※ The first 3 can be understood by analogy to how humans comprehend a natural language

The structure of a compiler

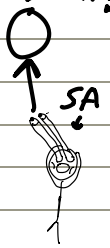
1. Lexical Analysis (Tokenization Analysis) → Same Step done by the interpreter
it divides Program text into "words" or "tokens"
ex. if x == y then z=1; else z=2;
keyword, identifier, equality predicate

2. Parsing (Syntax Analysis)

Once words are understood then the nexrb step is to understand sentence structure

Parsing = Diagramming Sentence
tree

3. Semantic Analysis
ex, scope checking and type checking
Once the structure is understood, we can try understanding the "meaning"
- but meaning is hard for compilers
- compilers use Semantic analysis to catch inconsistencies (because for understanding the full meaning of a sentence i need to have background knowledge
- but compilers has limited analysis power
- For checking the type of error we don't need background knowledge ex. type mismatch



4. Optimization

the butter (conserve resources)
1. Run faster → efficient
2. Use less memory → memory requirement

5. Code Generation (only when the code is valid)
Produces assembly code usually
A translation to another language

Programing language = Design

Compiler = implementation

Language design has big impact on compiler

- Determines what is easy and hard to compile
- many trade-offs in language design called (design and compilation trade-off)

The proportions has changed since (Fortran) → first high level programming language

Before: lexing, Parsing => expensive

now: optimization is King

- application domains have distinctive and conflicting needs
- Science => Parallelism, floating point, arrays and ops
- business => Presistence, Report generation, Data analysis ex→ SQL
- System Languages => Embedded and Operating Systems

For such applications you need => very low level control over resources
Like working with registers

Lexical Analyzer

- Scans the pure HLL code line by line
- takes Lexemes as input and produces Tokens
- Removes comments and whitespaces from pure HLL code
- Helps in micro expansion in pure HLL code

*in the implementation of a lexical analyzer it works on a DFA

Errors in a lexical Analysis
- Identifier that are way too long. C: 247 C++: 2048 Python: 79
- Exceeding length of numerical constants. ex. int i = 4567891; size 2Byte -32,768 → 32,767
- ill-formed numerical constants. ex => int i = 47$8
- illegal characters that are absent from the source code ex char x[] = "Hello" ; $

Issues with lexical analysis
- Lookahead
- Ambiguities

we Specify lexers using => RE or Examples of a RE

1950's
*Fortran
white spaces are insignificant

Output of a lexical analyzer is a stream of tokens ==go to==> Parser (syn anal)

How can a lexical analyzer be implemented:
1. Recognizing substrings corresponding to tokens
2. Return the value of a lexeme of a token (lexeme is the substring)

"Lookahead" might be used to decide where one token ends and the other token begins
So we have to use Lookahead

C++ ex Foo<Bar<Bazz>> (✗ can't) > ws > (✓)

There are several ways to specify tokens ==Pop==> RL (Simple, easy to understand, Efficient)

a Language is a set of strings

Need some notation for specifying which sets we want (RL ==notation==> RE)

Atomic regex => Union: $A+B = \{s \mid s \in A \text{ or } s \in B\}$ concatenation: $AB = \{s \mid s \in A \text{ and } s \in B\}$
Iteration $A^* = \bigcup_{i \geq 0} A^i$ where $A^i = A \ldots i$ times A

Σ={0,1}

1* = ""+1+11+111+...

(1+0)1 = {ab| a ϵ 1+0 ^ b ϵ 1} ={ 11,01}

0*+1* = { 0ⁱ| i>=0} U {1ⁱ| i>=0} = {"",0,00,...,1,11,111,...}

(0+1)* = $U_{i>=0}$ (0+1)ⁱ = ""+ 0+1, (0+1)(0+1), ..., (0+1)...(0+1
= all strings of 0's and 1'a

Def. The RE over S are the smallest set of expressions ex

Syntax vs. Semantics

To be careful, we should distinguish syntax (the reg. exp.) and semantics (the langs. they denote)
Meaning function L maps syntax to semantics
L: Exp → Sets of Strings

$L(\varepsilon) = \{""\}$
$L('c') = \{"c"\}$
$L(A+B) = L(A) \cup L(B)$
$L(AB) = \{ab \mid a \in L(A) \text{ and } b \in L(B)\}$
$L(A^*) = \bigcup_{i \geq 0} L(A^i)$

identifier = letter (letter + digit)* ≠ (letter* + digit*)
a letter followed by a bunch of letters or digits | this means either all letters followed by all digits

in Summary RE describe many useful languages / RL = Language Specification

Lexical analysis C implementation

- Union: $A|B \equiv A+B$
- Option: $A+\varepsilon \equiv A?$
- Range: 'a'+'b'+...+'z' $\equiv [a-z]$
- excluded Range: complement of $[a-z] \equiv [\hat{}a-z]$

RE => lexical Spec Steps

1. write a regxp for the lexemes of each token ex. Number = digit+... Keyword = "if"+"else"+...
2. Construct R, matching all lexemes for all tokens R = Keyword + Identifier + Number = $R_1 + R_2 + R_3$
3. Let input be $x_1 \dots x_n$ for $1 \le i \le x$ check $x_1 \dots x_i \in L(R)$
4. If Success, then we know that $x_1 \dots x_i \in L(R_j)$ for some j
5. Remove $x_1 \dots x_i$ From input and go to (3)

Ambiguities

1. How much input is used? Like $\begin{matrix} x_1 \dots x_i \\ x_1 \dots x_n \end{matrix} \in L(R)$ Rule: Pick the longest possible string
   - it's called the maximal march

2. which Token is used ex. $x_1 \dots x_j \in L(R_j)$ $L(R_k)$ which one do we choose
   Rule: use rule listed first (Priority) (j if j > k)

Error Handling = Lowest priority

Summary

- RE provide a concise notation for string patterns
- Use in lexical analysis requires small extensions → resolve ambiguities; handle errors
- Good algorithm known
  - require only single pass over the input
  - Few operation per character (table lookup)

Finite Automata

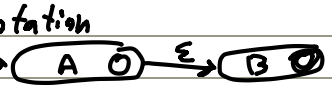
- RE = spec & FA = imple

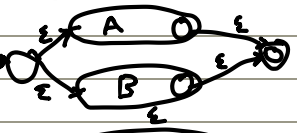
A FA is made of

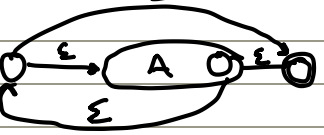
1. input alphabet Σ
2. Set of states S
3. Start state n
4. Set of accepting states $F \subseteq S$
5. Set of transitions state $\xrightarrow{input}$ state

Lexical Specification → RE → NFA → DFA → Table-driven implementation of DFA

NFA notation

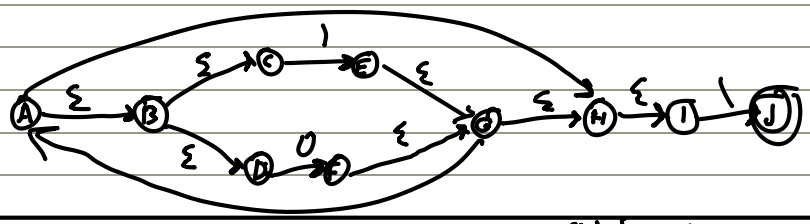
AB = 

A+B = 

A* = 

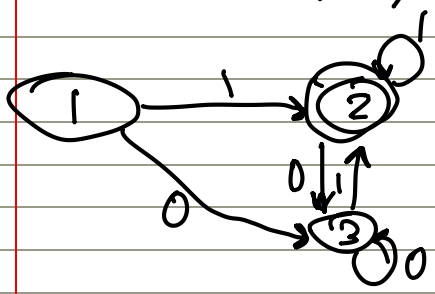
(1+0)*1

(ab*ab*) even
a(ab*ub*)ock



ε-(A)	1	0
ε-enclosure(A) {A,B,C,D,H,I} → 1	E J	F

ε-enclosure(E,J) {E,G,H,I,A,B,C,D} → 2
ε-enclosure(F) {G,H,I,A,B,C,D} → 3



Languages are developed around the von neumann architecture

Influences on Language design(Computer architechture/Programing methodologies

why is the von neumann architecture dominent

1. Data and programs stored in memory
2. memory is seperate from the cpu
3. instruction and data are piped from the cpu
4. basis for imperative languages
- variables model memory cells
- assignments statements model piping
- Iteration is efficient
- selection and branching statments

algorithm

init Prog Count
rep fev
get instructio pointed by the counter
counter ++
decode the instruction
execute the (/ /
end rep

* a connection between a Cpu and RAM = speed of computer

Load module : the user and system Load together
Linking and Loading : the process of collecting system program units and linkw user prog
von bottleneck : instruction excation speed > connection speed
↳ primary limiting Factor in PC speed

Interpreter

Older
used in Scripting Languages
Better portability
skips the compilation part, goes stright to exc
uses more space
runs programs as is (little to no preprocessing)

Compilers

newer
Better execution time
uses less space
compiles the code before execution
does extensive preprocessing

Compiled code

ready to Run
often faster
not cross platform
inflexible
Source code is Private

Interpreted Code

cross-platform
interpreter
simpler to test
often slower
easier to debug
source code is public

Compiled	Interpreted	Hybrids
C	PHP	Python
C++		Java
objective-C	JavaScript	C#
		VB.NET

ah Oh Open it up	Java	C++
Readability	more readable has a simple design and easy to read	bad in readability since it has pointers which are hard to read
Writeability	although Java is great in writeability we gotta give it to C++ since it uses Pointers	C++ is "Him!!" in terms of writeabilite becaus it uses pointers
Reliability	Java is better 1. usas type checking 2. exception handling and indexing 3. check the OoB	bad because as mistakes can happen when we use arithmetic Pointers
Portability	Java is better you can compile it to an intermediate language (byte code)	has to recompile it
Cost of execation	higher cost of execation since its a strong checking language ex. OoB while c++ doesn't chec	C++ translate directly into machine code, while Java → intermediate language → machine cost ↑

NFA

Can have multiple transition for the same input
ε-moves
Slower to execute (more paths to consider)
Simpler and smaller

DFA

One transition per input per state
No ε-moves
Faster to execute (no choices to consider
more complex and can be exponentially larger
$2^n - 1$ states

States : S	States : subset of S
Start state: $s \in S$	Start state: ϵ-closure(s)
Final states: F subset of S	Final state: $\{ X \mid X \cap F \neq \phi \}$
The transition function: $a(x)=\{y \mid x \in X \wedge x \rightarrow^a y\}$	The transition function: $X \rightarrow^a Y$ if $Y = \epsilon\text{-closure}(a(X))$