

## Linguaggi di Programmazione (Analisi Lessicale e Sintattica)



Sono le prime due fasi di un compilatore

Lucidi basati su materiale  
in inglese di Ras Bodik

<http://inst.eecs.berkeley.edu/~cs164/fa04>

# Three kinds of execution environments

• Traduzione a Run-Time

## Interpreters

- Scheme, lisp, perl
- popular interpreted languages later got compilers

Traduzione prima del Run-Time

## Compilers

- C
- Java (compiled to bytecode)

Traduce le righe di codice da ambiente virtuale per il processore fisico

## Virtual machines

- Java bytecode runs on an interpreter
- interpreter often aided by a JIT compiler

Il codice è pre-compilato in un bytecode (codice intermedio). La VM, che è un programma installato sulla macchina reale, agisce da interprete per questo bytecode.

Il lessico è l'insieme delle parole di una lingua, la sintassi studia le regole che governano la struttura e l'ordine delle parole nelle frasi, mentre la semantica si occupa del significato di queste parole e di come esse si combinano per creare un senso compiuto.

Il Front-End di un Compilatore è composto dalle prime tre fasi

## The Structure of a Compiler

Analogia Umana: Riconoscimento delle parole.

Analisi Lessicale

### 1. Scanning (Lexical Analysis)

Analisi Sintattica

### 2. Parsing (Syntactic Analysis)

Analisi Semantica

### 3. Type checking (Semantic Analysis)

### 4. Optimization

Miglioramento del codice intermedio per renderlo più veloce o più piccolo, senza alterarne la funzionalità.

### 5. Code Generation

(Il Compilatore è un traduttore: se il check del front-end è corretto, si esegue ottimizzazione e traduzione; se non è corretto, il compilatore si ferma e non continua con la traduzione)

Analogia Umana:  
Riconoscimento della struttura grammaticale della frase.

Analogia Umana:  
Comprensione del significato.

Il scanner legge il codice sorgente come una sequenza grezza di caratteri e li raggruppa in unità significative chiamate token.

Il parser prende il flusso di token generato dall'Analisi Lessicale e verifica se la loro sequenza rispetta la grammatica (le regole sintattiche) del linguaggio di programmazione.

Questa fase aggiunge significato e contesto. Il compilatore verifica se le strutture sintatticamente corrette hanno anche senso dal punto di vista del linguaggio (es. compatibilità dei tipi, variabili dichiarate).

The first 3, at least, can be understood by analogy to how humans comprehend English.

## 1

# Lexical Analysis

Divide il programma in token e controlla se sono corretti

---

- Lexical analyzer divides program text into “words” or “tokens”

if x == y then z = 1; else z = 2;

- Units:

if, x, ==, y, then, z, =, 1, ;, else, z, =, 2, ;

## 2 Parsing

(Cercare la struttura delle frasi) crea un albero sintattico, in base ai token trovati nella fase 1

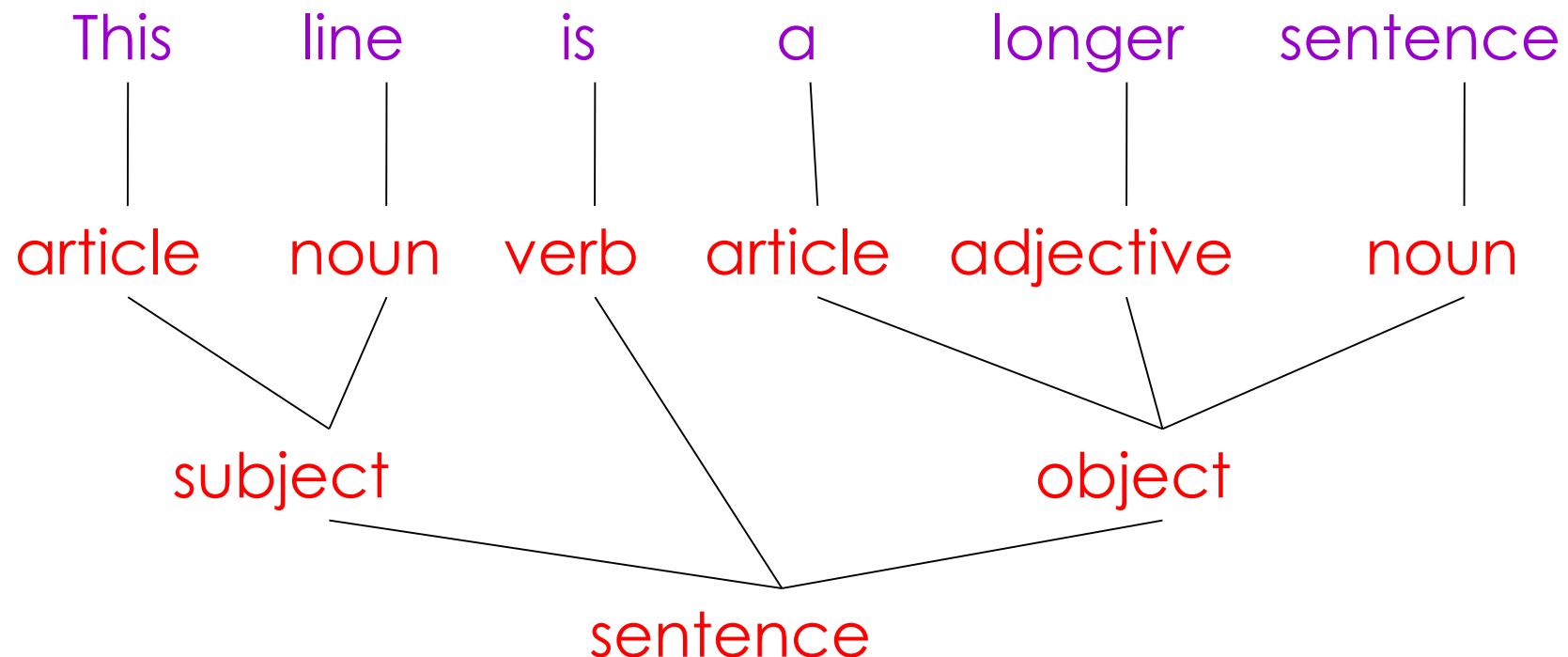
---

- Once words are understood, the next step is to understand sentence structure
- Parsing = Diagramming Sentences
  - The diagram is a tree

Cercare la struttura nelle frasi del linguaggio parlato umano,  
é uguale a farlo con il linguaggio di programmazione

## Diagramming a Sentence

(Parser Bottom-Up, anche se viene mostrato al contrario, cioè la radice si trova in basso)

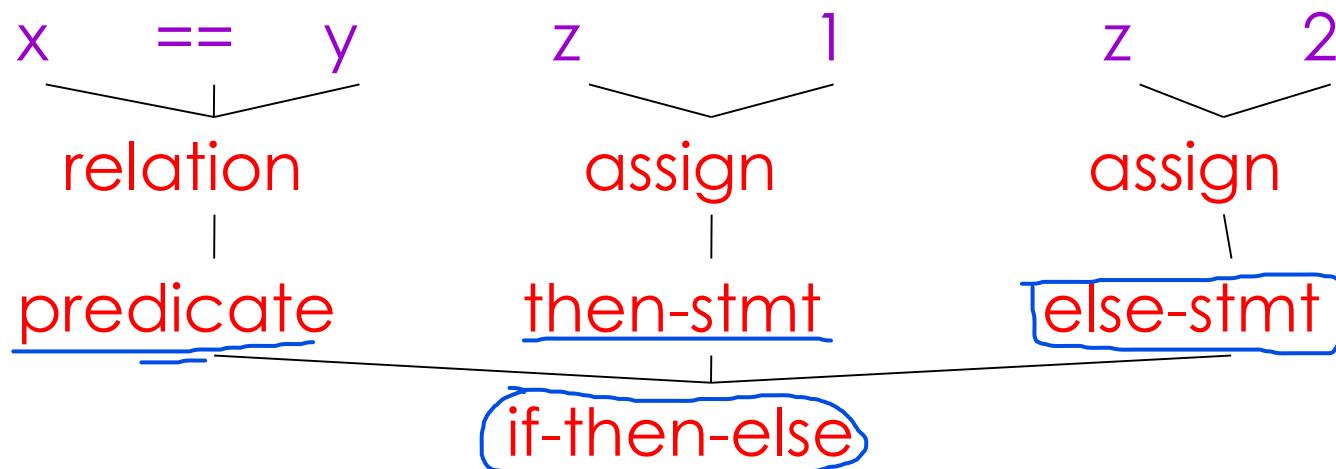


## Parsing Programs

- Parsing program expressions is the same
- Consider:

If x == y then z = 1; else z = 2;

- Diagrammed:



3

## Semantic Analysis in English

Type Checking

---

- Example:

Jack said Jerry left his assignment at home.  
What does “his” refer to? Jack or Jerry?

- Even worse:

Jack said Jack left his assignment at home?  
How many Jacks are there?  
Which one left the assignment?

# Semantic Analysis I

Legare l'uso delle variabili alle sue dichiarazioni (ci devono essere delle regole: non esiste il contesto come nelle lingue parlate in cui si interpreta in base all'esperienza)

- Programming languages define strict rules to avoid such ambiguities

- This C/C++ code prints "4"; the inner definition is used

In Java, puoi dichiarare una sola volta una variabile (quindi, in questo caso, avrebbe dato errore)

```
{  
    int Jack = 3;  
    {  
        int Jack = 4;  
        printf("%d", Jack);  
    }  
}
```

1. Risoluzione dei Nomi e Controllo delle Dichiarazioni
  - Risoluzione dei Nomi: Determina a quale specifica dichiarazione (variabile, funzione, classe) si riferisce ogni nome nel codice.
  - Controllo delle Dichiarazioni: Verifica che ogni entità utilizzata (variabile, funzione) sia stata effettivamente dichiarata prima di essere usata.
  - Gestione dello Scoping: Costruisce e aggiorna le Tabelle dei Simboli man mano che il codice viene analizzato, associando attributi (tipo, indirizzo, ecc.) ai nomi.
2. Controllo dei Tipi (Type Checking)
  - Verifica della Compatibilità: Controlla che gli operandi di un'operazione abbiano tipi compatibili (es. non sommare una stringa con una funzione).
  - Controllo delle Assegnazioni: Assicura che un valore sia assegnato a una variabile del tipo corretto.
  - Controllo della Firma delle Funzioni: Verifica che il numero e i tipi degli argomenti passati a una funzione corrispondano a quelli attesi dalla sua definizione.

## Semantic Analysis II

---

- Compilers also perform checks to find bugs
- Example:  
Jack left her homework at home.
- A “type mismatch” between her and Jack
  - we know they are different people  
(presumably Jack is male)

## Code Generation

---

- A translation into another language
  - Analogous to human translation
- Compilers for C, C++, ...
  - produce assembly code (typically)
- (e.g. GUI) code generators
  - produce C, C++, Java, ...

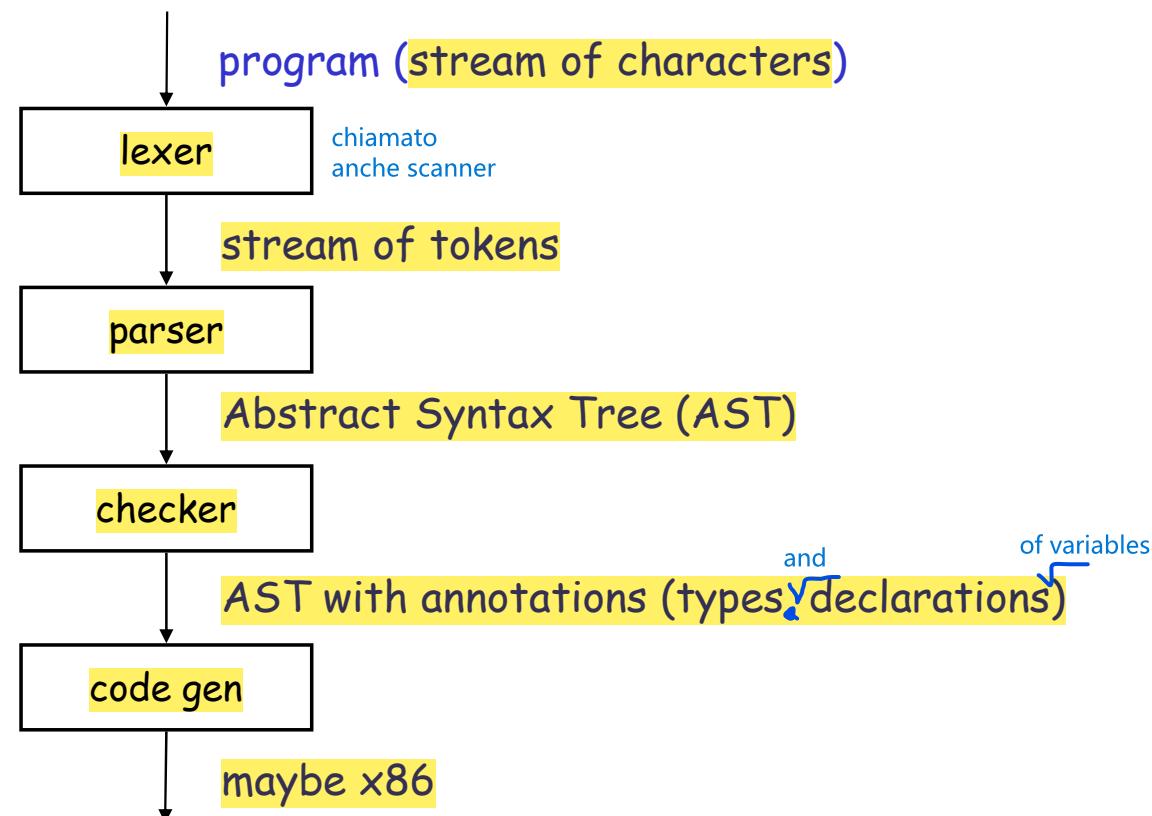
- Assembly Code: linguaggio a basso livello specifico per l'architettura della CPU (ancora rappresentazione testuale)  
- Codice Oggetto: traduzione finale del codice sorgente (o dell'assembly) in una rappresentazione binaria che il processore può eseguire direttamente.

Quindi, L'Assembly è la versione testuale, simbolica e leggibile dall'uomo; il Codice Oggetto è la versione binaria, numerica e leggibile dalla macchina.

Il Modulo di Generazione del Codice è un plug-in: si riferisce alla filosofia di progettazione del compilatore, nota come architettura a tre fasi (Front-end, Mid-end, Back-end):

- Front-end: Legge il codice sorgente e crea l'AST.
- Mid-end: Esegue l'analisi semantica e le ottimizzazioni sull'AST. È indipendente dal linguaggio sorgente e dall'architettura target.
- Back-end (Plug-in): Include la fase di Generazione del Codice. Questo modulo prende l'IR (Rappresentazione intermedia) ottimizzato e lo traduce in codice macchina specifico per una data CPU (es. x86).

## Summary: The Structure of a Compiler



L'Albero Sintattico Astratto (AST) è la rappresentazione interna, gerarchica e pulita del codice sorgente. È il risultato diretto delle fasi di Analisi Lessicale e Sintattica.

L'AST è la rappresentazione indipendente dalla destinazione del codice sorgente, mentre la Generazione del Codice è la fase dipendente dalla destinazione ed è strutturata come un modulo intercambiabile per garantire flessibilità e portabilità.

---

**Building a lexer**

## Recall: Lexical Analysis

- The input is just a sequence of characters. Example:

```
if (i == j)  
    z = 0;  
else  
    z = 1;
```

- More accurately, the input is string:

```
\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
```

Goal: find lexemes and map them to tokens:

- partition input string into substrings (called lexemes), and
- classify them according to their role (role = token)

I Lessemi sono sottostringhe che matchano con il lessico del linguaggio. Ai lessemi vengono attribuiti dei ruoli (token)

## Continued

---

- **Lexer input:**

\t if(i==j)\n\t\tz = 0;\n\telse\n\t\tz = 1;

- **partitioned into these lexemes:**

\t if(\t i == \t j)\n\t\tz = \t 0;\n\telse\n\t\tz = \t 1;

- **mapped to a sequence of tokens**

IF, LPAR, ID("i"), EQUALS, ID("j") ...

- **Notes:**

- whitespace lexemes are dropped, not mapped to tokens

- some tokens have **attributes**: the **lexeme** and/or **line number**

- why do we need them?

Abbiamo bisogno degli attributi per due motivi principali: l'Analisi Semantica e il Debug.

1: Gli attributi contengono l'informazione specifica che trasforma un tipo di token astratto in un'entità concreta che può essere analizzata, memorizzata e, infine, tradotta in codice macchina.

2: Includendo il numero di riga come attributo, il compilatore può generare messaggi chiari e utili: "Errore sintattico: Punto e virgola mancante alla riga 5."

---

## How to build a lexer?

È difficile e noioso da costruire in modo imperativo (es:  
ad ogni modifica del linguaggio, devo modificare il lexer)

## Writing the lexer

---

- Not by hand
  - tedious, repetitious, error-prone, non-maintainable
- Lexer generator!
  - once we have the generator, we'll only describe the lexemes and their tokens ...
    - that is, provide language's lexical specification (the What)
  - ... and generate code that performs the partitioning
    - generated code hides repeated code (the How)

 codice sempre uguale, indipendente  
da linguaggio di programmazione

## Code generator: key benefit

---

- The lexer generator allows the programmer to focus on:
  - What the lexer should do,
  - rather than How it should be done.
- what: declarative programming
- how: imperative programming

## Imperative lexer (in Java)

---

- Let's first build the lexer in Java, by hand:
  - to see how it is done, and where's the repetitious code that we want to hide e soprattutto mostrare perché è tedioso e difficile
- A simple lexer will do. Only four tokens:

TOKEN	Lexeme
ID	a sequence of one or more letters or digits starting with a letter
EQUALS	"=="
PLUS	"+"
TIMES	"*"

## Imperative lexer

---

```
c=nextChar();
if (c == '=') { c=nextChar(); if (c == '=') {return EQUALS;}}
if (c == '+') { return PLUS; }
if (c == '*') { return TIMES; }
if (c is a letter) {
    c=NextChar();
    while (c is a letter or digit) { c=NextChar(); }
    undoNextChar(c);
    return ID;
}
```

## Maximal match rule

- Il lexer legge il carattere extra (il Look-Ahead).
- Se il Look-Ahead non fa parte del lessema corrente, il lexer conferma il lessema.
- Per non perdere quel carattere (che appartiene al prossimo lessema o serve da separatore), l'operazione `undoNextChar()` riporta indietro l'indicatore del flusso di caratteri di una posizione.

- What is the need for `undoNextChar()`?

↳ Funzionamento: Il lexer deve leggere un carattere extra per determinare se il lessema corrente è terminato.

↳ it performs look-ahead, to determine whether the ID lexeme can be grown further

Per implementare la Regola del Match Massimale, il lexer deve spesso guardare oltre la fine effettiva del lessema. Questa è l'operazione di Look-Ahead.

- This is an example of maximal match rule:

- this rule followed by all lexers

- the rule: the input character stream is partitioned into lexemes that are as large as possible

↳ Ex.: in Java, “iffy” is not partitioned into “if” (the IF keyword) and “fy” (ID), but into “iffy” (ID)

Match Non Massimale: Se si fermasse a if, l'input iffy sarebbe scomposto in IF (token parola chiave) e poi fy (token ID), un errore grave.

Match Massimale (Corretto): Il lexer continua fino a che non può più crescere (fino a quando il carattere successivo non invalida il lessema o non esiste). In questo caso, iffy è il lessema valido più lungo e viene tokenizzato come un singolo ID("iffy").

Ogni volta che il lexer incontra una sequenza di caratteri che può formare un lessema valido, deve continuare a leggere il più a lungo possibile per formare il lessema più lungo e valido.

## Imperative Lexer: what vs. how

---

```
c=nextChar();
if (c == '=') { c=nextChar(); if (c == '=') {return EQUALS;}}
if (c == '+') { return PLUS; }
if (c == '*') { return TIMES; }
if (c is a letter) {
    c=NextChar();
    while (c is a letter or digit) { c=NextChar(); }
    undoNextChar(c);
    return ID;
}
```

- little logic....

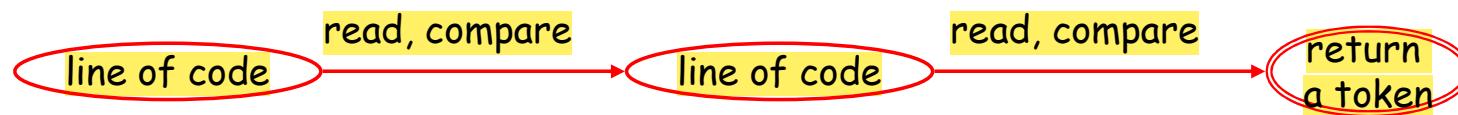
## Separate out the what

tramite DFA

```
c=nextChar();
if (c == '=') {c=nextChar(); if (c == '=') {return EQUALS;}}
if (c == '+') { return PLUS; }
if (c == '*') { return TIMES; }
if (c is a letter) {
    c=NextChar();
    while (c is a letter or digit) { cNextChar(); }
    undoNextChar(c);
    return ID;
}
```

## Separate out the **what**

- Is there a computational model that follows this behaviour?
  - yes, finite automata! DFA



- The code actually follows this simple pattern:
  - read next character and compare it with some predetermined character
  - if there is a match, jump to a different line of code
  - repeat this until you return a token.

Legge prossimo carattere ed in base a quello decido cosa fare e nel caso ritorna il risultato finale (token). Dopo avere ritornato il token, ritorno nello stato iniziale del DFA

## A declarative lexer

### Part 1: declarative (the what)

- describe each token as a finite automaton (or, better, as a regular expression) Ogni token viene definito da un DFA, o meglio da una Espressione Regolare.
  - must be supplied for each lexeme, of course (it specifies the lexical properties of the input language)

→ Oltre alla combinazione dei vari DFA (unione dei DFA ad uno stato iniziale), vengono aggiunti dei comandi negli stati di accettazione di ogni DFA

### Part 2: imperative (the how)

- connect these automata into a lexer automaton
- common to all lexers (like a library)
  - responsible for the mechanics of scanning

Il software generatore di lexer prende tutti gli automi (i DFA) generati dalle RE e li combina in un unico Automa Globale che riconosce simultaneamente tutti i possibili token.

In sintesi, il modello dichiarativo permette al progettista del linguaggio di concentrarsi solo sul vocabolario (RE), lasciando al motore imperativo (DFA combinato) il compito di gestire in modo efficiente l'intera logica di scansione.

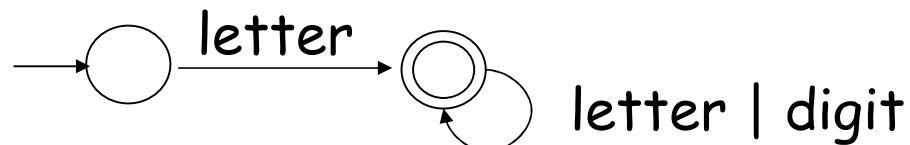
---

**the declarative lexer**

## Part 1: specify a FA for each token

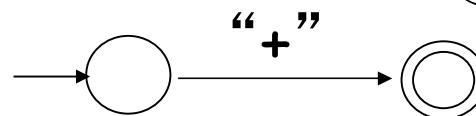
---

ID:



letter | digit

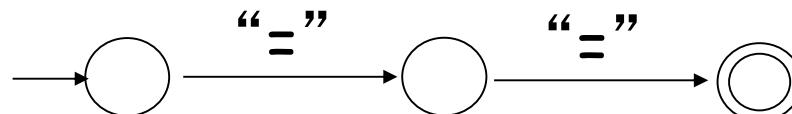
PLUS:



TIMES:



EQUALS:



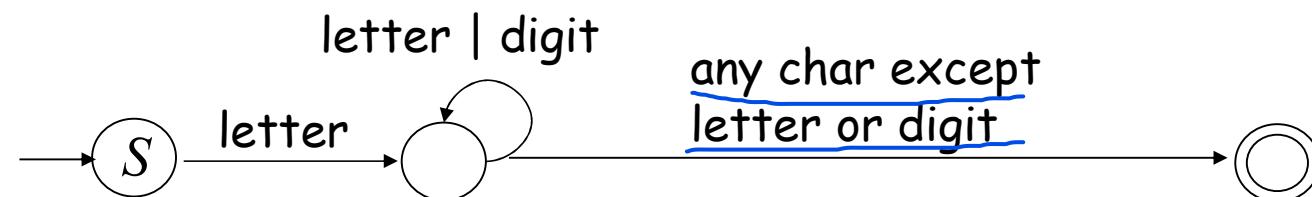
## Part 2: add actions on acceptance of the FA

---

- the action can be one of
  - "put back one character" (if look-ahead is needed)
  - "return token XYZ"

## Part 2: example of extending a FA

- The DFA recognizing identifiers is modified to:



action:

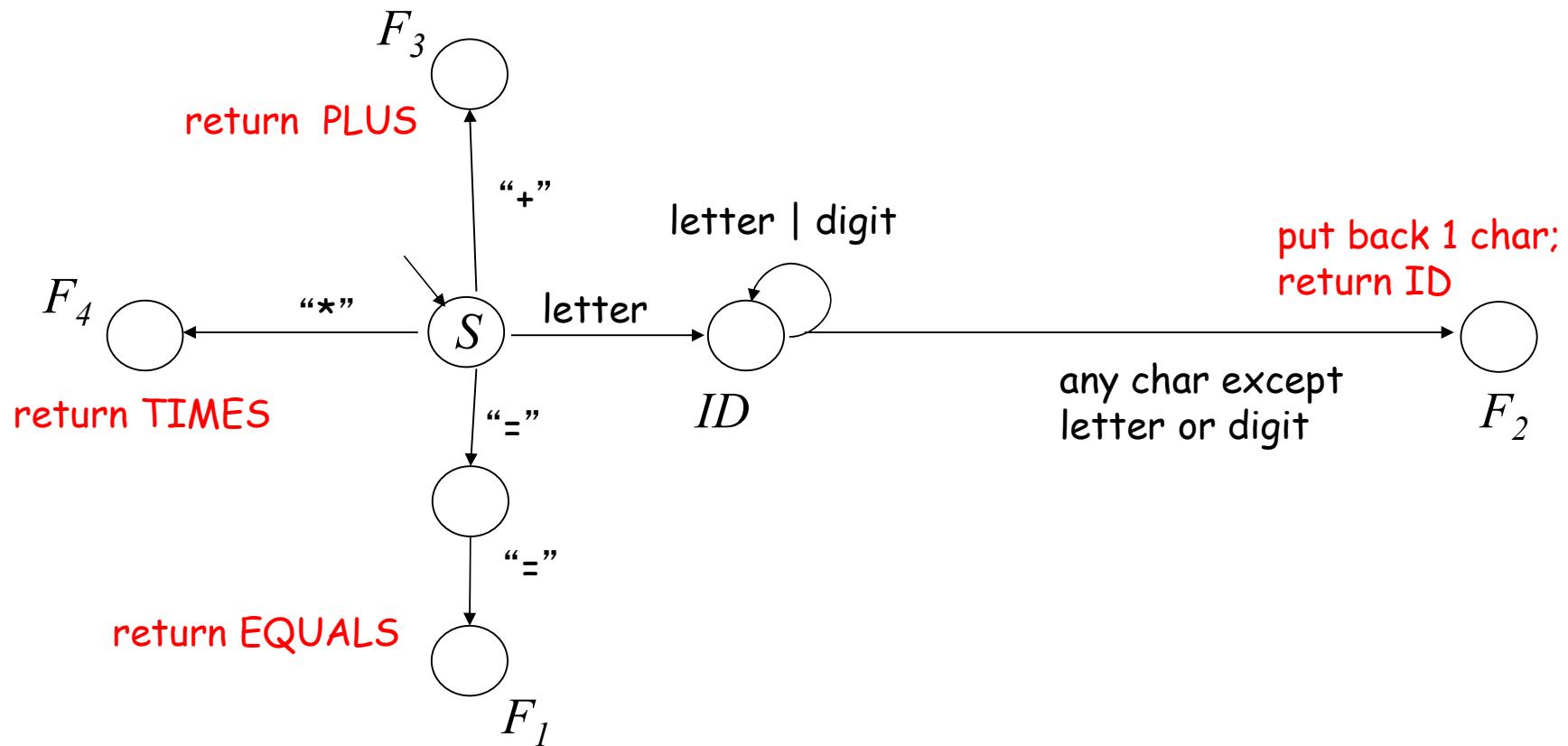
- put back 1 char
- return ID

Vedremo più avanti che non si applica solo ai lessemi con lunghezza variabile

- Look-ahead is added for lexemes of variable length
  - in our case, only ID needs lookahead
- A note on action “**return ID**”
  - resets the lexer back into start state S (recall that lexer is called by parser; each time, one token is returned) ricorda che

## Part 2: Combine the extended FA's

The algorithm: merge start states of the DFA's.



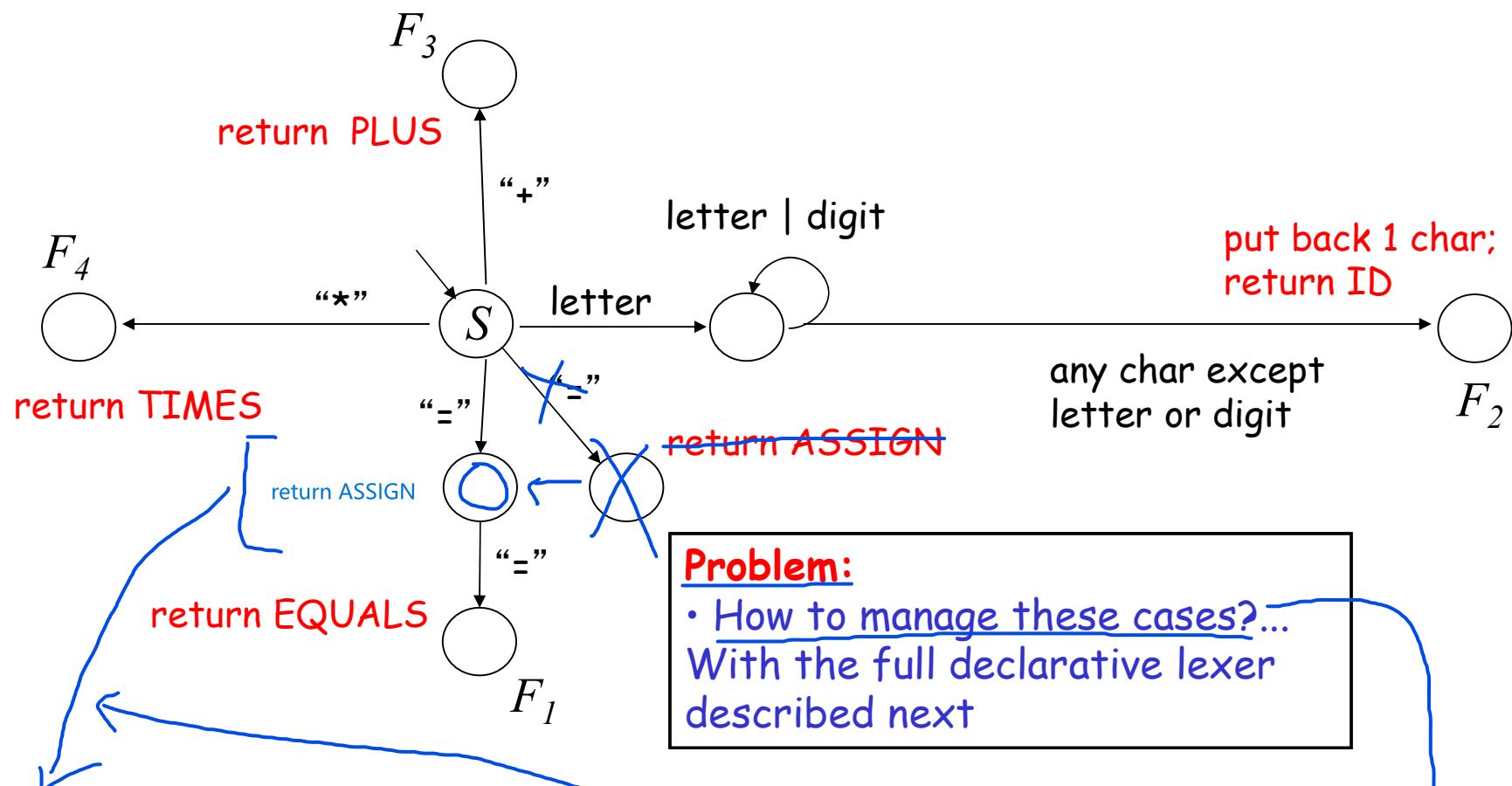
## Towards a realistic scanner

- Consider a fifth token type, for the assignment operator

TOKEN	Lexeme
ID	a sequence of one or more letters or digits starting with a letter
EQUALS	“==”
ASSIGN	“=”
PLUS	“+”
TIMES	“*”

Quindi il Maximal Match si applica sia ai lessemi di lunghezza variabile sia alle relazioni tra lessemi di lunghezza statica

## But above algorithm produces



32  
Inoltre, dobbiamo rendere il DFA da non deterministico a deterministico. Rendendolo deterministico però appunto dovremmo aggiungere anche il Maximal Match

---

**full declarative lexer**

## The algorithm

---

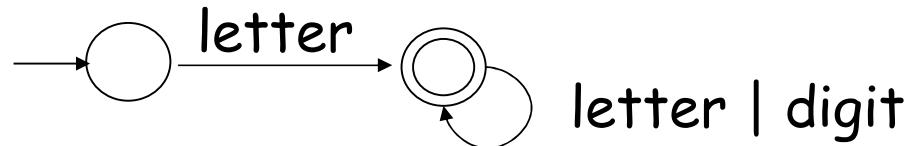
We can develop the lexer as follows:

1. Specify an automaton for each lexeme (as before)

## Step 1 in our example

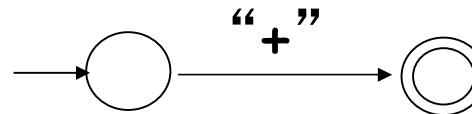
---

ID:



letter | digit

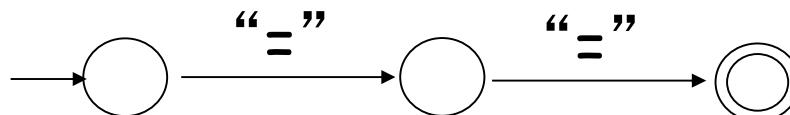
PLUS:



TIMES:



EQUALS:



ASSIGN:



## The algorithm

L'utente non si occupa dell'estensione (é l'algoritmo che applica maximal match)

We can develop the lexer as follows:

1. Specify an automaton for each lexeme (as before)
2. Merge them (start states) into the lexer automaton
3. If the obtained automaton is nondeterministic translate into a deterministic automaton
4. Consider the following rules (implementing maximal match):
  - whenever you reach a final state: (quindi, non si ferma più agli stati di accettazione)
    - remember position in input (so that you can undo reads)
    - keep reading more characters, moving to other states
  - whenever you get stuck (cannot make a move on next char):
    - return to the last final state (i.e., undo the reads)
    - return the token associated with this final state

(ricordare la posizione dell'input  
nello stream degli input)

## Notes

(Viene applicato il look-ahead fino a che l'algoritmo si blocca)

---

- Notice that reading past final state implements look-ahead
- This look-ahead is unbounded
  - can undo any number of characters

## Practical concerns

- Ambiguity
  - problem: lexer may reach multiple final states at once
  - ex.: "if" matches both ID and IF (the keyword)
  - solution: prioritize tokens (IF wins over ID) dare ordine di priorità ai token
- Discarding whitespace spazi, tabulazioni, newline
  - solution: final state for white-space lexeme is special: don't return a token, just jump to (common) start state alla fine del lessema degli spazi vuoti, non ritornare nulla, ma salta e ritorna allo stato iniziale del DFA combinato
- Error inputs
  - problem: discard illegal lexemes and print an error message
  - simple solution (discard char by char): add a lexeme that matches any character, giving it lowest priority; it will match when no other will

Si definisce una regola lessicale "catch-all" (cattura-tutto) che corrisponde a qualsiasi singolo carattere (es. con la RegEx ., dot). Questa regola riceve la priorità più bassa di tutte. Se l'automa non riesce a far corrispondere l'input a un lessema valido (ID, IF, NUM, ecc.), cadrà per forza su questa regola catch-all. L'azione associata a questa regola è: stampare un messaggio di errore e scartare il carattere, per poi ripartire dallo stato iniziale.

le Parole Chiave hanno una priorità maggiore rispetto agli Identificatori.

## We have a full declarative scanner

---

- imperative part,
  - The steps 2, 3, and 4 of the algorithm
- declarative part
  - Specify each token with an automaton

But how to specify the automata in practice?

---

**regular expressions**

## Regular Expressions

---

- Automaton is a good “visual” aid
  - but is not suitable as a specification  
(its textual description is too clumsy)
- regular expressions are a suitable specification
  - a compact way to define a language that can be accepted by an automaton.
- used as the input to a lexer generator
  - define each token, and also
  - define white-space, comments, etc
    - these do not correspond to tokens,  
but must be recognized and ignored.

## Example: identifier

- Lexical specification (in English):
  - a letter, followed by zero or more letters or digits.
- Lexical specification (as a regular expression):
  - $[a-zA-Z]. ([a-zA-Z] | [0-9])^*$

$[a-zA-Z]$  concatenato ( $[a-zA-Z]$  oppure  $[0-9]$ ) stella di clini

	means "or"	(nelle RE teoriche, sarebbe il '+')
.	means "followed by"	concatenazione
*	means zero or more instances of	stella di clini
()	are used for grouping	per dare precedenze

## Operands of a regular expression

---

- Operands are same as labels on the edges of an FA
  - single characters, or
  - the special empty string character  $\epsilon$   
(in ANTLR denoted by not writing any character!)
- sometimes we put the characters in quotes, e.g. '*a*'  
(required in ANTLR)
  - necessary when denoting | . \*
- [a-z] (written 'a'..'z' in ANTLR) is a shorthand for
  - a | b | c | ... | z
- [0-9] (written '0'..'9' in ANTLR) is a shorthand for
  - 0 | 1 | ... | 9

## | . \* operators have implicit precedence!

Regular Expression Operator	Analogous Arithmetic Operator	Precedence
	plus	lowest
.	times	middle
*	exponentiation	highest

- Consider regular expressions:
  - $[a-zA-Z] . [a-zA-Z] | [0-9]^*$
  - $[a-zA-Z] . ([a-zA-Z] | [0-9])^*$

## Example: Integer Literals

---

- An integer literal with an optional sign can be defined in English as:
  - “(nothing or + or -) followed by one or more digits”
- The corresponding regular expression is:
  - $(+| - | \varepsilon ).([0-9].[0-9]^*)$
- “.” can also be omitted, e.g.
  - $(+| - | \varepsilon ) ([0-9] [0-9]^*)$
- A new convenient operator ‘+’
  - same precedence as ‘\*’
  - $[0-9].[0-9]^*$  is the same as
  - $[0-9]^+$  which means “one or more digits”

---

## **Lexer generators**

## Lexer generators

L'utente manda in input al Generatore di Lexer le RE.  
Il Generatore trasforma le RE in FA.

- Receive in input the regular expressions describing the lexemes
- Generate code (*C*, *C++*, *Java*, ...) that implements the full declarative lexer algorithm:
  - Translate the regular expressions into FA
  - Merge the FA in a unique automaton
  - Translate the merged automaton to a Deterministic FA (more efficient to be simulated)
  - Produce the code that implements the "special" simulation of the DFA (lookahead for maximal match rule, priorities in case of multiple match, operations to be executed upon matching, and return to initial state)

# Implementation

Invece di scrivere codice imperativo con molti if/else o switch per ogni stato e per ogni carattere, il DFA è rappresentato da una struttura dati molto efficiente: una matrice bidimensionale (Tabella T)

- A DFA can be implemented by a 2D table T

- One dimension is “states” Le righe corrispondono agli stati interni del DFA ( $S_0, S_1, \dots, S_n$ ).

- Other dimension is “input symbols” Le colonne corrispondono a tutti i possibili caratteri di input (l'alfabeto, es. lettere, cifre, simboli).

- For every transition  $S_i \xrightarrow{a} S_k$  define  $T[i,a] = k$  →

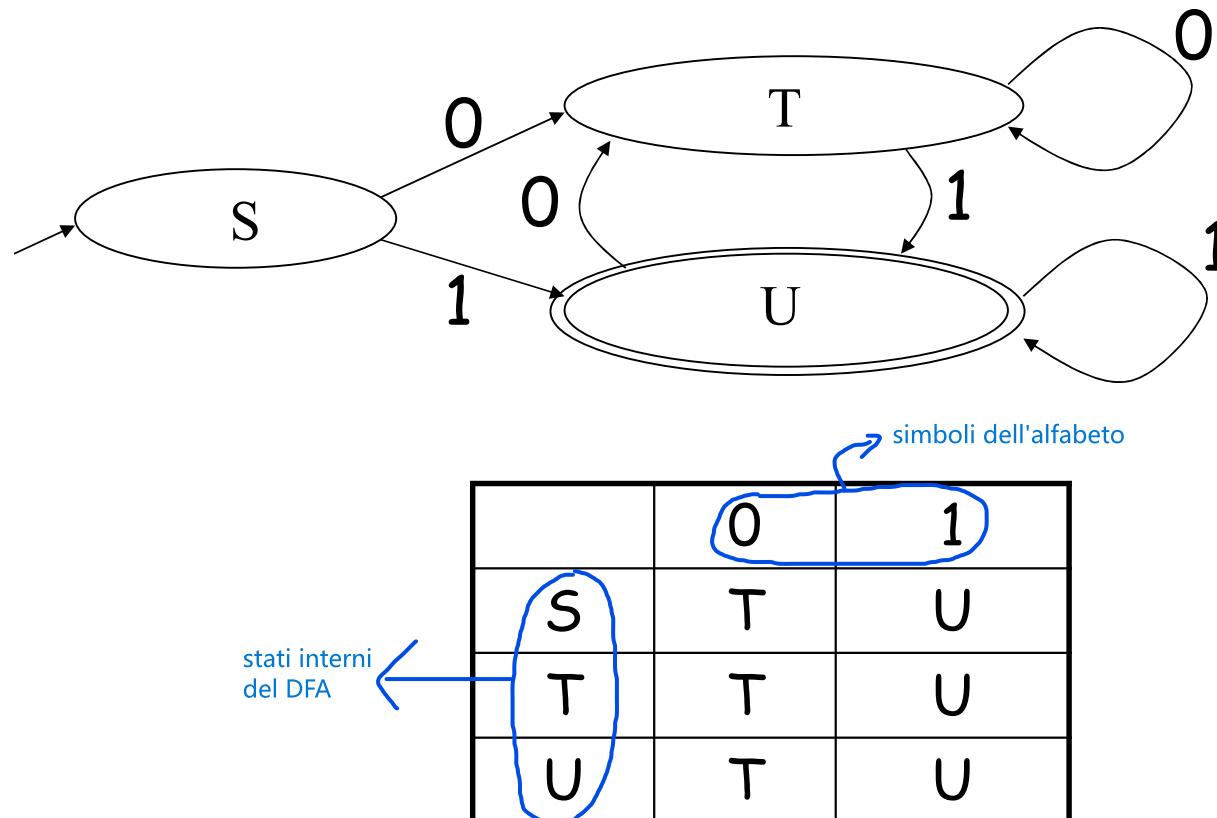
In ogni cella della tabella  $T[i, a]$ , è memorizzato l'indice dello stato successivo ( $S_k$ ) che deve essere raggiunto.

## DFA “execution”

- If in state  $S_i$  and input a, read  $T[i,a] = k$  and skip to state  $S_k$
- Very efficient

L'esecuzione del lexer si riduce a una serie di semplici e veloci operazioni di ricerca in tabella (lookup) e cambio di stato.

## Table Implementation of a DFA



Lexer generator per parser bottom-up

## Example: flex (a fast lexer generator)

---

- **flex generates lexers**
- **flex input:**
  - file with regular expressions defining the tokens, and for each token some **C** instructions
- **flex output:**
  - a **C** file named **lex.yy.c** that contains the definition of a function **yylex()**
  - **yylex()** reads the input and finds the tokens
  - when a token is found the corresponding **C** instructions are executed

Se non si matcha nessuno, allora genera errore

## Example of flex input file

Dall'alto verso il basso, il lessema con più priorità

```
%{  
#include <stdio.h>  
%}  
%%  
[ \t\n]+    printf("white space, length %i\n",yylen);  
/*      printf("times\n");      "/"      printf("div\n");  
+      printf("plus\n");      "-"      printf("minus\n");  
(      printf("left-par\n");  ")"      printf("right-par\n");  
0|[1-9][0-9]*          printf("integer %s\n",yytext);  
[a-zA-Z_][a-zA-Z0-9_]*  printf("identifier %s\n",yytext);  
.                printf("illegal char %s\n",yytext);  
%%  
main(){yylex();}
```

lunghezza del lessema

lessema matchato

funzione che lancia la funzione del lexer

---

**Building a parser**

Il Parse Tree è la prima rappresentazione generata dal parser (Analizzatore Sintattico) e riflette l'intera struttura sintattica del codice sorgente, basandosi direttamente sulla Grammatica Context-Free (CFG) del linguaggio.

- Scopo Principale (Sintassi Corretta): Eseguire il Controllo Sintattico (Syntax Checking). Dimostra che la sequenza di token fornita dal scanner può essere derivata dalle regole della grammatica.
- Contenuto: Contiene tutti gli elementi della derivazione: I nodi interni sono simboli non terminali (le variabili grammaticali); I nodi foglia sono i token terminali.
- Dettagli Inclusi: Mantiene tutti i simboli della grammatica ma anche quelli irrilevanti per il significato logico del programma (es. parentesi, punti e virgola).
- Ruolo per il Parser: Il parser usa l'esistenza implicita o esplicita del Parse Tree per definire le relazioni di annidamento (nesting) tra le espressioni (sotto-alberi e sopra-alberi).

## Overview

- What does a parser do, again?

its two tasks

Albero Sintattico (teoria)

- parse tree vs. AST

- A hand-written parser

- and why it gets hard to get it right

Controlla la sequenza di token  
dati dal lexer/scanner e forma  
un programma corretto

Genera un albero sintattico (top-down o bottom-up)



L'AST è la rappresentazione finale, ottimizzata per le fasi successive di analisi e traduzione. Viene costruito trasformando o filtrando il Parse Tree.

- Scopo Principale (Sintassi Astratta): Fornire una rappresentazione pulita e essenziale del programma, contenente solo ciò che è necessario per l'Analisi Semantica, l'Ottimizzazione e la Generazione del Codice.

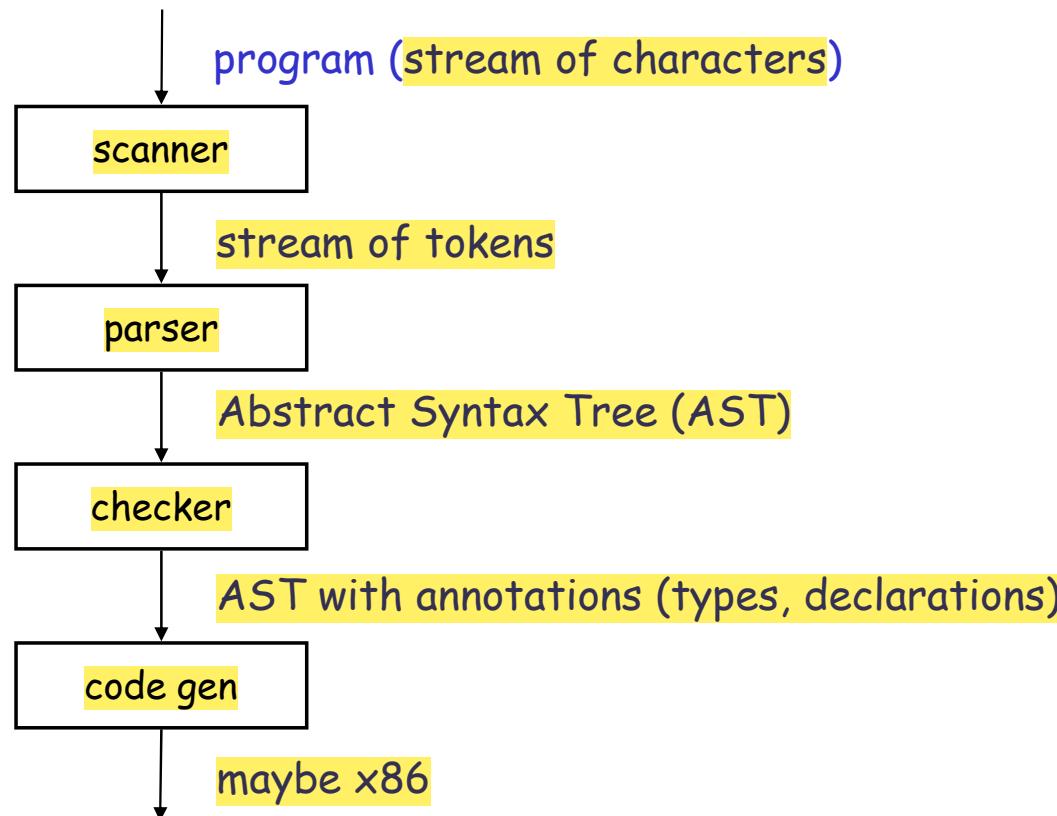
- Contenuto: I nodi interni rappresentano gli operatori o le costruzioni logiche (es. assegnazione, somma, ciclo while); I nodi foglia rappresentano gli operandi o i valori (es. variabili/identificatori, costanti).
- Dettagli Esclusi: L'AST rimuove tutto ciò che è superfluo ai fini della logica del programma, come parentesi, punti e virgola e simboli non terminali intermedi.
- Flusso: Il parser genera il Parse Tree (implicitamente o esplicitamente) prima di generare l'AST. La costruzione dell'AST è il risultato finale dell'analisi sintattica.

---

**What does a parser do?**

## Recall: The Structure of a Compiler

---



## Recall: Syntactic Analysis

Obiettivo: Verificare se sequenza di input da scanner rispetta la Grammatica Context-Free (CFG) del linguaggio e, in caso affermativo, rappresentarne la struttura gerarchica.

- **Input:** sequence of tokens from scanner
- **Output:** abstract syntax tree (AST)
- Actually,
  - parser first considers the parse tree, i.e. the (non abstract) syntax tree
  - AST is then built by translating the parse tree
  - parse tree rarely built explicitly; only determined by, say, how parser pushes stuff to stack
  - our lectures first focus on constructing the parse tree; later we'll show the translation to AST.

## Example

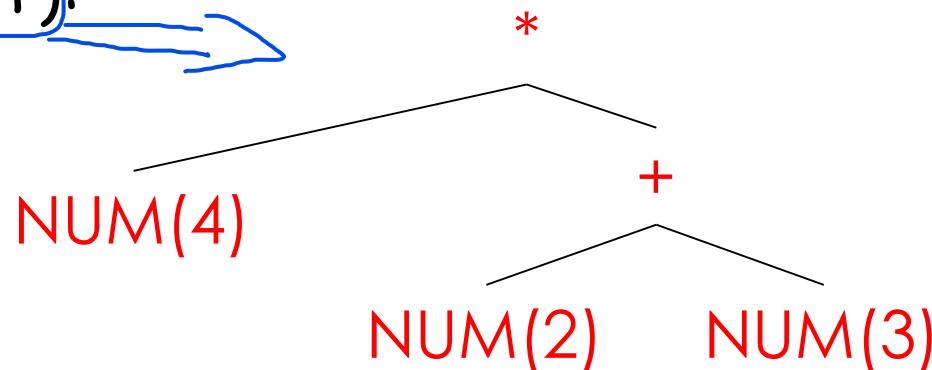
- Expression

$4*(2+3)$

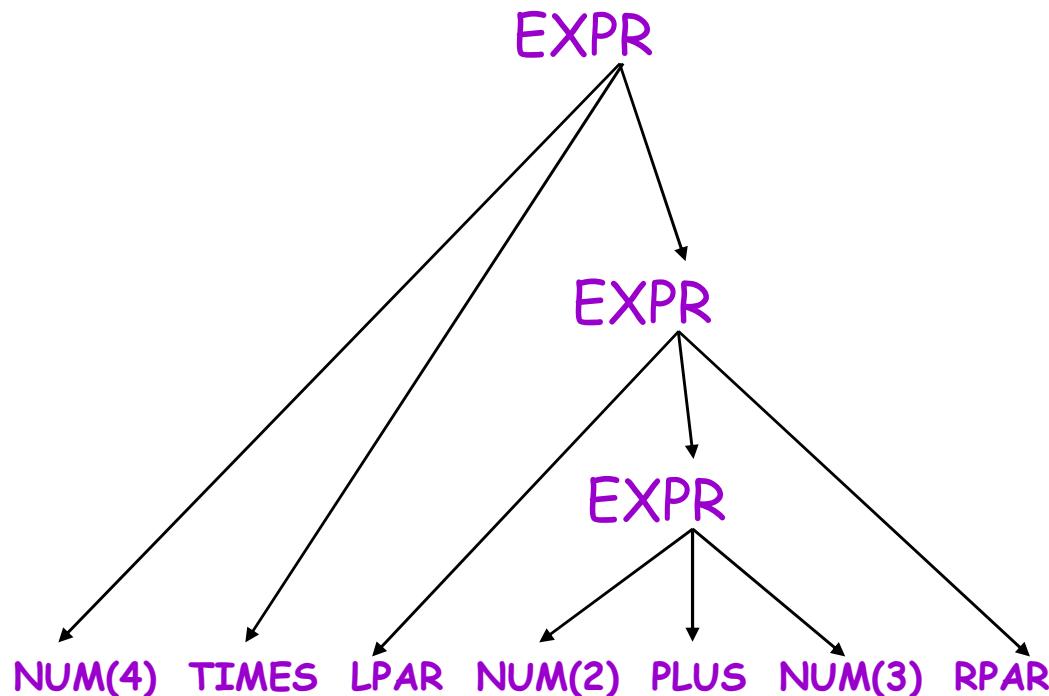
- Parser input sequenza di token individuati dallo scanner/lexer

NUM(4) TIMES LPAR NUM(2) PLUS NUM(3) RPAR

- Parser output (AST):



## Parse tree for the example



leaves are tokens

Parser bottom-up (dai token genera il Parse Tree)

## Another example

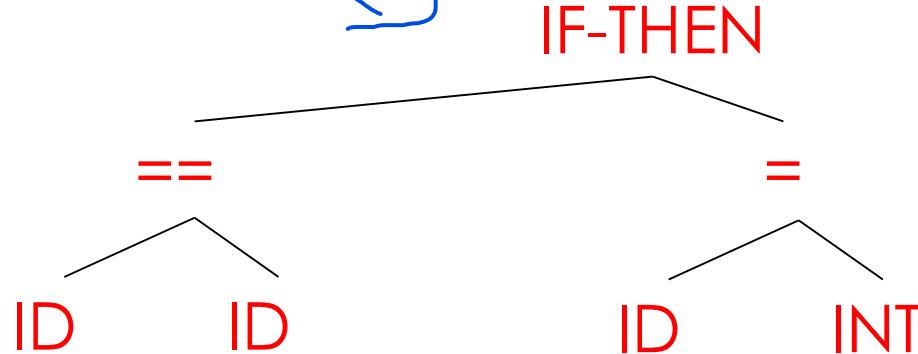
- Other expression

if (x == y) { a=1; }

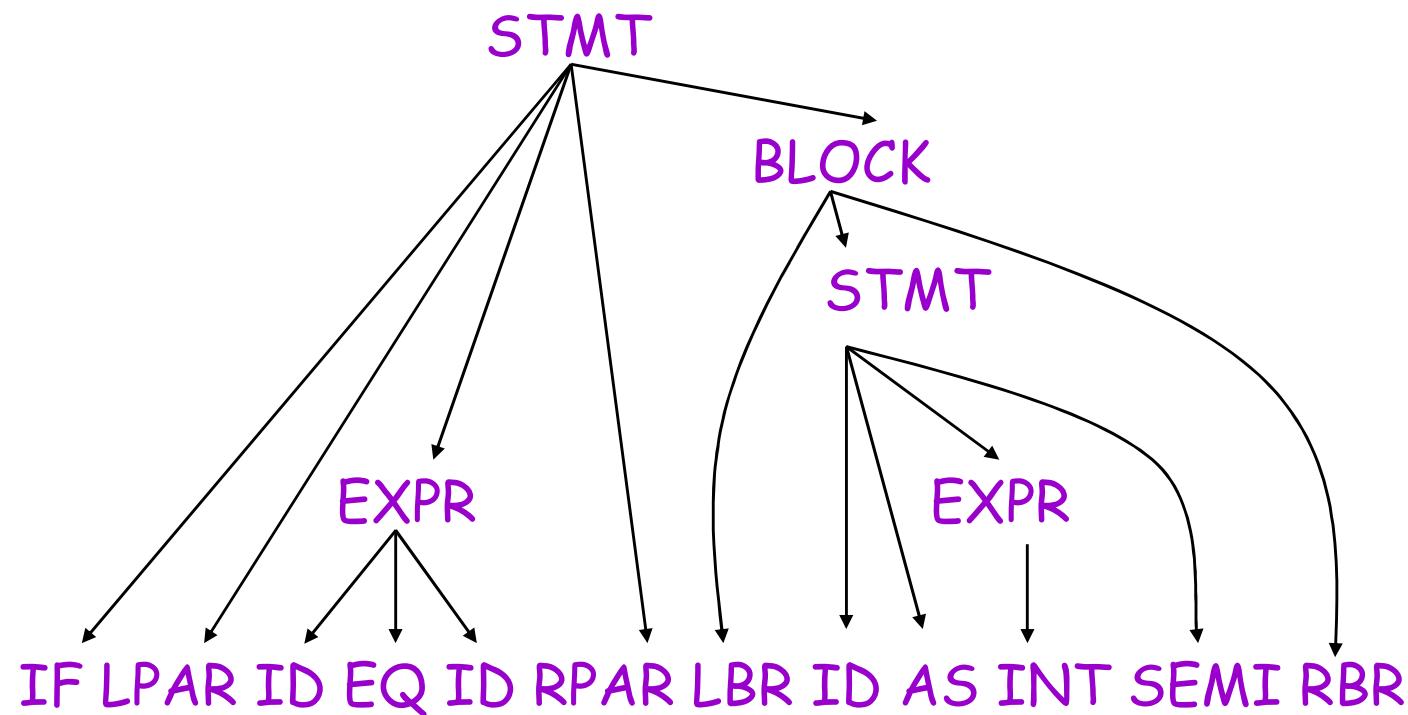
- Parser input

IF LPAR ID EQ ID RPAR LBR ID AS INT SEMI RBR

- Parser output (AST):



## Parse tree for the example



## Parse tree vs. abstract syntax tree

---

- Parse tree
  - contains all tokens, including those that parser needs “only” to discover
    - intended nesting: parentheses, curly braces
    - statement termination: semicolons
  - technically, parse tree shows **concrete syntax**
- Abstract syntax tree (AST)
  - abstracts away artifacts of parsing, by flattening tree hierarchies, dropping tokens, etc.
  - technically, AST shows **abstract syntax**

## Comparison with Lexical Analysis

---

Phase	<i>Input</i>	<i>Output</i>
Lexer	Sequence of characters	Sequence of tokens
Parser	Sequence of tokens	AST, built from parse tree

## Summary

Il Parse Tree viene utilizzato sia per la fase 1 (Syntax Checking) sia per la fase 2 (generazione AST)

- Parser performs two tasks:

### 1 - Syntax checking

- a program with a syntax error is rejected and information about error given

### 2 - Parse tree construction

- usually implicit (needed for syntax checking)
- exploited to build the AST

---

**How to build a parser?**

## Writing the parser

---

- Can do it all by hand, of course
  - ok for small languages, but hard for real programming languages
- Just like with the scanner, it is possible to use a parser generator
  - one concisely describes the syntactic structure
    - that is, how expressions, statements, definitions look like
  - and the generator produces a working parser
- Let's start with a hand-written parser
  - to see why we want a parser generator

## First example: balanced parens

---

- Our problem: check the syntax
  - are parentheses in input string balanced?
- The simple language
  - parenthesized number literals
  - Ex.: 3, (4), ((1)), (((2))), etc
- Before we look at the parser
  - why aren't finite automata sufficient for this task?

## Parser code preliminaries

---

- Let `TOKEN` be an enumeration type of tokens:
  - `NUM, LPAR, RPAR, PLUS, MINUS, TIMES, DIV`
- Let the global `in[]` be the input string of tokens
- Let the global `next` be an index in the token string (initially is 0)

## Parsers use stack to implement infinite state

---

Balanced parentheses parser:

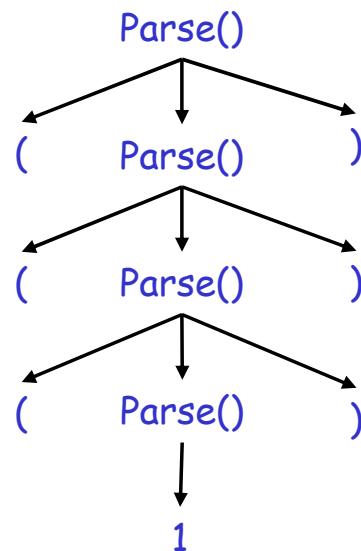
```
void Parse() {
    nextToken = in[next++];
    if (nextToken == NUM) return;

    if (nextToken != LPAR) print("syntax error");
    Parse();
    if (in[next++] != RPAR) print("syntax error");
}
```

## Where's the parse tree constructed?

---

- In this parser, the parse is given by the call tree:
- For the input string  $((1))$  :



## Second example: subtraction expressions

---

The language of this example:

1, 1-2, 1-2-3, (1-2)-3, (2-(3-4)), etc

```
void Parse() {
    if ((nextToken = in[next++]) == NUM) {
        if (in[next] == MINUS) { next++; Parse(); }
    } else if (nextToken == LPAR) {
        Parse();
        if (in[next++] != RPAR) print("syntax error");
    } else print("syntax error");
}
```

## Second example: subtraction expressions

---

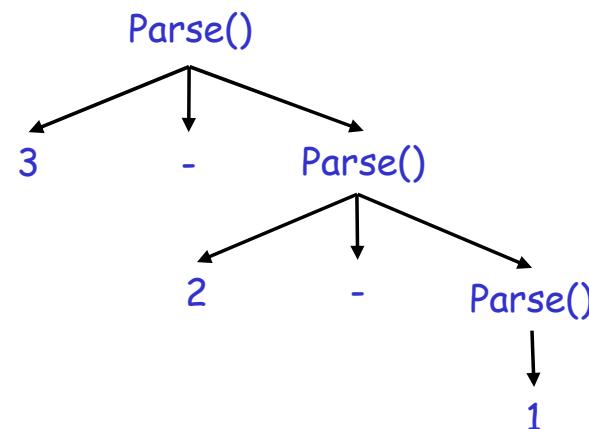
The language of this example: (fixed!!)

1, 1-2, 1-2-3, (1-2)-3, (2-(3-4)), etc

```
void Parse() {
    if ((nextToken = in[next++]) == NUM) {
        if (in[next] == MINUS) { next++; Parse(); }
    } else if (nextToken == LPAR) {
        Parse();
        if (in[next++] != RPAR) print("syntax error");
        if (in[next] == MINUS) { next++; Parse(); }
    } else print("syntax error");
}
```

## Subtraction expressions continued

- Observations:
  - a more complex language
    - hence, harder to see how the parser works (and if it works correctly at all)
  - the parse tree is actually not really what we want
    - consider input 3-2-1
    - what's undesirable about this parse tree's structure?



## We need a clean syntactic description

---

- Just like with the scanner, writing the parser by hand is painful and error-prone
  - consider adding +, \*, / to the last example!
- So, let's separate the what and the how
  - **what:** the syntactic structure, described with a context-free grammar
  - **how:** the parser, built from the grammar, which reads the input and produces the parse tree

## Idea...

---

- We can describe the syntactic structure by using context-free grammars!
- What's next?
  - Grammars
  - Derivations

## Grammars

---

- Programming language constructs have recursive structure.
  - which is why our hand-written parser had this structure, too
- Example: an expression is either:
  - number, or
  - variable, or
  - expression + expression, or
  - expression - expression, or
  - ( expression ), or
  - ...

## Context-free grammars (CFG)

---

- a natural notation for this recursive structure
- grammar for our balanced parens expressions:  
 $\text{BalancedExpression} \rightarrow a \mid (\text{BalancedExpression})$
- describes (generates) strings of symbols:
  - $a, (a), ((a)), (((a))), \dots$
- like regular expressions but can refer to
  - other expressions (here, `BalancedExpression`)
  - and do this recursively (thus, it is “non-finite state”)

## Example: arithmetic expressions

---

- Simple arithmetic expressions:

$$E \rightarrow n \mid id \mid (E) \mid E + E \mid E * E$$

- Some strings of this language:

- $id$
- $n$
- $(n)$
- $n + id$
- $id * (id + id)$

## How do derivations help us in parsing?

---

- A program (a string of tokens) has no syntax error if it can be derived from the grammar.
  - grammars, however, define how to derive some (any) string, not how to check if a given string is derivable
- So how to do parsing?
  - a naïve solution: derive all possible strings and check if your program is among them
  - not as bad as it sounds: there are parsers that do this.  
Coming soon.

## Example

---

A fragment of a simple language:

$$\begin{array}{l} \text{STMT} \rightarrow \text{while ( EXPR ) STMT} \\ | \text{id ( EXPR ) ;} \end{array}$$
$$\begin{array}{l} \text{EXPR} \rightarrow \text{EXPR + EXPR} \\ | \text{EXPR - EXPR} \\ | \text{EXPR < EXPR} \\ | (\text{EXPR}) \\ | \text{id} \end{array}$$