Compilatori

Corso di Laurea in Informatica

Mauro Leoncini

A.A. 2024/2025

1. Generazione dell'AST
   - Architettura del front-end
   - Verso l'ASD di LFM

Compilatori

Organizzazione modulare

- L'obiettivo è la progettazione e poi l'implementazione di un front-end per il Linguaggio Funzionale Minimo
- La versione 1.0 del front-end produce solo l'*Abstract Syntax Tree* (AST) del programma in input
- In queste slide descriviamo la struttura dell'applicazione
- Potremmo scrivere tutto in due sole unità di programma: lo scanner e il parser, includendo in quest'ultimo un *main program* che coordini tutte le attività
- La soluzione scelta è più *modulare*, manutenibile e "debuggabile"
- La stessa architettura modulare costituisce da sola una migliore documentazione dell'applicazione

I moduli dell'applicazione

- L'applicazione consta dunque dei seguenti moduli
- Un modulo *scanner* per la lettura del file e l'analisi lessicale, generato utilizzando `Flex`
- Un modulo per il *parsing* e la costruzione dell'AST, generato utilizzando `Bison`
- Un modulo *driver*, che include tutte le classi C++ di cui i nodi dell'AST costituiscono istanze
- Il modulo driver definisce anche una classe (omonima) che rappresenta il c.d. *parsing context*, ovvero una struttura dati che consente un'efficace condivisione di dati fra parser e scanner (e main program) prescindendo dall'uso di variabili globali
- Un *client* che include il *main program* e il cui compito è di effettuare il parsing dei parametri forniti dall'utilizzatore nella riga di comandi e di "lanciare" la compilazione

Il programma client (1)

```
#include <iostream>
#include "driver.hpp"
int main (int argc, char *argv[]) {
   driver drv;
   bool verbose = false;
   for (int i = 1; i < argc; ++i)
      if (argv[i] == std::string ("-p"))
         drv.trace_parsing = true;
      else if (argv[i] == std::string ("-s"))
         drv.trace_scanning = true;
      else if (argv[i] == std::string ("-v"))
         verbose = true;
      else if (!drv.parse (argv[i])) {
         std::cout << "Parse successful\n";
         if (verbose) drv.root->visit();
      } else return 1;
   return 0;
}
```

Il client e l'uso della classe `driver`

- Come si vede, l'architettura modulare consente di avere un client molto semplice
- A parte le ovvie operazioni di parsing della linea di comando, dal codice presentato si può iniziare a comprendere l'uso della classe `driver`
  - In due opportune variabili della classe, `trace_scanning` e `trace_parsing`, il driver inserisce informazioni che saranno utlizzate dallo scanner e (indirettamente) dal parser
  - Un metodo della classe (`parse`) consente di dare inizio al processo di compilazione
  - A fine compilazione, la variabile `root` "contiene" la radice dell'AST, da cui deve iniziare il processo di visita
- Le prossime due slide riportano la definizione della classe `driver`; quelle immediatamente seguenti mostrano invece "graficamente" l'architettura generale

La classe driver: definizione

```
class driver {
public:
   driver();
   void scan_begin();      // Implementata nello scanner
   void scan_end();        // Implementata nello scanner
   int parse (const std::string& f);

   RootAST* root;          // Radice dell'AST costruito
   yy::location location; //
   std::string file;       // File sorgente

   bool trace_parsing;     // Per trace debug nel parser
   bool trace_scanning;    // Per trace debug nello scanner
};
```

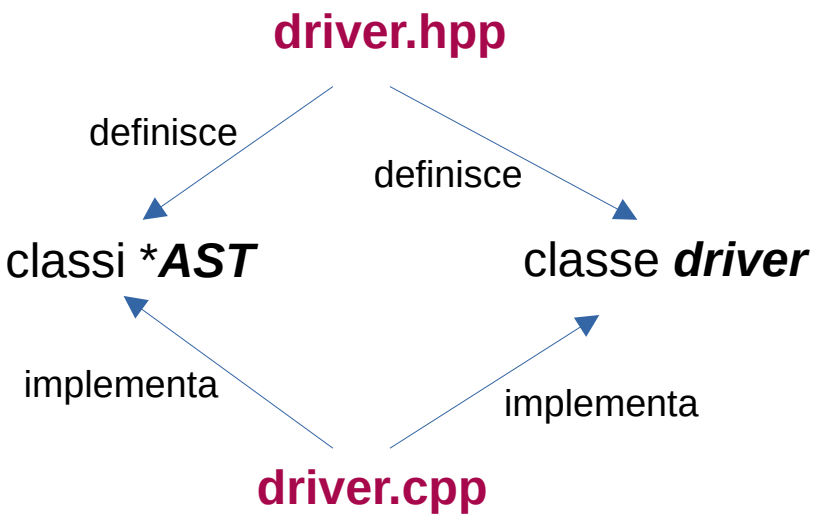
La classe driver: implementazione (parziale)

```
driver::driver(): trace_parsing (false),
                  trace_scanning (false) {};

int driver::parse (const std::string &f) {
   file = f;
   location.initialize(&file);
   scan_begin();
   yy::parser parser(*this);
   parser.set_debug_level(trace_parsing);
   int res = parser.parse();
   scan_end();
   return res;
}
```

File coinvolti

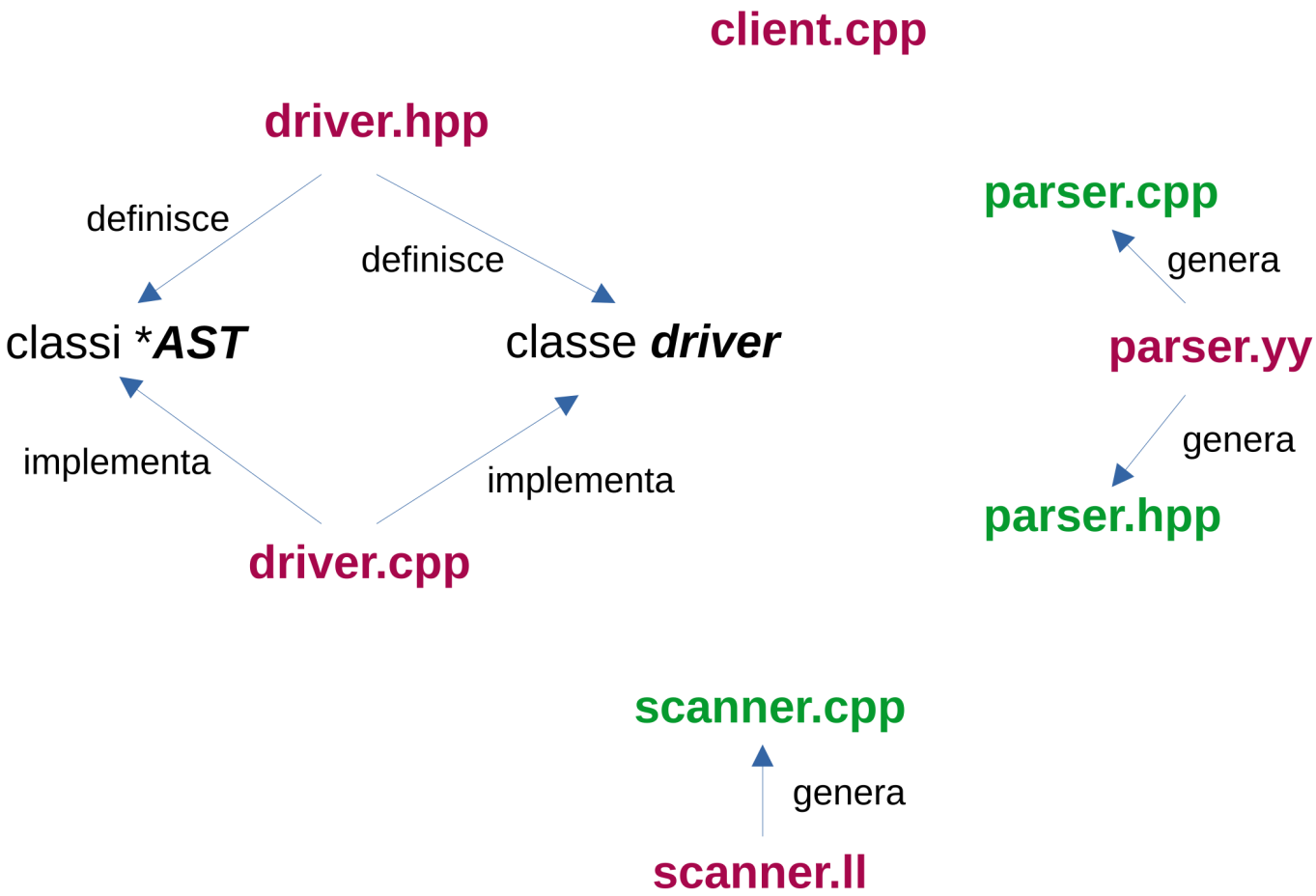
client.cpp



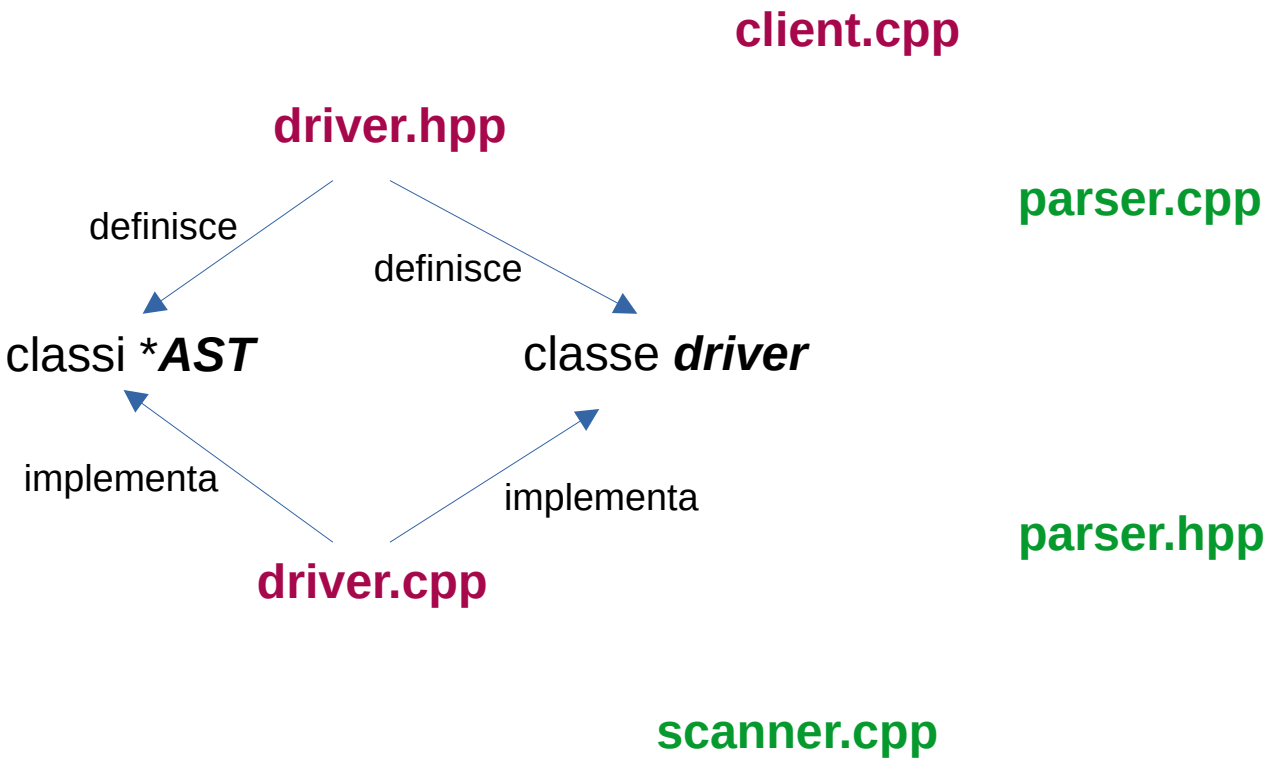
parser.yy

scanner.ll

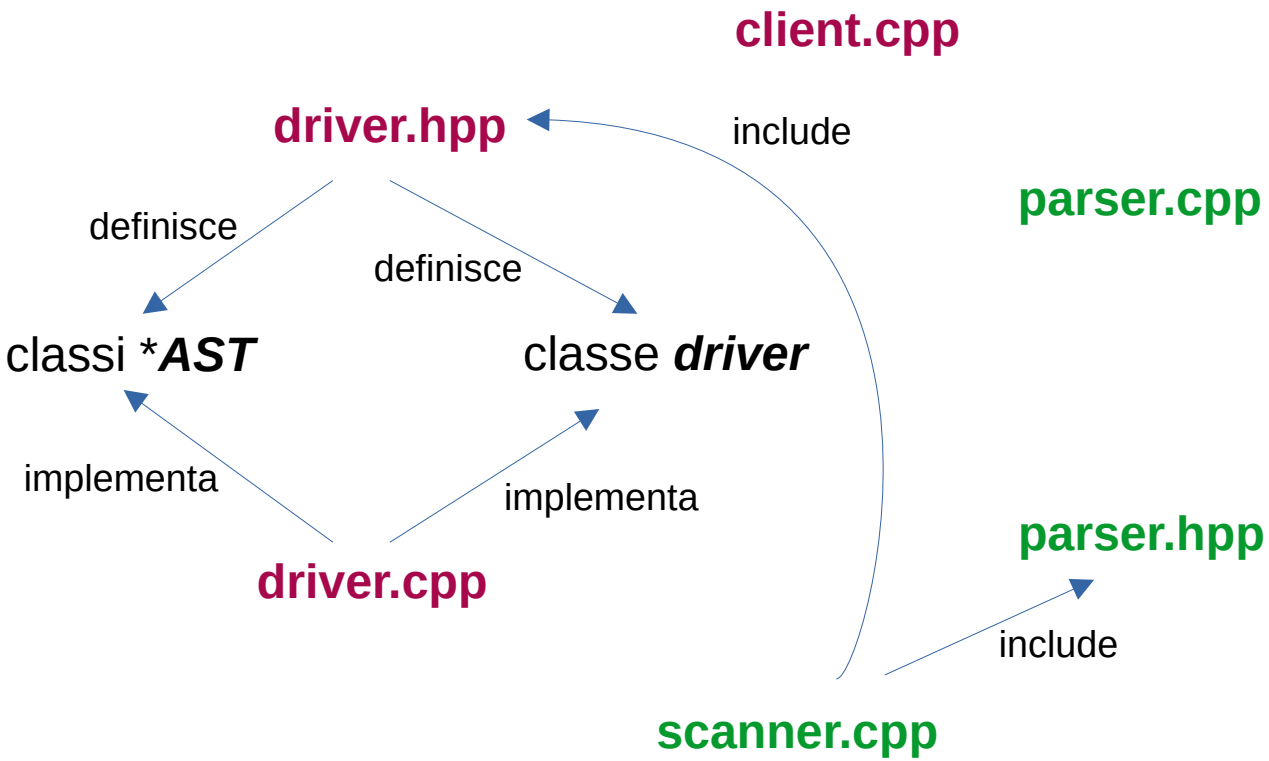
Compilazione di Lexer e Parser



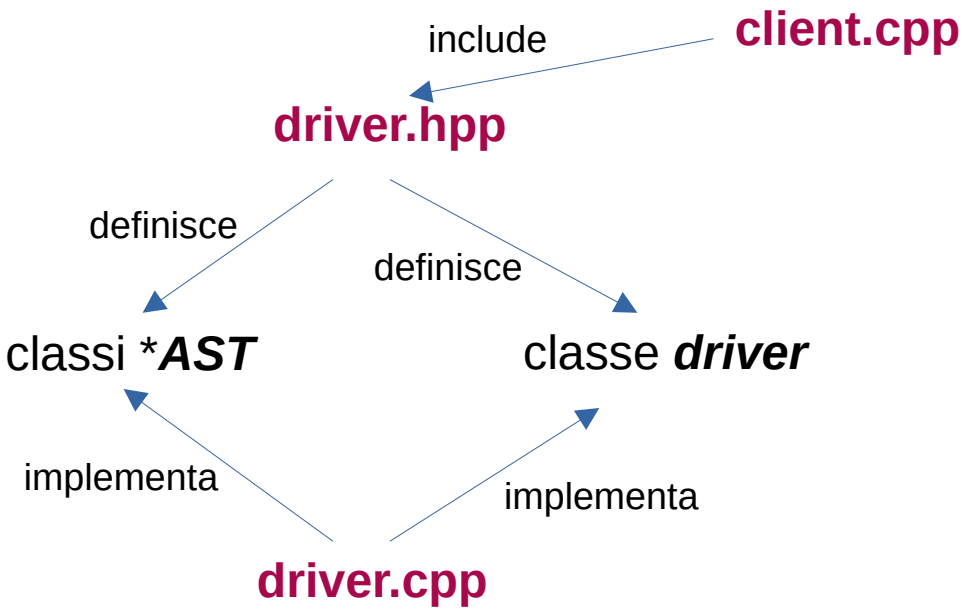
Dopo la compilazione



Dipendenze


client.cpp
driver.hpp
include
parser.cpp
definisce
definisce
classi *AST
classe driver
implementa
implementa
parser.hpp
driver.cpp
include
scanner.cpp

Dipendenze



parser.cpp

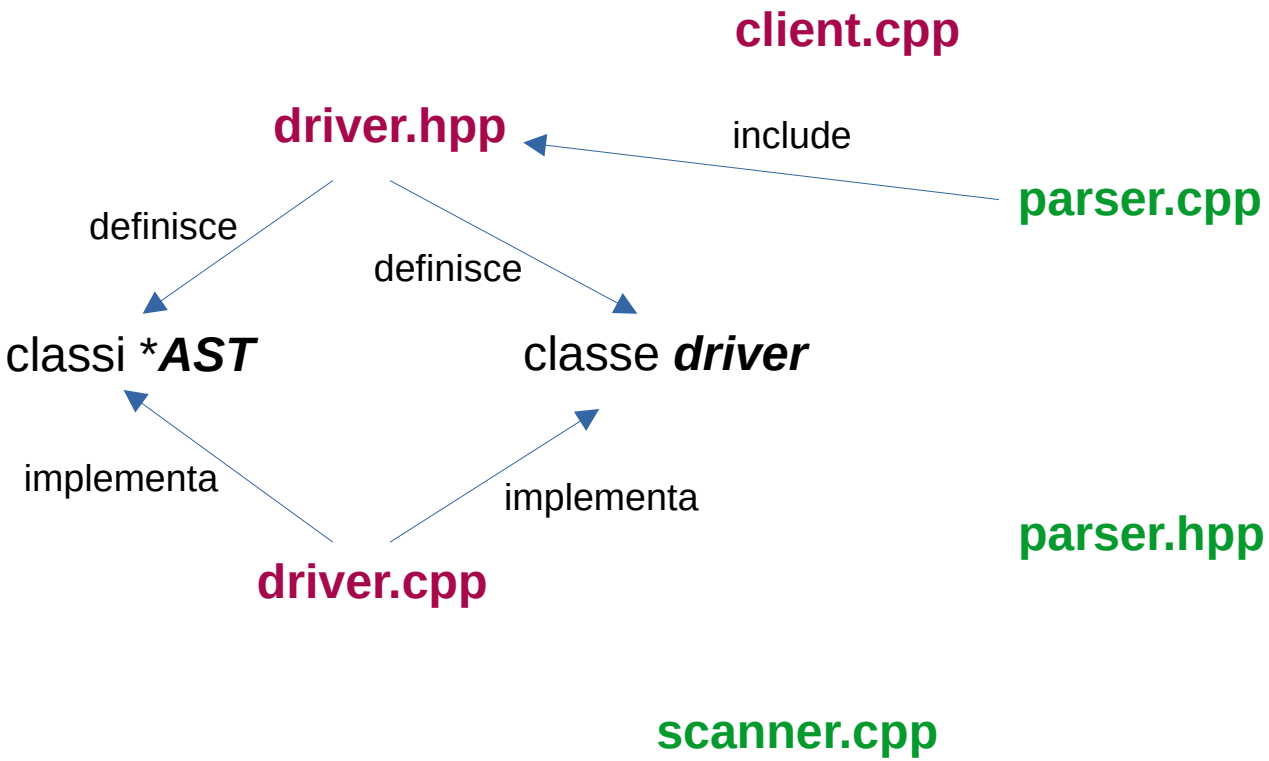
parser.hpp

scanner.cpp

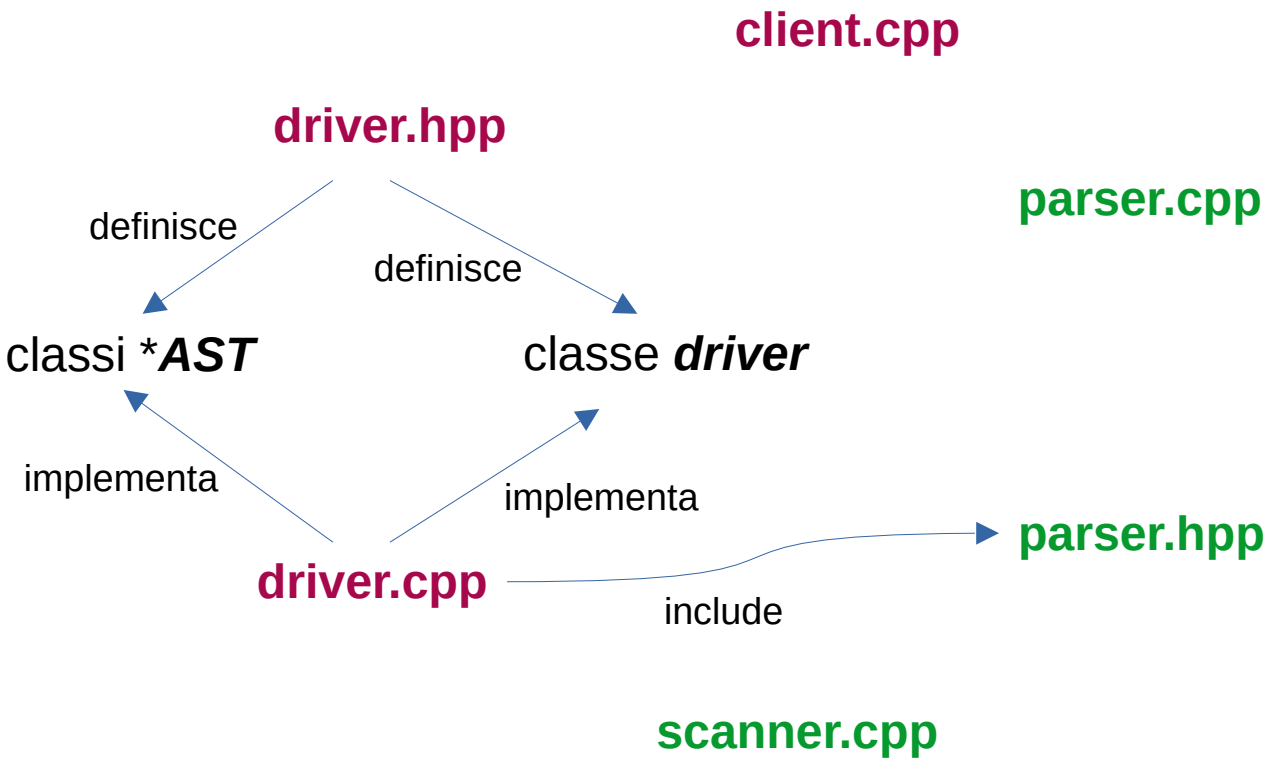
Esistenza di una dipendenza "circolare"

- Le prossime due slide mostrano l'esistenza di una mutua dipendenza di `driver` e `parser`
- Per un verso, infatti, il parser necessita di includere `driver.hpp`
- Questa è la dipendenza più evidente perché il driver include le classi `*AST` (che il parser deve usare per costruire l'albero sintattico) e la classe `driver` che funge da contesto
- Tuttavia, anche il driver necessita di includere `parser.hpp`:
  - perché è nel parser che è definita la classe `location` e questa classe è un fondamentale dato da inserire nel contesto
  - perché la definizione del tipo di ritorno di `yylex` è una enumerazione dei vari token name, che sono definiti dal `parser`
- Per risolvere la circolarità, `Bison` ha la direttiva `%code requires`, che consente di inserire *forward declaration* nell'header file

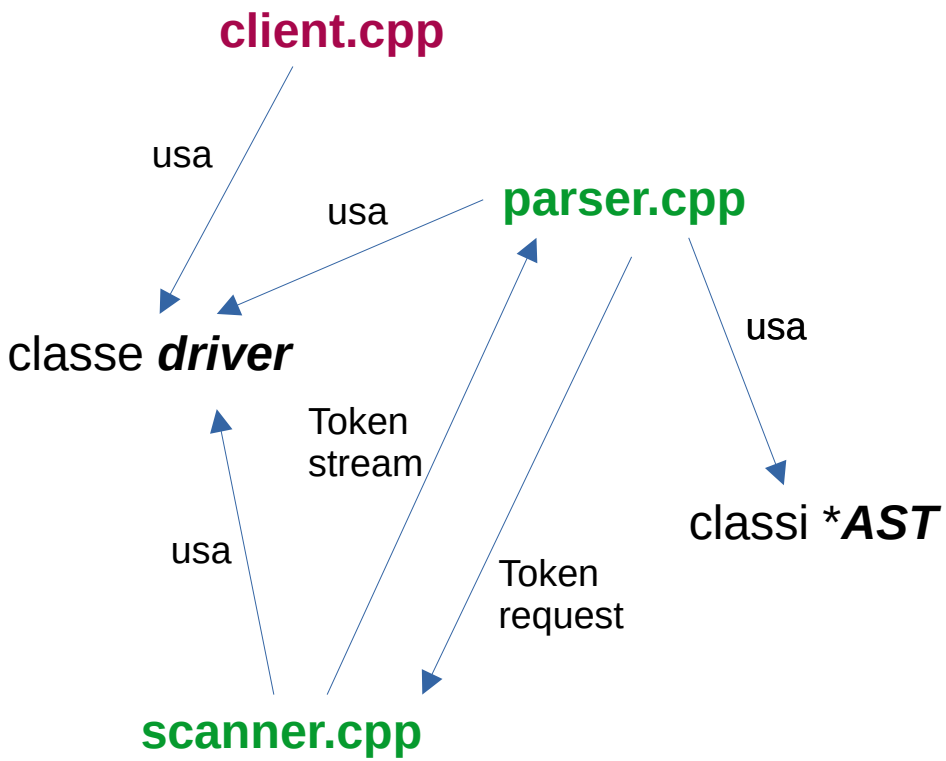
Dipendenze


client.cpp
driver.hpp
include
parser.cpp
definisce
definisce
classi *AST
classe driver
implementa
implementa
parser.hpp
driver.cpp
scanner.cpp

Dipendenze


client.cpp
driver.hpp
parser.cpp
definisce
definisce
classi *AST
classe driver
implementa
implementa
parser.hpp
driver.cpp
include
scanner.cpp

Il flusso informativo



Compilatori

I nodi dell'AST

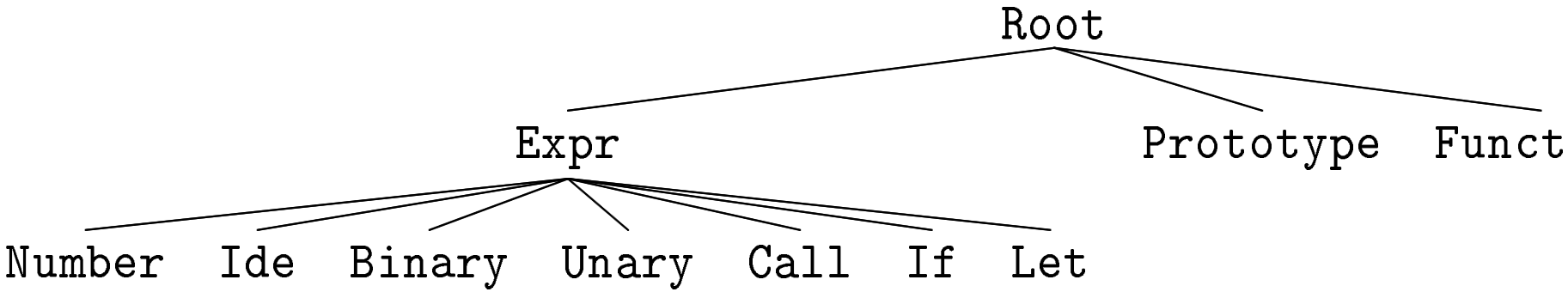
- Dal punto di vista "programmativo", l'AST altro non è che una struttura dati e dunque la prima decisione da prendere riguarda proprio la struttura da utilizzare
- L'idea è di rappresentare ogni nodo come *oggetto* di una classe
- La classe di appartenenza di un nodo può variare (naturalmente in dipendenza di ciò che il nodo vuole rappresentare), tuttavia tutte le classi dovranno ereditare da una superclasse comune
- Come vedremo, con qualche eccezioni, le classi che andremo a definire corrisponderanno ai simboli non terminali della grammatica
- In generale non esiste però una corrispondenza uno-a-uno fra classi e non terminali
- Inizieremo a riflettere su come rappresentare le espressioni

Rappresentazione delle espressioni

- Per le espressioni introduciamo una classe da cui deriveremo 4 classi, corrispondenti a: (1) operatori binari, (2) variabili, (3) costanti numeriche e (4) chiamata di funzione
- Le 4 sottoclassi rappresentano altrettanti "elementi" costituenti un'espressione. Esse rappresentano però caratteristiche diverse:
  - variabili e costanti numeriche corrispondono a foglie dell'ASD e sono elementi con valore lessicale (di natura differente)
  - gli operatori binari denotano nodi interni con due figli; tralasciando il particolare operatore, richiedono tutti lo stesso trattamento
  - come variabili e costanti, anche una chiamata di funzione etichetta una foglia di un AST ma il trattamento degli argomenti (che naturalmente sono a loro volta espressioni) richiede un collegamento ad un numero arbitrario di altri sotto-alberi.

La gerarchia completa

- In tutto introdurremo 8 classi, così organizzate



- I nomi effettivi che daremo alle classi sono leggermente differenti rispetto alla figura (dove sono stati "accorciati" per ragioni di spazio)
- I nomi della classe `Root` e delle classi che da essa derivano direttamente hanno suffisso `AST`
- I nomi che derivano dalla classe `Expr` hanno suffisso `ExprAST`
- Le classi vengono definite nel file `driver.hpp` mentre l'implementazione dei metodi è data nel file `driver.cpp`

La classe RootAST

- La classe base della gerarchia include solo metodi *virtuali* (si veda la prossima slide per un veloce "ripasso" sull'uso di tali metodi)

```
typedef std::variant<std::string,double> lexval;
const lexval NONE = 0.0;

class RootAST {
public:
   virtual ~RootAST() = default;
   virtual RootAST *left() {return nullptr;};
   virtual RootAST *right() {return nullptr;};
   virtual lexval getLexVal() {return NONE;};
   virtual void visit() {};
};
```

- I metodi saranno opportunamente ridefiniti nelle sottoclassi

Una digressione: metodi virtuali e loro utilizzo

- Si definisce `virtuale` un metodo dichiarato all'interno di una *classe base* e ridefinito (*overriden*) in una *classe derivata*
- Si tratta di uno strumento per implementare *polimorfismo* a tempo di esecuzione
- Per varie ragioni, può essere necessario fare riferimento ad un metodo M di una classe derivata D usando un "puntatore" il cui tipo è quello di una classe base B
- In questo caso, tecnicamente, il compilatore potrebbe sollevare un'eccezione se B non definisce M oppure se la definisce in modo diverso
- Se M è definita in B come virtuale, a tempo di esecuzione viene (cercata e) chiamata la funzione corretta definita in D
- Per i distruttori questa "costruzione" è di fatto obbligatoria (per far sì che venga distrutto l'oggetto giusto)

La classe base per le espressioni

- La classe `ExprASt` non include metodi virtuali e potrebbe in teoria non essere presente
- In tal caso tutte le classi erediterebbero direttamente da `RootAST`
- La classe viene introdotta principalmente per una maggiore "pulizia" del progetto complessivo
- Consideriamo il caso di un'espressione E costituita, ad es., dalla somma di due (sotto)-espressioni E_1 ed E_2
- E_1 ed E_2 possono essere di uno qualsiasi dei 4 tipi che abbiamo individuato (numero, identificatore, espressione binaria o chiamata di funzione)
- È chiaro dunque che ci deve essere una classe comune per indicare queste espressioni
- Se la classe fosse direttamente `RootAST`, potrebbe risultare lecito comporre un'espressione anche con oggetti delle altre tre classi, ad esempio `FunctionAST`, ma questo non sarebbe corretto

La classe NumberExprAST

Definizione

```
class NumberExprAST : public ExprAST {
private:
   int Val;
public:
   NumberExprAST(double Val);
   void visit();
   lexval getVal() const;
};
```

Implementazione

```
NumberExprAST::NumberExprAST(double Val): Val(Val) {};
void NumberExprAST::visit() {
   std::cout << Val << " "; };
lexval NumberExprAST::getLexVal() {
```

La classe IdeExprAST

Definizione

```
class IdeExprAST : public ExprAST {
private:
   std::string Name;
public:
IdeExprAST(std::string &Name);
void visit();
lexval getLexVal(); };
```

Implementazione

```
IdeExprAST::IdeExprAST(std::string &Name):
         Name(Name) {};
void IdeExprAST::visit() {
    std::cout << Name << " "; };
lexval IdeExprAST::getLexVal() {
```

La classe BinaryExprAST: definizione

```
class BinaryExprAST : public ExprAST {
private:
    char Op;
    ExprAST* LHS;
    ExprAST* RHS;
public:
    BinaryExprAST(char Op, ExprAST* LHS, ExprAST* RHS);
    ExprAST* left();
    ExprAST* right();
    void visit();
};
```

La classe BinaryExprAST: implementazione

```
BinaryExprAST::BinaryExprAST(char Op, ExprAST* LHS,
       ExprAST* RHS): Op(Op), LHS(LHS), RHS(RHS) {};
ExprAST* BinaryExprAST::left() {
   return LHS;
};
ExprAST* BinaryExprAST::right() {
   return RHS;
};
void BinaryExprAST::visit() {
   std::cout << "(" << Op << " ";
   LHS->visit();
   if (RHS!=nullptr) RHS->visit();
   std::cout << ")";
};
```

La classe CallExprAST: definizione

```
class CallExprAST : public ExprAST {
private:
    std::string Callee;
    std::vector<ExprAST*> Args;

public:
    CallExprAST(std::string Callee,
                std::vector<ExprAST*> Args);
    lexval getLexVal() const;
    void visit();
};
```

La classe CallExprAST: implementazione

```
CallExprAST::CallExprAST(std::string Callee,
                         std::vector<ExprAST*> Args): Callee(Callee),
                         Args(std::move(Args)) {};
lexval CallExprAST::getLexVal() const {
   lexval lval = Callee;
   return lval;
};
void CallExprAST::visit() {
    std::cout<< std::get<std::string>(getLexVal())<< "( ";
    for (ExprAST* arg : Args) {
       arg->visit();
    };
    std::cout << ')';
};
```

Le classi `FunctionAST` e `PrototypeAST`

- Le ultime due classi sono di natura differente, in quanto non implementano espressioni
- La classe `FunctionAST` rappresenta la definizione *completa* di una funzione
- La classe `PrototypeAST` rappresenta la definizione del *prototipi* di funzione ed è ovviamente utilizzata dalla classe `FunctionAST`

La classe PrototypeAST: definizione

```
class PrototypeAST : public RootAST {
private:
    std::string Name;
    std::vector<std::string> Args;

public:
    PrototypeAST(std::string Name, std::vector<std::string>
    lexval getLexVal() const;
    const std::vector<std::string> &getArgs() const;
    void visit();
    int argsize();
};
```

La classe PrototypeAST: implementazione

```
PrototypeAST::PrototypeAST(std::string Name,
                           std::vector<std::string> Args):
                           Name(Name), Args(std::move(Args)) {};
lexval PrototypeAST::getLexVal() const {
   lexval lval = Name; return lval; };
const std::vector<std::string>& PrototypeAST::getArgs()
   const { return Args; };
void PrototypeAST::visit() {
   std::cout << "extern " << Name << "( ";
   for (auto it=Args.begin(); it!=Args.end(); ++it) {
      std::cout << *it << ' '; };
   std::cout << ')';
};
int PrototypeAST::argsize() {
   return Args.size(); };
```

La classe FunctionsAST: definizione

```
class FunctionAST : public RootAST {
private:
    PrototypeAST* Proto;
    ExprAST* Body;
    bool external;

public:
    FunctionAST(PrototypeAST* Proto, ExprAST* Body);
    void visit();
    int nparams();
};
```

La classe FunctionAST: implementazione

```
FunctionAST::FunctionAST(PrototypeAST* Proto,
      ExprAST* Body): Proto(Proto), Body(Body) {
   if (Body == nullptr) external=true;
   else external=false; };
void FunctionAST::visit() {
   std::cout << std::get<std::string>
       (Proto->getLexVal()) << "( ";
   for (auto it=Proto->getArgs().begin();
        it != Proto->getArgs().end(); ++it) {
      std::cout << *it << ' ';
   };
   std::cout << ')';
   Body->visit();
};
int FunctionAST::nparams() {
   return Proto->argsize(); };
```