- Come abbiamo già sottolineato, i BB sono i nodi del CFG e giocano un ruolo fondamentale nella pianificazione della sintesi di codice
- Si pensi al semplice caso di un'istruzione condizionale a due vie
- In tal caso, il generatore del codice dovrà creare (nel BB corrente) le istruzioni per eseguire il test e dovrà poi generare tre BB per le due possibili vie del condizionale e per l'inserimento delle istruzioni successive (dove il flusso si "riunisce")

La classe `IRBuilder`

- Un unico oggetto della classe è sufficiente, il cui scopo è tenere traccia del punto di inserzione del codice e di generare, attraverso i suoi numerosi metodi, tutte le istruzione del set
- I metodi che utilizzeremo noi sono:
  - `CreateLoad()`, `CreateStore`, istruzioni load e store
  - `CreateFNeg()`, meno unario
  - `CreateFAdd()`, `CreateFSub()`, `...`, operazioni aritmetiche
  - `CreateFCmpULE()`, `CreateFCmpULT()`, , confronti fra numeri float
  - `CreateCall()`, chiamata di funzione
  - `CreateBr()`, `CreateCondBr()`, salto incondizionato e condizionato
  - `CreatePHI()`, istruzione `phi`
  - `CreateRet()`, istruzione `return`
  - `SetInsertPoint()`, stabilisce il punto di inserimento delle istruzioni
  - `GetInsertBlock()`, restituisce il blocco dove vengono correntemente inserite le istruzioni

La classe `Value`

- `Value` è la classe base per qualsiasi valore calcolato dal programma
- `Function` e `BasicBlock`, ad esempio, sono sottoclassi di `Value`
- I metodi che generano codice per ogni costrutto del linguaggio restituiscono un puntatore ad un oggetto della classe `Value`
- Si può pensare che tale oggetto denoti il *registro virtuale* dove verrà memorizzato il risultato della computazione rappresentata da "quel" codice

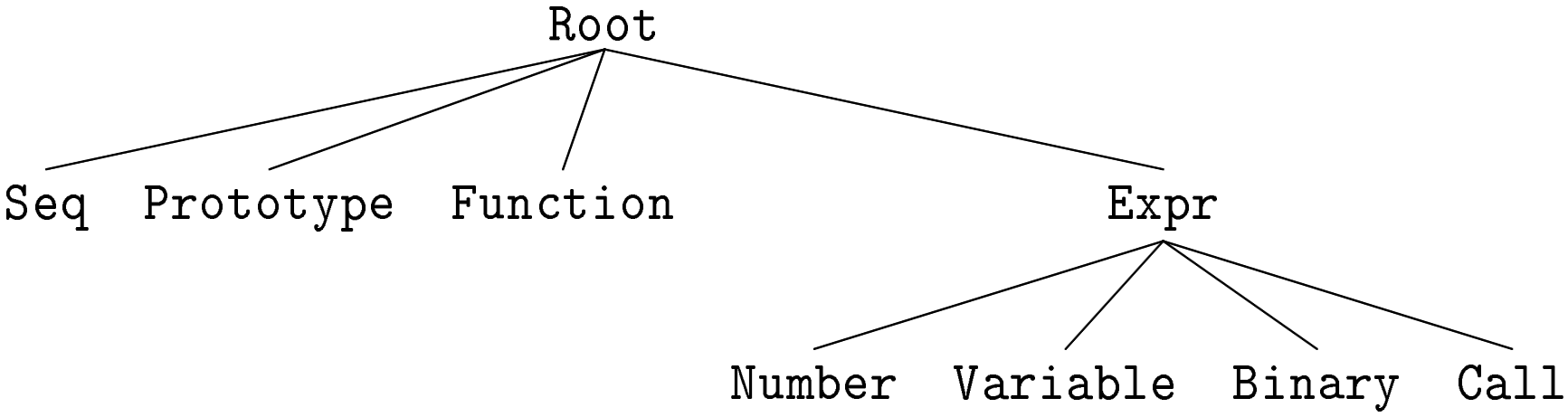
Linguaggi e compilatori

Il piano generale di generazione del codice

- Inizialmente creiamo un unico contesto (cioè un oggetto della classe LLVMContext) per il coordinamento di tutto il processo di generazione
- In modo esplicito il contesto verrà utilizzato solo nella generazione dei tipi e delle costanti, oltre che in quella dei `BuildingBlock`
- La generazione vera e propria viene poi eseguita con l'ausilio dei metodi di `IRBuilder`
- Un oggetto di questa classe può essere utilmente pensato come un *file pointer* il cui stato include il punto attuale di lettura/scrittura nel file
- La classe contiene quindi l'equivalente di metodi per determinare il punto di scrittura (in realtà comunque sequenziale ma con la possibilità di iniziare nuovi BuildingBlock o Funzioni)
- Come abbiamo già notato, la classe include poi metodi per la generazione di istruzioni nel punto di inserimento

AST e generazione del codice

- Dispondendo esplicitamente dell'AST, la fase di generazione del codice procede eseguendo una visita dell'AST; visita che viene "guidata" dai singoli costrutti rappresentati dai nodi dell'albero stesso
- Nel seguito, consideriamo i nodi dell'albero prodotto dal parser del codice prototipo e illustriamo le corrispondenti operazioni di costruzione della IR
- Per comodità, riportiamo la gerarchia delle classi cui possono appartenere i nodi dell'AST (relativi a Kaleidoscope 1.0)



Espressioni Top-Level

- Il trattamento delle espressioni nella generazione dell'IR LLVM richiede una spiegazione preliminare
- L'idea è che un'espressione sia trattata come funzione anonima
- Ad esempio, l'espressione $3 \times (5 + 2)$ può divenire la funzione (scritta in style `C/C++`)

$$\texttt{anon\_fun()\{return } 3 \times (5 + 2); \}$$

- Si noti, di passaggio, che un'espressione che contiene variabili può comparire solo all'interno di una funzione (non anonima) e se corrisponde ad un parametro formale
- Questo non a motivo di una non corretta sintassi: al di fuori di una funzione, infatti, una variabile non può avere valore perché non esistono (ancora) istruzioni di assegnamento e dunque l'eventuale variabile risulta certamente indefinita

Espressioni Top-Level

- Un'espressione è in generale composta da sotto-espressioni
- Ad esempio, l'espressione $3 \times (5 + 2)$ ha 3 e $5 + 2$ come sotto-espressioni e qust'ultima, a sua volta, ha sotto-espressioni 5 e 2
- Se ogni espressione venisse trasformata in funzione anonima la scrittura $3 \times (5 + 2)$ darebbe dunque origine a ben 5 tali funzioni e chiaramente non è questo l'obiettivo
- L'idea è di "rivestire" come funzioni solo le espressioni *top-level*
- Con riferimento alla grammatica, le espressioni top-level sono quelle che compaiono nelle due seguenti produzioni

```
top : expr
definition : "def" proto exp
```

- Nel secondo caso, l'espressione è il body di una funzione e dunque non deve dare origine ad una funzione anonima
- Rimangono solo le espressioni riconducibili alla prima produzione

Costanti e variabili

- In corrispondenza del nodo che rappresenta una costante numerica (classe `NumberExprAST`) non viene generato codice
- La chiamata `codegen()` genera direttamente una costante, la inserisce nel contesto e ne restituisce un riferimento mediante l'istruzione

  ```
  return ConstantFP::get(context,APFloat(Val))
  ```

  dove `Val` è il valore presente nel nodo dell'AST
- Neppure in corrispondenza del nodo che rappresenta una variabile (classe `VariableExprAST`) viene generato codice.
- In questo caso la `codegen()` restituisce il valore associato alla variabile nella *symbol table* (`NamedValue`):

  ```
  return NamedValue[Name]
  ```

  dove `Name` è il valore presente nel nodo dell'AST
- Si ricordi che qui il "valore" è un `Value` (dunque un registro SSA)

Espressioni binarie

- I nodi per le espressioni binarie sono istanze di `BinaryExprAST`
- Il metodo `codegen` di questa classe procede in due passi: dapprima invoca ricorsivamente `codegen` sui nodi figli

```
Value *L = LHS->codegen();
Value *R = RHS->codegen();
```

- Il risultato è di ciascuna chiamata è il riferimento ad un `Value`
- In concreto, `L` ed `R` andranno a memorizzare il riferimento ai registri SSA che conterranno i risultati delle due sottoespressioni
- A seconda del particolare operatore, `codegen` invoca quindi il builder per generare il codice ad esso corrispondente e memorizzare il risultato in un differente registro.Nel caso di addizione:

```
return Builder.CreateFAdd(L, R, "addregister");
```

dove `addregister` è il nome simbolico che verrà attribuito (con un eventuale indice incrementale) al registro SSA di destinazione

Chiamata di funzione

- Anche il metodo codegen della classe `CallExprAST` (di cui i nodi chiamata sono istanze) è di relativamente facile comprensione
- Dapprima viene interrogato il modulo corrente per recuperare l'oggetto di tipo `Function` (sottoclasse di `Value`, come già anticipato) che rappresenta la funzione

```
Function *CalleeF = module->getFunction(Callee);
```

dove `Callee` è il nome della funzione nel nodo AST

- Dopo questo codegen controlla che il numero di argomenti (le espressioni rappresentate dai nodi figli nell'AST) coincida con il numero di parametri specificati nella definizione di funzione (recuperabili interrogando `CalleeF`)

```
if (CalleeF->arg_size() != Args.size())
   return LogErrorV("Incorrect # arguments passed");
```

Chiamata di funzione

- Se il controllo ha esito positivo, il processo continua con la generazione del codice per il calcolo degli argomenti

```
std::vector<Value*> ArgsV;
for (unsigned i = 0, e = Args.size(); i != e; ++i){
    ArgsV.push_back(Args[i]->codegen());
```

- Le ripetute chiamate al metodo codegen dei singoli argomenti, oltre a generare il codice per il loro calcolo (e potrebbero essere espressioni complesse) restituiscono oggetti `Value`, ovvero (una rappresentazione de) i registri dove saranno memorizzati i risultati di tali calcoli
- Una volta completato questo processo, codegen invoca il builder per la generazione del codice che effettua la chiamata

```
return Builder.CreateCall(CalleeF, ArgsV, "callregister");
```

Definizione di funzione

- Il metodo codegen `FunctionAST` (di cui i nodi funzione dell'AST sono istanze) è composto da diversi "passi"
- Il primo consiste nell'interrogazione del modulo corrente per verificare che la funzione non sia già definita

  ```
  Function *function = module->getFunction(Proto->getName());
  ```

  dove `Proto` è l'attributo presente nel nodo dell'AST
- Se la funzione esiste già il metodo "esce", altrimenti il secondo passo è la generazione del prototipo (che discuteremo subito dopo)

  ```
  function = Proto->codegen();
  ```
- Il terzo passo consiste nell'allocazione di un blocco base dove andrà inserito il codice della funzione

  ```
  BasicBlock *BB =
      BasicBlock::Create(context, "entry", function);
  Builder.SetInsertPoint(BB);
  ```

Definizione di funzione

- Il passo successivo consiste nell'inserire i nomi dei parametri nella symbol table, dove potranno essere acceduti dalle istruzioni del body

```
NamedValues.clear();
for (auto &Arg : function->args())
    NamedValues[Arg.getName()] = &Arg;
```

  Si noti che i parametri (`function->args()`) sono stati inseriti nell'oggetto `function` durante la generazione del prototipo (il secondo passo di codegen)
- Si rifletta poi sul (e si trovi una giustificazione per) il fatto che il contenuto della symbol table viene preliminarmente cancellato

Definizione di funzione

- L'ultimo passo (salvo ulteriori verifiche) consiste nella generazione del codice per il body della funzione e l'inserimento nel blocco di una istruzione `ret`
- Quest'ultima viene utilizzata per restituire il flusso di controllo (e opzionalmente un valore) da una funzione al chiamante.

```
if (Value *RetVal = Body->codegen())
   Builder.CreateRet(RetVal);
```

Prototipo di funzione

- Le istanze della classe `PrototypeAST` includono un *nome* (`Name`) e un vettore di parametri (`Args`), ognuno di un determinato tipo
- La prima azione di codegen consiste nel create un c.d. *function type*, che include il tipo del risultato e un vettore con i tipi dei parametri
- Poiché in Kaleidoscope l'unico tipo è `double`, con l'istruzione

```
std::vector<Type*> Doubles(Args.size(),
                           Type::getDoubleTy(TheContext));
```

  creiamo il vettore `Doubles` i cui elementi, tanti quanti gli argomenti del prototipo, sono tutti di tipo `double` (nella rappresentazione LLVM)
- Con la successiva istruzione creiamo (o recuperiamo) un tipo costituito da un `double` e da un vettore di `double`

```
FunctionType *FT =
FunctionType::get(Type::getDoubleTy(TheContext),
                  Doubles, false);
```

Prototipo di funzione

- La funzione, il cui tipo è appunto il nuovo tipo creato, viene inserita nel modulo corrente

```
Function *F = Function::Create(FT, Function::
                                ExternalLinkage, Name, *module);
```

- `ExternalLinkage` indica che la funzione può essere definita (o essere richiamata) al di fuori del modulo corrente
- L'ultimo passaggio consiste nell'attribuzione ai parametri dei nomi specificati dal programmatore

```
unsigned Idx = 0;
for (auto &Arg : F->args())
    Arg.setName(Args[Idx++]);
```

- Si noti bene la struttura del ciclo `for`: l'iterazione è sulle posizioni del vettore di argomenti nella funzione LLV appena definita e vi inserisce i nomi presenti nell'AST

Linguaggi e compilatori

Classe Module e operazioni principali

- Un modulo è un contenitore per tutti gli altri oggetti IR di LLVM.
- Memorizza *variabili globali*, *funzioni* e *librerie* (e anche altri moduli) da cui esso dipende
- `Module::getFunction(<Function name>)`
  - Supposto che `module` sia un oggetto della classe `Module`, la scrittura
    `module.getFunction(Callee)`
    recupera la funzione (oggetto della classe `Function`) di nome `callee`, o restituisce `nullptr` se questa non è presente
- `Function::Create(<params>)`
  - Se `module` è un modulo e `function` un oggetto della classe `Function`, la scrittura
    `Function::Create(type, Function::ExternalLinkage, Name, module)`
    crea e inserisce in `module` una funzione di tipo `type`, nome `name` e con visibilità anche all'esterno del modulo stesso (`ExternalLinkage`)

Classe IRBuilder

- IRBuilder è un *class template*: scrivendo IRBuilder<> implicitamente si dichiara di usare i valori default per istanziare la classe
- La classe viene usata per creare istruzioni e inserirle in un BasicBlock
- IRBuilder::SetInsertPoint(<Basic Block>)
- NOTA: in questa e nella slide successiva supponiamo che builder sia un oggetto della classe IRBuilder, ragione per cui useremo la notazione builder.<metodo> anziché builder-><metodo> (da usare quando builder è un puntatore ad un oggetto IRBuilder)
  - builder.SetInsertPoint(BB)
    dove BB è un blocco di base (BasicBlock). Specifica al builder che le istruzioni che verranno create dovranno essere "appese" al blocco specificato.

Classe IRBuilder

- `IRBuilder::GetInsertBlock()`
  - `builder.GetInsertBlock()`
    restituisce il riferimento al blocco attualmente utilizzato dal builder per l'inserimento di instruzioni.
  - `builder.GetInsertBlock()->getParent()`
    È sempre la stessa funzione ma evidenziamo qui uno dei modi più comuni in cui viene utilizzata, e cioè per recuperare il riferimento alla funzione (oggetto delle classe `Function`) in cui il blocco è inserito. Al riguardo, ricordiamo che può trattarsi anche di dunzione *anonima.*

Istruzione Create di IRBuilder

- `IRBuilder::Create<Istruzione_IR>(<params>)`
- Fra i parametri può essere presente una stringa con cui il programmatore desidera sia identificato il resistro SSA in cui viene lasciato il risultato dell'operazione
  - `builder.CreateFAdd(LHS,RHS,"addregister")`
    dove `LHS` e `RHS` sono (puntatori a) oggetti di tipo `Value`, ovvero a loro volta rappresentazioni di registri SSA introdotti come target di precedenti istruzioni. Genera un'istruzione di addizione float.
  - `builder.CreateCall(CalleeF, ArgsV, "calltmp")`
    dove `CalleeF` è una funzione definita nel modulo corrente (o dichiarata `extern`) e `ArgsV` è un vettore di puntatori a oggetti `Value`. Genera una chiamata alla funzione con nemorizzazione del risultato in un registro SSA il cui nome è `calltmp`.
  - `builder.CreateRet(RetVal);`
    dove `RetVal` è il puntatore ad un oggetto `Value` che denota un registro SSA assegnato in una precedente istruzione

Istruzione Create di IRBuilder

- `builder.CreateBr(Label)`
  dove `Label` è l'etichetta di un blocco. Genera un'istruzione di salto alla prima istruzione del blocco con quell'etichetta.
- `builder.CreateCondBr(Cond, Label1, Label2)`
  dove `Cond` è un `Value` interpretato come valore logico e `Label1` e `Label2` etichette di altrettanti blocchi. Genera un'istruzione di salto condizionato. Il target è il blocco etichettato `Label1` se `Cond` valuta `true`, altrimenti il target è `Label2`.
- `builder.CreatePHI(Type::getDoubleTy(context),2,"x");`
  In questo caso crea un'istruzione *PHI* il cui target è la variabile `"x"`, che potrà assumere due valori, a seconda di qual è il blocco da cui il flusso di esecuzione proviene (cioè entra nel blocco che includie l'istruzione *PHI*)

Istruzione Create per i confronti

- `IRBuilder::CreateICmp<cond1>(<params>)` e `IRBuilder::CreateFCmp<cond2>(<params>)`
  - Le due funzioni si differenziano per il tipo dei parametri (interi o float)
  - Le singole istruzioni differiscono poi per la condizione
  - Valori possibili per `<cond1>` sono, ad esempio: `eq` e `neq` (dal significato ovvio) e poi `ult`, `ule`, `slt` e `sle`, dove `u` e `s` indicano che i termini a confronto sono interpretati senza senso o con segno , mentre `lt`, `le` indicano, al solito, minore stretto e minore o uguale.
  - Valori possibili per `<cond2>` sono, ad esempio: `oeq`, `one`, `olt` e `ueq`, `une`, `ult`, dove `o` e `u` indicano qui `ordered` e `unordered`
  - `ordered` significa che nessuno dei due termini a confronto è considerato un `QNaN`. Se uno o entrambi gli operandi sono `QNaN` il risultato è `false` .
  - `unordered`, al contrario, implica che uno o entrambi gli operandi possono essere `QNaN`. Se uno o entrambi gli operando sono `QNaN` il risultato è `true`

Classe LLVMContext

- Come già anticipato, è una classe fondamentale ma "opaca".
- Testualmemte, dalla documentazione ufficiale LLVM: "(LLVMContext) owns and manages the core global data of LLVM's core infrastructure, including the type and constant uniquing tables"
- Deve essere utilizzata in opportune istruzioni, alcune delle quali elencate di seguito, dove supponiamo che `context` indichi un oggetto della classe `LLVMContext<>`
- `Type::getDoubleTy(context)`
  restituisce (il riferimento un oggetto che rappresenta) il tipo di dati `DoubleTyID`, ovvero (almeno attualmente), floating-point a 64 bit
- `BasicBlock::Create(context, "entry", function)`
  restituisce (un riferimento a) un `BasicBlock`, con label `entry`, inserito all'interno della funzione `function`

Classe LLVMContext

- `ConstantFP::get(context, APFloat(Val))`
  crea e restituisce la rappresentazione di una costante (comunque un oggetto di una classe che è sottoclasse di `Value`) floating point corrispondente al valore `val`
- Si noti che `val` è una variabile C++ mentre `APFloat` è una funzione che fornisce la rappresentazione di `val` secondo LLVM
- `APFloat` sta per *Arbitrary Precision Float* ma questo vuol semplicemente dire che tutta la precisione secondo la rappresentazione C++ viene preservata in LLVM
- Il contesto deve inoltre essere specificato quando viene creato un modulo o un builder
  `Module(ID, context)`
  `IRBuilder(context)`
  dove `ID` è una stringa che identifica il modulo