Type Checking

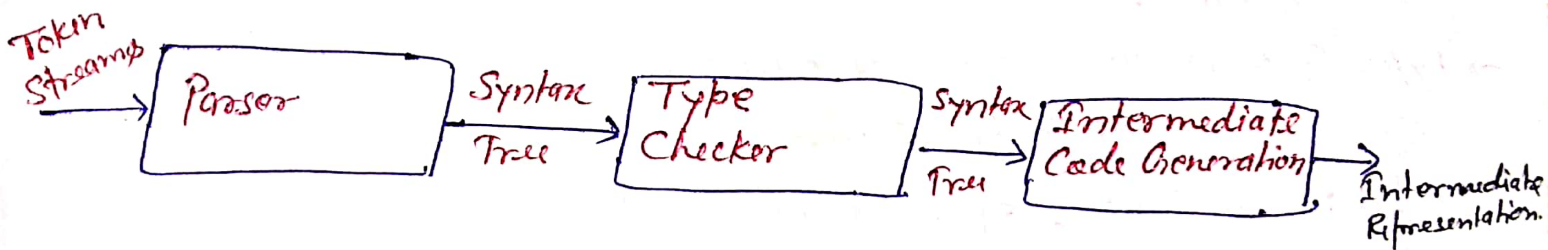
→ Type Checker is a module of compiler, devoted to type checking tasks.

→ Type Checking → Static – is done at Compile time
→ Dynamic – is done at Run time

The design of type checker Depends on

→ Syntatic, structure of language constructs.
→ The type expression of language
→ The rules for assigning types to Construct

Position of Type checker



Scanned with CamScanner

⊛ Type Expression & Type Systems

→ Type Expression will denotes the type of language Construction.

→ Type Expression
- combo → Basic Type {like, int, real, boolean, char, ... etc}
- → Type name (Type Constructor) {Arrays, product, pointer, function ... etc}

⇒ if T = Type. Expression (T.E.)
array (I, T) = T.E

eg:- array [1 .... 10] of int

array[1 ..... 10] of int = array[1...10, int];
(array … → Type Constructor; int → Basic Type)

Type Systems -

→ Type Systems is collection of rules for assigning type expression

Components of Type System
- → Basic Type, eg:- int, char, float, ... etc
- → Type Constructor, eg.. array, structure, String, function .... etc.
- → Type Equivalance, eg.
  - → name Equivalance, eg:- Char a, b; a = 'A'; b = a;
  - → Structural equivalence. eg:. Struct a, b;

Scanned with CamScanner

⊛ Peephole Optimization :-

→ This technique works locally on source code to transform it into an optimized code.

→ The peep hole optimization is a short sequence of target instruction that can be replaced by shorter or faster sequence instruction.

→ It examine at most a few instruction transforming instruction into other less expensive ones such as turning multiplication of x by 2 into an addition of x will itself.

eg $x \times 2 \Longrightarrow x+x$

⇒ Characteristics of Peephole optimization

1. Redundant instruction elimination
2. Unreachable code
3. Flow of control optimization
4. Algebraic Simplifications

⇒ Redundant Instruction Elimination :.

At source code level, the following can be done by user.

①
```
int add_ten(int x)
{
  int y, z;
  y = 10;
  z = x+y;
  return z;
}
```
⇒

②
```
int add_ten(int x)
{
  int y;
  y = 10;
  y = x+y;
  retur y;
}
```
⇒

③
```
int add_ten(int x)
{
  int y = 10;
  return x+y;
}
```

④
```
int add_ten(int x)
{
  return x+10;
}
```

Scanned with CamScanner

⇒ **Unreachable Code:-**

it is a part of program code that is never accessed because of program constructs.

→ Programmers may have accidently written a piece of code that can never be reached.

eg -
```
Void add_ten (int x)
{
    return x+10;
    Printf ("Value of x is %d," x);
}
```

In this stmt. printf stmt will never executed as prog. control return back before it execute, hence stt. can be removed.

⇒ **Flow of Control optimization -**

These are instances in a code where the program control jumps back & forth without performing any significant task, these jumps can be removed.

eg:-
```
MOV R1, R2
Go to L1
...
L1: Go to L2
L2: INC R1
...
```

In this code L1 can be removed as it passes the control to L2, so, instead of jumping to L1 & then to L2, the control directly reach L2.

```
MOV R1, R2
Go to L2
...
L2: INC R1
```

⇒ **Algebraic Simplifications:-**

These are occasions where algebraic Expression can be made simple.

for eg:

$a = a + 0$ // Can be replaced by a itself

$a = a + 1$ // Can simply by replaced by increment (inc) a)

⇓

INC a.