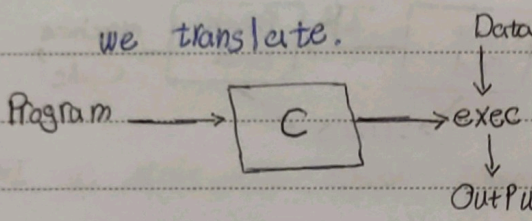
Subject:
Date

What is a compiler : Takes as input a program written in one language and translates it into a functionally equivalent program in another language.

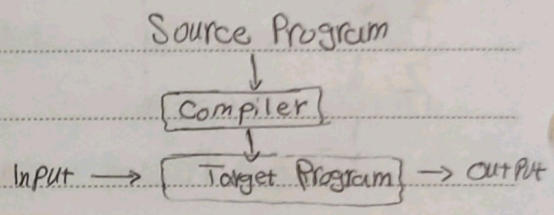
* Source is usually high-level, target is usuly Low Level.

Trans Lation Approaches

newer ← 1) Compiler : By following physical structure of Program we translate.

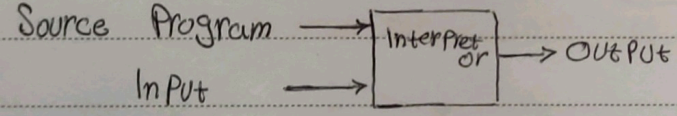


offline



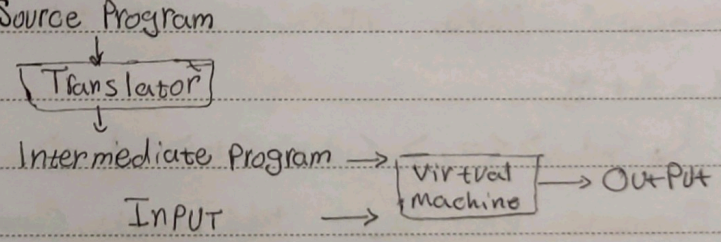
older ← 2) Interpreter : translate the source code to machine Code online

* The code is translated while it is running.



on-line

3) hybrid Approach : Source Program → Translator → Intermediate Program → virtual machine → OutPut; Input → virtual machine



How Does Compiler work?

tockenize 1) Lexical analysis (Scanning) : Identifying logical Piece of the description

Diagraming sentence 2) Syntax analysis (Parsing) : Identifying how ~~does~~ those Pieces relate to each other.

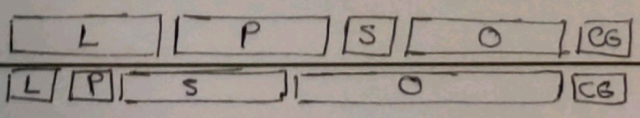
3) Semantic analysis : Identifying the meaning of the overall structure.

4) IR Generation : Design one Possible structure.

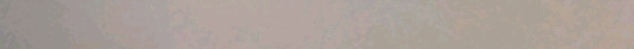
5) IR optimization : Simplify the intendended structure.

6) Generation : Fabricate the structure.

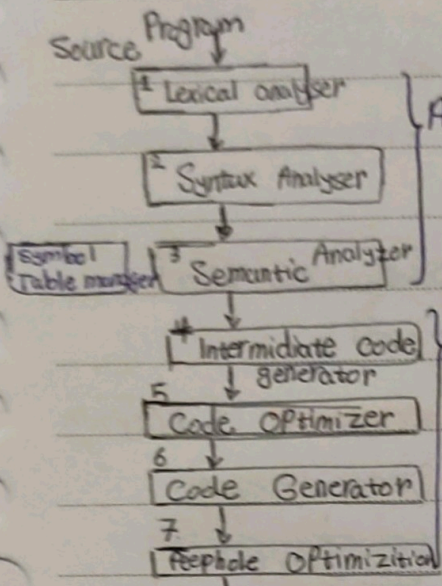
7) Optimization : Improving the resulting structure.



PAPCO



Subject:
Date

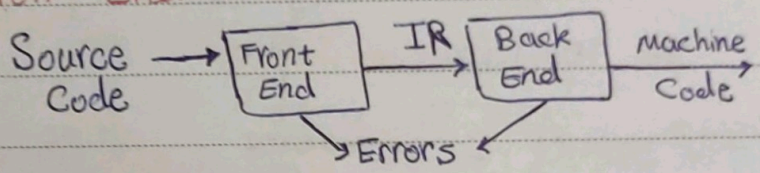


1,2,3,4,5 & Front-end

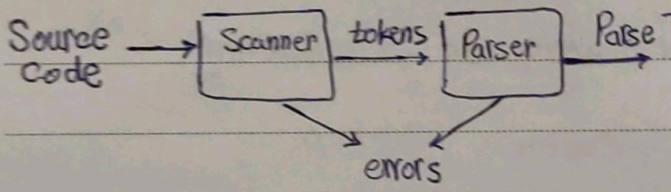
6,7 & Back-end

Error Handler

Front End



* Front End maps source code into an IR representation
* Back End maps IR onto machine code
* Simplifies retargeting



Scanner & Maps characters into tokens

$X = X + y$

↳ <id, x><=, > <id, x><+, ><id, y>

Eliminate white space (tabs, blancks, Comments)

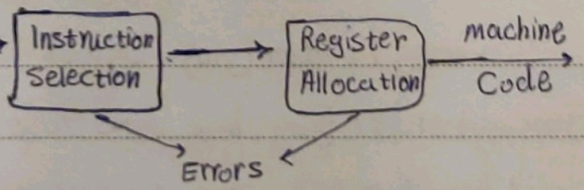
Parser &

Recognize context-free syntax

Guide context-sensitive analysis

Produce meaningful error messages

Back-end IR →

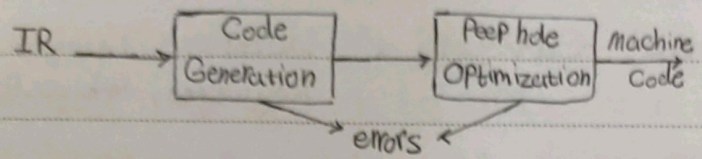


* Translate IR into machine code
* Choose instructions for each IR operation
* Decide what to keep in registers at each Point

PAPCO

Subject:
Date

Two Main Components of Back-end



Code Generator

Produce compact fast code

Use available addressing modes

Peephole Optimization :

Limited resources

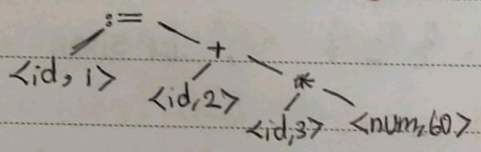
Optimal Allocation is difficult

Position = initial + rate * 60

lexical analyzer (Scanner)

<id, 1> <=> <id, 2> <+> <id, 3> <*> <num, 60>

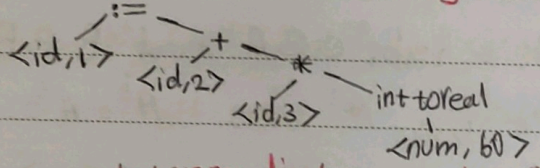
Syntax analyzer (Parser)



Symbol Table

1	Position	real
2	initial	real
3	rate	real

Semantic analyzer



Intermediate Code Generator

```
t1 := int to real (60)
t2 := id3 * t1
t3 := id2 + t2
id1 := t3
```

Code optimizer

```
t1 := id3 * 60.0
id1 := id2 + t1
```

Final Code Generator

```
LD  R1, id3
MUL R1, R1, #60.0
LD  R2, id2
ADD R1, R1, R2
ST  id1, R1
```

PAPCO

3/

7/

Subject:
Date

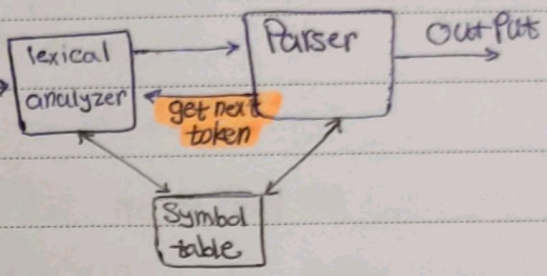
Lexical Analysis
Partition input string into tokens

* Token is A syntatic Category : Identifier, Integer keyword, whitespace

Lexical Analysis in Prespective

key	Symbol Table lexeme	type
1	position	real
2	initial	real
3	rate	real

token
<type, attribute>



Atomic Regular Expressions
Single character 'c' = {"c"} , Epsilon $\varepsilon = \{""\}$

Compound Regular Expressions
Union $A+B = \{s \mid s \in A \text{ or } s \in B\}$

Concatenation $AB = \{ab \mid a \in A \text{ and } b \in B\}$

Iteration $A^* = \bigcup_{i \geq 0} A^i$ where $A^i = A$ — i times — A

Regular Expressions The regular expressions over Σ are the smallest set of expressions including → L( )

ε , 'c' where $c \in \Sigma$, A+B where A,B are rexp over Σ

A^* where A is a rexp over Σ, AB " " " " " "

Syntax vs. Semantic
$L(\varepsilon) = \{""\}$, $L('c') = \{"c"\}$, $L(A+B) = L(A) \cup L(B)$

$L(AB) = \{ab \mid a \in L(A) \text{ and } b \in L(B)\}$, $L(A^*) = \bigcup_{i \geq 0} L(A^i)$

* integer = digit digit* , $A^+ = AA^*$

identifier = letter (letter + digit)*

Option $A + \varepsilon$ $A?$ صفر یا یکی

Subject:
Date

Regular Expressions ⇒ Lexical Spec.

1) Write a rexp for the lexemes of each token

* Number ≈ digit + * keyword ≈ 'if' + 'else' + --- * Identifier ≈ letter(letter+digit)*

2) Construct R, matching all lexemes for all tokens.

R = keyword + Identifier + Number + --- = $R_1 + R_2 + \cdots$

3) Let input be $x_1 \cdots x_n$ For $1 \le i \le n$ check $x_1 \cdots x_i \in L(R)$

4) If success, then we know that $x_1 \cdots x_i \in L(R_J)$ for some J

5) Remove $x_1 \cdots x_i$ from input and go to (3)

What if $x_1 \cdots x_i \in L(R)$ and also $x_1 \cdots x_k \in L(R)$, $i \neq k$

↳ Rule "maximal munch" Pick longest possible string in L(R)

What if $x_1 \cdots x_i \in L(R_J)$ and also $x_1 \cdots x_i \in L(R_k)$, $J \neq k$

↳ Rule use rule listed first (J if J<k)

درحقیقت به keyword نسبت به identifier اولویت بدیم.

What if $x_1 \cdots x_i \notin L(R_J)$ No rule matches a prefix of input?

هیچی دیگه error میدیم

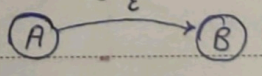
نکته ≈ از accepting state هم میشه خارج شد ولی وقتی رشته تموم میشه که توی accepting state باشیم

حالا مثلا توی زبانی ε هست که start state هم accepting state باشه خیلی خوبه برای پیدا کردن زبون

اول کوتاه ترین شو پیدا کنیم بعد تحلیل کنیم.

* IF (end of input) and (in accepting state) ⇒ accept

Epsilon moves ~~Another kind~~ ε-moves



یعنی دیگه نمیشه روی ε حرکت کنیم

Machine can move from state A to state B without reading input

Deterministic Finite Automata (DFA)	Nondeterministic Finite Automata (NFA)
One transition per input per State	Can have multiple transitions for one input in a given state.
No ε-move	Can have ε-move

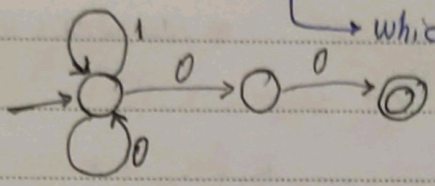
چون عدم قطعیت داریم گنده تره

PAPCO

* هر DFA ای یک NFA هم هست.

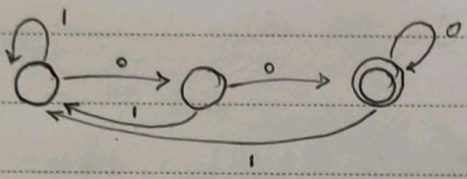
Subject:
Date

* A DFA can take only one Path through the State graph. (Completely determinde by input)
* NFAs can choose → whether to make ε-moves
  ↳ which of multiple transitions for a single input to take

NFA 8 

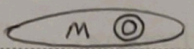
* Acceptance
  ↳ NFA accepts if it can get to a final state

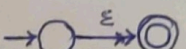
* DFAs are faster to execute (there are no choises to consider)
* NFAs are, in general, Smaller (Sometime exponentially Smaller)

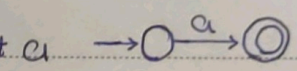
DFA 8 

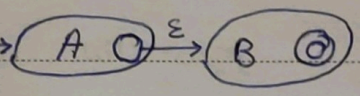
Regular expression to Finite Automata (High-Level Sketch)

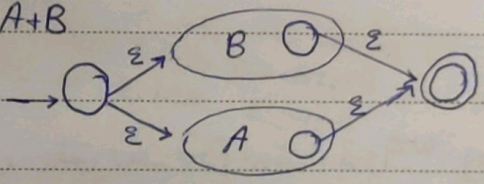
Lexical Specification → Regular expressions → NFA → DFA → Table-Driven Implementation of DFA

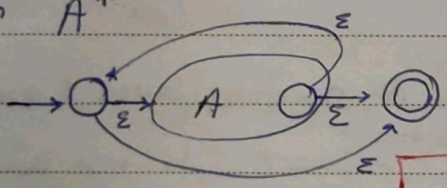
* For each kind of rexp, define an NFA
  ↳ NFA for rexp M → 

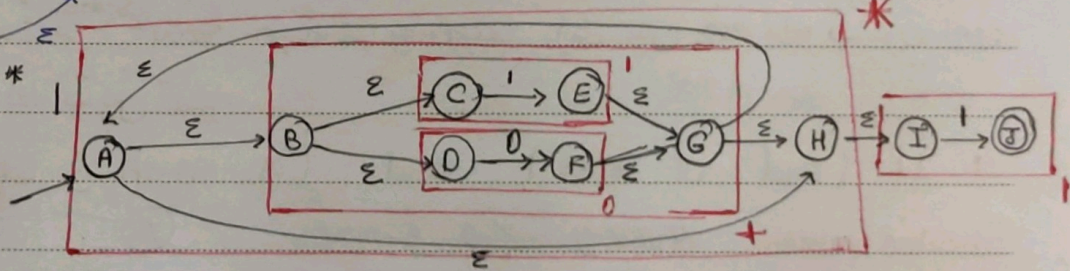
For ε → 

For input a → 

For AB → 

For A+B 

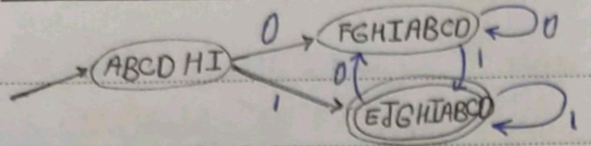
For A* 

Ex 8 $(1+0)^*1$ 

NFA to DFA 8 The Trick

1) Simulate the NFA 2) Each state of DFA = a non-empty subset of states of the NFA
3) Start State = ε-closure of the start state of NFA
4) Add a transition $S \xrightarrow{a} S'$ to DFA iff
   S' is the set of NFA states reachable from any state in S after seeing the input a, considering ε-moves as well
5) Final States 8 Subsets that include at least one final state of NFA.

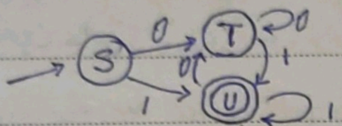
Subject:
Date



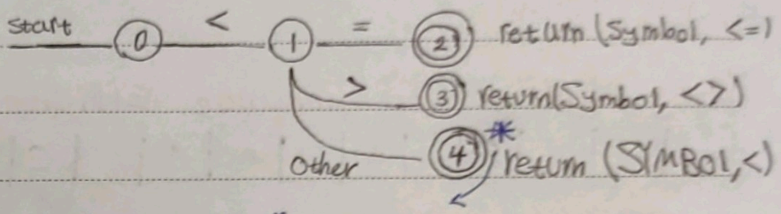
قبلی NFA ، DFA :

* A DFA can be implemented by a 2D table T : states vs. input symbol

$S_i \xrightarrow{a} S_k$ define $T[i, a] = k$

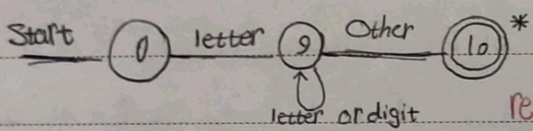


	0	1
S	T	U
T	T	U
U	T	U



we have accepted "<" and have read "other" character that must be unread. That is moving the input pointer one character back.

و برمی‌گرده Start

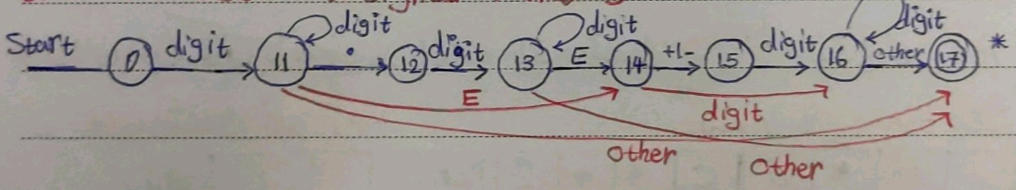


return (get_token (), install_id ())

→ returns either a keyword or ID based on the typo of the token.

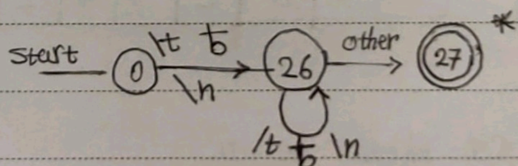
→ If the token is an ID, its lexeme is inserted into the symbol table (only one record for each lexeme) and lexeme of token returned.

DFA of Pascal Unsigned Numbers



return (NUM, lexeme of the number)

DFA of whitespace



تقریبی Lexical %10
Syntax %60
semantic %30

Error Recovery abc $ de $:= 0

(Panic حالت) de $ می‌خونه از چپ به راست می‌خونه تا به $ می‌رسه می‌ریزه دور

درباره de$ رو میاد بخونه به $ می‌رسه Panic می‌کنه همه رو می‌ریزه دور و به هیچی نمی‌رسه!

Cascaded errors

error handling ⟶ error recovery یا error correction

هیچ‌کدام لغتیم | تصحیح خطا

PAPCO

7

Subject:
Date

1) Delete one character from the remaining input
2) Insert a missing character into the remaining input
3) Replace a character by another charachter
4) Transpose two adJacent character

* یکی از این چهار تا کار رو امتحان می‌کنه و کد رو کامپایل می‌کنه
حالا یه ایده هست minimal distance که میاد برای هر ایراد... هر چهار تا رو می‌زنه
آخرین جایی که به جواب درست می‌رسه میاد کوتاه‌ترین رو می‌گیره.

Using Buffer to Enhance Efficiency

چون برنامه بزرگه buffer buffer می‌خونه

			E		=		M	*		C	*	*	2	eof					

lexeme begining (arrow to C)
forward (Scans ahead to find Pattern match) (arrow to eof)

```
if forward at end of first half then begin
    reload second half ;  ← Block I/O
    forward := forward + 1
end
else if forward at end of second half then begin
    reload first half ;  ← Block I/O
    move forward to begining of first half
end
else forward := forward +1
```

اول کار forward و lexeme begining = 0

مثلاً اگه به state * برسه F = F-1

اندازه بافر: read () : 1kb
↳ طول بلاک‌ها

چرا 2 نیمه داریم؟
↳ اگه توکن رو نصفه خونده باشیم چی؟

* طولانی‌ترین توکن با معنی در اندازه بافر موثره!

Buffered I/O with Sentinels: نگهبان

برای بهتر کردن الگوریتم بالا

			E		=		M	*	eof	C	*	*	2	eof					eof

```
forward := forward +1
if forward is at eof then begin
    if forward at end of first half then begin
        reload second half;
        forward := forward +1
    end
    else if forward at end of second half the begin
        reload first half;   move forward to begging of first half
    end
    else /* eof within buffer signifying end of Point */
        terminate lexical analysis
end
```

2nd eof ⇒ no more input!

PAPCO

Subject:
Date

Chomsky Hierarchy

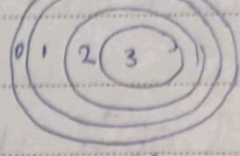
Parser مدیر کامپایلر

0 Unrestricted $\alpha A \beta \rightarrow \alpha \gamma \beta$ ← این یعنی سمت چپ حتما یک غیرپایانه باید باشد

1 Context-Sensitive $|LHS| \leq |RHS|$

2 Context-Free $|LHS| = 1$

3 Regular $|RHS| = 1$ or 2, $A \rightarrow a \mid aB$ (Right linear), or $A \rightarrow a \mid Ba$ (left linear) — یکی پایانه یکی غیرپایانه



$G = <N, T, S, P>$

N → non-terminal, T → terminal, S → start symbol, P → Product-rule

هر زبانی یک گرامر دارد.

a, b, c یک پایانه ، A, B, C یک غیرپایانه ، α, β, γ یک رشته به طول صفر یا بیشتر از پایانه ها و غیرپایانه ها

x, y, z یک رشته به طول یک یا بیشتر از پایانه ، X, Y, Z یک علامت (هرچی) ، ε رشته به طول ۰

مثال $G: < \{E\}, \{+, *, (,), id\}, E, P>$ که همه را از P بدست آوردیم

$P: E \rightarrow E+E$, $E \rightarrow E*E$, $E \rightarrow (E)$, $E \rightarrow id$

قاعده اپسیلون : $A \rightarrow \varepsilon$ تبدیل 2 به 1

* درخت نحو فشرده شده درخت Parse هستش دیگه terminal ها رو نشون نمیدیم

* Non-terminals are written upper case , Terminals lower case

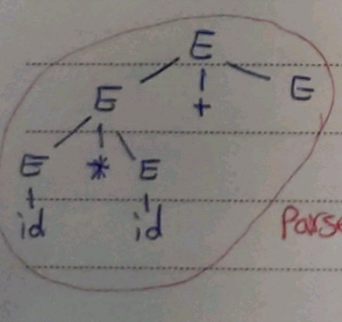
* an in-order travesal of the leaves is the original input

The right-most and left-most derivations have the same Parse tree.

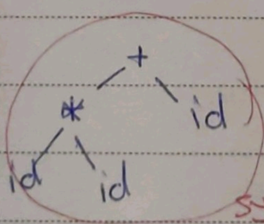
درخت نحو (syntax tree) VS. درخت (Parse)

↳ فقط terminal

id * id + id



Parse



syntax

LMD (Left Most Derivation)

↳ اول چپ ترین غیرپایانه رو با پایانه جایگزین می کنیم.

* درخت Parse اولویت های بالاتر پایین اند

↳ شکل پذیری رو هم نشون میده.

تبدیل مبهم به غیر مبهم : $E \rightarrow E+E \mid E*E \mid (E) \mid id$ تبدیل به ⟹

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

↳ enforce Precedence of * over +

PAPCO