Corso di Laurea Magistrale in Ingegneria Informatica
A.A. 2013-2014

Linguaggi Formali e Compilatori

Premessa

Le pagine che seguono hanno lo scopo di presentare una sintesi dei principali elementi utili alla realizzazione del tema d'anno, che, come è noto, rappresenta il principale oggetto di colloquio e valutazione nel corso dell'esame.

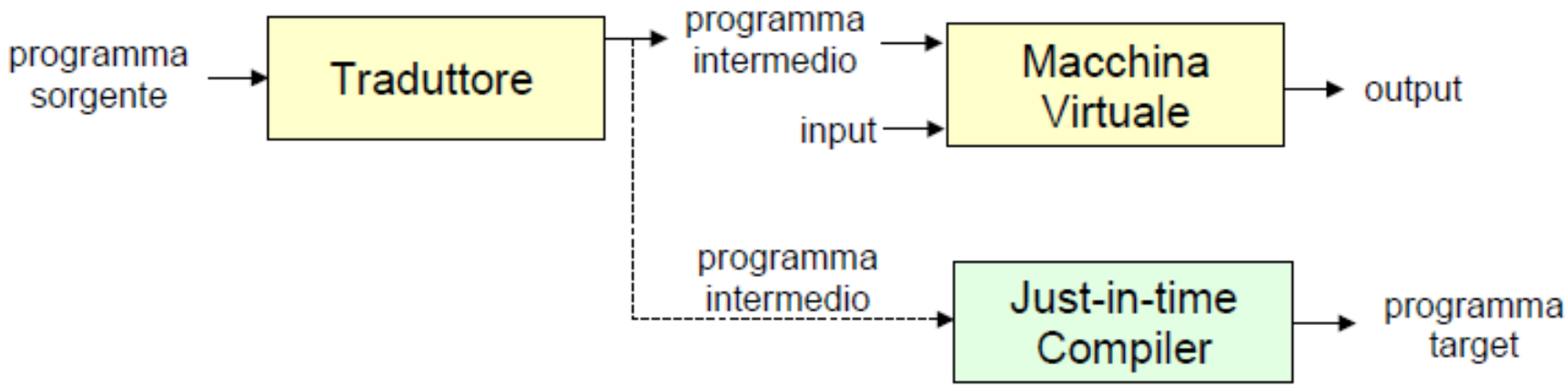
Nel seguito, perciò, si riportano schemi, figure e brani contenuti nelle letture e nei testi consigliati, oltre che suggerimenti e considerazioni del docente circa i principali aspetti di alcuni temi d'anno svolti negli scorsi anni accademici da alcuni studenti.

A language translator (or compiler)

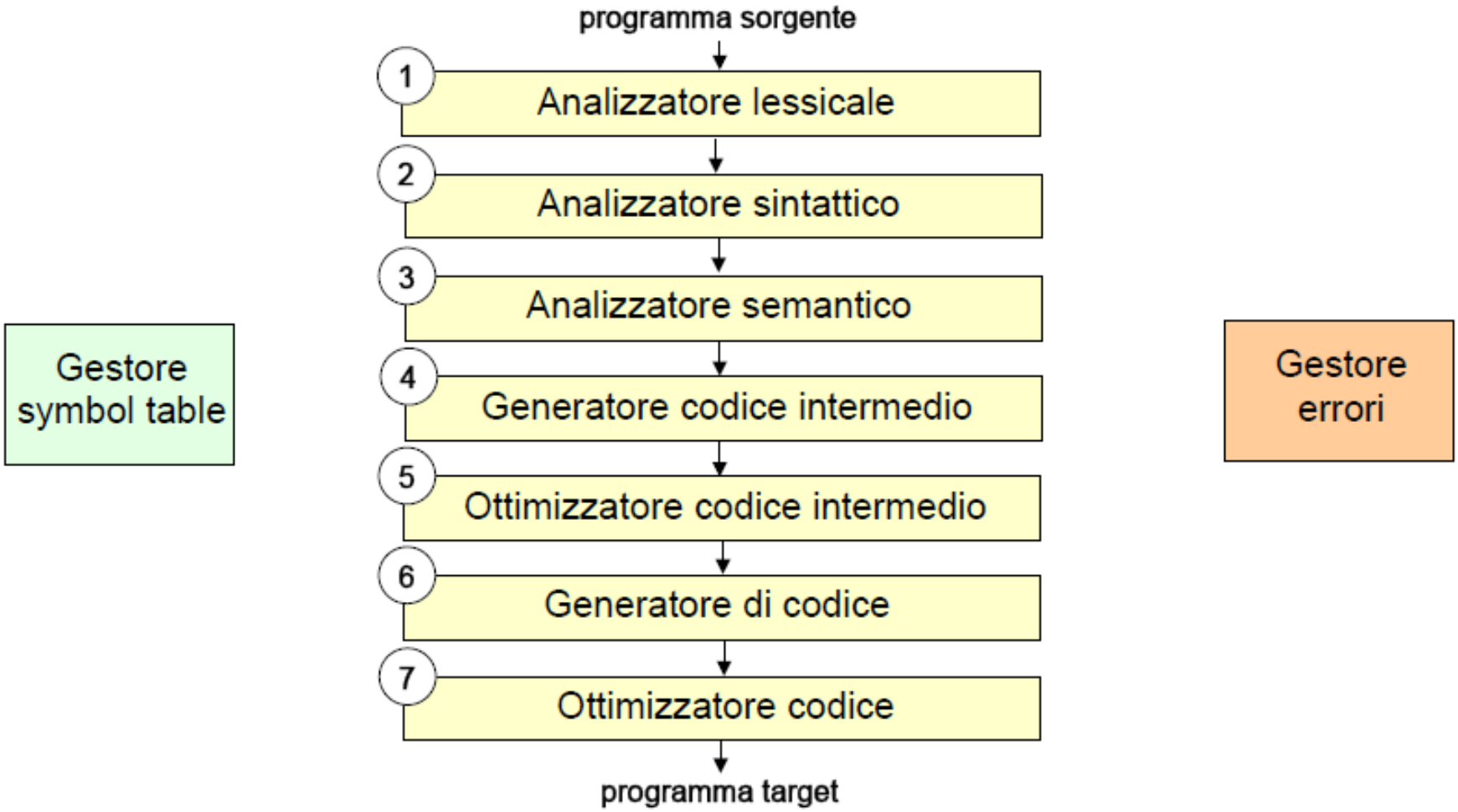
A language translator (or compiler) is a program which translates programs written in a source language into an equivalent program in an object language.

The source language is usually a high-level programming language and the object language is usually the machine language of an actual computer.

From the pragmatic point of view, the translator defines the semantics of the programming language, it transforms operations specified by the syntax into operations of the computational model to some ***real*** or ***virtual*** machine.



The phases of a language translator (1/3)



Lamperti, Università di Brescia

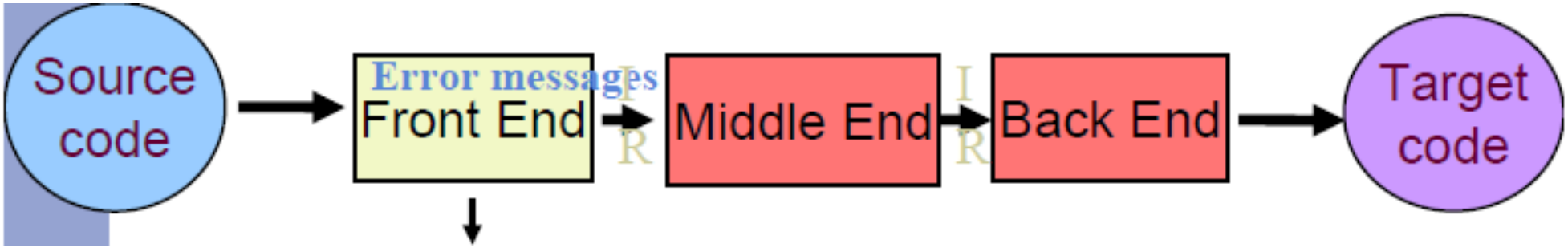
The phases of a language translator (2/3)

The typical compiler consists of several phases each of which passes its output to the next phase:

- The lexical phase (***scanner***) groups characters into lexical units or tokens.
  The input to the lexical phase is a character stream. The output is a stream of tokens.
  Regular expressions are used to define the tokens recognized by a scanner (or lexical analyzer).
  The scanner is implemented as a finite state machine.
  Lex and Flex are tools for generating scanners: programs which recognize lexical patterns in text. Flex is a faster version of Lex.
- The ***parser*** groups tokens into syntactical units.
  The output of the parser is a parse tree representation of the program.
  Context-free grammars are used to define the program structure recognized by a parser.
  The parser is implemented as a push-down automata.
  Yacc and Bison are tools for generating parsers: programs which recognize the structure grammatical structure of programs. Bison is a faster version of Yacc.
- The ***semantic analysis*** phase analyzes the parse tree for context-sensitive information often called the static semantics.
  The output of the semantic analysis phase is an annotated parse tree.
  Attribute grammars are used to describe the static semantics of a program.
  This phase is often combined with the parser.
  During the parse, information concerning variables and other objects is stored in a ***symbol table***.
  The information is utilized to perform the context-sensitive checking.

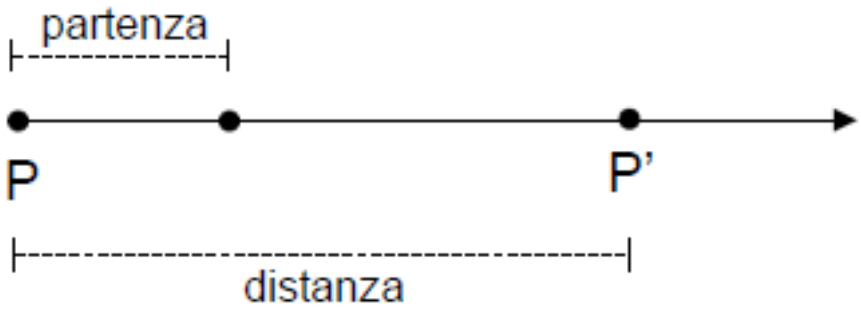
The phases of a language translator (3/3)

- The ***intermediate code generator*** transforms the simplified annotated parse tree into intermediate code using rules which denote the semantics of the source language. The code generator may be integrated with the parser.
- The ***intermediate code optimizer*** applies semantics preserving transformations ***(machine independent code improvements)*** to the annotated parse tree to simplify the structure of the tree and to facilitate the generation of more efficient code.
- The ***code generator*** transforms the optimized intermediate code into object code using the target machine intructions.
- The ***peep-hole optimizer*** examines the object code, a few instructions at a time, and attempts to do ***machine dependent code improvements***.

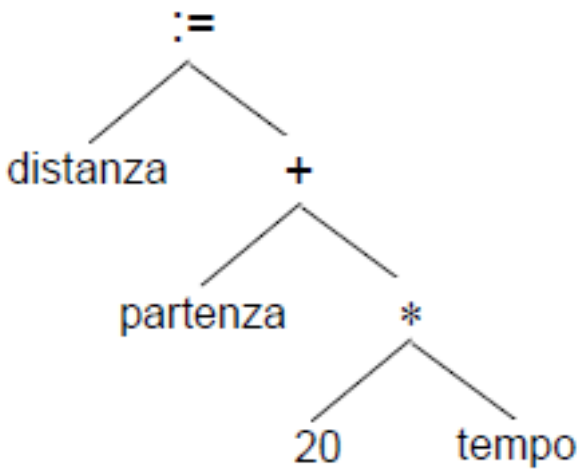


In contrast with compilers ***an interpreter*** is a program which simulates the execution of programs written in a source language.
Interpreters may be used either at the source program level or as an object code for an idealized machine.
This is the case when a compiler generates code for an idealized machine whose architecture more closely resembles the source code.

Trasformazione del Programma Sorgente




```
distanza := partenza + 20 * tempo
```

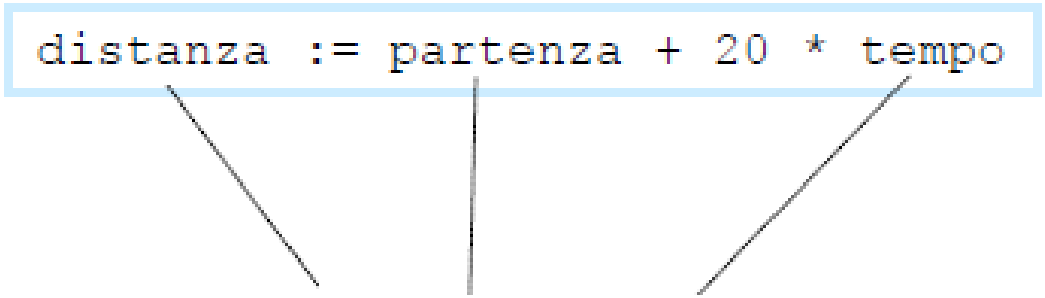


Le fasi della trasformazione

- Stessa computazione espressa secondo diversi livelli di astrazione

Lamperti, Università di Brescia

La trasformazione nell'analisi lessicale



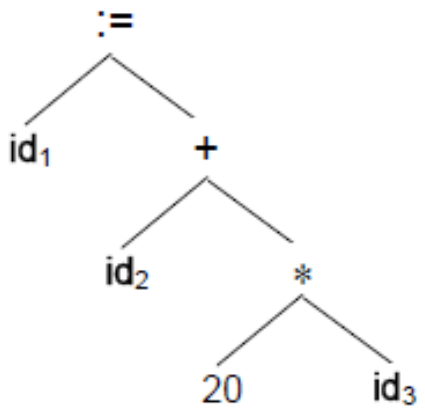
- **Analisi lessicale** id1 := id2 + 20 * id3

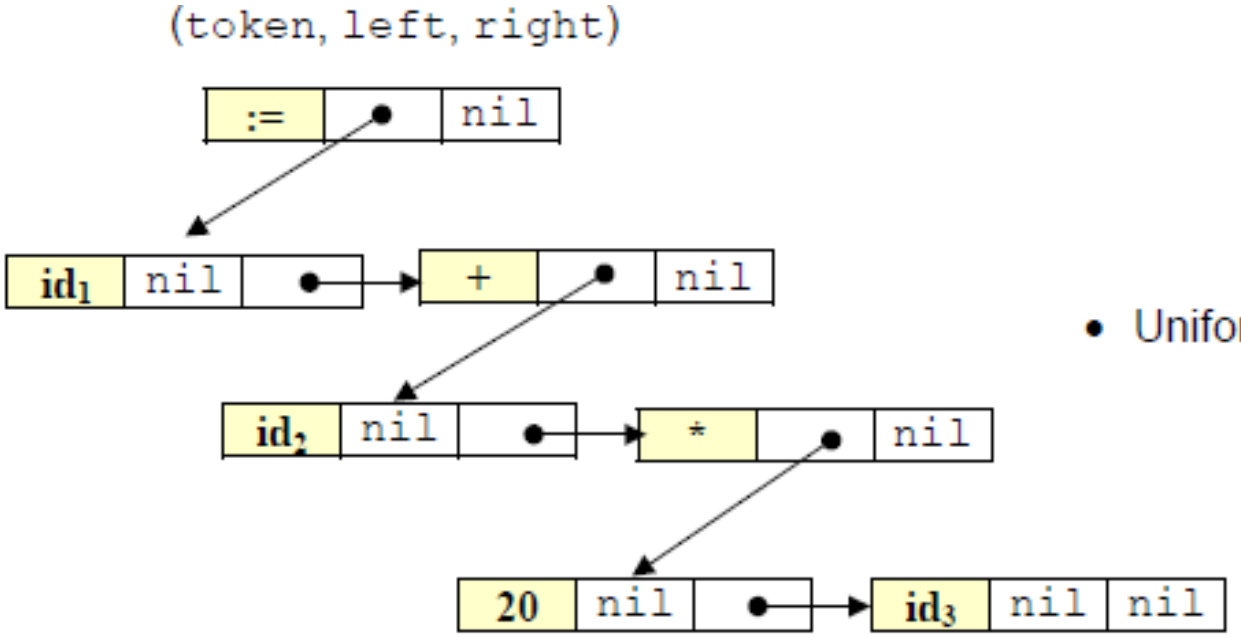
Lamperti, Università di Brescia

La trasformazione nell'analisi sintattica

Analisi sintattica

$id_1 := id_2 + 20 * id_3$





- Uniformità nodi → pb
  - disomogeneità
  - cardinalità prole

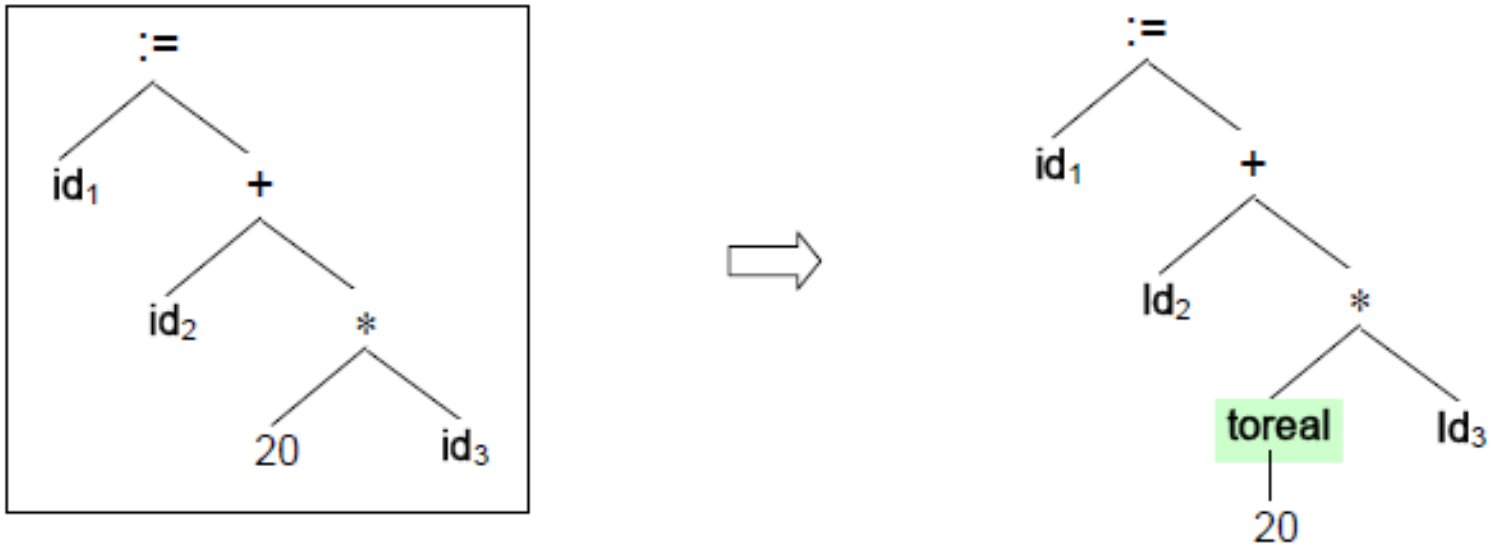
union

binarizzazione

Lamperti, Università di Brescia

La trasformazione nell'analisi semantica

Analisi semantica

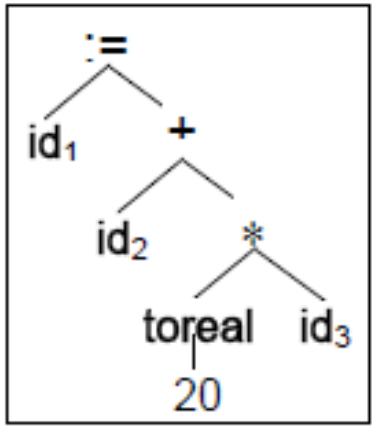


- Rivisitazione dell'albero per controllare i vincoli semantici
- Eventuale decorazione/alterazione dell'albero

Lamperti, Università di Brescia

La trasformazione nella generazione del codice intermedio

- **Generazione del codice intermedio → ∃ una istruzione per ogni operatore dell'albero sintattico**



t_1 := toreal(20)
t_2 := t_1 * id_3
t_3 := id_2 + t_2
id_1 := t_3

Codice intermedio = programma scritto nel linguaggio di una macchina astratta (virtuale)
Proprietà: facile da produrre e da tradurre in codice target
Natura: svariata, tipicamente: codice a tre indirizzi (quadruple) ≈ Assembler in cui le locazioni di memoria sono assimilabili a registri

operatore ind_1 ind_2 ind_3 ⇨ * ind_1 ind_2 ind_3 ≡ ind_3 := ind_1 * ind_2

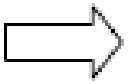
∃ al più un operatore <u>esplicito</u> (oltre all'assegnamento) → sequenzializzazione delle operazioni
Generazione di temporanei per i risultati intermedi
Non necessariamente tre operandi (anche meno, es: `toreal`)

Lamperti, Università di Brescia

La trasformazione nell'ottimizzazione del codice intermedio

Ottimizzazione del codice intermedio

$t_1 := \text{toreal}(20)$
$t_2 := t_1 * id_3$
$t_3 := id_2 + t_2$
$id_1 := t_3$

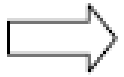


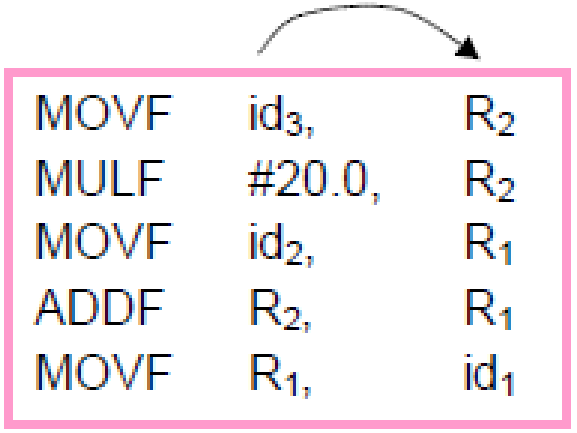
$t := 20.0 * id_3$
$id_1 := id_2 + t$

La trasformazione nell'ottimizzazione del codice intermedio

Generazione di codice

$t := 20.0 * id_3$
$id_1 := id_2 + t$





Tipo di codice target	Assembler (tipicamente) Macchina rilocabile → variabili trasformate in locazioni di memoria
Mapping	istruzioni → istruzioni macchina equivalenti variabili → registri

Lamperti, Università di Brescia

Symbol Table

Struttura dati contenente informazioni sugli identificatori (catalogo)

attributi (informazione rilevante)

id					
distanza					
partenza					
tempo					

⇨

- spazio allocato
- tipo
- scope
- se nome di procedura
  - numero parametri
  - tipo parametri
  - metodo di passaggio par.
  - [tipo parametro uscita]

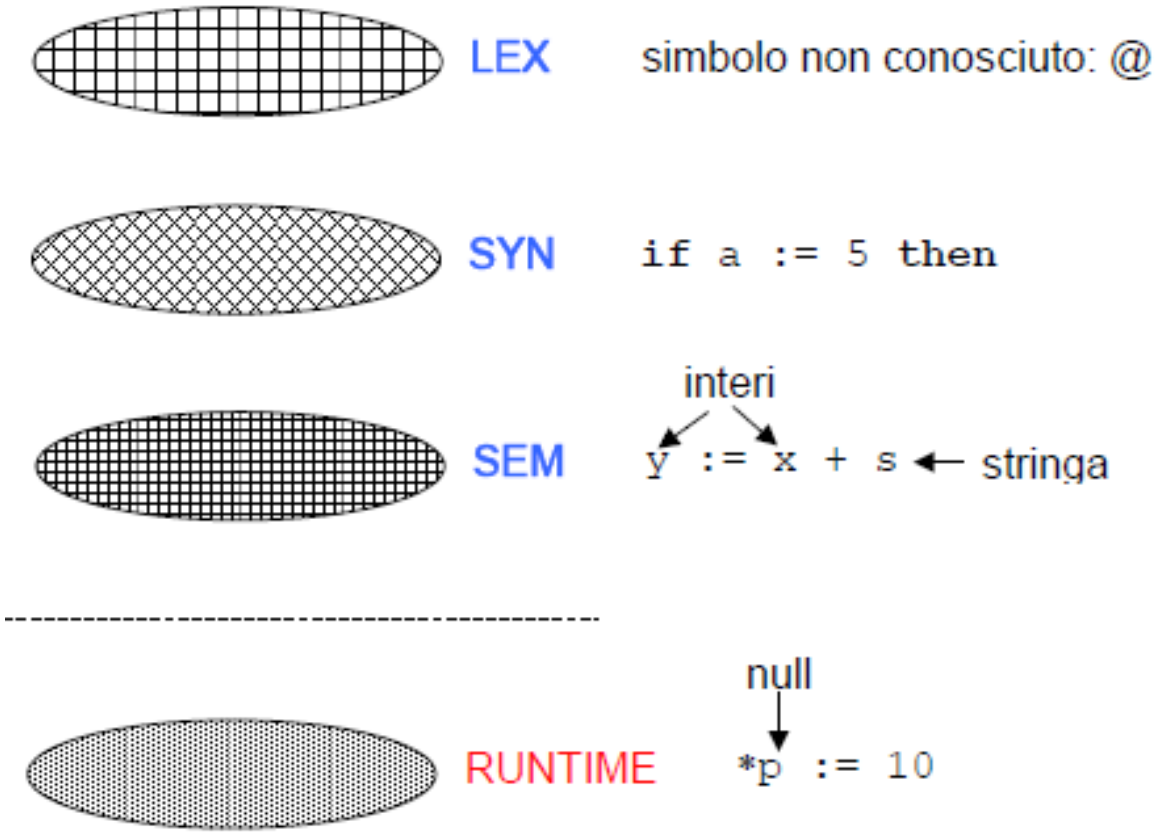
Deve permettere accesso efficiente agli attributi di un identificatore in lettura
scrittura

aggiornamento incrementale degli attributi

Lamperti, Università di Brescia

Gestione Errori

∀ fase F → gestione degli errori pertinenti ad F



Lamperti, Università di Brescia

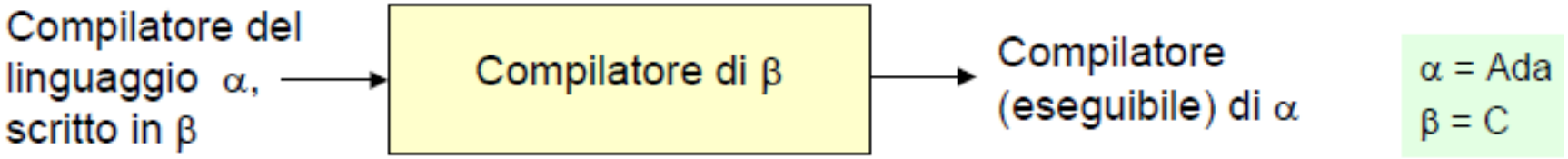
Bootstrapping & Porting

L coinvolti nella realizzazione di un compilatore

sorgente
target
implementazione (host)

⇧

L macchina nei primi compilatori



Cross-compilatore: quando Comp(β) gira su macchina $\neq$ target

target = Intel
Comp(β) su Motorola

Lamperti, Università di Brescia

LEX / FLEX

Lex is a lexical analyzer generator developed at AT&T Bell Laboratories.

The input to Lex/Flex is a file containing tokens defined using regular expressions.

Lex/Flex produces an entire scanner module that can be compiled and linked to other compiler modules.

Lex/Flex generates a file containing the function **`yylex()`** which is called from the function **`yyparse()`** and returns an integer denoting the token recognized.

An input file for Lex is of the form:

```
C and scanner declarations          first section
%%
Token definitions and actions       second section
%%
C subroutines                       third section
```

An example of first section of FLEX

```
%{
#include "y.tab.h" /* The tokens */
%}
DIGIT [0-9]
ID [a-z][a-z0-9]*
%%
Token definitions and actions
%%
C subroutines
```

The first section of the Lex file contains the C declaration to include the file (**y.tab.h**) produced by Bison which contains the definitions of the multi-character tokens.

The routine is supposed to return the type of the next token as well as putting any associated value in the global **yylval**. The function **yyparse** expects to find the textual location of a token just parsed in the global variable **yylloc**. So **yylex** must store the proper data in that variable. The data type of **yylloc** has the name **yyltype**.
To use Flex with BISON, one specifies the '**-d**' option to yacc to instruct it to generate the file '**y.tab.h**' containing definitions of all the `**%tokens**' appearing in the yacc input.
The first section also contains Lex definitions used in the regular expressions.

In this case, DIGIT is defined to be one of the symbols 0 through 9 and ID is defined to be a lower case letter followed by zero or more letters or digits.

An example of second section of FLEX (1/2)

C and scanner declarations

```
%%
":=" { return(ASSGNOP); }
{DIGIT}+ { return(NUMBER); }
do { return(DO); }
else { return(ELSE); }
end { return(END); }
fi { return(FI); }
if { return(IF); }
in { return(IN); }
integer { return(INTEGER); }
let { return(LET); }
read { return(READ); }
skip { return(SKIP); }
then { return(THEN); }
while { return(WHILE); }
write { return(WRITE); }
{ID} { return(IDENTIFIER); }
[ \t\n]+ /* blank, tab, new line: eat up whitespace */
. { return(yytext[0]); }
%%
```

C subroutines

Riconoscimento di simboli lessicali in un linguaggio di programmazione

```
%{
#include <stdlib.h>
#include "def.h" /* IF, THEN, ELSE, ID, NUM, RELOP, LT, LE, EQ, NE, GT, GE */
int lexval;
%}
delimitatore                [ \t\n]
spaziatura                  {delimitatore}+
lettera                     [A-Za-z]
cifra                       [0-9]
id                          {lettera}({lettera}|{cifra})*
num                         {cifra}+
%%
{spaziatura}                ;
if                          {return(IF);}
then                        {return(THEN);}
else                        {return(ELSE);}
{id}                        {lexval = assegna_id(); return(ID);}
{num}                       {lexval = atoi(yytext); return(NUM);}
"<"                         {lexval = LT; return(RELOP);}
"<="                        {lexval = LE; return(RELOP);}
"="                         {lexval = EQ; return(RELOP);}
"<>"                        {lexval = NE; return(RELOP);}
">"                         {lexval = GT; return(RELOP);}
">="                        {lexval = GE; return(RELOP);}
%%
assegna_id() /* tabella di simboli senza keywords */
{    int riga;
     if((riga=cerca(yytext)) == 0) riga = inserisci(yytext);
     return(riga);
}
```

An example of second section of FLEX (2/2)

The second section of the Lex file gives the regular expressions for each token to be recognized and a corresponding action.

Strings of one or more digits are recognized as an integer and thus the value **`INT`** is returned to the parser.

The reserved words of the language are strings of lower case letters (upper-case may be used but must be treated differently). Blanks, tabs and newlines are ignored.

All other single character symbols are returned as themselves (the scanner places all input in the string **`yytext`**).

The values associated with the tokens are the integer values that the scanner returns to the parser upon recognizing the token.

There is a global variable **`yylval`** accessible by both the scanner and the parser and is used to store additional information about the token.

The third section of FLEX

The third section of the file may contain C code associated with the actions.

Compiling the Flex file with the command flex file.lex results in the production of the file **`lex.yy.c`** which defines the C function **`yylex()`**.

On each invocation, the function **`yylex()`** scans the input file and returns the next token.

Uso accorto del manuale di BISON

Manuale BISON del GNU

by Charles Donnelly and Richard Stallman
october 23, 2013; Bison Version 3.0.2

Leggere par.	1.1 – 1.4	
	1.6	
	1.7	
	2.1 (2.1.1 – 2.1.7)	
	2.5 (2.5.1 – 2.5.3)	
Tenere presente	Appendix A	Bison Symbols
	Appendix B	Glossary

Le fasi dell'uso di BISON

The actual language-design process using Bison, from grammar specification to a working compiler or interpreter, has these parts:

1. Formally specify the grammar in a form recognized by Bison (see Chapter 3 [Bison Grammar Files]). For each grammatical rule in the language, describe the action that is to be taken when an instance of that rule is recognized. The action is described by a sequence of C statements.

2. Write a lexical analyzer to process input and pass tokens to the parser. The lexical analyzer may be written by hand in C or it could also be produced using Flex.

3. Write a controlling function that calls the Bison-produced parser.

4. Write error-reporting routines.

To turn this source code as written into a runnable program, you must follow these steps:

a) Run Bison on the grammar to produce the parser.

b) Compile the code output by Bison, as well as any other source files.

c) Link the object files to produce the finished product.

I temi d'anno da considerare

Ricci, Valle – *SIMPLE compile&execute VM*

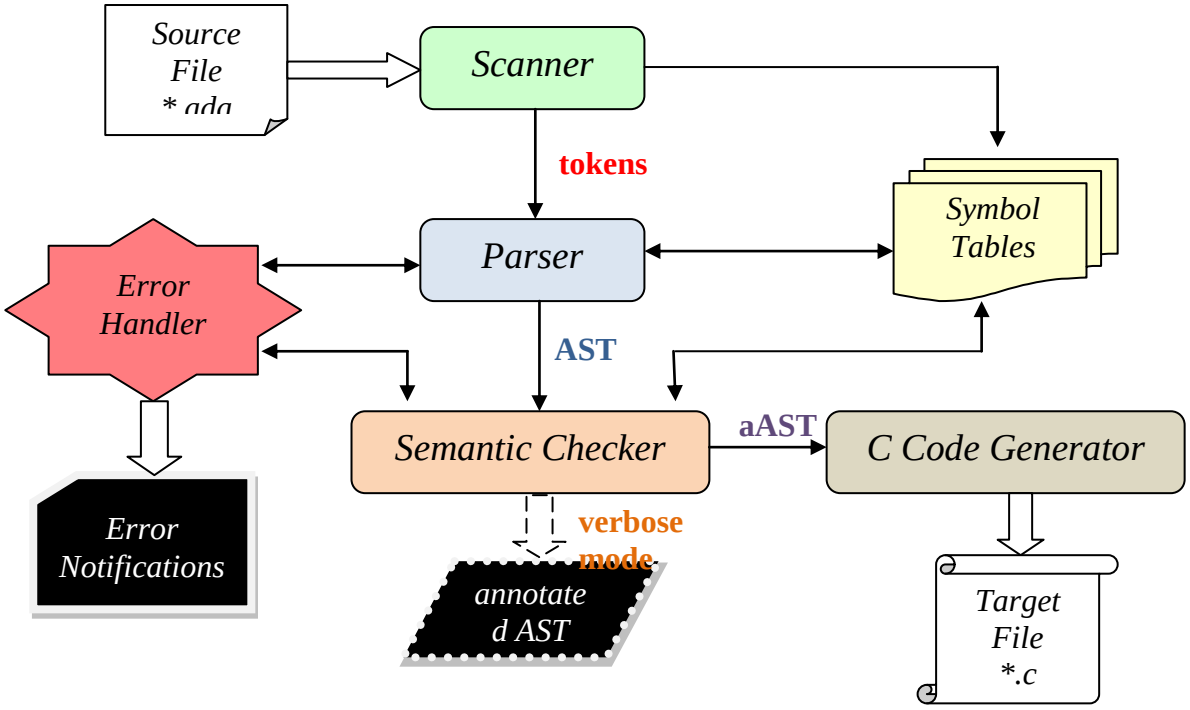
Prendono l'esempio (linguaggio SIMPLE) riportato nella lettura consigliata di Anthony A. Aaby (Compiler Construction using Flex and Bison) e, costruendo le due necessarie macchine virtuali, sviluppano l'intero compilatore fino all'effettiva esecuzione del codice sorgente.

Dibitonto, Fiorella, Leoci – *C dialect analyzer*

C-Analyzer consente di effettuare l'analisi lessicale, sintattica e semantica di un codice scritto nel linguaggio di programmazione C, in particolare in un dialetto del C. La caratteristica saliente è che il software fa uso di NetBeans, un ambiente di sviluppo che ha consentito di realizzare un'interessante interfaccia grafica in Java.

Campese - *da ADA a C*

Realizzazione del *Front-End* di un Compilatore dedicato all'analisi ed alla traduzione di un linguaggio ADA-like. Il progetto del Compilatore si è articolato nello sviluppo di vari moduli. Interessante è l'uso di 6 symbol table (mediante il package **`uthash-1.9.3`**) e di un Traduttore di alto livello che genera codice C.



Ambiente *Linux - Ubuntu 10.10 Maverick Meerkat*, distribuzione in cui sono inclusi i tools di sviluppo *Flex* e *Bison*

Uthash, scritta da Troy Hanson, è composto da un header file da includere nel sorgente dove utilizzare l'Hash Table implementata in C; supporta operazioni di aggiunta, ricerca e rimozione su strutture dati C.

Il progetto del Compilatore si è articolato nello sviluppo di vari moduli:

- un analizzatore lessicale (Scanner);
- un analizzatore sintattico (Parser);
- un analizzatore semantico (Semantic Checker);
- sei Tabelle dei Simboli (Symbol Tables);
- un Gestore degli errori (Error Handler);
- un Traduttore di alto livello che genera codice C.

Ci sono due modi per compilare ed ottenere l'eseguibile finale. Il primo è procedere per passi generando e compilando i singoli file uno per volta ed eseguendo quindi il link dei file-oggetto creati; l'altro consiste nell'eseguire lo script esistente che contiene una sequenza preimpostata di tutti i comandi precedenti necessari alla creazione dell'eseguibile. Chiaramente se si vuole specificare particolari flag, opzioni di compilazione o scegliere il file eseguibile finale, è consigliabile la prima modalità.

```
bison -d parser.y
gcc -c parser.tab.c
flex scanner.l
gcc -c lex.yy.c
gcc -o A2C parser.tab.o lex.yy.o -lm
```

In sintesi occorre innanzitutto generare il parser tramite il comando

`bison -d parser.y`

dove **`parser.y`** è il file che contiene le direttive specificate dalla grammatica del linguaggio; viene usata l'opzione **`-d`** per istruire BISON della generazione di un file header contenente i codici dei token associati alla grammatica, che sarà incluso e utilizzato dallo scanner. BISON genera quindi il file **`parser.tab.c`** contenente il codice C che è in grado di eseguire l'analisi sintattica. Si compila il file,

`gcc -c parser.tab.c`

e si istruisce il compilatore con l'opzione **`-c`** a creare il relativo file oggetto.

Possiamo ora occuparci della generazione dello scanner con il comando

`flex scanner.l`

e relativa compilazione del file **`lex.yy.c`** generato da Flex e creazione del file oggetto,

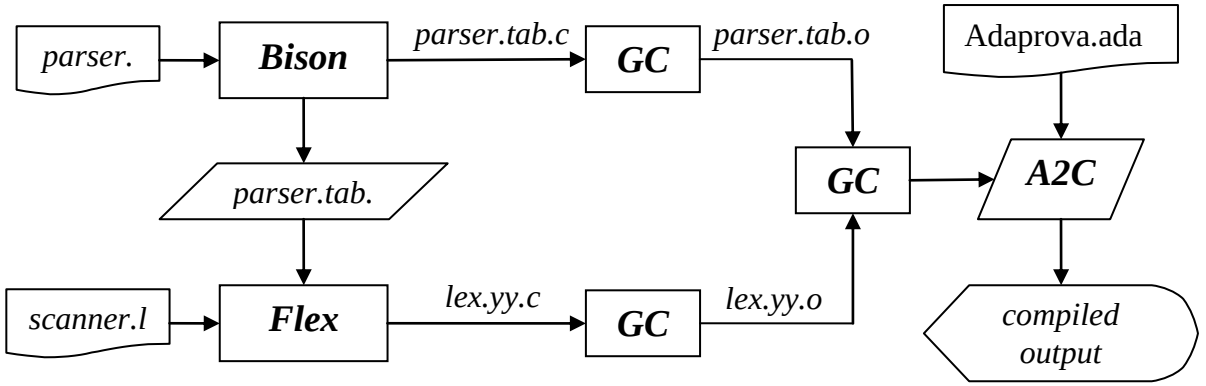
`gcc -c lex.yy.c`

A questo punto sono stati ottenuti i file oggetto che devono essere sottoposti ad operazione di *linking* per la generazione del file eseguibile finale

```
gcc -o A2C parser.tab.o lex.yy.o -lm
```

che nello script è chiamato **A2C**; l'opzione **`-lm`** indica al compilatore di cercare le funzioni matematiche specificate nel codice.
Ottenuto il file eseguibile è pertanto possibile usare il compilatore da linea di comando inserendo il nome del file eseguibile e il percorso del sorgente ADA-like da tradurre. Il compilatore mostrerà a video gli eventuali errori di compilazione e genererà il relativo file di traduzione in codice C, salvandolo nella cartella di lavoro.

diagramma a blocchi della generazione del Compilatore



E' possibile anche utilizzare il Compilatore in modalità *'verbose'*, specificando l'opzione *-v*. In questa modalità oltre al classico output il Compilatore mostrerà a video alcuni dei simboli non terminali più significativi che ha riconosciuto nella costruzione dell'albero sintattico insieme all'arricchimento di attributi semantici che si hanno con le operazioni sulle *Symbol Tables*.