Compilatori

Corso di Laurea in Informatica

Mauro Leoncini

A.A. 2024/2025

1. Generazione di codice intermedio LLVM
   - Generazione del codice del prototipo

Compilatori

Il piano generale di generazione del codice

- Al punto in cui siamo arrivati nella realizzazione del front-end, possiamo "abbandonare" ogni attenzione a scanner e parser,
- La generazione del codice procede infatti mediante una visita dell'AST (o meglio, della *ASF*, *Abstract Syntax Forest*) già generato/a.
- Questo significa che andremo a lavorare solo sul modulo driver(con piccole modifiche al main program)
- Nel driver (include file) é necessario includere alcuni moduli di `llvm`

```
#include "llvm/IR/BasicBlock.h"
#include "llvm/IR/Constants.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/Instructions.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Type.h"
#include "llvm/IR/Verifier.h"
```

Il piano generale di generazione del codice

- Nel driver vero e proprio dobbiamo poi istanziare le classi più importanti per la generazione (si veda, al riguardo, il precedente set di slide su *LLVM Basics*)
- Si tratta della classe `LLVMContext` e delle classi `Module` e `IRBuilder`

```
LLVMContext *context = new LLVMContext;
Module *module = new Module("LFMCompiler", *context);
IRBuilder<> *builder = new IRBuilder(*context);
```

- Un oggetto della classe `IRBuilder` può essere pensato come un *file pointer* il cui stato include il punto attuale di scrittura nel file
- La classe contiene quindi, oltre ai metodi per generare specifiche istruzioni, anche l'equivalente di metodi per determinare il punto di scrittura

Il piano generale di generazione del codice

- Nella definizione della classe `driver` (presente nel file `driver.hpp`) deve poi essere inclusa la definizione della *symbol table*
- Si tratta di una *tabella associativa* (una `map` nella *stl* di C++) che mette in corrispondenza identificatori e "istruzioni di allocazione"
- Si ricordi, al riguardo, che nel modello SSA, un'istruzione che scrive in un registro è univocamente associata al registro stesso

```
class driver {
public:
   driver();   // Costruttore
   ...
   std::map<std::string, AllocaInst*> NamedValues;
   ...
}
```

Il piano generale di generazione del codice

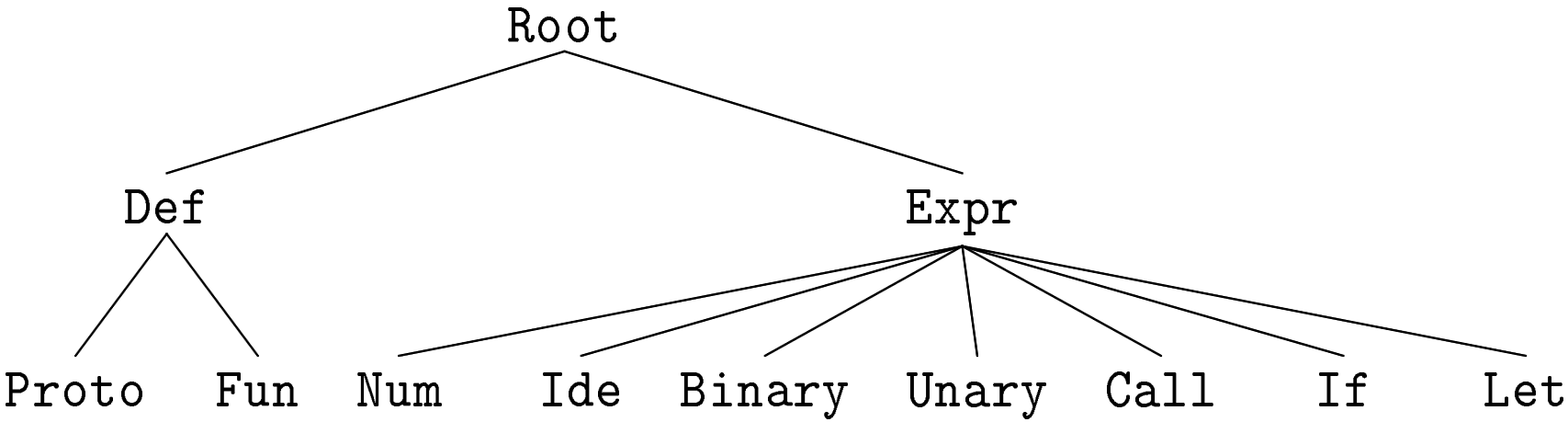
- Ci sono due momenti in cui il compilatore scrive nella symbol table
  - all'inizio della generazione di una funzione
  - nella generazione dei "binding" di un blocco `let`

  In entrambi i casi, all'identificatore si associa il registro in cui verrà scritto (a run-time) il valore corrispondente (argomento, nel caso di funzioni, o right-end side dell'assegnamento, in quello di `let`)
- Per l'allocazione, useremo la seguente funzione di utilità

```
static AllocaInst *MakeAlloca(Function *fun,
       StringRef ide,
       Type* T = IntegerType::get(*context,32)) {
  IRBuilder<> TmpB(&fun->getEntryBlock(),
                   fun->getEntryBlock().begin());
  return TmpB.CreateAlloca(T, nullptr, ide);
```

AST e generazione del codice

- Ricordiamo nuovamente le classi definite nel driver con la loro struttura gerarchica



- A queste chiaramente si aggiunge poi la speciale classe `driver`
- Per la generazione del codice, alla classe `RootAST` viene aggiunto il metodo virtuale codegen, che viene poi diversamente implementato nelle classi "foglia" della sopra-riportata gerarchia
- Il metodo vene anche definito e implementato nella classe `driver`

AST e generazione del codice

- L'implementazione di codegen nella classe driver è semplice
- Esso deve solo dare "il via" alla visita, con generazione, di tutti gli AST generati durante il parsing

```
void driver::codegen() {
   // Il metodo codegen effettua "semplici" chiamate
   // al metodo omonimo presente nei nodi radice
   // degli AST generati dal parser.
   fprintf(stderr,
           "target triple=\"x86_64-pc-linux-gnu\"\n");
      for (DefAST* tree: root) {
         tree->codegen(*this);
      }
};
```

AST e generazione del codice

- Durante visita in post-ordine di un AST viene eseguito il metodo codegen presente in ogni nodo
- Nel seguito, descriviamo per linee essenziali l'implementazione del metodo per tutte le classi foglia della gerarchia
- Iniziamo con le espressioni costanti (i numeri) e gli identificatori
- 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

```
Constant *NumberExprAST::codegen(driver& drv) {
 return ConstantInt::get(*context, APInt(32,Val));
};
```

dove `Val` è il valore presente nel nodo dell'AST

Costanti e variabili

- Nel caso di identificatore (classe `IdeExprAST`), il metodo `codegen()` restituisce il valore presente nella symbol table ad esso associato:

```
Value *IdeExprAST::codegen(driver& drv) {
 AllocaInst *L = drv.NamedValues[Name];
 if (L) {
  Type *type = L->getAllocatedType();
  return builder->CreateLoad(type, L, Name);
 };
 return LogErrorV("Variabile "+Name+" non definita");
};
```

  dove `Name` è il valore presente nel nodo dell'AST
- Si ricordi che qui il "valore" è un `Value` (dunque un registro SSA)
- Il controllo della presenza dell'identificatore nella symbol table rientra nell'*analisi semantica*

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 (auto arg : Args) {
 ArgsV.push_back(arg->codegen(drv));
}
```

- 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, "callreg");
```

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

```
for (auto &Arg : function->args()) {
   AllocaInst *Alloca = MakeAlloca(function,
                                   Arg.getName());
   builder->CreateStore(&Arg, Alloca);
       drv.NamedValues[std::string(Arg.getName())] =
           Alloca;
}
```

Si noti che i parametri (`function->args()`) sono stati inseriti nell'oggetto `function` durante la generazione del prototipo (il secondo passo di codegen)

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 LFM l'unico tipo è `int32`, con l'istruzione

```
std::vector<Type*> intarray(Params.size(),
                  IntegerType::get(*context,32));
```

  creiamo il vettore `intarray` i cui elementi, tanti quanti gli argomenti del prototipo, sono tutti di tipo `int32` (nella rappresentazione LLVM)
- Con la successiva istruzione creiamo (o recuperiamo) un tipo costituito da un `int32` e da un vettore di `int32`

```
FunctionType *FT =
   FunctionType::get(IntegerType::get(*context,32),
                     intarray, 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(Params[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