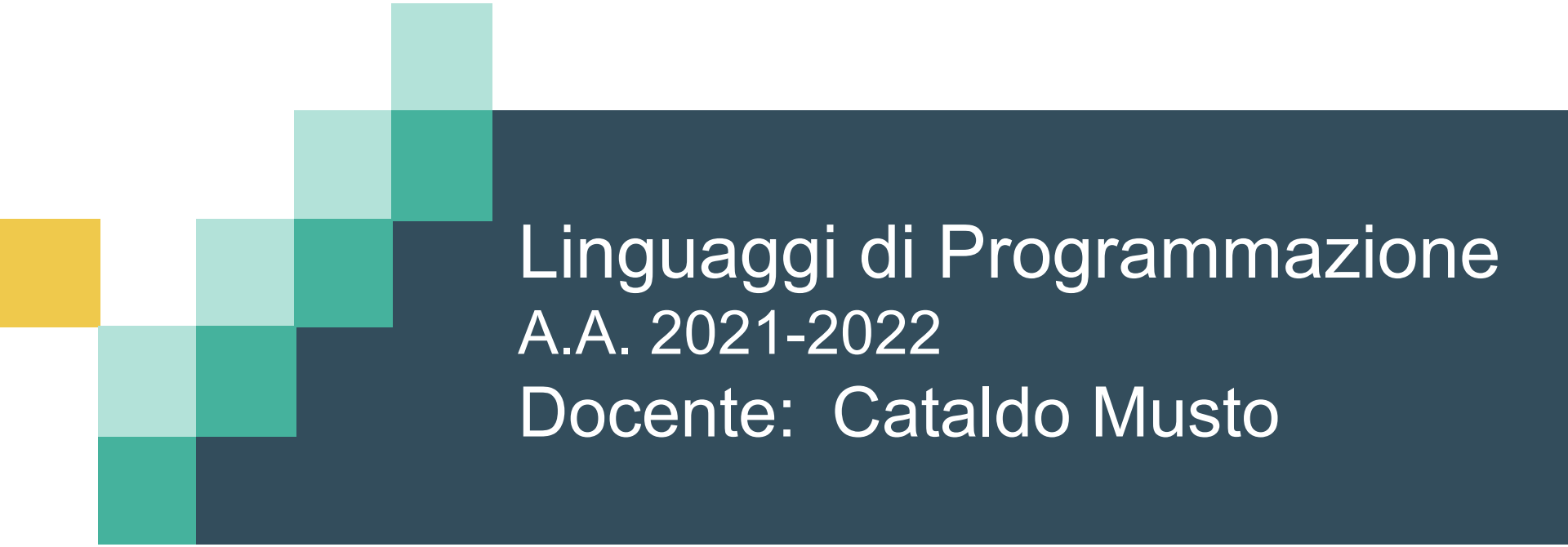


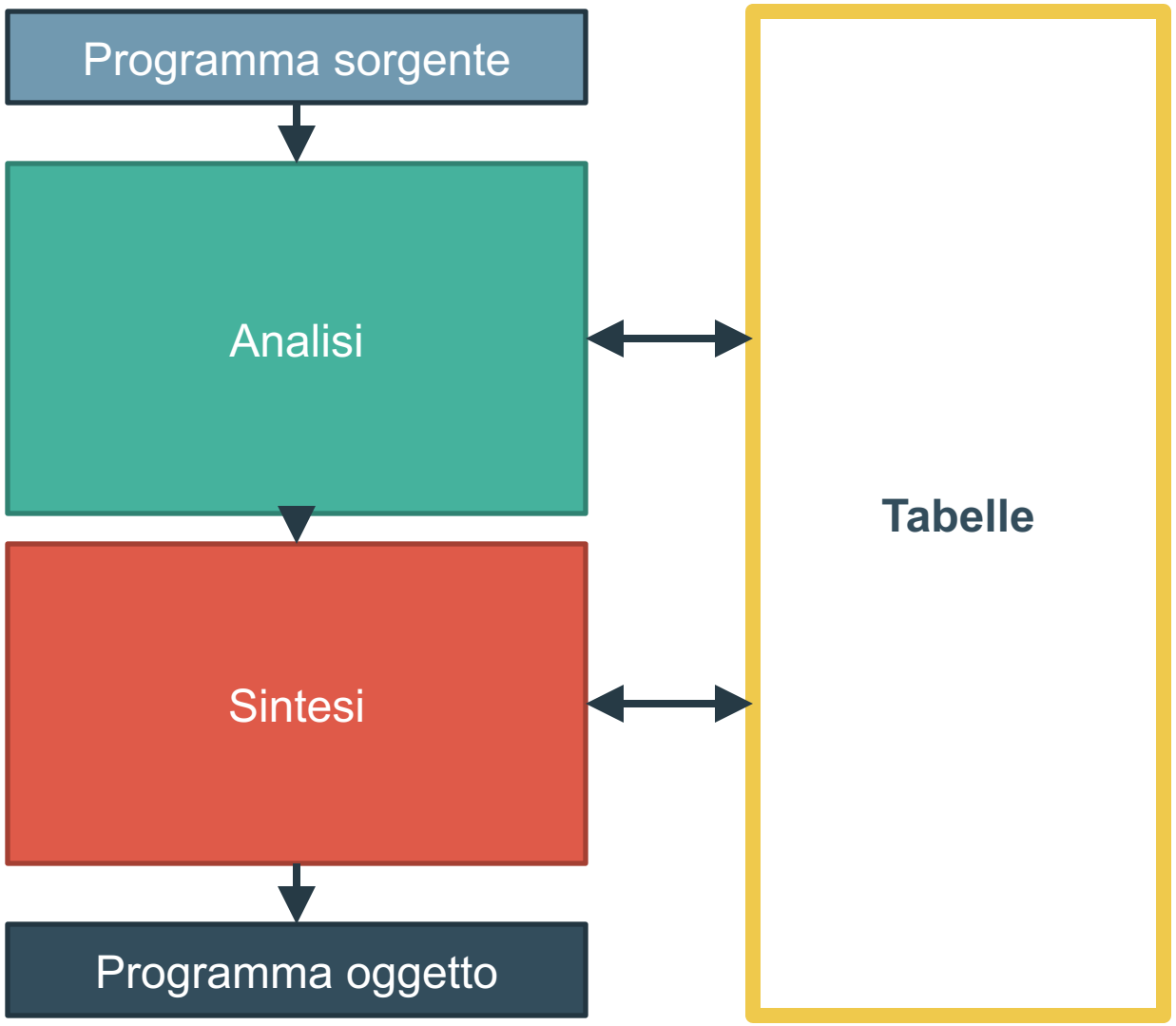
Modello di un compilatore

Il Modello di un Compilatore

- La costruzione di un compilatore per un particolare linguaggio di programmazione è abbastanza complessa.
- La complessità dipende dal linguaggio sorgente.

- Il processo di compilazione è diviso in due macro-passaggi
  - Analisi del programma sorgente;
  - Sintesi del programma oggetto.

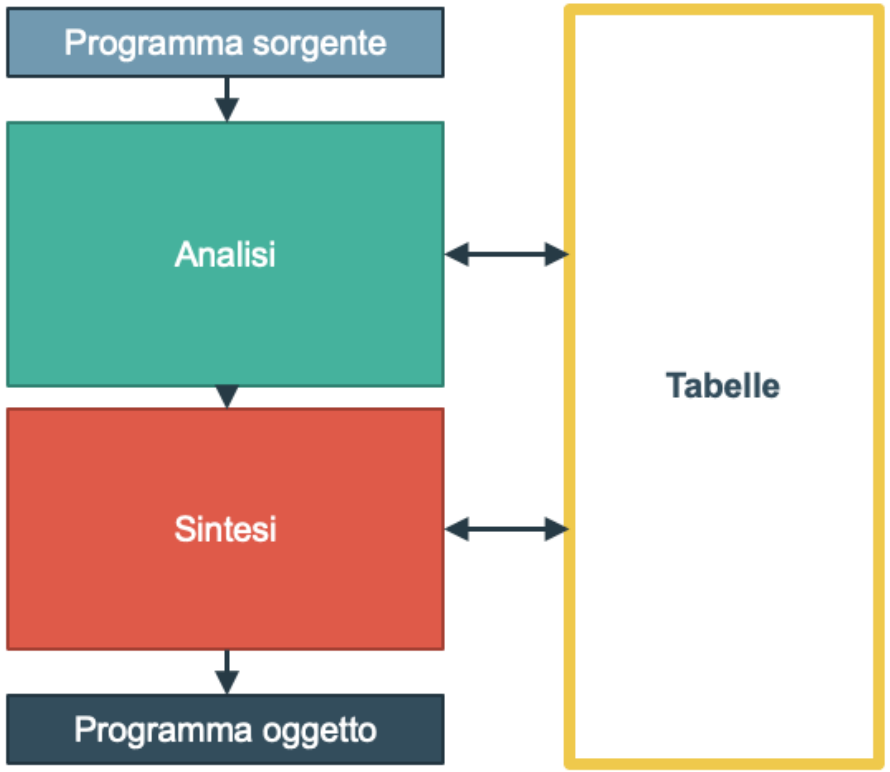
Compilatore



Programma sorgente

- È una stringa di simboli.
- Esempio

```
if A>B then X:=Y;
```



Analisi

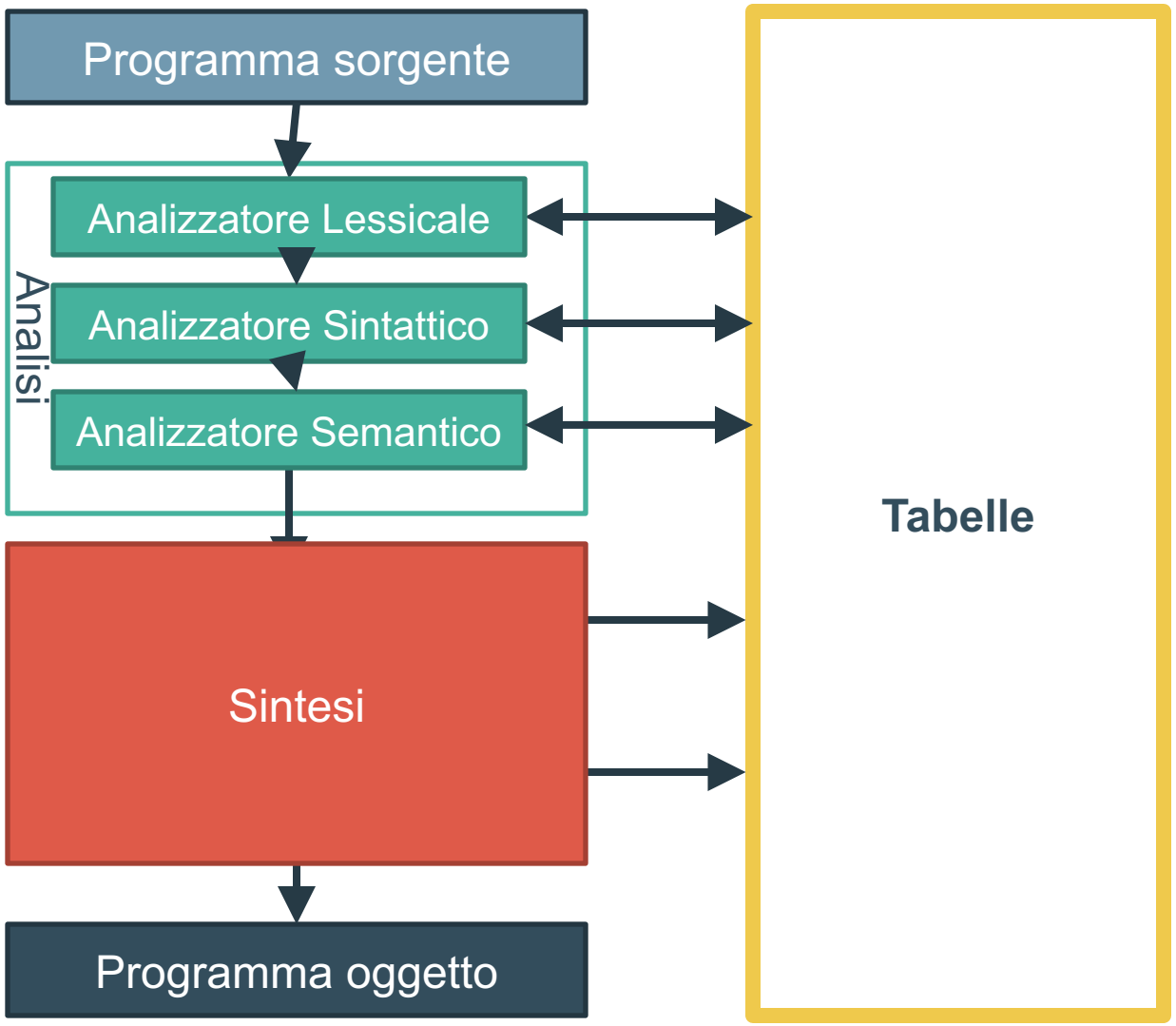
- Verifica della correttezza **sintattica e semantica** di un programma
  - Svolta in fase di compilazione;
  - Verifica che:
    - I simboli utilizzati siano legali, cioè appartengano all'alfabeto (analisi lessicale);
    - Le regole grammaticali siano rispettate (analisi sintattica);
    - I vincoli imposti dal contesto siano rispettati (analisi semantica).

- Esempio:

```
var A: integer;
…
if A then …
```

Errore!

Compilatore



Analizzatore lessicale (scanner)

- Lo scanner **rappresenta un passaggio intermedio** fra il programma sorgente e l'analizzatore sintattico o parser.
- Lo scanner, attraverso un esame carattere per carattere dell'ingresso, separa il programma sorgente in parti chiamate **token** che rappresentano i nomi delle variabili, operatori, label, ecc.

Analizzatore lessicale (scanner)

- **Input:** un programma sorgente
- Legge uno stream di caratteri e raggruppa i caratteri in sequenze che abbiano un significato. Individua i simboli (token) che lo compongono classificando parole chiave, identificatori, operatori, costanti, ecc.
- Per ragioni di efficienza ad ogni classe di token è dato un numero unico che la identifica.
- **Output:** lista di token

Analizzatore lessicale o scanner

- Lo scanner suddivide il programma sorgente in *token.*
- Il tipo di token è rappresentato con un numero intero unico (esempio, variabile con il numero 1, costante 2, label 3).
- Il token, che è una stringa di caratteri, è memorizzato in una tabella.
- I valori delle costanti sono memorizzati in una *constant table,* mentre i nomi delle variabili in una *symbol table.*

Analizzatore lessicale (scanner)

- Esempio

```
IF A>B THEN X:=Y;

IF          20
A           1
>           15
B           1
THEN        20
X           1
:=          10
Y           1
;           27
```

- Si noti che vengono ignorati spazi bianchi e commenti. Inoltre alcuni scanner inseriscono label, costanti e variabili in tavole appropriate.
- Un elemento della tavola per una variabile, ad esempio, contiene nome, tipo, indirizzo, valore e linea in cui è dichiarata.

Analizzatore lessicale (scanner)

- Esempio

Programma in input

```
x1:=a+bb*12;
x2:=a/2+bb*12;
```

Sequenza di token

“x1“	Id
“:=”	Op
"a"	Id
“+”	Op
"bb"	Id
“*”	Op
12	Lit
;	Punct
“x2”	Id
“:=”	Op
“a”	Id
"/”	Op
2	Lit
“+”	Op
"bb"	Id
“*”	Op
12	Lit
;	Punct

Contenuti della tabella dei simboli

- Una TS è costituita da una serie di righe ognuna delle quali contiene una lista di valori di attributi associati con una particolare variabile.
- Gli attributi dipendono dal linguaggio di programmazione che si compila (se non ha i tipi non comparirà tale attributo).

Variable Name	Address	Type	Dimension	Line Declared	Line Referenced	Pointer
COMPANY	0	2	1	2	9,14,25	7
X3	4	1	0	3	12,14	0
FORM1	8	3	2	4	36,37,38	6
B	48	1	0	5	10,11,13,23	1
ANS	52	1	0	5	11,23,25	4
M	56	6	0	6	17,21	2
FIRST	64	1	0	7	28,29,30,38	3

Il progetto di uno scanner e la sua realizzazione

- Compiti:
  - **Tutti i task sono legati alla manipolazione delle stringhe**
  - Eliminare spazi bianchi, commenti, ecc;
  - Isolare il prossimo token dalla sequenza di caratteri in input;
  - Isolare identificatori e parole-chiave;
  - Generare la symbol-table.

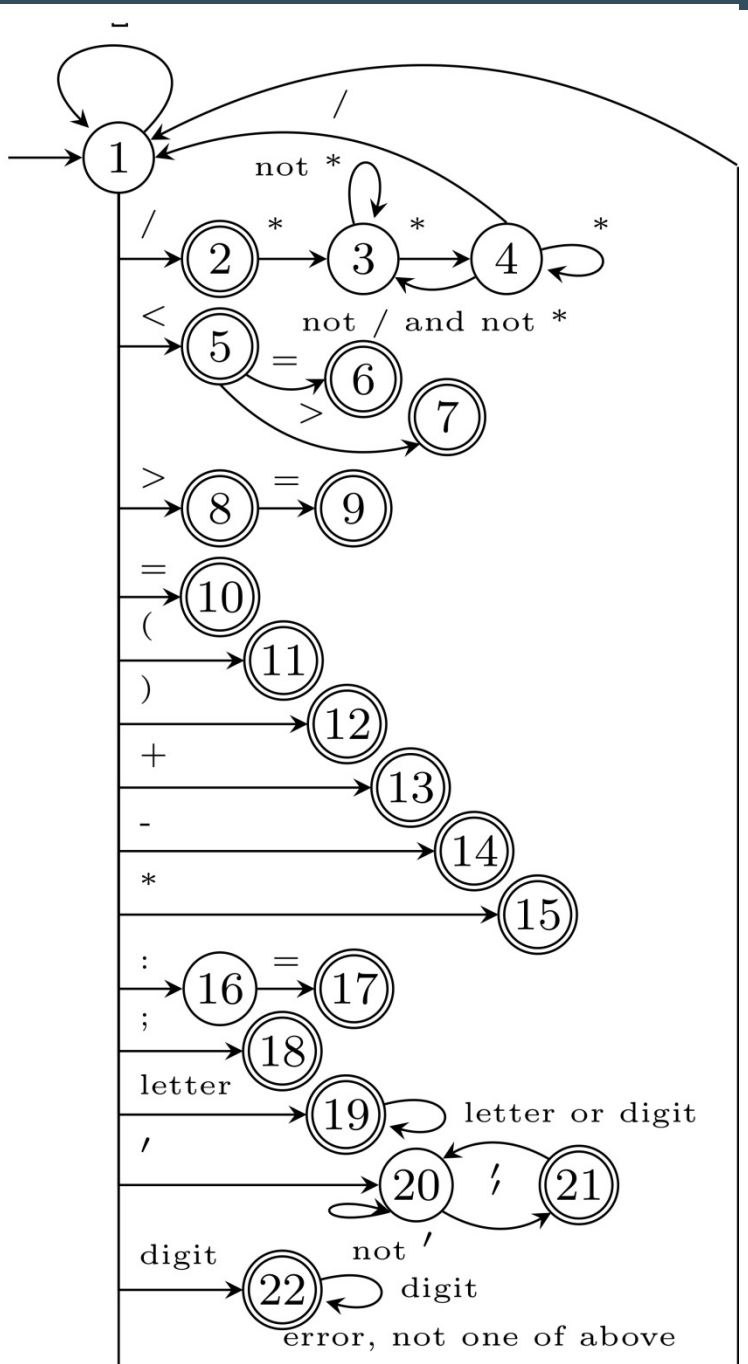
Esempio

- I token possono essere descritti in diversi modi. Spesso si utilizzano le **grammatiche regolari**.
- Grammatica regolare per generare i numeri naturali:

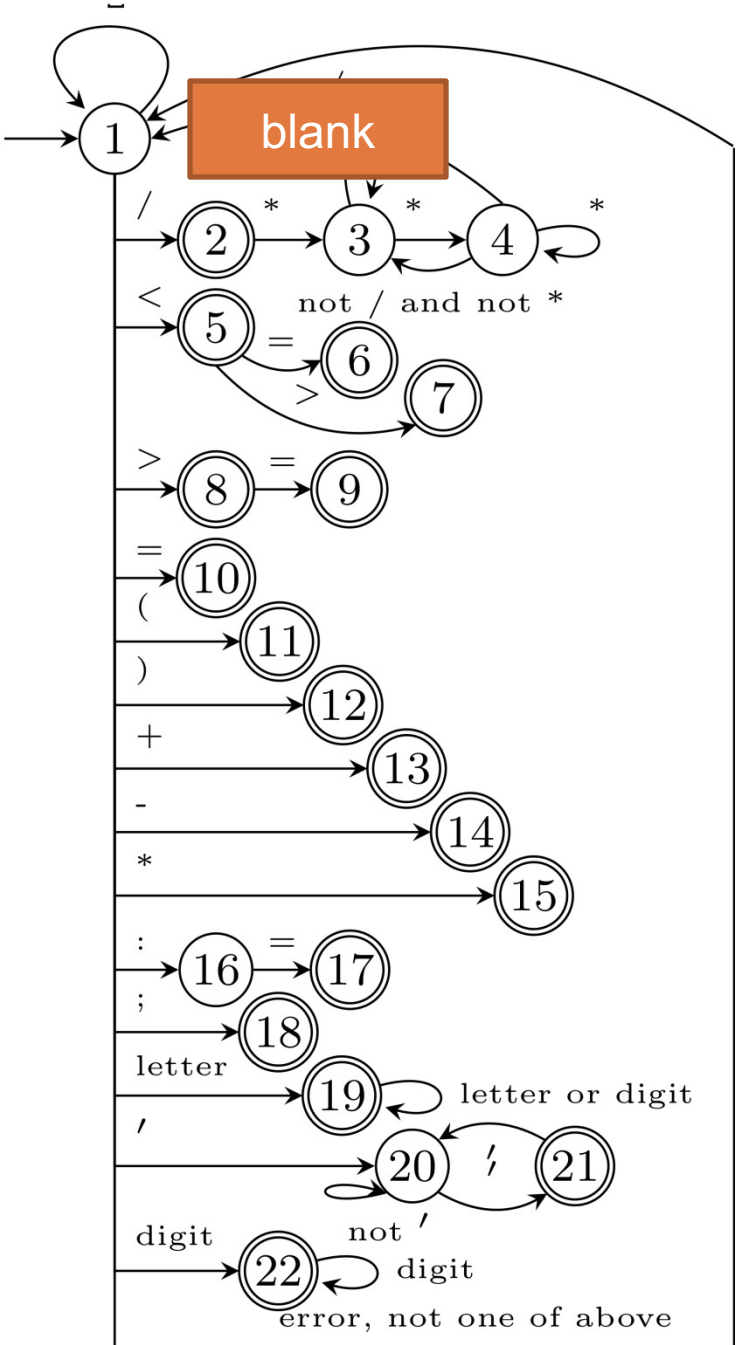
<unsigned integer>::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
0<unsigned integer> |
1<unsigned integer> |
…
9<unsigned integer>

- Un altro modo per descrivere i token è in modo riconoscitivo piuttosto che generativo mediante **automi a stati finiti**.

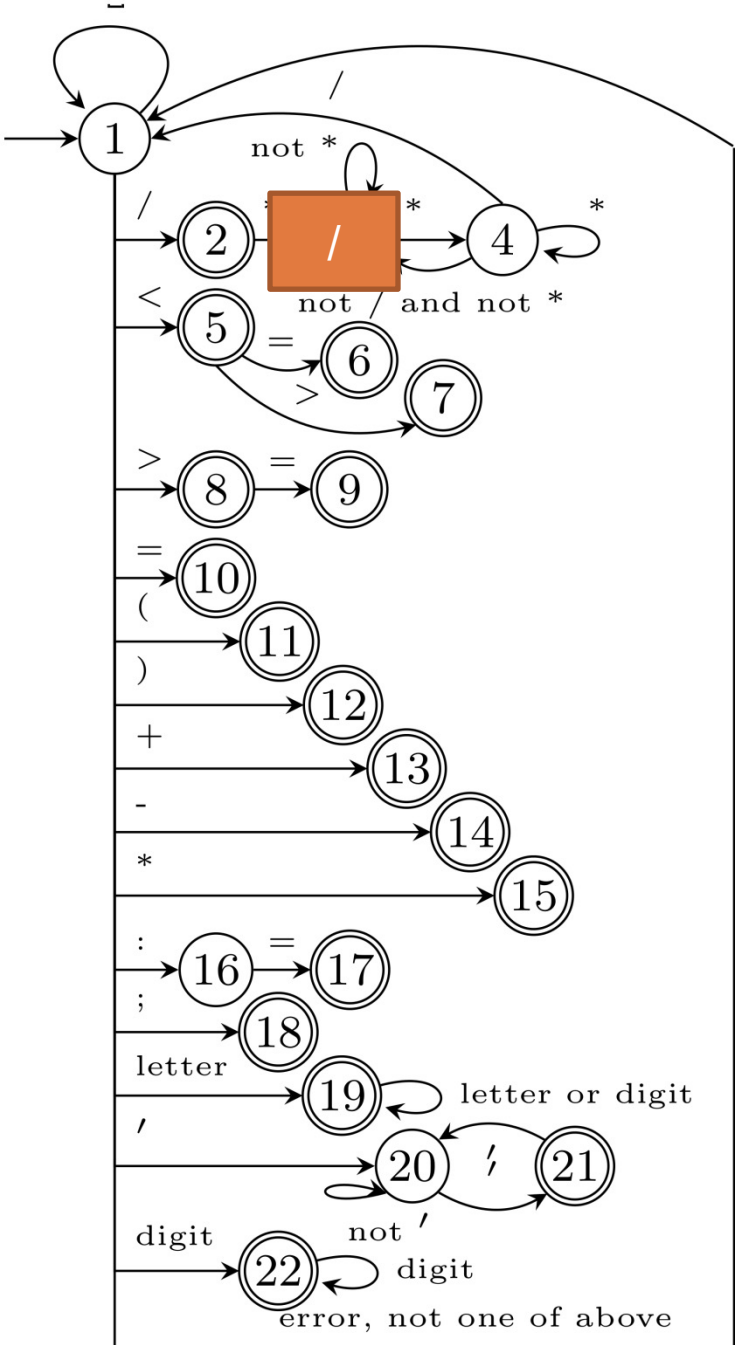
Algoritmo


1
/
not *
/
*
*
*
2
3
4
<
5
not / and not *
=
6
>
7
>
=
8
9
=
10
(
11
)
12
+
13
-
14
*
15
:
=
16
17
;
18
letter
19
letter or digit
'
20
'
21
digit
not '
22
digit
error, not one of above

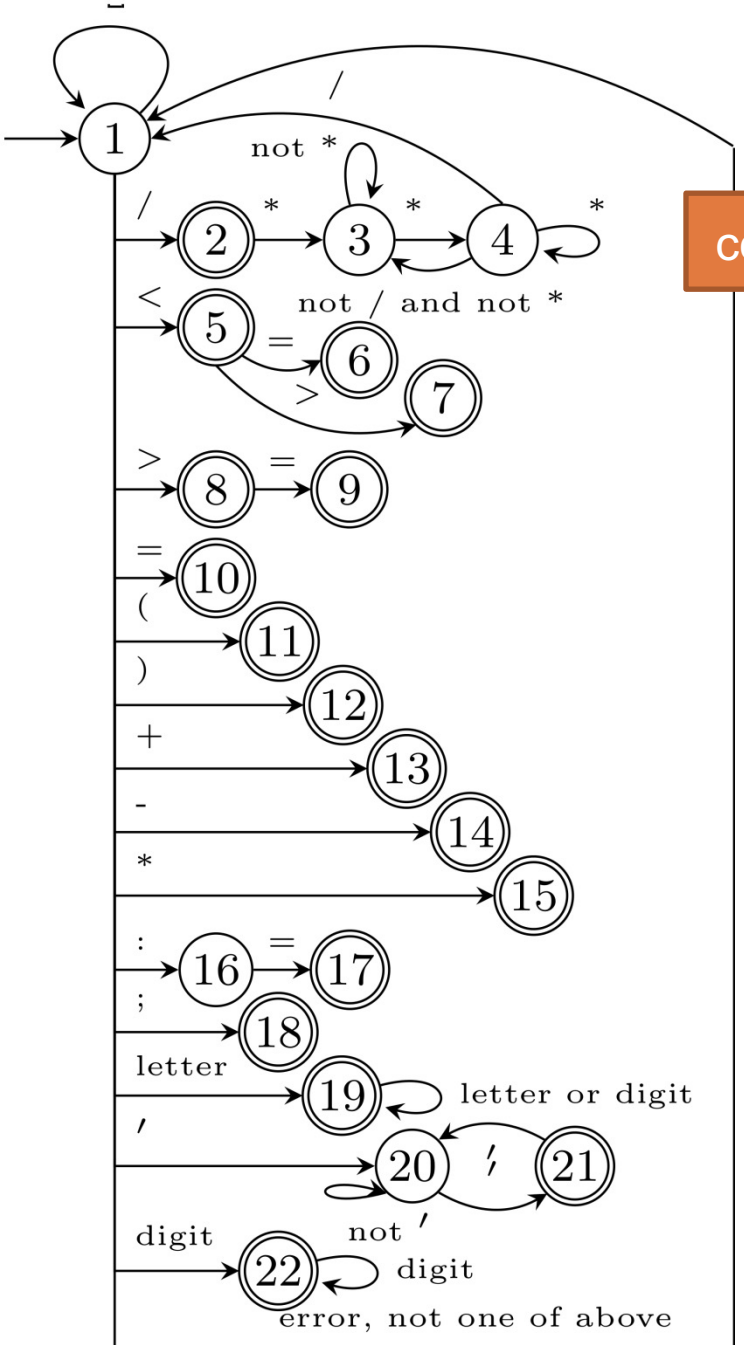
Algoritmo


blank
1
/
2
*
3
*
4
*
not / and not *
<
5
=
6
>
7
>
8
=
9
=
10
(
11
)
12
+
13
-
14
*
15
:
16
=
17
;
18
letter
19
letter or digit
'
20
'
21
not '
digit
22
digit
error, not one of above

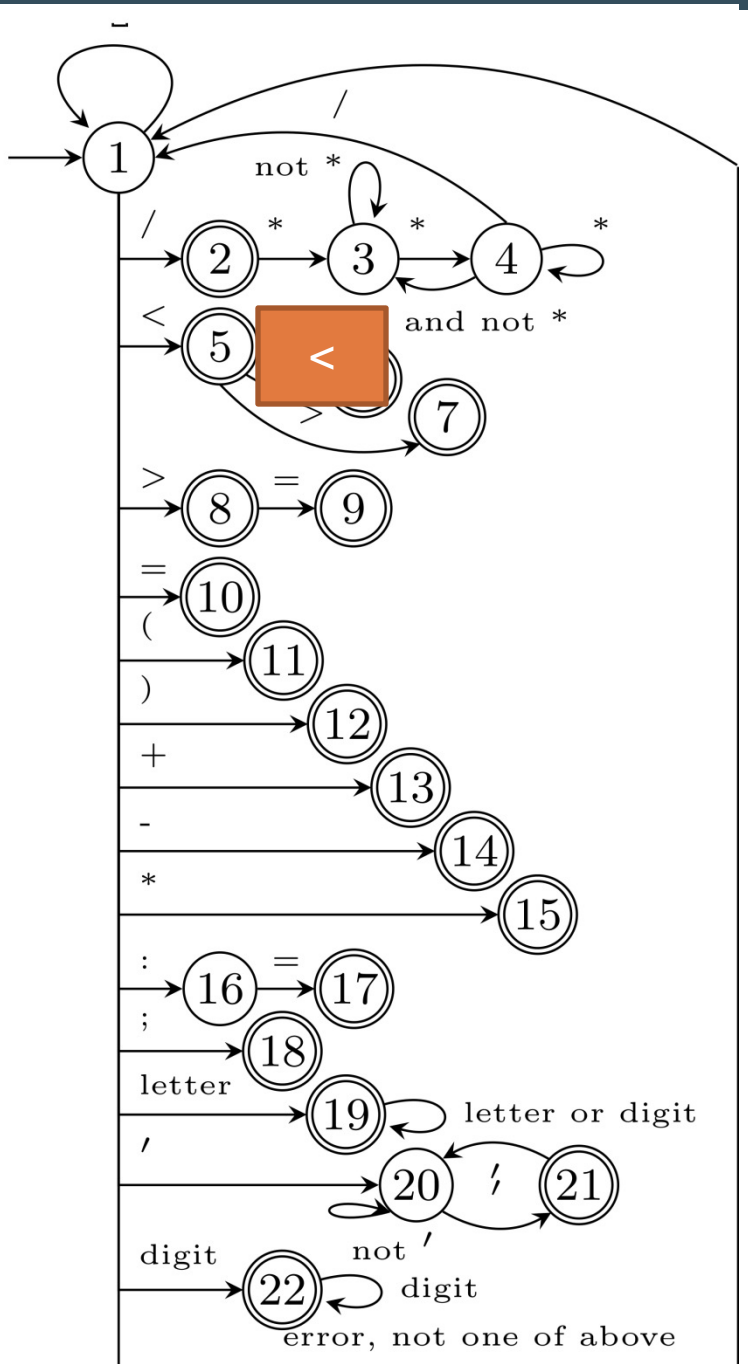
Algoritmo


1
/
not *
/
2
/
*
4
*
not / and not *
<
5
=
6
>
7
>
8
=
9
=
10
(
11
)
12
+
13
-
14
*
15
:
16
=
17
;
18
letter
19
letter or digit
'
20
/
21
not '
digit
22
digit
error, not one of above

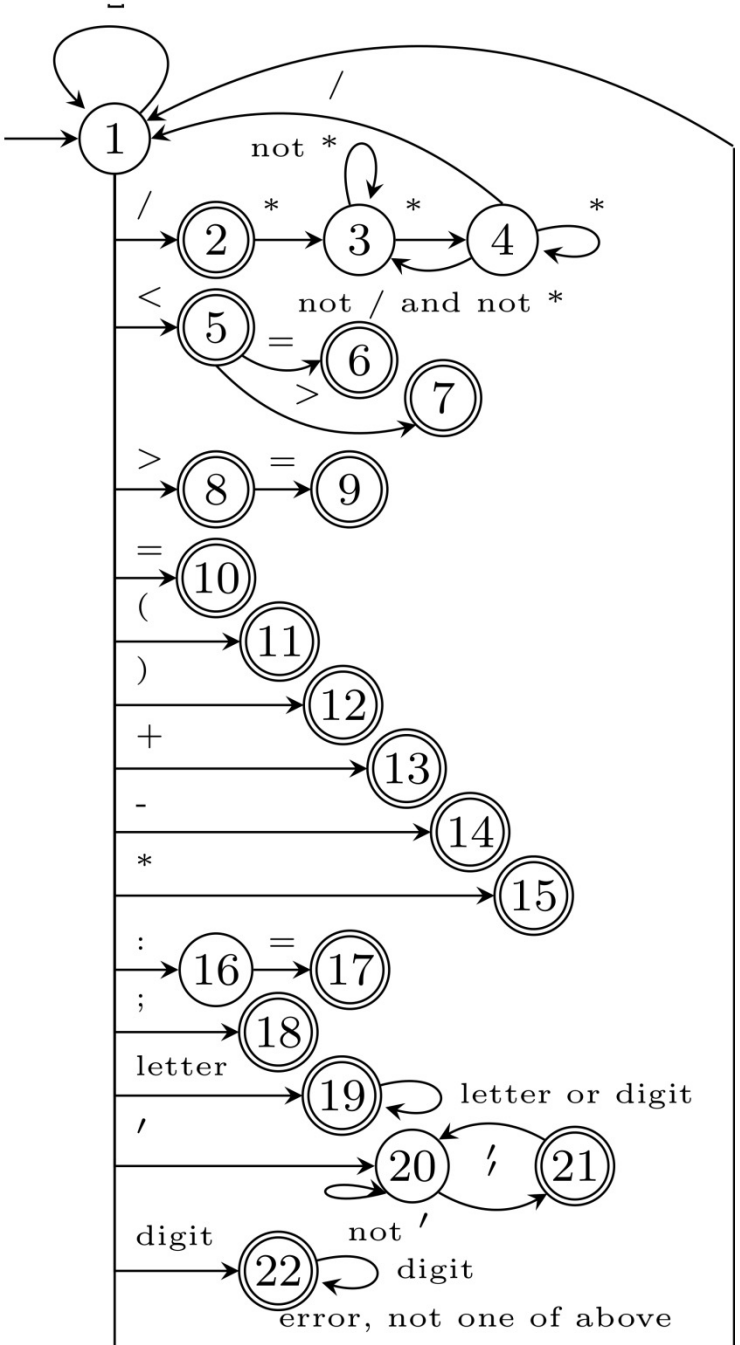
Algoritmo


1
/
not *
/
*
*
*
2
3
4
comment
<
5
not / and not *
=
6
>
7
>
8
=
9
=
10
(
11
)
12
+
13
-
14
*
15
:
16
=
17
;
18
letter
19
letter or digit
'
20
'
21
not '
digit
22
digit
error, not one of above

Algoritmo

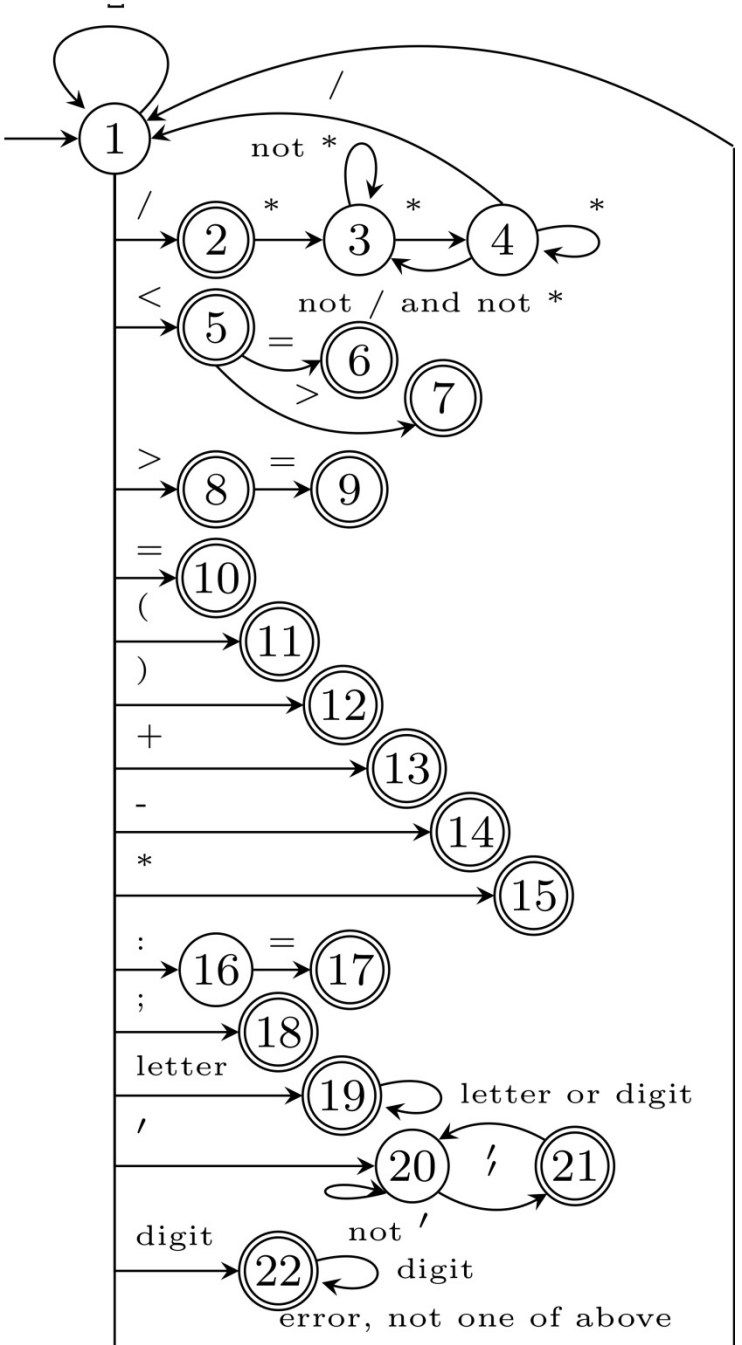

1
/
not *
/
*
2
*
3
*
4
*
<
5
and not *
<
>
7
>
=
8
9
=
10
(
11
)
12
+
13
-
14
*
15
:
=
16
17
;
18
letter
19
letter or digit
'
20
'
21
digit
not '
22
digit
error, not one of above

Algoritmo


1
/
not *
/
*
*
*
2
3
4
<
not / and not *
5
=
6
>
7
>
=
8
9
=
10
(
11
)
12
+
13
-
14
*
15
:
=
16
17
;
18
letter
19
letter or digit
'
20
'
21
digit
not '
22
digit
error, not one of above

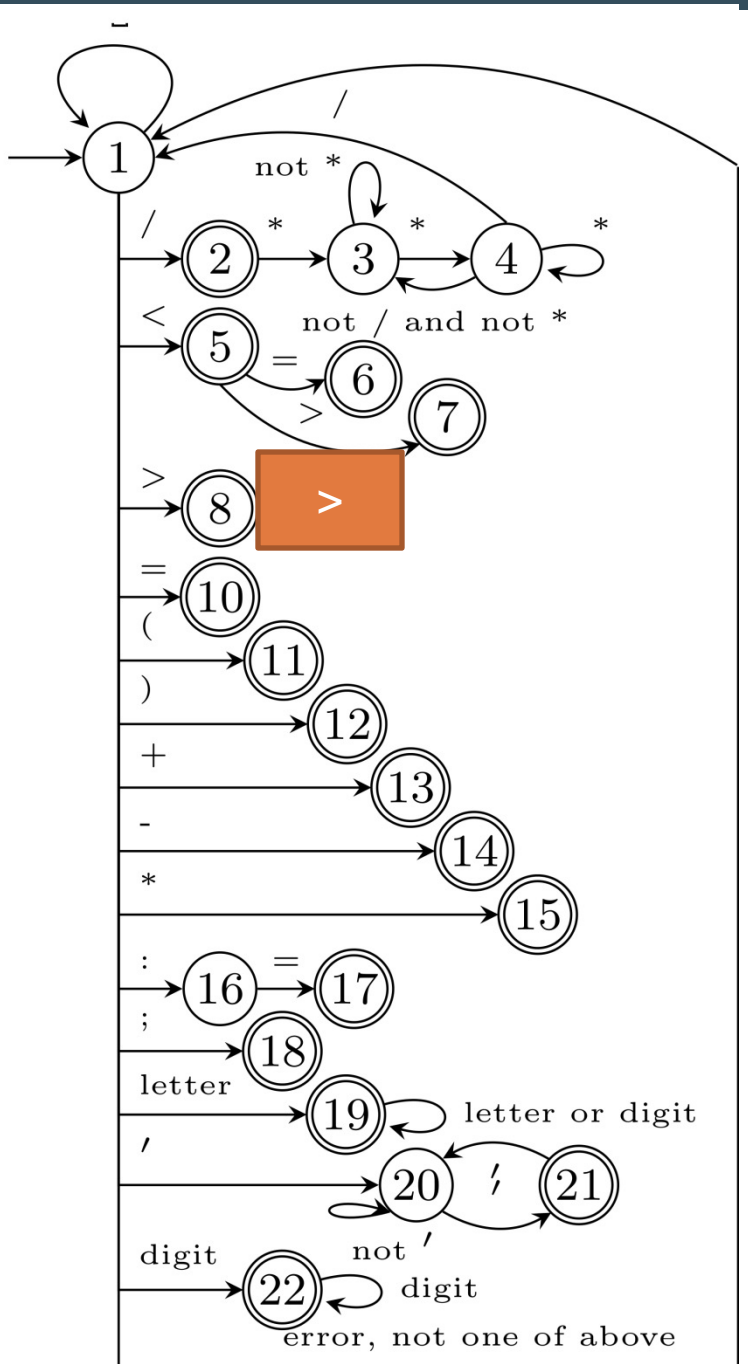
<=

Algoritmo

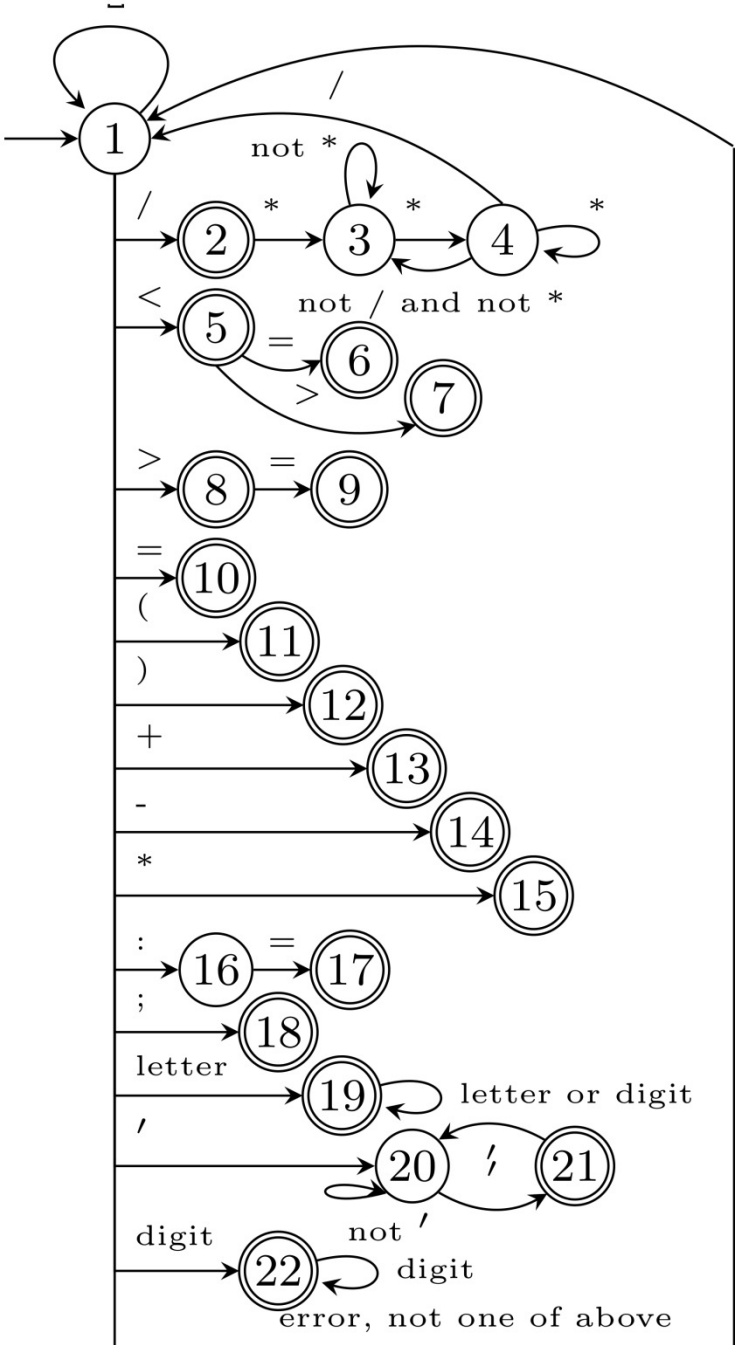

1
/
not *
/
*
*
*
2
3
4
<
not / and not *
5
=
6
>
7
>
=
8
9
=
10
(
11
)
12
+
13
-
14
*
15
:
=
16
17
;
18
letter
19
letter or digit
'
20
'
21
digit
not '
22
digit
error, not one of above

<>

Algoritmo

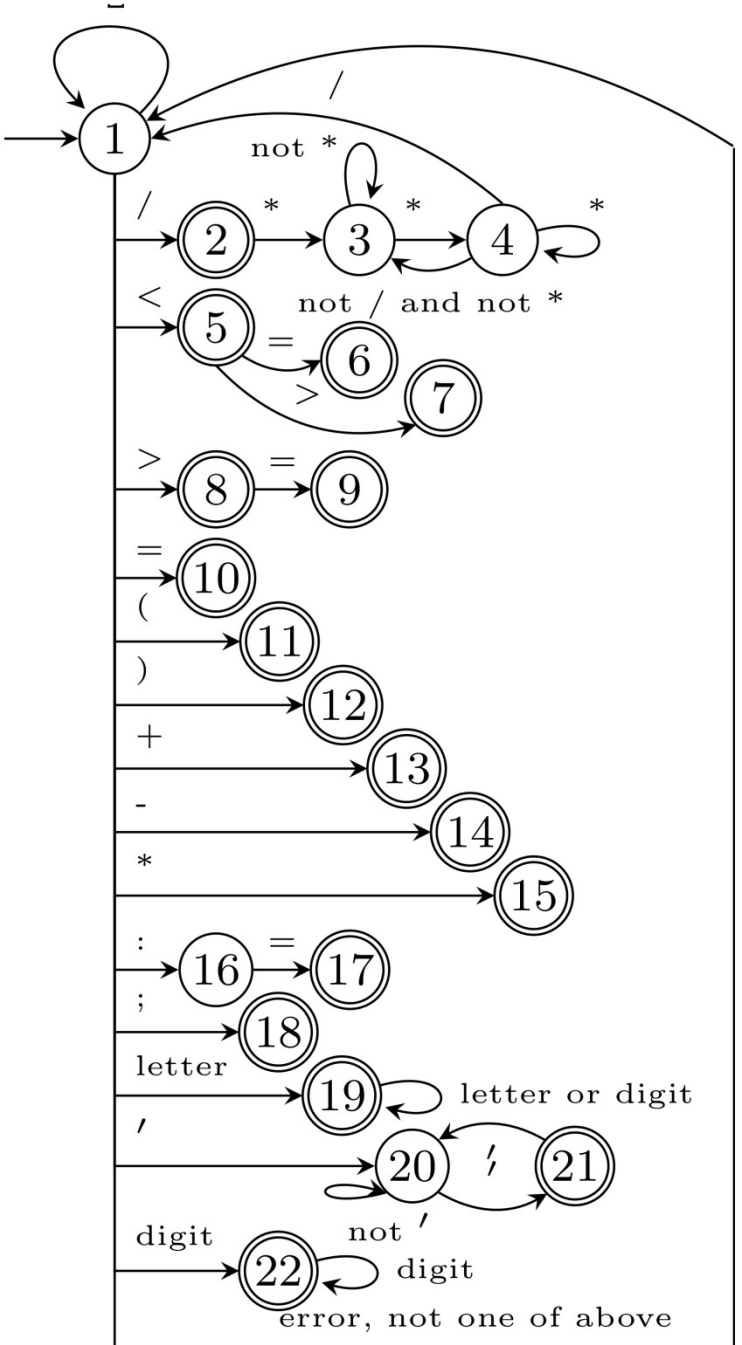

1
/
not *
/
*
*
*
2
3
4
<
5
not / and not *
=
6
>
7
>
8
>
=
10
(
11
)
12
+
13
-
14
*
15
:
=
16
17
;
18
letter
19
letter or digit
'
20
'
21
digit
not '
22
digit
error, not one of above

Algoritmo


1
/
not *
/
*
*
*
2
3
4
<
5
not / and not *
=
6
>
7
>
=
8
9
=
10
(
11
)
12
+
13
-
14
*
15
:
=
16
17
;
18
letter
19
letter or digit
'
20
'
21
digit
not '
22
digit
error, not one of above

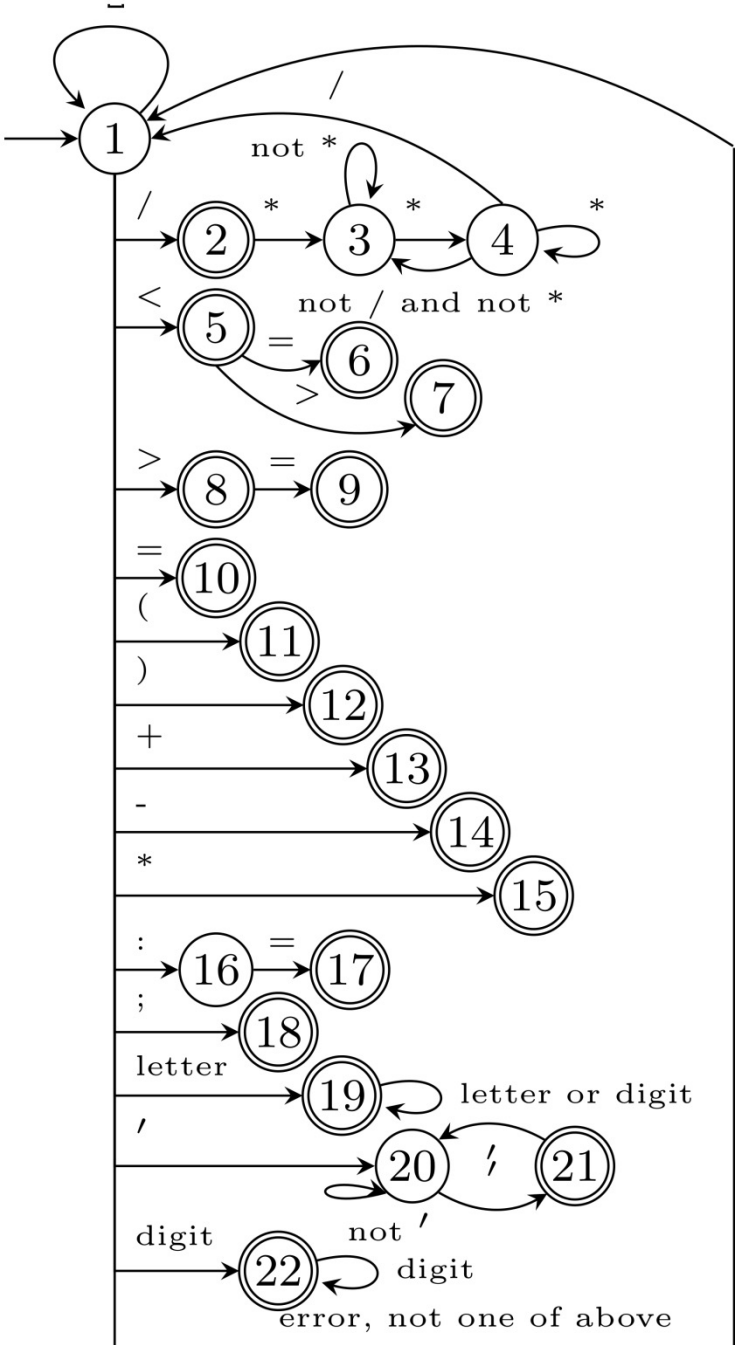
>=

Algoritmo


1
/
not *
2
3
4
*
*
*
/
<
5
not / and not *
=
6
>
7
>
8
=
9
=
10
(
11
)
12
+
13
-
14
*
15
:
16
=
17
;
18
letter
19
letter or digit
'
20
'
21
not '
digit
22
digit
error, not one of above

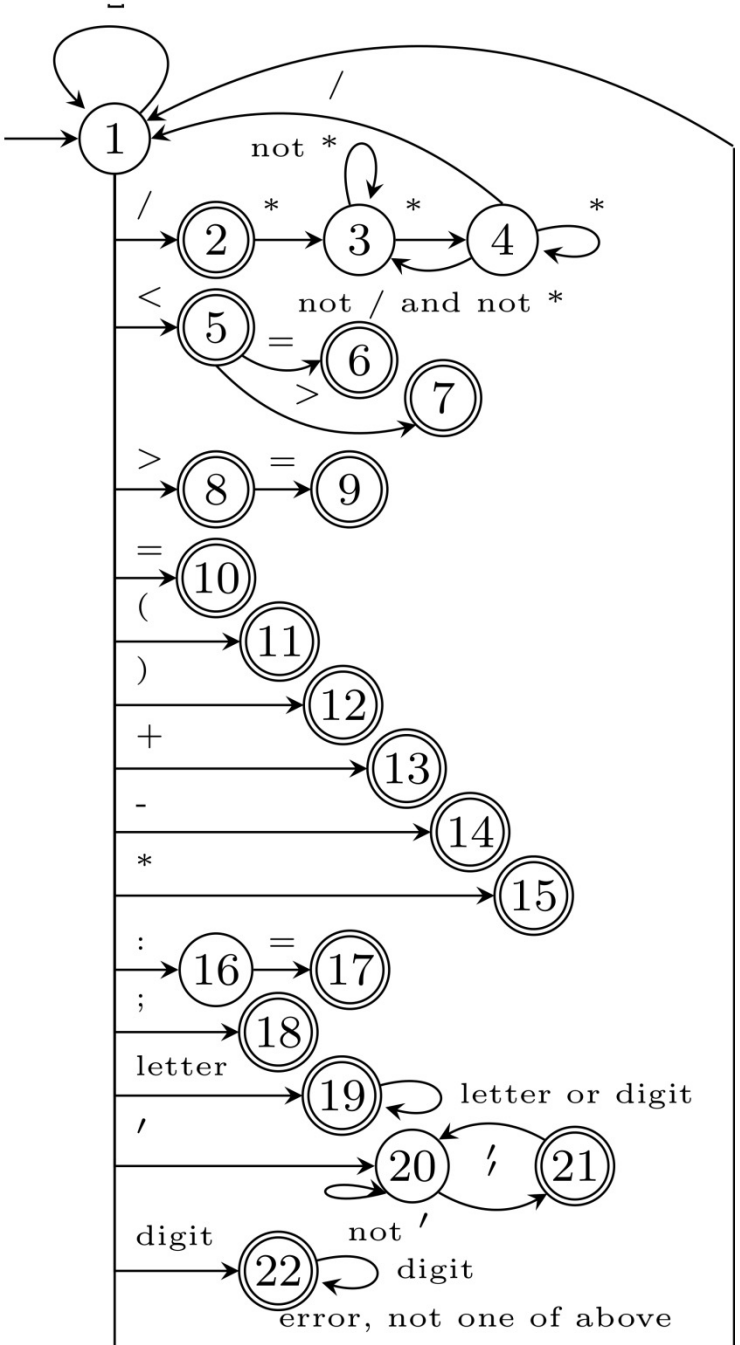
=

Algoritmo


1
/
not *
2
3
4
*
*
*
/
<
5
not / and not *
=
6
>
7
>
8
=
9
=
10
(
11
)
12
+
13
-
14
*
15
:
16
=
17
;
18
letter
19
letter or digit
'
20
21
digit
not '
22
digit
error, not one of above

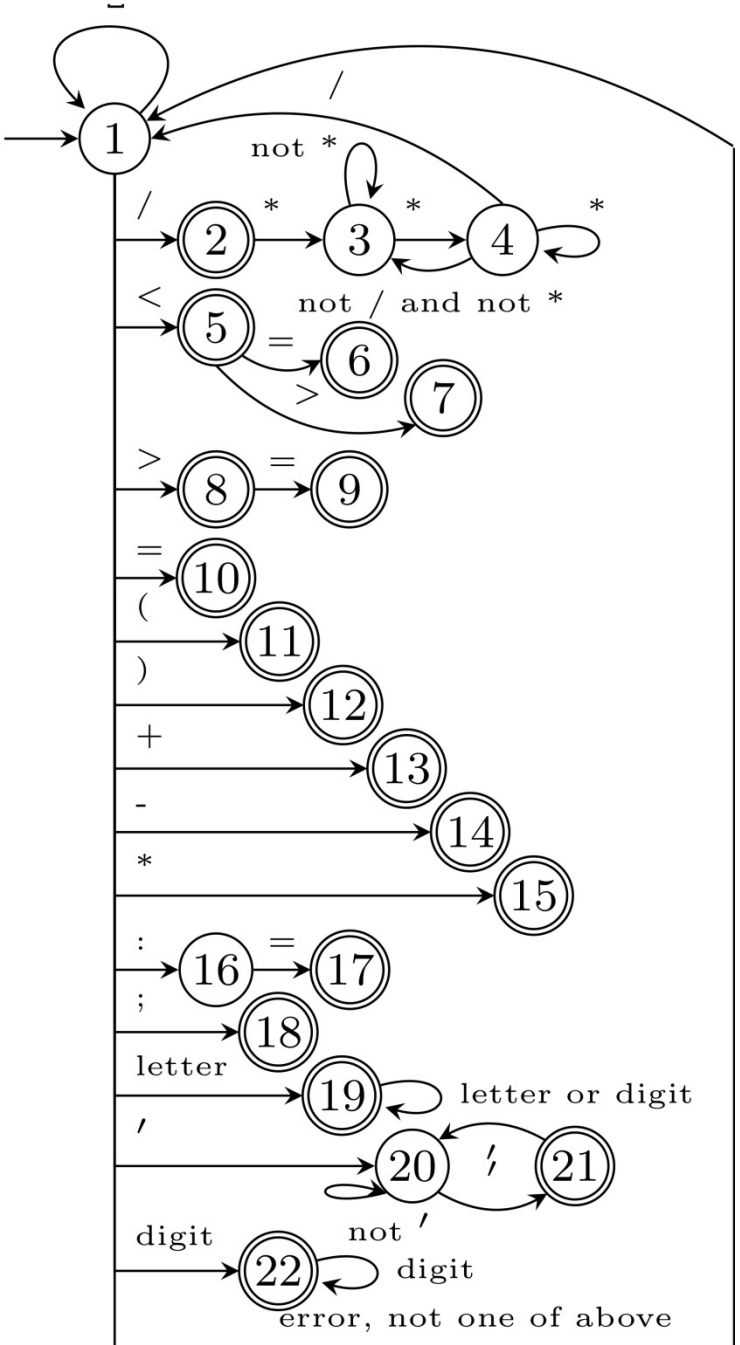
(

Algoritmo


1
/
not *
/
*
*
*
2
3
4
<
not / and not *
5
=
6
>
7
>
=
8
9
=
10
(
11
)
12
+
13
-
14
*
15
:
=
16
17
;
18
letter
19
letter or digit
'
20
'
21
digit
not '
22
digit
error, not one of above

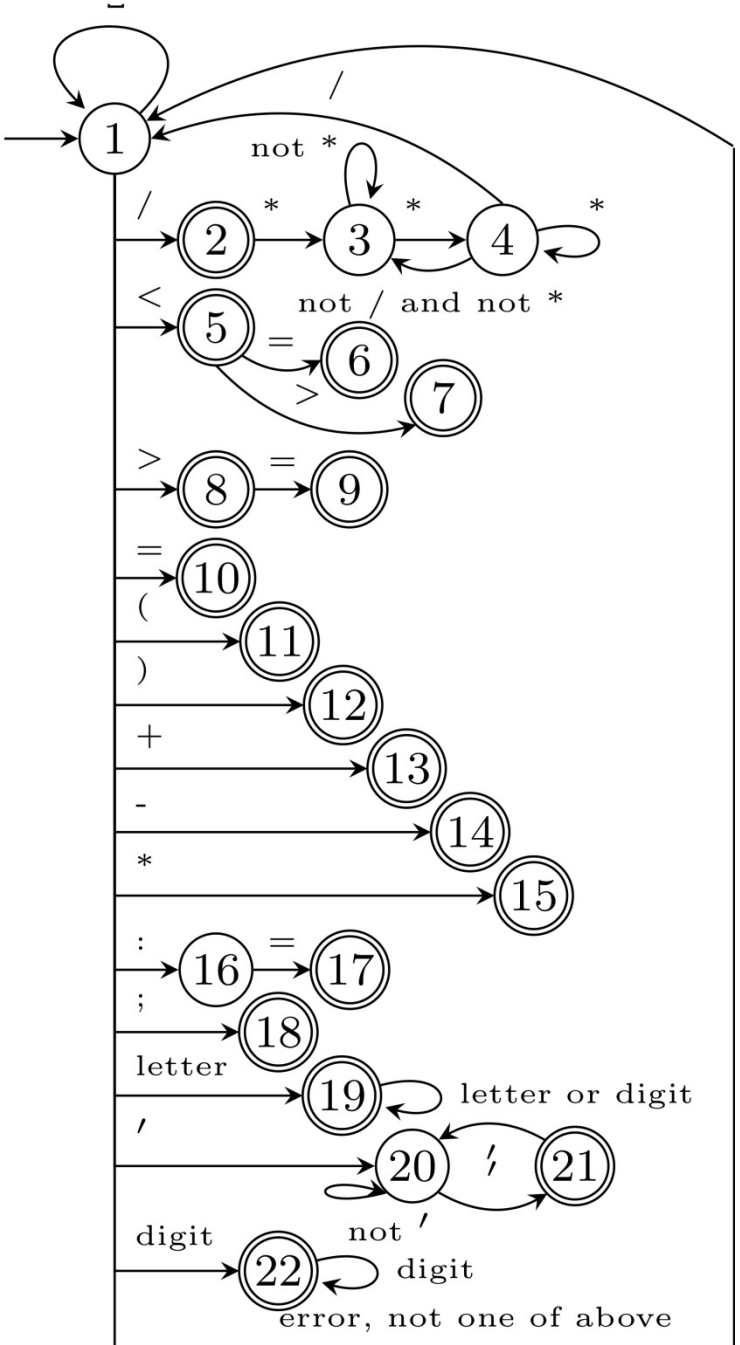
)

Algoritmo


1
/
not *
2
3
4
*
*
*
/
<
5
not / and not *
=
6
>
7
>
8
=
9
=
10
(
11
)
12
+
13
-
14
*
15
:
16
=
17
;
18
letter
19
letter or digit
'
20
'
21
digit
not '
22
digit
error, not one of above

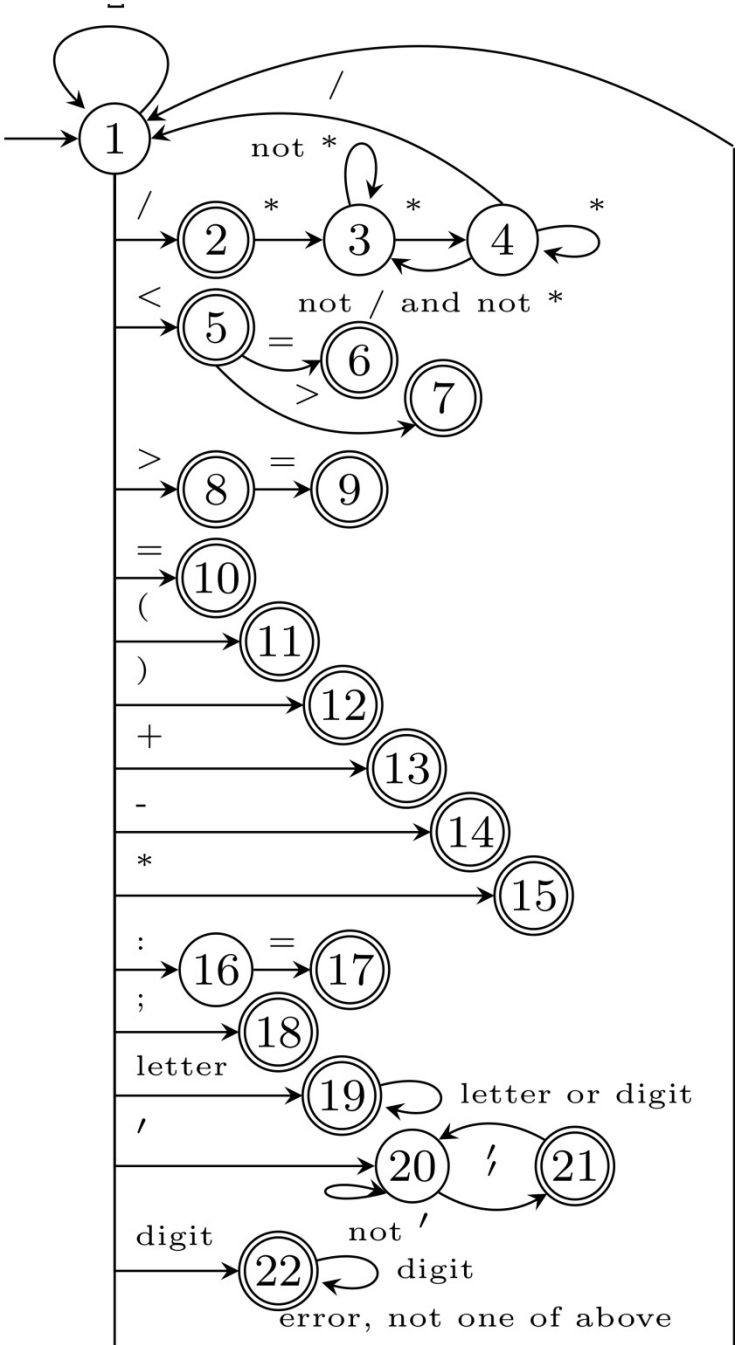
+

Algoritmo


1
/
not *
/
*
*
*
2
3
4
<
not / and not *
5
=
6
>
7
>
=
8
9
=
10
(
11
)
12
+
13
-
14
*
15
:
=
16
17
;
18
letter
19
letter or digit
'
20
'
21
digit
not '
22
digit
error, not one of above

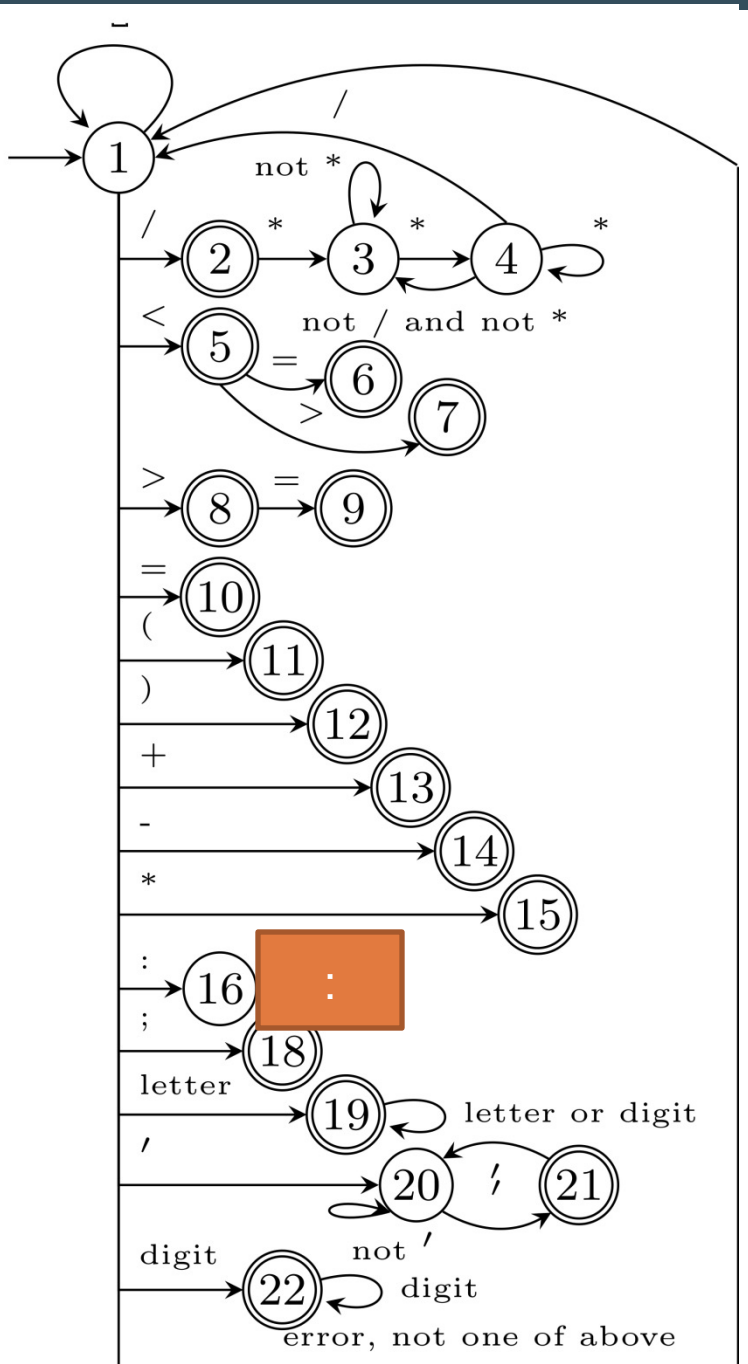
-

Algoritmo

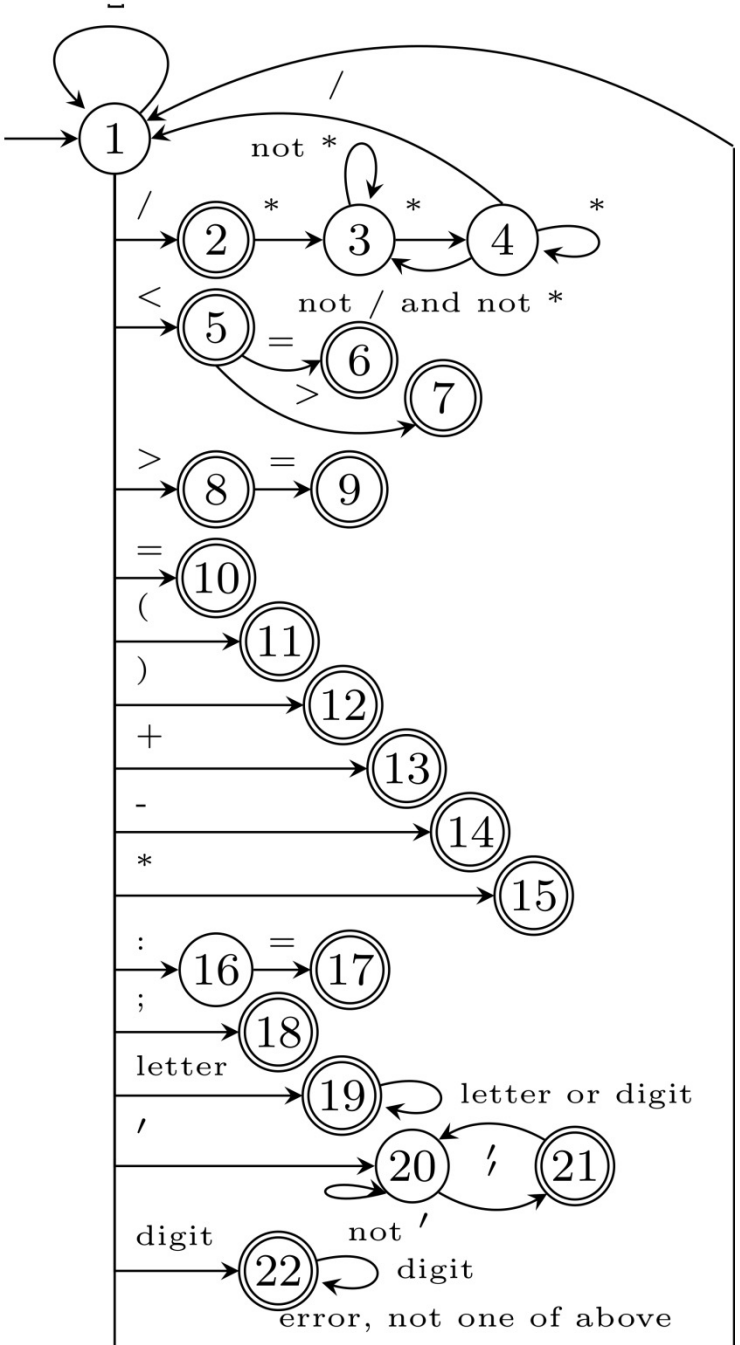

1
/
not *
/
*
*
*
2
3
4
<
not / and not *
5
=
6
>
7
>
=
8
9
=
10
(
11
)
12
+
13
-
14
*
15
:
=
16
17
;
18
letter
19
letter or digit
'
20
'
21
digit
not '
22
digit
error, not one of above

*

Algoritmo

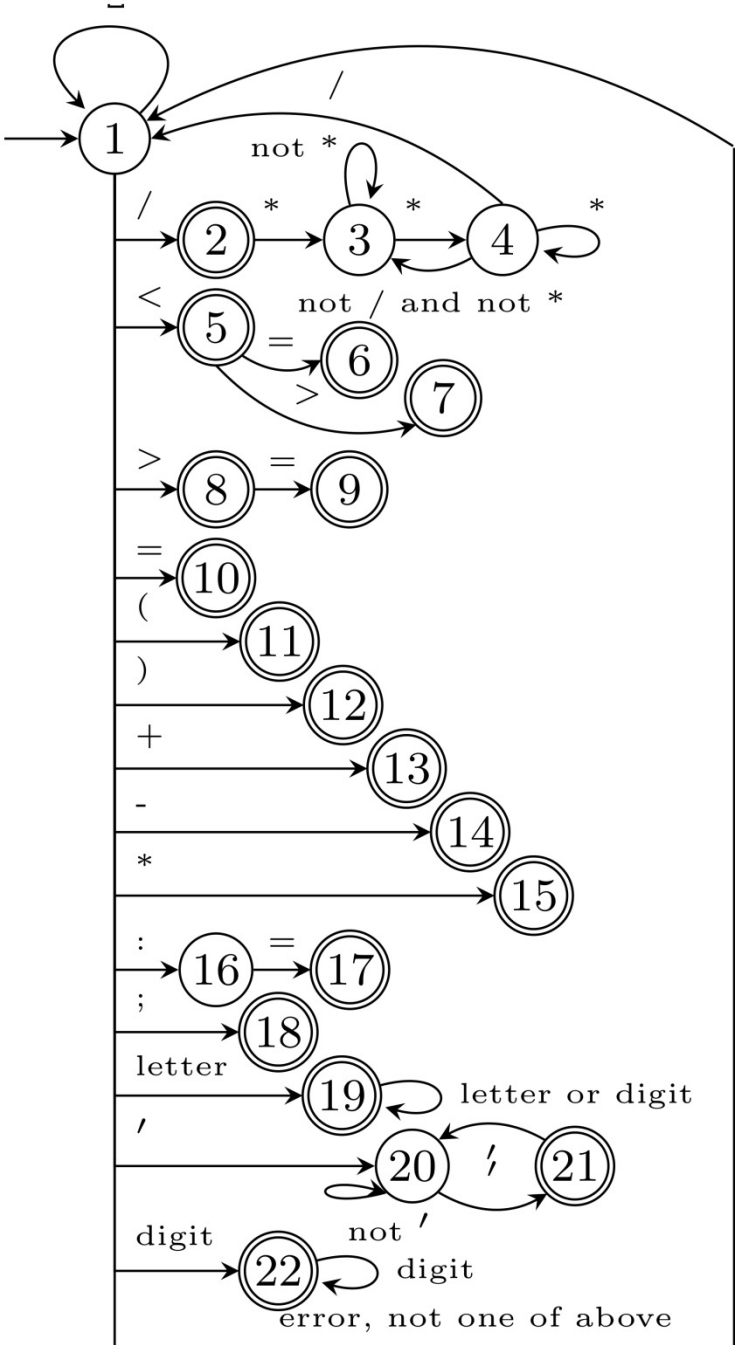

1
/
not *
/
*
*
*
2
3
4
<
not / and not *
5
=
6
>
7
>
=
8
9
=
10
(
11
)
12
+
13
-
14
*
15
:
16
:
;
18
letter
19
letter or digit
'
20
'
21
not '
digit
22
digit
error, not one of above

Algoritmo


1
/
not *
/
*
*
*
2
3
4
<
not / and not *
5
=
6
>
7
>
=
8
9
=
10
(
11
)
12
+
13
-
14
*
15
:
=
16
17
;
18
letter
19
letter or digit
'
20
'
21
digit
not '
22
digit
error, not one of above

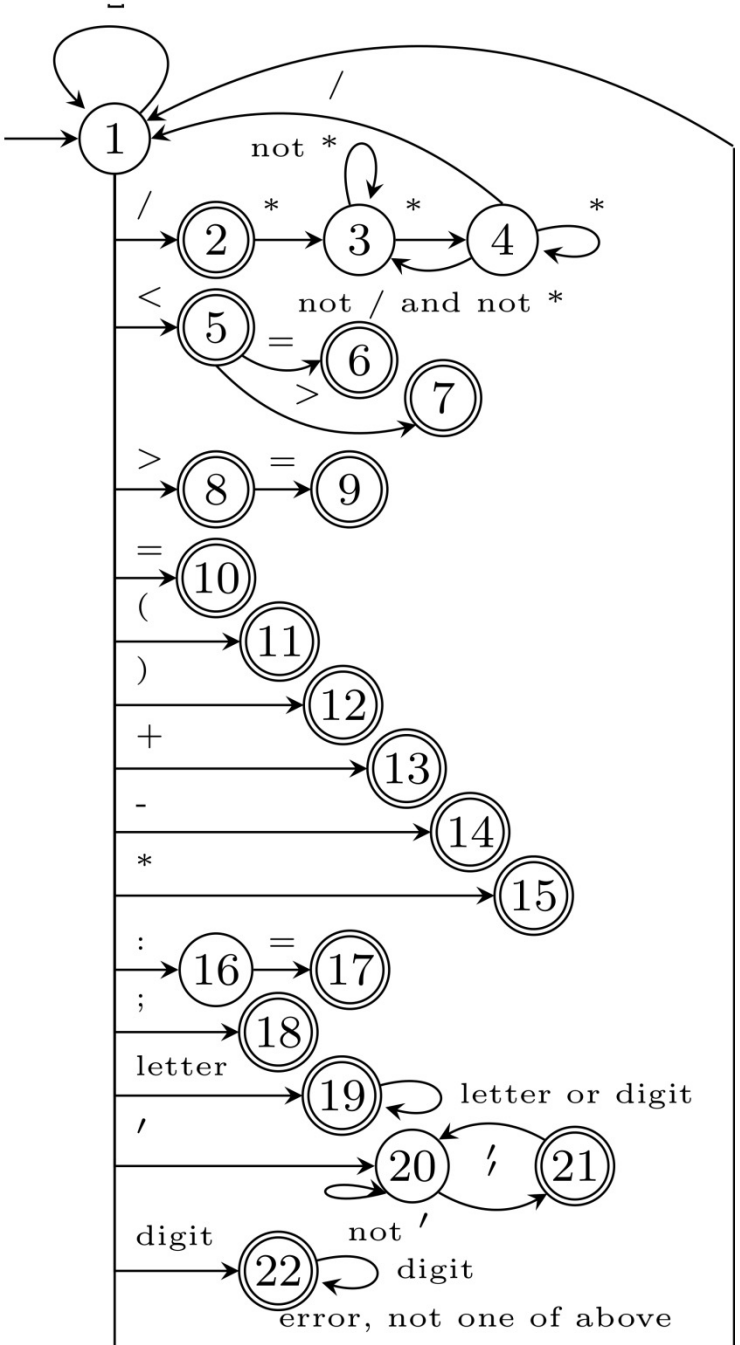
:=

Algoritmo


1
/
not *
/
*
2
*
3
*
4
*
<
5
not / and not *
=
6
>
7
>
=
8
9
=
10
(
11
)
12
+
13
-
14
*
15
:
=
16
17
;
18
letter
19
letter or digit
'
20
'
21
digit
not '
22
digit
error, not one of above

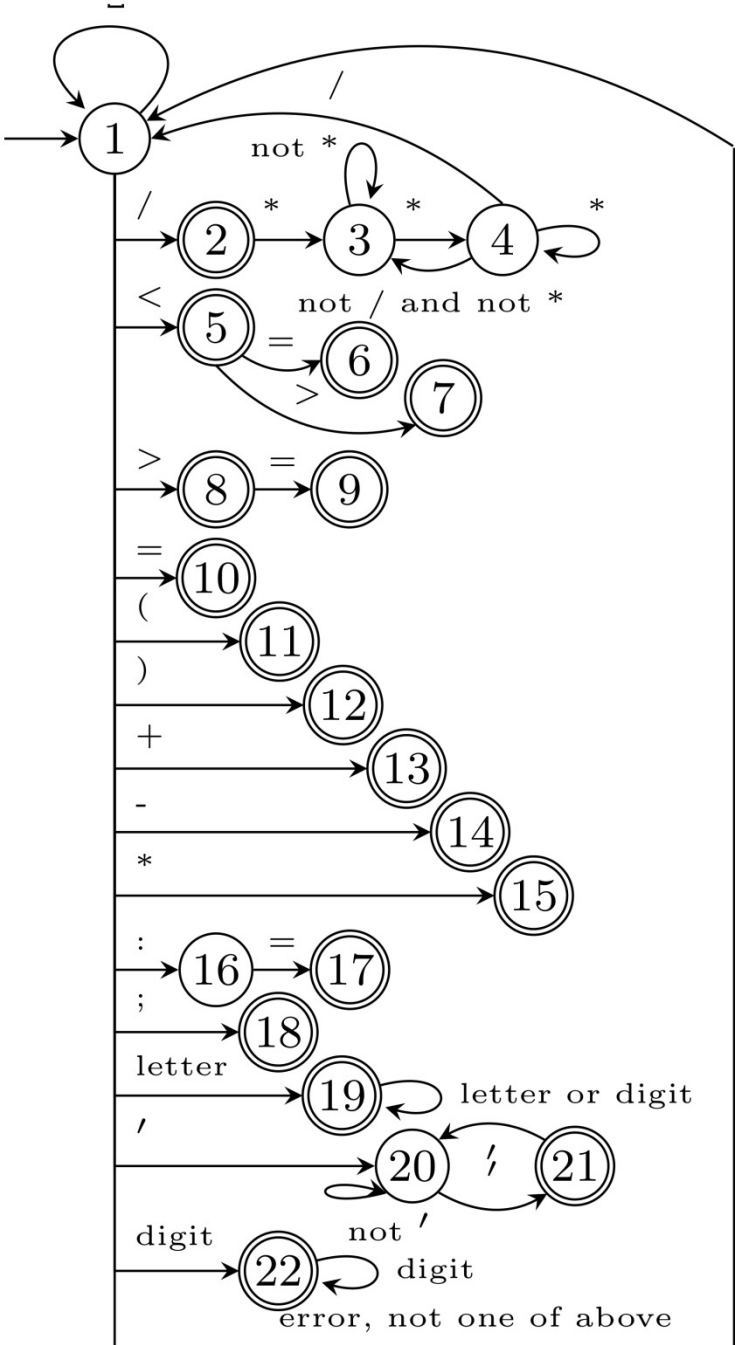
;

Algoritmo



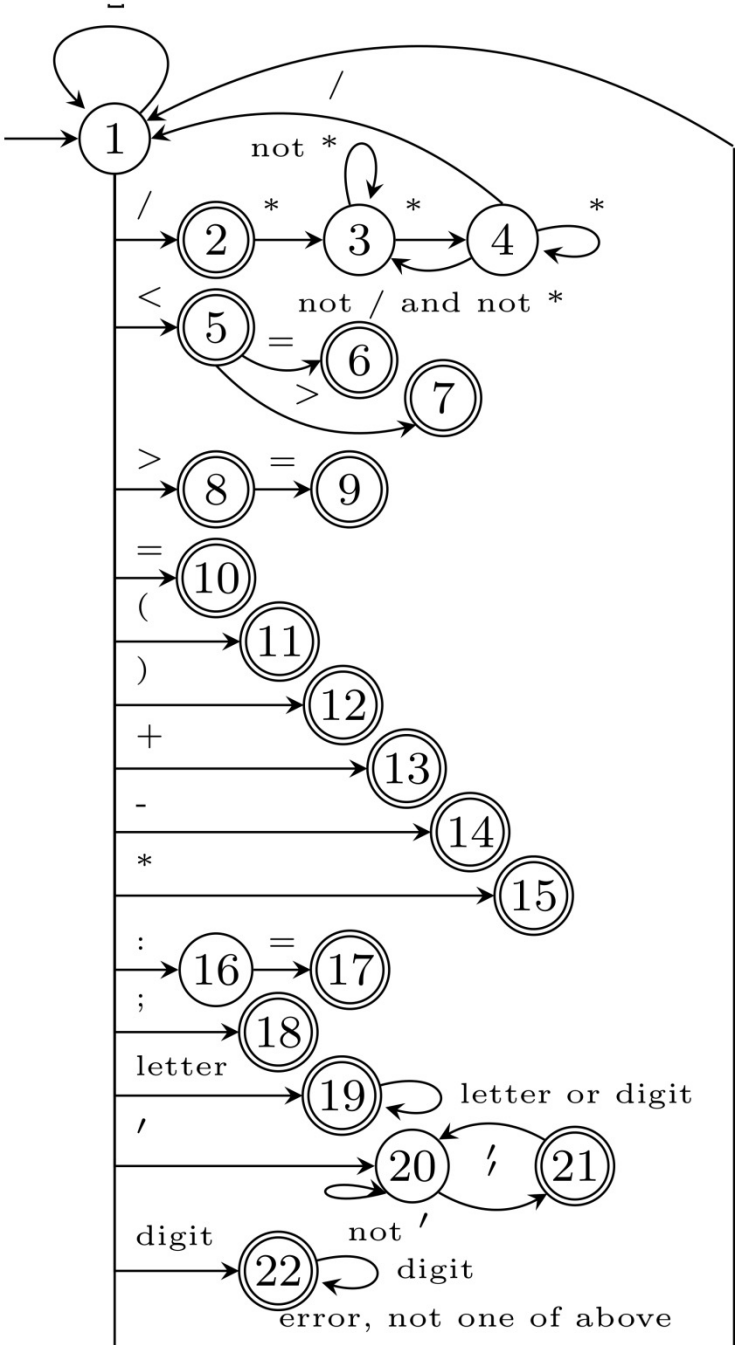
If token in the keyword table, then keyword;
else identifier: add to symbol table

Algoritmo



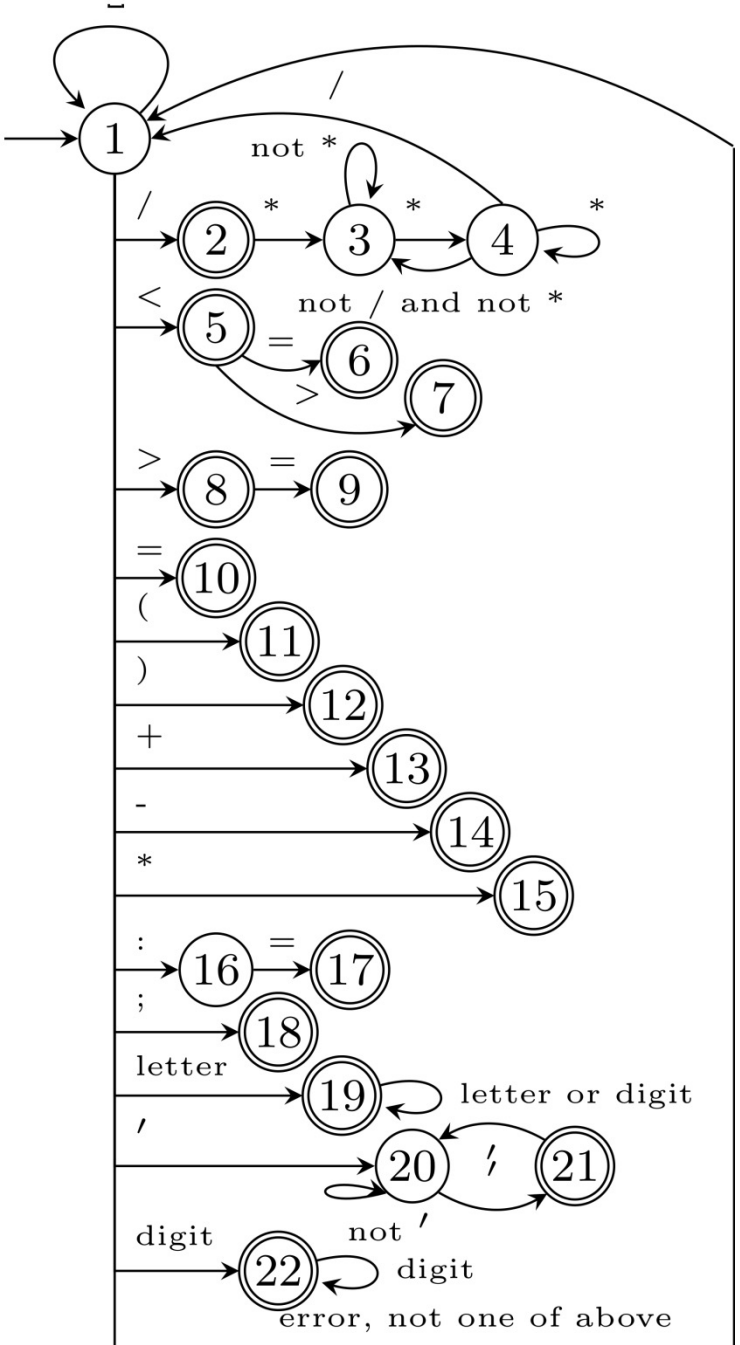
Literal: add to symbol table

Algoritmo



Constant: add to symbol table

Algoritmo



Print erro message for invalid character

Linguaggi regolari e scanner

- I linguaggi regolari vengono riconosciuti in modo efficiente attraverso automi a stati finiti. **Tuttavia, sono linguaggi abbastanza limitati** e con le loro grammatiche non si possono descrivere anche semplici costrutti dei linguaggi di programmazione.
- **L'utilizzo delle grammatiche regolari è perciò limitato alla costruzione (e riconoscimento) dei simboli base usati in un programma** (quello che si indica generalmente come lessico), ad esempio gli identificatori.

Linguaggi regolari e scanner

- Le grammatiche regolari possono essere usate per rappresentare scanner che sono implementati come automi a stati finiti.
- Poiché realizzare un automa a stati finiti è banale **sono stati progettati programmi in grado di generare automaticamente degli scanner** usando un metodo formale per specificare l'automa
- Un noto generatore di scanner è Lex (1975) che utilizza espressioni regolari per specificare lo scanner.
  - Lo scanner generato da Lex può essere usato in congiunzione con un parser (YACC) per eseguire sia l'analisi lessicale che sintattica.

Linguaggi regolari e scanner

- L'analisi del programma prosegue poi con tecniche più complesse (**analisi sintattica e analisi semantica**).
- Il parser potrebbe fare direttamente anche l'analisi sintattica, ma non è conveniente in quanto la grammatica per i token è una grammatica regolare più semplice quindi di quella che tratta il parser.
- Lo scanner può interagire con il parser in due modi differenti:
  - Lavorare in un passo separato, producendo i token in una grossa tabella in memoria di massa;
  - Interagire direttamente con il parser che chiama lo scanner quando è necessario il prossimo token nell'analisi sintattica (preferibile).

Analizzatore sintattico (parser)

- Input: lista di token
- **Individua la struttura sintattica** della stringa in esame a partire dal programma sorgente sotto forma di token.
- Identifica quindi espressioni, istruzioni, procedure.
- Output: albero sintattico

Analizzatore sintattico (parser)

- Esempio `ALFA1:=5+A*B`

Analizzatore sintattico (parser)

- Esempio `ALFA1:=5+A*B`
- Lo scanner ha già riconosciuto alcuni elementi dell'istruzione come operatori (+, *), identificatori (A, B) e costanti.
- Seguendo la sintassi del linguaggio, l'intera stringa `5+A*B` è riconosciuta come `<espressione>`
- La stringa completa è riconosciuta come `<assegnazione>` in accordo alla regola sintattica:

```
<assegnazione>::= <variabile> := <espressione>
```

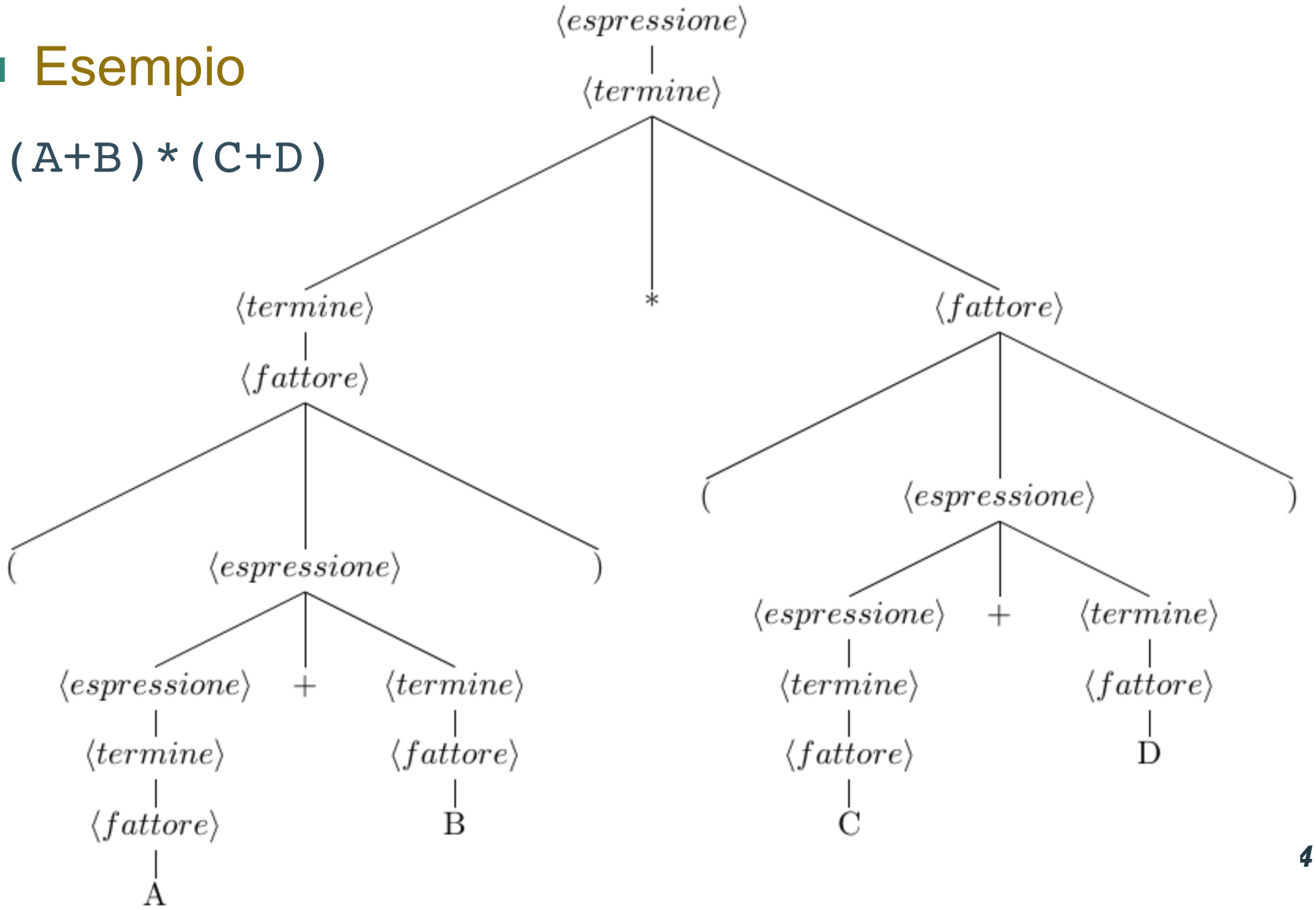
Analizzatore sintattico (parser)

- Esempio `(A+B)*(C+D)`
- L'analisi produce le classi sintattiche:
  - `<fattore>`
  - `<termine>`
  - `<espressione>`
- Il controllo sintattico si basa sulle *regole grammaticali* utilizzate per definire formalmente il linguaggio.
- Durante il controllo sintattico si genera l'albero di derivazione (*albero sintattico*)

Analizzatore sintattico (parser)

- Esempio

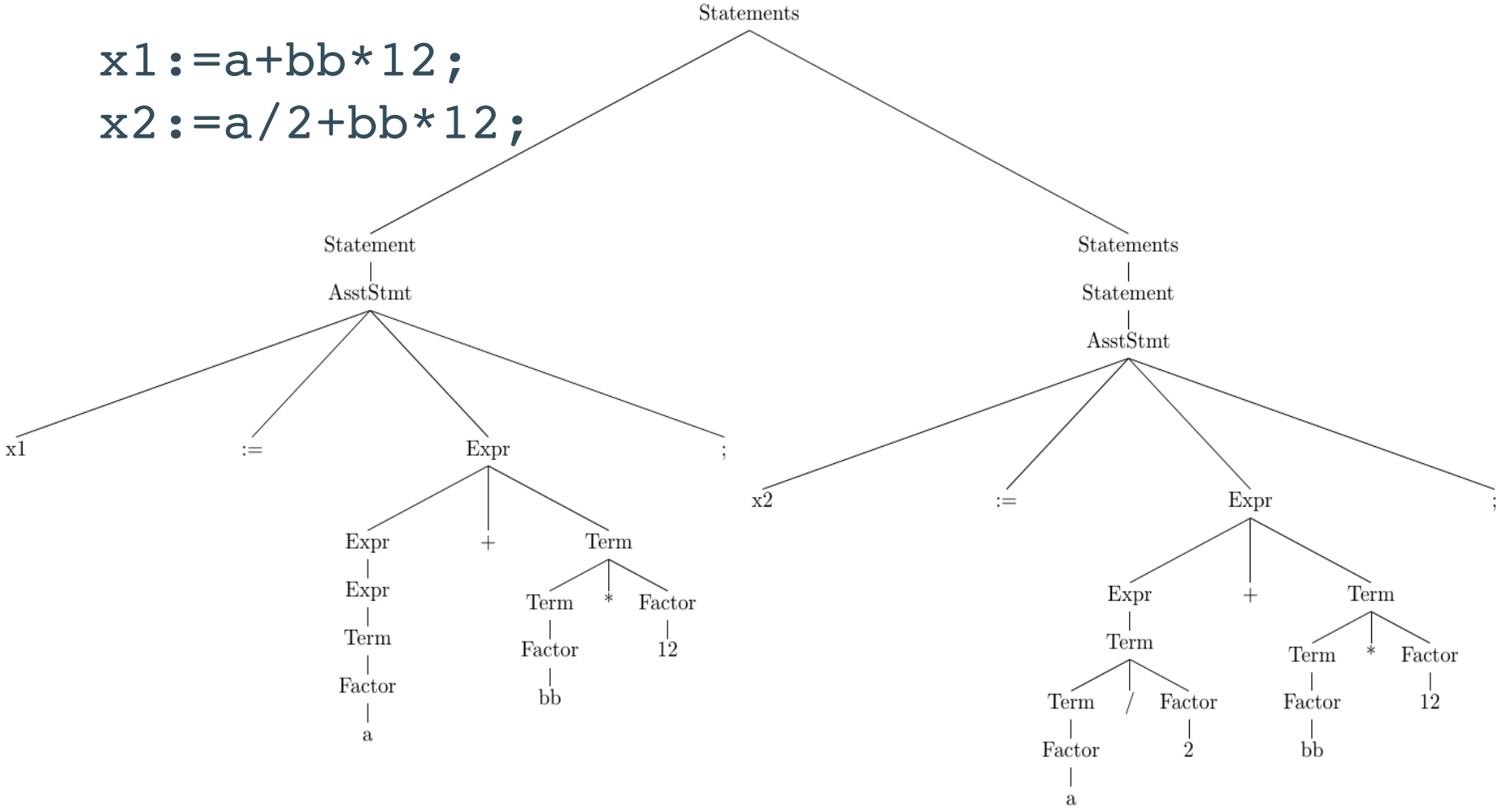
`(A+B)*(C+D)`



Analizzatore sintattico (parser)

- Esempio

```
x1:=a+bb*12;
x2:=a/2+bb*12;
```



Analizzatore sintattico

- Alla fine del processo, lo strumento restituisce un'indicazione della **correttezza sintattica** del codice sorgente

Analizzatore semantico

- Input: albero sintattico generato dal parser.
- Si compone di due fasi principali:
  1. Controlli statici (static checking).
  2. Generazione di una rappresentazione intermedia (IR)
- **Output:** albero arricchito con informazioni sui vincoli sintattici contestuali

Analizzatore semantico

- Input: albero sintattico generato dal parser.
- Si compone di due fasi principali:
  1. **Controlli statici (static checking).**
     Sono svolti vari controlli sui *tipi, dichiarazioni, numero parametri funzioni, etc.*

     Per l'espressione `(A+B)*(C+D)`, ad esempio, l'analizzatore semantico deve determinare quali azioni sono specificate dagli operatori aritmetici di **addizione e moltiplicazione**.

     Ad ogni token che corrisponde ad un identificatore di variabile è associato: tipo, luogo di dichiarazione, etc memorizzate nella tabella dei simboli.

     Quando riconosce il + od il * invoca allora una routine semantica che specifica le azioni da svolgere. Ad esempio, che gli operandi siano stati dichiarati, abbiano lo stesso tipo ed un valore.

Analizzatore semantico

- Input: albero sintattico generato dal parser.
- Si compone di due fasi principali:
  2. **Generazione di una rappresentazione intermedia (IR)**
     Spesso la parte di analisi semantica produce anche una forma intermedia di codice sorgente.

     Ad esempio può produrre il seguente insieme di quadruple per `(A+B)*(C+D)`:

     ```
     (+,A,B,T1)
     (+,C,D,T2)
     (*,T1,T2,T3)
     ```

     Od altri tipi di codice intermedio.

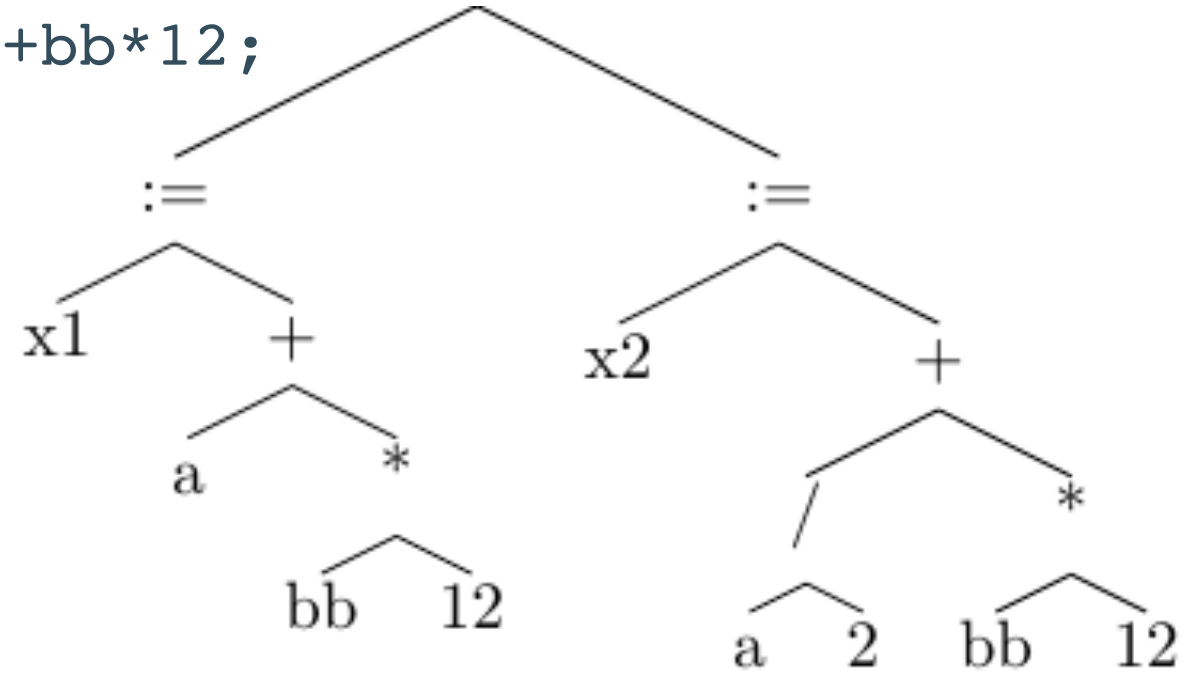
Analizzatore semantico

- Esempio: codice intermedio **che rimuove dall'albero sintattico alcune delle categorie intermedie** e mantiene solo la struttura essenziale (albero sintattico astratto).
  - Tutti i nodi sono token.
  - Le foglie sono operandi.
  - I nodi intermedi sono operatori.
- Spesso a valle dell'analizzatore semantico ci può essere un ottimizzatore del codice intermedio.

Analizzatore semantico

- Esempio:

```
x1:=a+bb*12;
x2:=a/2+bb*12;
```



Analizzatore semantico

- **Ottimizzazione del codice intermedio**: propagazione di costanti
- Esempio

```
X:=3;
A:=B+X;
```

- Si può ottimizzare come:

```
X:=3;
A:=B+3;
```

evitando un accesso alla memoria.

Analizzatore semantico

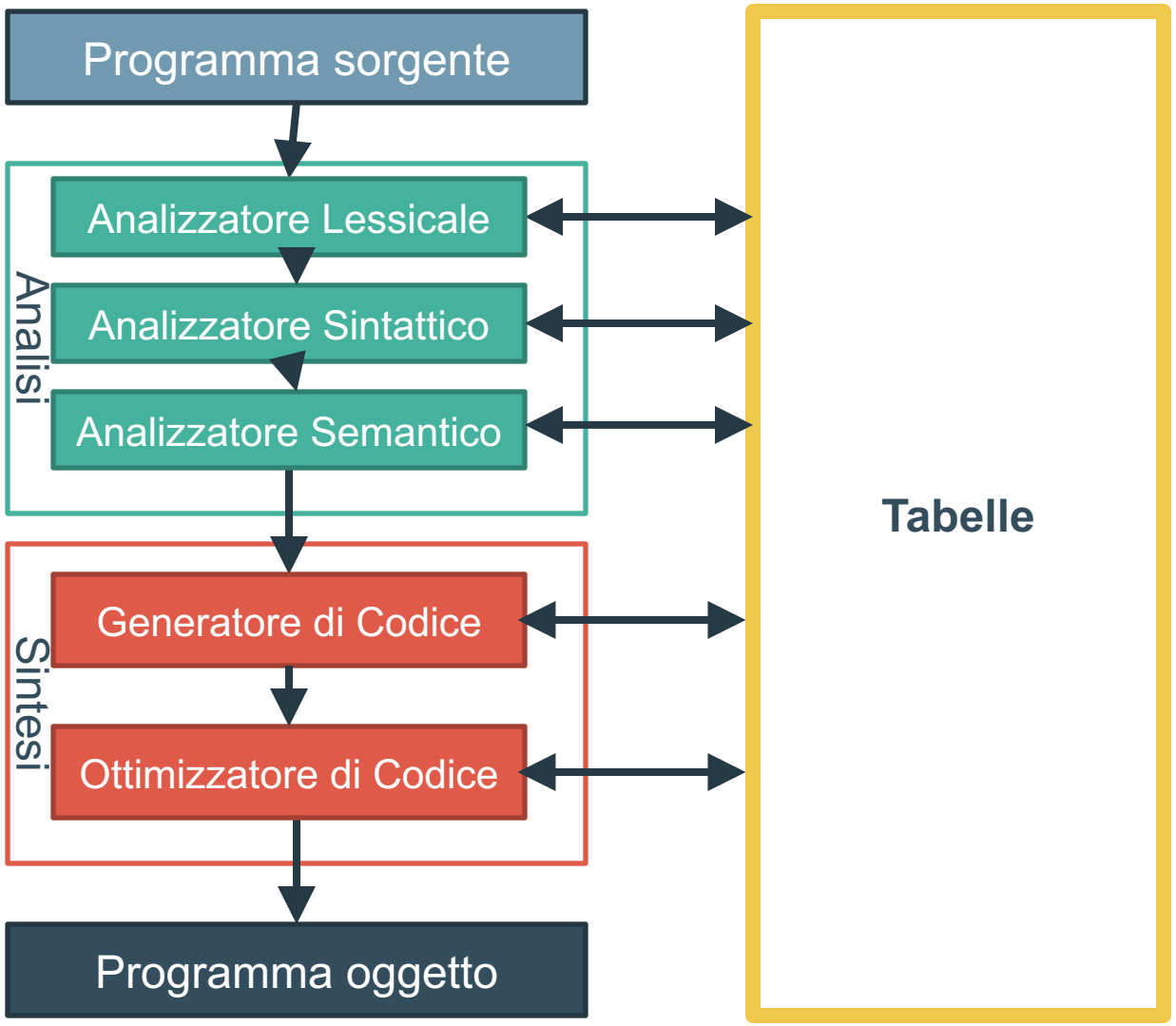
- Ottimizzazione del codice intermedio: eliminazione di sotto-espressioni comuni
- Esempio

```
A:=B*C;
D:=B*C;
```

- Si trasforma in:

```
T:=B*C;
A:=T;
D:=T;
```

Compilatore



Generatore di codice

- L'output dell'analizzatore semantico è passato al **generatore di codice che trasla la forma intermedia in linguaggio assembler o macchina**;
- Prima della generazione del codice oggetto ci sono delle fasi di preparazione:
  - **Allocazione della memoria:** può essere allocata staticamente oppure è uno stack o heap la cui dimensione cambia durante l'esecuzione;
  - **Allocazione dei registri:** poiché l'accesso ai registri è più rapido dell'accesso alle locazioni di memoria, i valori cui si accede più spesso andrebbero mantenuti nei registri.

Generatore di codice

- Esempio

```
x1:=a+bb*12;
x2:=a/2+bb*12;
```

- Potremmo pensare di allocare l'espressione `bb*12` al registro 1, ed una copia del valore di `a` al registro 2 assieme al valore `a/2`.
- Le variabili si potrebbero allocare sullo stack con `a` al top, e poi, nell'ordine, `bb`, `x1`, `x2`. Il registro S punta al top dello stack.
- Segue poi la vera e propria generazione di codice.

Generatore di codice

- Esempio

```
(A+B)*(C+D):
(+,A,B,T1)
(+,C,D,T2)
(*,T1,T2,T3)
```

- Si possono produrre quindi le seguenti istruzioni assembler

```
LOADA A
LOADB B
ADD
STOREA T1
LOADA C
LOADB D
ADD
STOREA T2
LOADA T1
LOADB T2
MULT
STOREA T3
```

Ottimizzatore di codice

- L'output del generatore di codice è passato in input **all'ottimizzatore di codice**, presente nei compilatori più sofisticati.
- Ottimizzazioni indipendenti dalla macchina: ad esempio la rimozione di istruzioni invarianti all'interno di un loop, fuori dal loop, etc.
- Ottimizzazioni dipendenti dalla macchina: ad esempio ottimizzazione dell'uso dei registri

Ottimizzatore di codice

- L'output del generatore di codice è passato in input all'ottimizzatore di codice, presente nei compilatori più sofisticati
- Esempio: ottimizzazione del codice precedente

```
LOADA A
LOADB B
ADD
STOREA T1
LOADA C
LOADB D
ADD
LOADB T1
MULT
STOREA T3
```

Ottimizzatore di codice

- L'output del generatore di codice è passato in input all'ottimizzatore di codice, presente nei compilatori più sofisticati
- Esempio: ottimizzazione del codice precedente

```
LOADA A
LOADB B
ADD
STOREA T1
LOADA C
LOADB D
ADD
LOADB T1
MULT
STOREA T3
```

```
LOADA A
LOADB B
ADD
STOREA T1
LOADA C
LOADB D
ADD
STOREA T2
LOADA T1
LOADB T2
MULT
STOREA T3
```

Passi di un compilatore

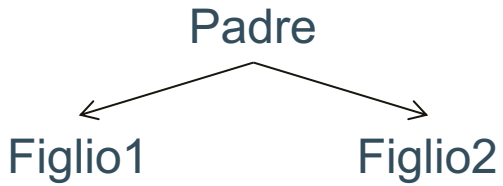
- Abbiamo ignorato altri aspetti importanti della compilazione:
  1. **Error Detection e Recovery;**
  2. Le **Tabelle dei Simboli** prodotte dai vari moduli;
  3. La **Gestione della Memoria** implicata da alcuni costrutti del linguaggio di alto livello.
- Le fasi di più semplice progettazione, con un apparato formale ben sviluppato e quindi facilmente automatizzabili sono scanner e parser, mentre maggiore difficoltà si trova nella progettazione di analizzatori semantici, generatori ed ottimizzatori di codice.

Linking e Caricamento

- Il programma oggetto prodotto dal compilatore contiene una serie di riferimenti esterni (es. riferimenti a programmi di libreria, funzioni).
- **I riferimenti esterni vengono risolti dal linker.**
- Il programma è rilocabile: può essere allocato in diverse zone di memoria cambiando indirizzo `ind` (indirizzamento relativo).
- Fase di caricamento compiuta dal **loader** che assegna un valore numerico all'indirizzo `ind`, trasformando gli indirizzi relativi in assoluti.

Linking e Caricamento

- Esempio



Padre	indirizzi relativi	da 0 a 300
Figlio1		da 0 a 120
Figlio2		da 0 a 150

Il linker riceve in ingresso questi tre moduli e genera un unico modulo con riferimento ad indirizzi contigui a partire da un indirizzo simbolico `ind`.

Ogni riferimento a moduli esterni viene sostitutito con l'indirizzo così calcolato.

Linking e Caricamento

- Esempio

Indirizzo	Contenuto	Commento
ind	Inizio Padre	
	…	
	salta ad ind + 301	rif. a Figlio1
	...	
	salta ad ind + 421	rif. a Figlio2
	…	
ind + 300	fine Padre	
ind + 301	inizio Figlio1	
	…	
ind + 420	fine Figlio1	
ind + 421	inizio Figlio2	
	…	
ind + 570	fine Figlio2	

Esecuzione del Codice