Bootstrapping ⇒ It is a means of developing a compiler in the target programming language which it is intended to compile.

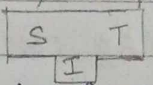
Bootstrapping and porting :-

Third language for compiler Construction

- Machine language
  - compiler to execute immediately.
- Another language with existed compiler on the same target machine : (First scenario)
  - Compile the new compiler with existing compiler.
- Another language with existed compiler on different m/c : (Second scenario)
  - compilation produce a cross compiler.

for bootstrapping purposes, a compiler is characterized by 3 languages.

Thombstone diagram

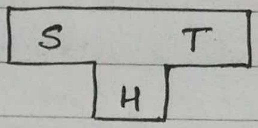


S - Src lang that it compiles
T - target " " " generates code for
I - Impln " " " " written in

represented by a T-diagram

T - Diagram Describing Complex situation.

- A compiler written in language H that translates language S into lang T

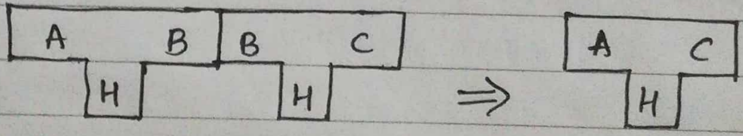


$S \xrightarrow{\text{translated}} T$

H - compiler is written in H.

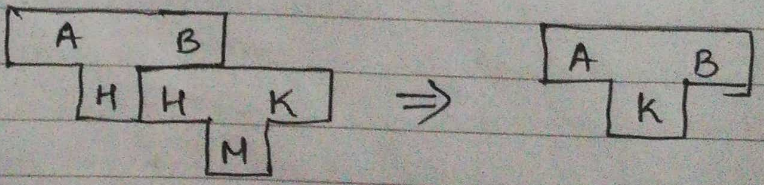
- T-diagram can be combined in two basic ways.

1) T-diagram Combination.



- Two compilers run on the same m/c H
  - first from A to B
  - 2nd from B to C
  - Result from A to C on H.

2) T-diagram Combination

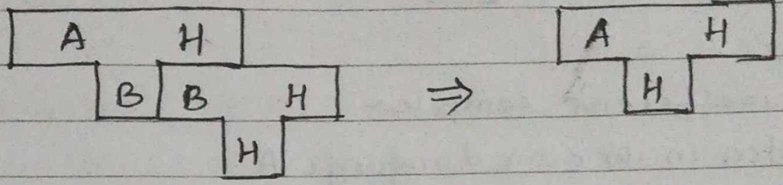


translates H → K

- Translate implementation language of a compiler from H to K

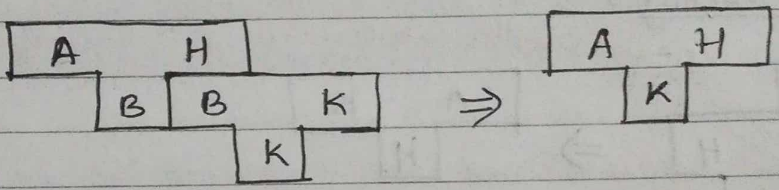
- Use another compiler from H to K.

First scenario :-



- Translate a compiler from A to H written in B
  - use an existing compiler for language B on m/c H.

Second scenario :-

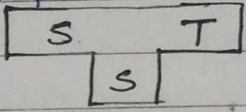


- Use an existing compiler for language B on different m/c K.
  - Result in a cross compiler.

Process of Bootstrapping

→ build up a compiler for larger & larger subsets of a language

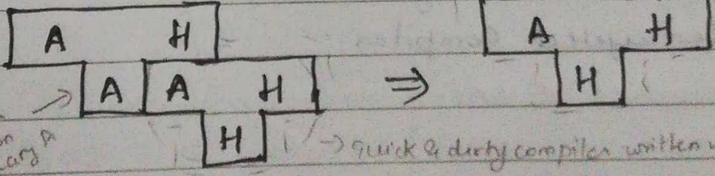
- write a compiler in the same language.



In Unix C compilers are written in C.

- No compiler for source language yet
- Porting to a new host machine.

First step in bootstrap :-



compiler written in its own lang A

→ quick & dirty compiler written in m/c lang

⇒ running but inefficient compiler

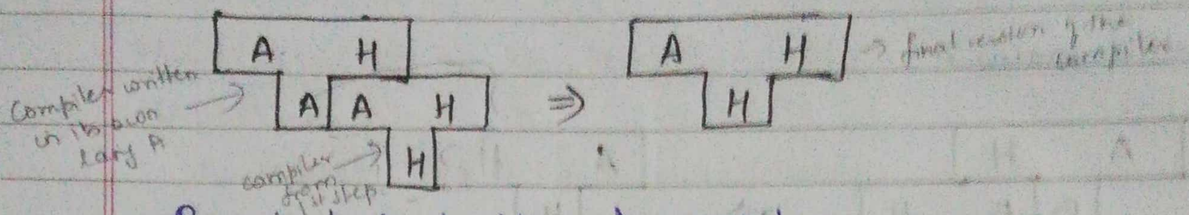
- "quick and dirty" compiler written in m/c language H
- Compiler written in its own language A
- Result in running but inefficient compiler.

Q&D compiler ⇒ may also produce extremely inefficient code

once we have the running Q&D we use it to compile the good compiler.

Then we compile the good compiler to produce final version. This is called bootstrapping.

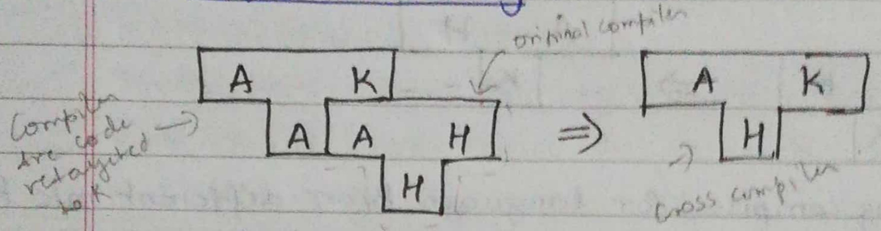
The second step in bootstrap


A
H
compiler written in its own lang A
A
A
H
compiler from 1st step
H
⇒
A
H
H
final version of the compiler

- Running but inefficient compiler
- compiler written in its own language A
- Result in final version of the compiler.

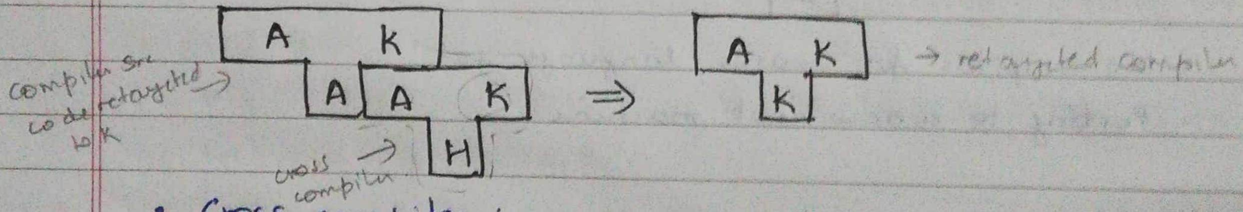
Porting :- → Porting the compiler to a new host computer only requires that the back end of the src be rewritten to generate the code for a new m

Step 1 in Porting :-



- original compiler
- Compiler source code retargeted to K
- Result in Cross compiler.

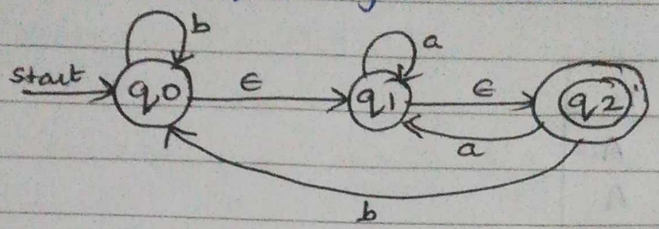
step 2 in porting



- Cross compiler.
- compiler source code retargeted to K
- Result in Retargeted Compiler.

NFA to DFA

Convert the following NFA with ϵ to equivalent DFA.



Sol^n:- To convert this NFA we first find ϵ-closures.

ϵ-closure $\{q0\} = \{q0, q1, q2\}$

ϵ-closure $\{q1\} = \{q1, q2\}$

ϵ-closure $\{q2\} = \{q2\}$.

Let us start from ϵ-closure of start state

ϵ-closure $\{q0\} = \{q0, q1, q2\} \Rightarrow$ A

Now let us find transitions on A with every input symbol.

$$\delta'(A,a) = \epsilon\text{-closure}(\delta(A,a))$$
$$= \epsilon\text{-closure}(\delta(q0,q1,q2),a)$$
$$= \epsilon\text{-closure}\{\delta(q0,a) \cup \delta(q1,a) \cup \delta(q2,a)\}$$
$$= \epsilon\text{-closure}\{q1\}$$
$$= \{q1,q2\} \Rightarrow B$$

$$\delta'(A,b) = \epsilon\text{-closure}(\delta(A,b))$$
$$= \epsilon\text{-closure}(\delta(q0,q1,q2),b)$$
$$= \epsilon\text{-closure}\{\delta(q0,b) \cup \delta(q1,b) \cup \delta(q2,b)\}$$
$$= \epsilon\text{-closure}\{q0\}$$
$$= \{q0,q1,q2\} \Rightarrow A$$

Hence we can state that

$\delta'(A,a) = B$

$\delta'(A,b) = A$.

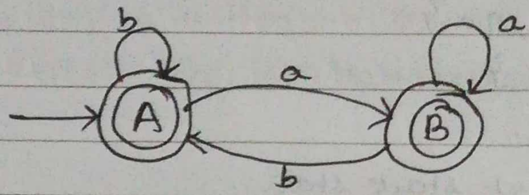
Now let us find transitions for state $B = \{q1, q2\}$.

$$\delta'(B,a) = \epsilon\text{-closure}(\delta(q1,q2),a)$$
$$= \epsilon\text{-closure}\{q1\}$$
$$= \{q1,q2\} \Rightarrow B$$

$$\delta'(B,b) = \epsilon\text{-closure}(\delta(q1,q2),b)$$
$$= \epsilon\text{-closure}\{\delta(q1,b) \cup \delta(q2,b)\}$$
$$= \epsilon\text{-closure}\{q0\}$$
$$= \{q0,q1,q2\} \Rightarrow A.$$

Hence the generated DFA is

	a	b
→Ⓐ	B	A
Ⓑ	B	A



Recognition of Tokens

token = token type + token value

- For a programming languages there are various types of tokens such as identifier, keywords, constants and operator and so on.
- The token is usually represented by a pair token type and token value.

category

token attribute
infm about token

Token type	Token value

fig:- Token representation

- The token type tells us the category of token and token value gives us the infm regarding token.
- The token value is also called token attribute. During lexical analysis process the symbol table is maintained.
- The token value can be a pointer to symbol table in case of identifier and constants.
- The LA reads the input program and generates a symbol table for tokens.

Eg: Consider some encoding of tokens as follows :-

Token	Code	Value
if	1	–
else	2	–
while	3	–
for	4	–
identifier	5	Ptr to S.T
constant	6	" "

<	7	1
<=	7	2
>	7	3
>=	7	4
!=	7	5
(	8	1
)	8	2
+	9	1
-	9	2
=	10	-

Consider a program code as

scanner scans i/p string →

```
if (a<10)
      i = i+2;
else
      i = i-2;
```

token type, token value

(8, 1)
(5,100)
(7,1)
(5,105)

token type

parenthesis

- Our LA will generate following token stream:

token stream
1, (8,1), (5,100), (7,1), (6,105), (8,2), (5,107), 10, (5,107), (9,1), (6,110), 2, (5,107), 10, (5,107), (9,2), (6,110).

- The corresponding symbol table for identifiers and Constants will be,

Location counter	Type	Value
100	identifier	a
⋮	⋮	
105	constant	10
⋮	⋮	
107	identifier	i
⋮	⋮	
110	constant	2

- In above example scanner scans the input string and recognizes "if" as a keyword and returns token type as 1 since in given encoding code 1 indicates keyword "if" and hence 1 is at the beginning of token stream.
- Next is a pair (8,1) where 8 indicates parenthesis and

'c'. Then we scan the input 'a' recognizes it as identifier and searches the symbol table to check whether the same entry is present.

- If not it inserts the inf^{m} about ~~its~~ this identifier in symbol table and returns 100.
- If the same identifier or variable is already present in S.T then LA does not insert it into the table instead it returns the location where it is present .