Linguaggi formali e compilazione

Corso di Laurea in Informatica

A.A. 2012/2013

Linguaggi formali e compilazione

Che cosa è un compilatore

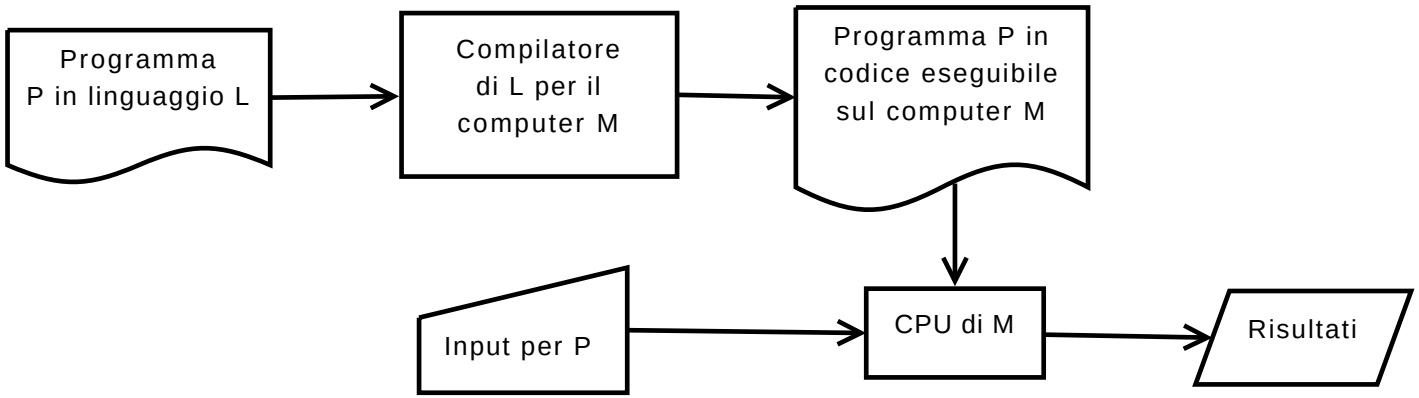
- Un compilatore è un *traduttore*.
- Questo vuol dire un *programma* che:
  - riceve in input la descrizione di un "oggetto" X scritto in un certo linguaggio L;
  - produce in output la descrizione di X in un altro linguaggio L'.
- La traduzione deve essere *corretta* nel senso che il *significato* (o *semantica*) dell'input deve essere preservato.

Traduttori in generale

- Alcuni semplici esempi:
  - un algoritmo che trasforma numeri dal sistema romano a quello posizionale decimale;
  - i programmi `ps2pdf`, `pdftotext`, `text-to-html`, ... disponibili in Linux;
  - un software che trasforma programmi scritti in C in programmi scritti in binario per l'architettura x86.
- Si pensi alla nozione di significato (e quindi alla correttezza) negli esempi menzionati.
- Solo nell'ultimo caso si parla propriamente di *compilazione*.

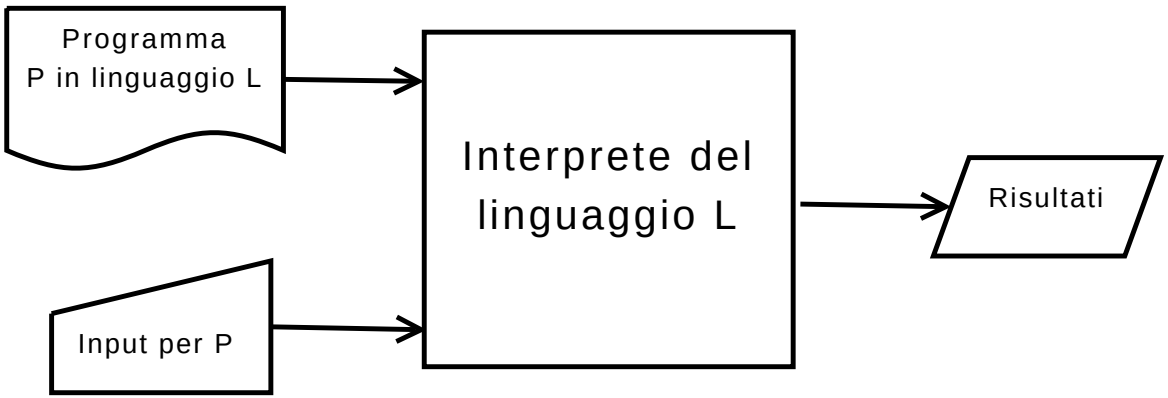
Compilazione ed esecuzione di programmi in linguaggio ad altro livello

- La compilazione è il più importante caso di traduzione in ambito informatico.
- Un *compilatore* per un linguaggio L e una macchina $\mathcal{M}$ è un traduttore che, data una stringa (programma) in linguaggio L, produce un programma "equivalente" nel linguaggio macchina di $\mathcal{M}$.
- Compilazione e successiva esecuzione di un programma



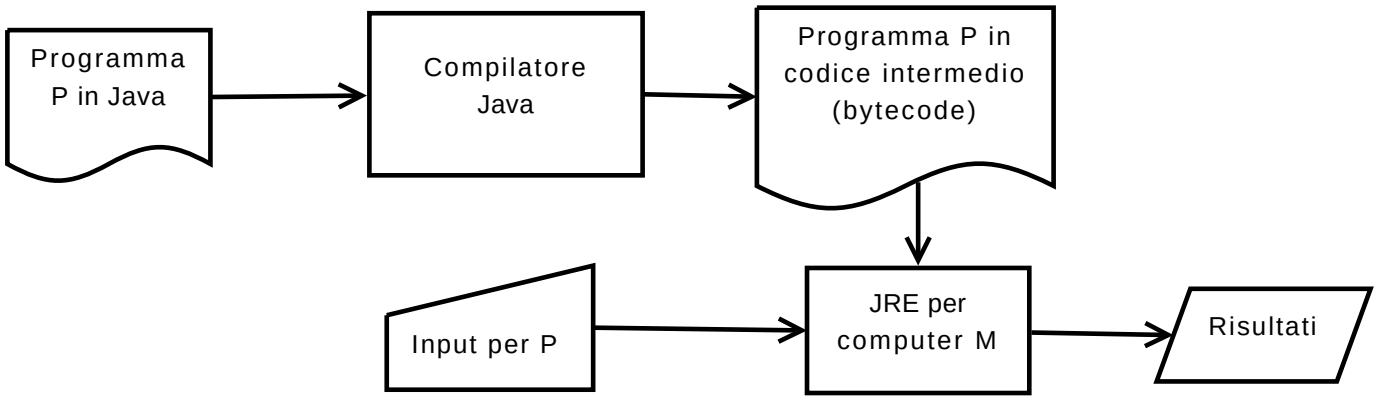
Uno schema alternativo: l'interpretazione pura

- Un'alternativa alla compilazione è l'*interpretazione* diretta dei programmi.
- Nel caso dell'interpretazione di L su $\mathcal{M}$, un programma in esecuzione su $\mathcal{M}$ prende direttamente in input frasi in linguaggio L e le "esegue", producendo in output il risultato dell'esecuzione.
- Lo schema per l'interpretazione pura



Esistono anche soluzioni “miste”

- Compilazione ed intepretazione in Java



- In tutti i casi (anche nell’interpretazione “pura”), le tecniche di compilazione giocano un ruolo fondamentale.

Inciso: linguaggi compilati e linguaggi interpretati

- In linea di principio, di uno stesso linguaggio L potrebbero esistere implementazioni compilate ed implementazioni interpretate.
- Nella pratica, i linguaggi "si specializzano" in una sola delle due alternative, anche (o forse soprattutto) perché compilazione ed interpretazione offrono vantaggi e svantaggi complementari, che sono poi riflessi nei linguaggi stessi.
- C, C++, Fortran e Pascal sono linguaggi compilati.
- I linguaggi dinamici (Perl, Python, PHP, ...) sono linguaggi interpretati.
- Il caso di Java.

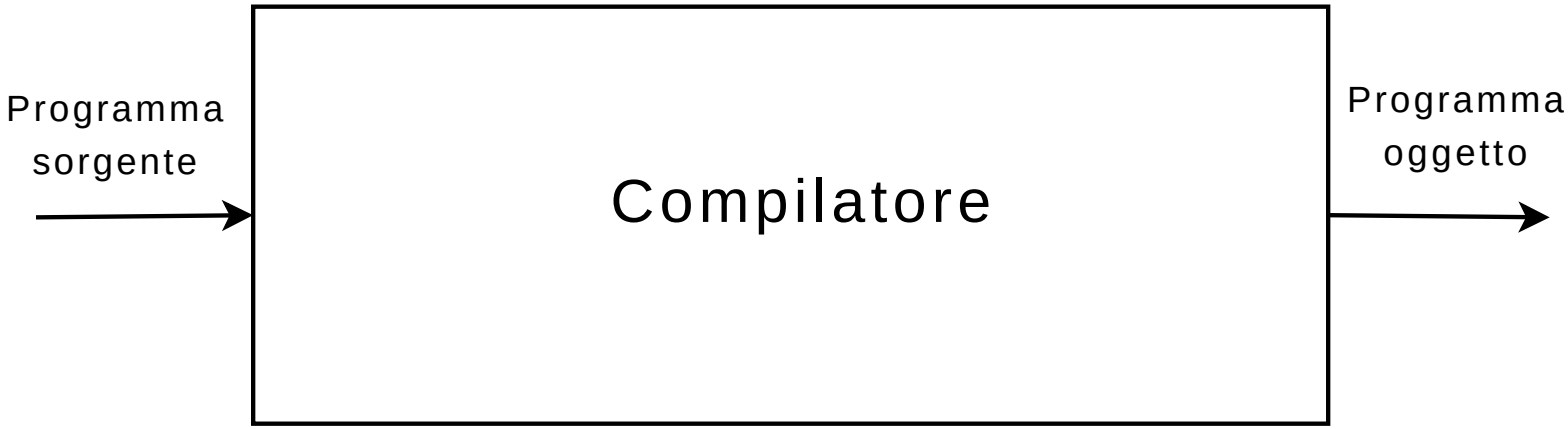
Ancora un inciso: traduzione automatica dei linguaggi naturali

- L'obiettivo più ambizioso nell'ambito della traduzione automatica sono le lingue "naturali" (italiano, inglese, ecc.)
- Da questo punto di vista i risultati sono modesti (chi non ha provato le traduzioni di Google?)
- Qualche successo nella traduzione automatica di manualistica e documentazione tecnica (eventualmente con un "post-processing" umano).
- Impossibile tradurre, ad esempio, un romanzo, una poesia o i testi di un film.
- Perché? È (solo) un problema di semantica?

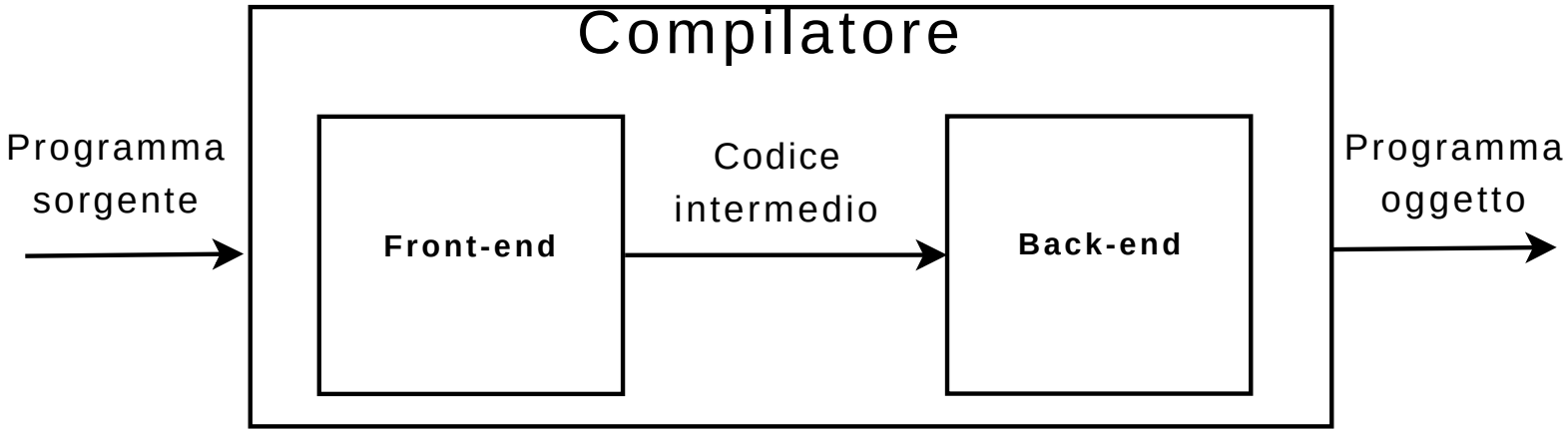
Traduzione automatica di linguaggi artificiali

- Esistono invece molti esempi di traduttori per linguaggi formali, cioè linguaggi che obbediscono a precise regole formali (*formalismi*) di composizione delle frasi.
- Abbiamo già citato esempi importanti in ambito informatico:
  - linguaggi di programmazione, come C, C++ e Java;
  - linguaggi di descrizione e/o manipolazione di dati, come XML e SQL;
  - linguaggi di descrizione di pagine quali *PostScript* e *Portable Document Format*.
- Linguaggi formali "manipolabili" in modo automatico esistono anche al di fuori dell'ambito informatico (si pensi ai formalismi sviluppati in Matematica e Chimica, ma anche per descrivere la musica).
- Da qui deriva l'importanza dello studio dei linguaggi formali.

Schema generale di un compilatore



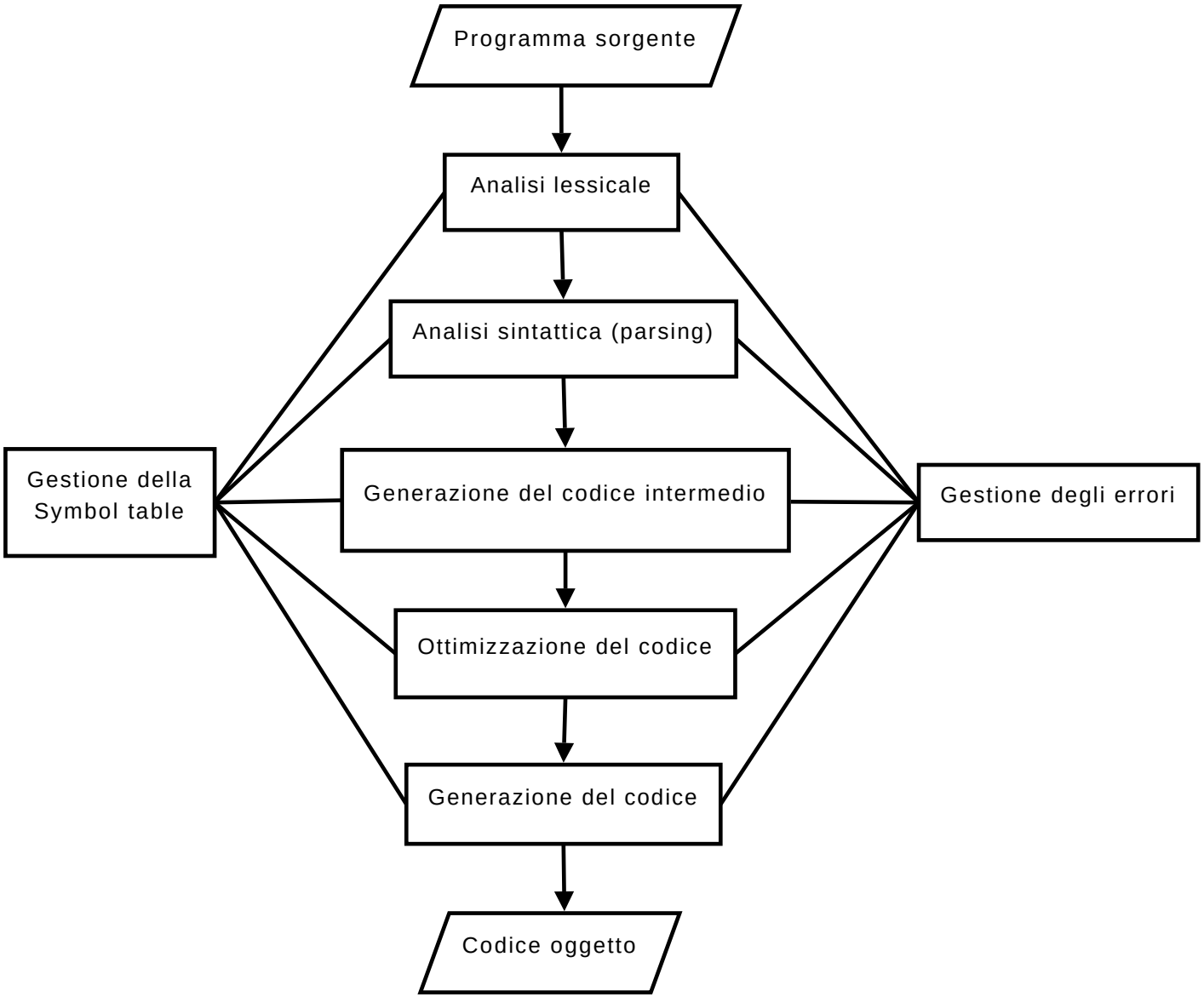
- Cominciamo a guardare "dentro la scatola" ...



Due componenti principali

- Il front-end rappresenta la fase di *analisi* dell'input.
- Il back-end è la fase di *sintesi* dell'output.
- Il codice intermedio è indipendente dall'architettura hardware (*i386*, *powerpc*, *sparc*, ...).
- Vantaggi di questa organizzazione:
  - modularità;
  - portabilità;
  - economicità.
- Noi ci occuperemo "esclusivamente" del front-end

Uno schema ancora più dettagliato



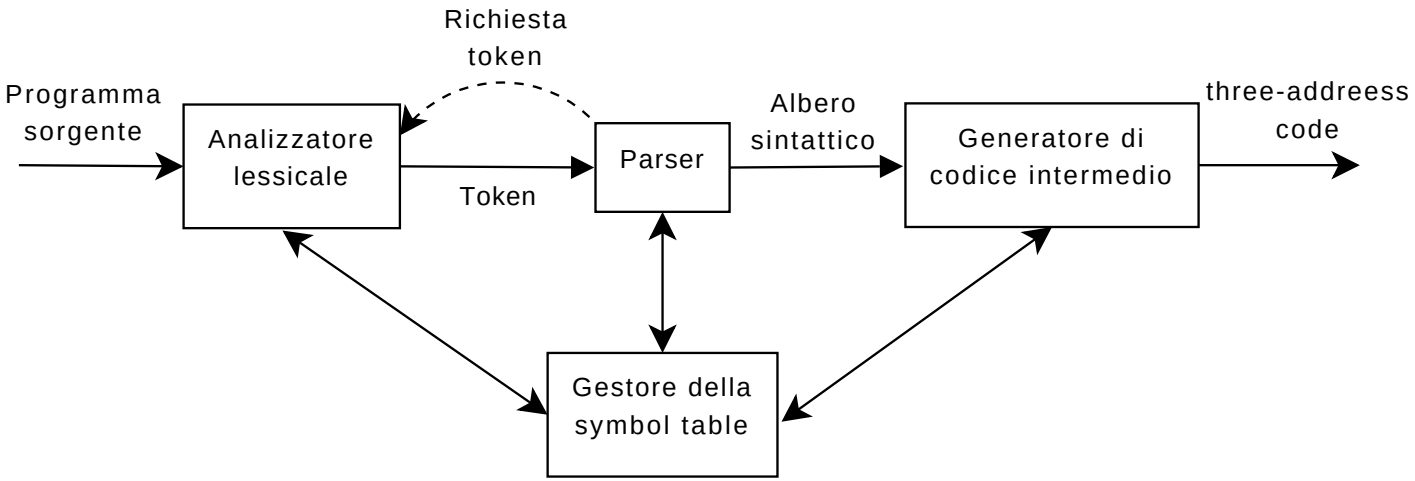
Schema tratto dal "Dragon book" (1977)

Il front-end più in dettaglio

- Il front-end di un compilatore (cioè la parte che termina con la creazione di una rappresentazione intermedia del programma) è costituita dai seguenti moduli:
  - analizzatore lessicale (scanner);
  - analizzatore sintattico (parser);
  - symbol table (e routine di gestione);
  - generatore di codice intermedio.
- Per ragioni di tempo non ci potremo occupare, in questo corso, della gestione degli errori.

Il modello di front-end

- La seguente figura illustra un tipico schema di organizzazione del front-end di un compilatore (tratto da Aho, Lam, Sethi, Ullman (2007)).



Analizzatore lessicale

- L'*analizzatore lessicale*, detto anche *scanner*, è l'unico modulo che legge il file di testo che costituisce l'input per il compilatore.
- Il suo ruolo è di raggruppare i caratteri in input in *token*, ovvero "oggetti" significativi per la successiva analisi sintattica.
- Esempi di token sono i numeri, gli identificatori e le parole chiave di un linguaggio di programmazione.
- Ad esempio, la sequenza di caratteri:

```
X = 3.14;
```

potrebbe venire trasformata nella sequenza di token:

id assign number sep

Analizzatore lessicale (continua)

- Ci sono però altri compiti che deve svolgere l'analizzatore lessicale:
  - riconoscere e "filtrare" commenti, spazi e altri caratteri di separazione;
  - associare gli eventuali errori trovati da altri moduli del compilatore (in particolare dal parser) alle posizioni (righe di codice) dove tali errori si sono verificati allo scopo di emettere appropriati messaggi diagnostici;
  - procedere all'eventuale espansione delle macro (se il compilatore le prevede).

Parser

- Il parsing opera esclusivamente al livello di *sintassi* (cioè della corretta formazione delle frasi).
- Un *parser* (detto appunto anche *analizzatore sintattico*) è uno strumento che, in termini generali, consente di *dare struttura ad una descrizione lineare*.
- Per questa ragione il parsing è un processo molto comune, anche in altre ambiti dell'Informatica.
- Alcuni esempi:
  - passare dalla descrizione testuale di un grafo ad una sua rappresentazione in liste di adiacenze;
  - dato un insieme di pagine web, creare un grafo che ne descriva la struttura dei collegamenti;
  - dato un documento XML, creare una rappresentazione dell'informazione in esso contenuta (detta *information set*) da rendere disponibile all'applicazione client.

Parser (continua)

- La struttura viene conferita alla descrizione lineare in input in accordo a regole formali.
- Nel caso dei linguaggi di programmazione tali formalismi sono generalmente *grammatiche*.
- L'output viene a sua volta specificato mediante un formalismo in grado di esprimere le *proprietà di struttura*.
- Generalmente tale formalismo è un *albero* (con varie proprietà).
- In tutti i casi, il parsing include un'analisi della *correttezza formale* (well-formedness) delle stringhe da analizzare.

Generazione del codice intermedio

- A partire dalla struttura prodotta dal parser e (come vedremo) dalle informazioni raccolte nella symbol table, questa fase produce un programma scritto in un linguaggio intermedi
- Il codice intermedio è indipendente dalla macchina e quindi è teoricamente portabile.
- Esistono diverse soluzioni per rappresentare tale codice e, fra queste, il cosiddetto *three-address code* (codice a tre indirizzi).

Breve descrizione delle altre fasi

Ottimizzazione del codice In questa fase operano algoritmi che *manipolano* il programma scritto in codice intermedio in modo da ottenere un programma equivalente che (in genere) utilizza una minore quantità di spazio e/o che "gira" più velocemente.

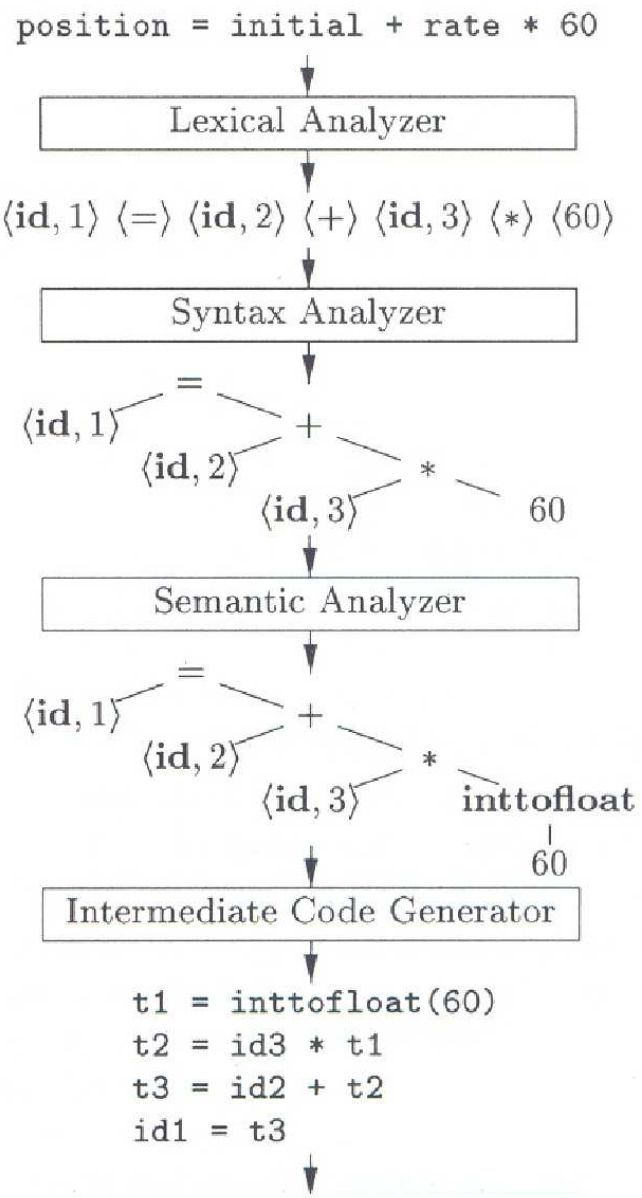
Generazione del codice In questa fase il programma in codice intermedio (eventualmente ottimizzato) viene trasformato in codice macchina. Qui vengono prese le decisioni sull'allocazione dei dati in memoria, il codice di accesso e l'utilizzo dei registri.

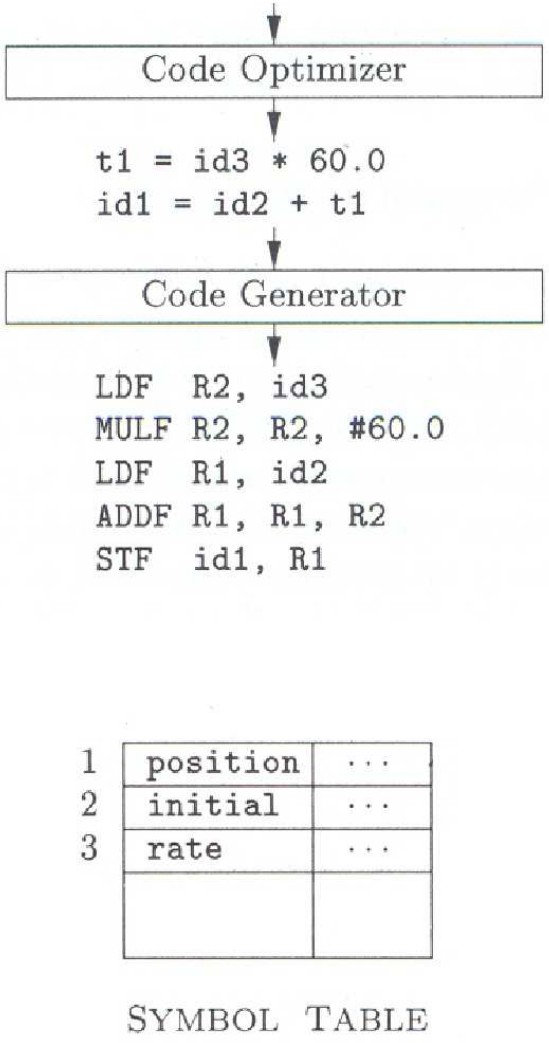
Gestione della tabella dei simboli È un'attività che accompagna tutte le fasi della compilazione. La *tabella dei simboli* è essenzialmente un dizionario che registra tutti i nomi usati nel programma e le informazioni ad esso associate (esempio, un identificatore e il suo tipo).

Breve descrizione delle fasi

Gestione degli errori Durante una qualunque delle fasi si possono verificare degli errori, che devono essere individuati e segnalati con opportuni messaggi diagnostici. Gli esempi più facilmente comprensibili di errore riguardano le fasi di analisi lessicale (ad esempio, un identificatore che contiene un carattere non ammesso) e di analisi sintattica (ad esempio un'espressione aritmetica mal formata). In generale se vengono rilevati errori non viene prodotto il codice oggetto.

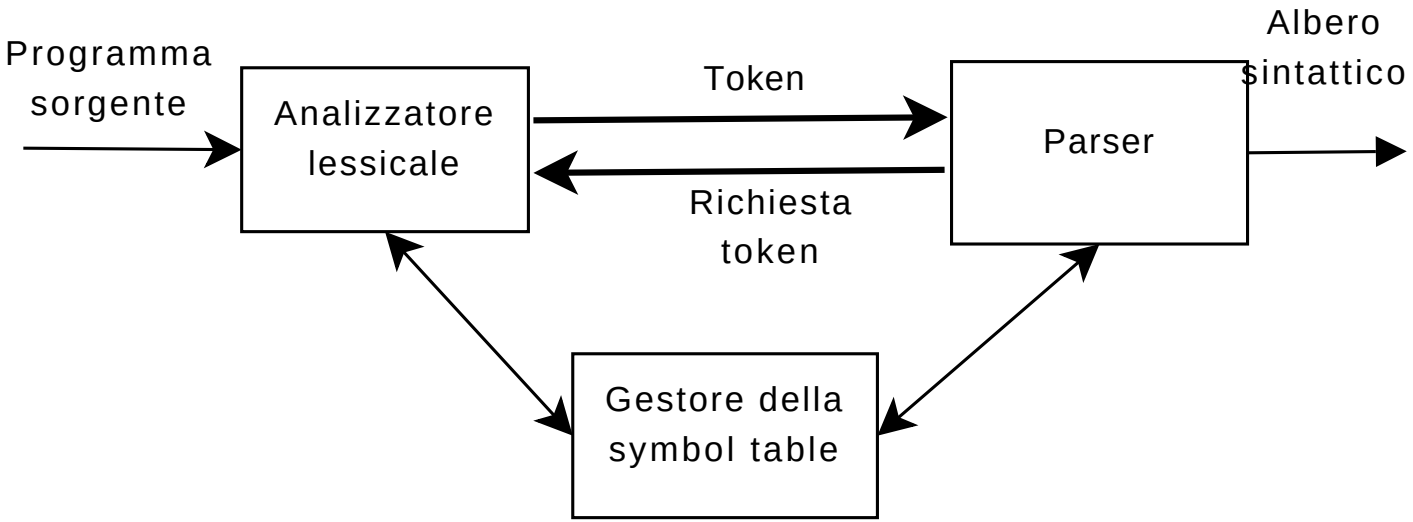
Le fasi attraverso un semplice esempio





Interazione parser-lexical analyzer

- Lo scanner opera come "routine" del parser.



- Quando il parser deve leggere il prossimo simbolo di input esegue una chiamata allo scanner.
- Lo scanner legge la "prossima" porzione di input fino a quando non riconosce un *token*, che viene restituito al parser.
- Ciò che viene propriamente restituito è un *token name*, una delle due componenti che costituiscono il token vero e proprio.

Token

- Un token è un oggetto astratto caratterizzato da due attributi, un *token name* (obbligatorio) e un *valore* opzionale.
- Ecco alcuni esempi di sequenze di caratteri riconosciuti come token da un compilatore C++:
  - "`0.314E+1`", cui corrisponde la coppia (token) in cui il nome è **number** e il valore è il numero razionale 3.14, opportunamente rappresentato;
  - "`x1`", cui corrisponde un token in cui il nome è **id** e il cui valore è, a sua volta, un insieme di informazioni elementari (la stringa di caratteri che forma l'identificatore, il suo tipo, il punto del programma dove è stato definito);
  - "else", cui corrisponde il solo token name **else**.

Token name

- Il nome del token serve principalmente al parser, per stabilire correttezza sintattica delle frasi (e costruire l'albero sintattico).
- Il valore dei token è invece, da questo punto di vista, non rilevante (almeno in prima istanza).
- Ad esempio, le frasi "`x<10`" e "`pippo!=35`" sono del tutto equivalenti e riconducibili alla frase generica in cui il valore di un identificatore è confrontato con un numero.
- Una tale frase può quindi essere astrattamente descritta come "**id comparison number**", dove abbiamo utilizzato il nome **comparison** per il token che descrive gli operatori relazionali.

Token value

- Il valore di un token gioca il suo ruolo principale nella traduzione del codice.
- Con riferimento agli esempi del lucido precedente:
  - di un identificatore è importante sapere (tra gli altri) il tipo, perché da questo dipende (in particolare) la quantità di memoria allocare;
  - di un operatore di confronto è necessario sapere esattamente di quale confronto si tratta, per poter predisporre il test e l'istruzione (macchina) di salto opportuna;
  - di una costante numerica è necessario sapere il valore per effettuare l'inizializzazione corretta.

Token (continua)

- È naturalmente il progettista del linguaggio che stabilisce quali debbano essere considerati token.
- Ad un token name possono corrispondere (in generale) molte stringhe alfanumeriche nel codice sorgente (si pensi ai token che descrivono numeri o identificatori).
- Il progettista deve quindi stabilire esattamente anche la *mappatura* fra stringhe e token name (cioè, quali stringhe corrispondono a quali token).
- Tale mappatura viene descritta utilizzando opportuni formalismi, il più importante dei quali è costituito (come vedremo) dalle *espressioni regolari*.

Un minimo di terminologia

- Le sequenze di simboli che possono legalmente comparire in un programma e che corrispondono ai token sono dette *lessemi*.
- Un *pattern* è invece una descrizione (formale o informale) delle stringhe (lessemi) che devono essere riconosciute (dall'analizzatore lessicale) come istanze di un determinato token name.
- Le espressioni regolari, delle quali ci occuperemo adesso, sono un formalismo per specificare tali pattern.

Pattern, lessemi e token name (continua)

- La seguente tabella illustra, mediante alcuni esempi, i concetti che stiamo analizzando.

Token name	Pattern	Esempio di lessema
id	*letter*(*letter* | *digit*)*	`pippo1`
number	[+ | -]*nzd digit** [.]*digit**	`-3.14`
comparison	`<, >, <=, >=, =, !=`	`<`
literal	[:*alpha*:]	"`Pi greco`"
if	`if`	`if`
while	`while`	`while`

Si noti che, per ragioni di spazio, il pattern che descrive il token **number** non prevede la notazione scientifica.

Che cosa è Lex

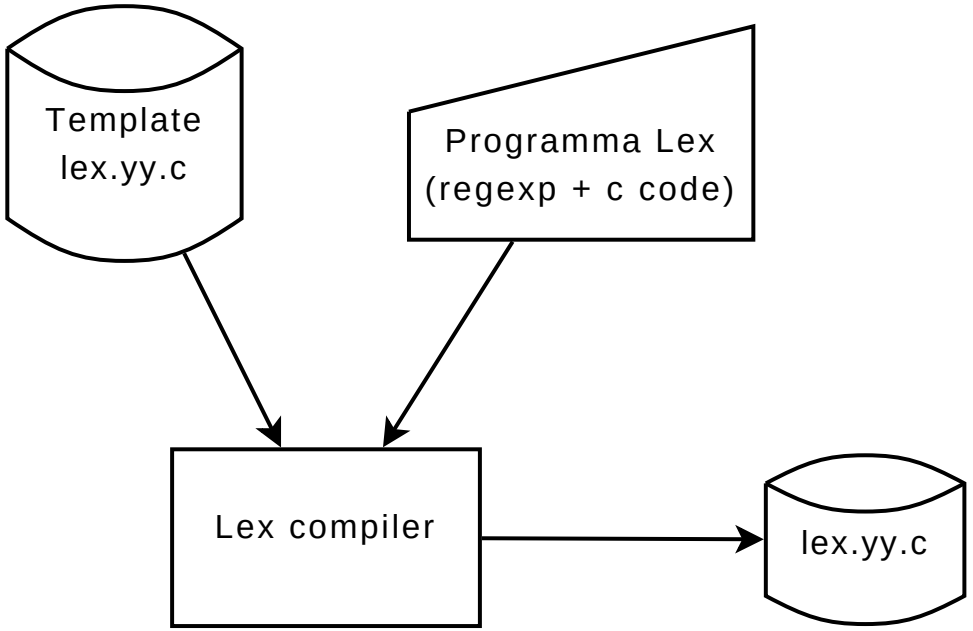
- *Lex* è un *generatore di scanner*.
- Si tratta cioè di un software in grado di generare automaticamente un altro programma che riconosce stringhe di un linguaggio regolare.
- Non solo, il software generato da Lex ha "capacità di scanning", cioè di acquisire le stringhe da analizzare in sequenza (da file o standard input) e di passare l'output ad un altro programma, tipicamente un parser.
- Lex può quindi essere uno strumento prezioso nella realizzazione di un compilatore.

Come funziona Lex

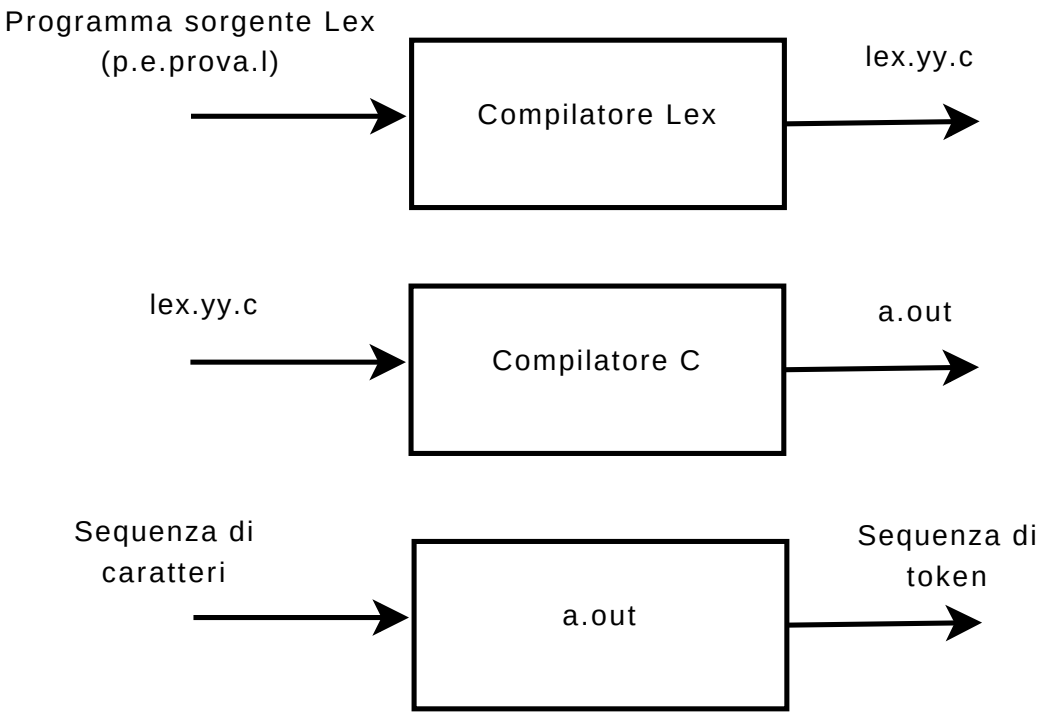
- L'input per un programma Lex è costituito "essenzialmente" da un insieme di espressioni regolari/pattern da riconoscere.
- Inoltre, ad ogni espressione regolare $\mathcal{E}$, l'utente associa un'*azione*, espressa sotto forma di codice C.
- Lex "trasforma" $\mathcal{E}$ nella descrizione di un "automa" che riconosce il linguaggio descritto da $\mathcal{E}$.
- Lex inoltre associa, ad ogni stato terminale dell'automa che riconosce $\mathcal{E}$, il corrispondente software fornito dall'utente.
- Nel programma generato da Lex, tale software andrà in esecuzione quando viene riconosciuto un lessema di $\mathcal{E}$.

Come funziona Lex (2)

- Lex inserisce la descrizione dell'automa e il codice fornito dall'utente in uno scheletro di programma (*template*) per ottenere così il programma finale "completo", per default chiamato `yy.lex.c`.
- La parte più "complessa" dal punto di vista teorico consiste proprio nella trasformazione delle espressioni regolari in automi.
- Noi sappiamo già "che cosa ci sta sotto"!



Schema d'uso di Lex



Un primo programma Lex

- Programma Lex per riconoscere (un sottoinsieme de) i token del linguaggio Pascal

```
delim [ \t\n]
ws {delim}+
letter [A-Za-z]
digit [0-9]
id {letter}({letter}|{digit})*
number [+-]?{digit}+(\.{digit}+)?(E[+-]?{digit}+)?
%%
{ws} { }
if {printf("IF ");}
then {printf("THEN ");}
else {printf("ELSE ");}
{id} {printf("ID ");}
{number} {printf("NUMBER ");}
"<" {printf("RELOP ");}
"<=" {printf("RELOP ");}
"=" {printf("RELOP ");}
"<>" {printf("RELOP ");}
">" {printf("RELOP ");}
">=" {printf("RELOP ");}
":=" {printf("ASSIGNMENT ");}
%%
main()
{ yylex();
  printf("\n"); }
```

Esecuzione del programma

- Se mandiamo in esecuzione il programma C ottenuto dopo le due compilazioni (si rammenti lo schema d'uso) con il seguente input:

```
if var1<0.0 then
     sign := -1
     else sign := 1
```

otteniamo come output la sequenza di token

IF ID RELOP NUMBER THEN ID ASSIGNMENT NUMBER ELSE ID ASSIGNMENT NUMBER

in realtà poiché i token name vengono internamente rappresentati mediante interi, la sequenza le output potrebbe essere:

12 2 6 1 13 2 18 1 14 2 18 1

Struttura generale di un programma Lex

- Un generico programma Lex contiene tre sezioni, separate dal dalla sequenza %%

```
Dichiarazioni
%%
Regole di traduzione
%%
Funzioni ausiliarie
```

- La sezione "Dichiarazioni" può contenere definizione di costanti e/o variabili, oltre alle cosiddette *definzioni regolari*, cioè espressioni che consentono di "dare un nome" ad espressioni regolari.
- La sezione più importante è quella relativa alle regole di traduzione che contiene le descrizioni dei pattern da riconoscere e, corrispondentemente, le azioni che devono essere eseguite dallo scanner.

Struttura generale di un programma Lex (continua)

- L'ultima sezione può contenere funzioni aggiuntive (che vengono tipicamente invocate nella parte relativa alle regole di traduzione).
- Se lo scanner non è utilizzato come routine del parser o di altro programma, quest'ultima sezione contiene anche il main program.
- Se presente, il main conterrà ovviamente la chiamata allo scanner (funzione `yylex`).

Non solo riconoscimento di token

- Il seguente programma Lex conta caratteri, parole e linee presenti in un file (assimiglia dunque al programma `wc` di Unix/Linux).

```
%{
unsigned charCount = 0, wordCount = 0, lineCount = 0;
%}

word [^ \t\n]+
eol \n

%%

{word}      {wordCount++; charCount += yyleng; }
{eol}       {charCount++; lineCount++}
.           charCount++;

%%
main()
{ yylex();
  printf("%d %d %d\n", lineCount, wordCount, charCount);
```