Unit-III

According to Chomsky hierarchy, grammars are divided of 4 types:

Type 0 known as unrestricted grammar.
Type 1 known as context sensitive grammar.
Type 2 known as context free grammar.
Type 3 Regular Grammar.

Type 0: Unrestricted Grammar:
In Type 0
Type-0 grammars include all formal grammars. Type 0 grammar language are recognized by turing machine. These languages are also known as the Recursively Enumerable languages.

Grammar Production in the form of

alpha \to \beta where alpha is (V + T)* V (V + T)*
V : Variables
T : Terminals.
beta is (V + T)*.
In type 0 there must be at least one variable on Left side of production.

For example,
Sab –>ba
A –> S.
Here, Variables are S, A and Terminals a, b.

Type 1: Context Sensitive Grammar)
Type-1 grammars generate the context-sensitive languages. The language generated by the grammar are recognized by the Linear Bound Automata
In Type 1
I. First of all Type 1 grammar should be Type 0.
II. Grammar Production in the form of

alpha \to \beta
alpha| <= |\beta|
i.e count of symbol in \alpha is less than or equal to \beta

For Example,
S –> AB
AB –>abc
B –> b

Type 2: Context Free Grammar:
Type-2 grammars generate the context-free languages. The language generated by the grammar is recognized by a Pushdown automata. Type-2 grammars generate the context-free languages.

In Type 2,
1. First of all it should be Type 1.
2. Left hand side of production can have only one variable.

alpha= 1.

Their is no restriction on \beta.

For example,
S –> AB
A –> a
B –> b

Type 3: Regular Grammar:
Type-3 grammars generate regular languages. These languages are exactly all languages that can be accepted by a finite state automaton.

Type 3 is most restricted form of grammar.
Type 3 should be in the given form only :

V –> VT* / T*.
(or)
V –> T*V /T*

for example :
S –> ab.

TypeChecking:

- A compiler has to do semantic checks in addition to syntactic checks.
- Semantic Checks
- Static –done duringcompilation
- Dynamic –done duringrun-time
- Type checking is one of these static checkingoperations.
- we may not do all type checking at compile-time.
- Some systems also use dynamic type checking too.
- A type system is a collection of rules for assigning type expressions to the parts of a program.
- A type checker implements a type system.
- A sound type system eliminates run-time type checking for type errors.
- A programming language is strongly-typed, if every program its compiler accepts will execute without type errors.
- In practice, some of type checking operations is done at run-time (so, most of the programming languages are not strongly yped).
- –Ex: int x[100]; … x[i] most of the compilers cannot guarantee that i will be between 0and 99

Type Expression:

• The type of a language construct is denoted by a typeexpression.

•A type expression can be:

–A basic type

•a primitive data type such as integer, real, char, Boolean, …

• type-error to signal a typeerror

- void: notype

A type name

- **a name can be used to denote a type expression.**
- **A type constructor applies to other type expressions.**
- arrays: If T is a type expression, then array (I,T)is a type expression where I denotes index range. Ex: array(0..99,int)
- products: If T1and T2 are type expressions, then their Cartesian product T1 x T2 is a type expression. Ex: int xint
- pointers: If T is a type expression, then pointer (T) is a type expression. Ex: pointer(int)
- functions: We may treat functions in a programming language as mapping from a domain type D to a range type R. So, the type of a function can be denoted by the type expression D→R where D are R type expressions. Ex: int→int represents the type of a function which takes an int value as parameter, and its return type is alsoint.

Type Checking of Statements:

S->d=E	**{ if (id.type=E.type thenS.type=void else S.type=type-error**
S ->if Ethen S1	{ if (E.type=boolean thenS.type=S1.type else S.type=type-error}
S->while EdoS1	{ if (E.type=boolean thenS.type=S1.type else S.type=type-error}

Type Checking of Functions:

E->E1(E2)	**{ if (E2.type=s and E1.type=s t)thenE.type=t else E.type=type-error}**

Ex: intf(double x, char y) { ... }

f: double x char->int

argumenttypes return type

Structural Equivalence of Type Expressions:

•How do we know that two type expressions are equal?
•As long as type expressions are built from basic types (no type names), we may use structural equivalence between two type expressions

Structural Equivalence Algorithm (sequin):

if (s and t are same basic types) then return true
else if (s=array(s1,s2) and t=array(t1,t2))

then return (sequiv(s1,t1)

andsequiv(s2,t2))

else if(s= s1 x s2and t = t1 x t2)

then return (sequiv(s1,t1)

and sequiv(s2,t2))

else if (s=pointer(s1) and t=pointer(t1)) then return (sequiv(s1,t1))

else if (s = s1 s2and t = t1 t2) then return (sequiv(s1,t1) and sequiv(s2,t2))

else return false

Names for Type Expressions:

In some programming languages, we give a name to a type expression, and we use that name as a type expressionafterwards.

type link= ↑cell; ? p,q,r,s have same

types ? varp,q :link;

varr,s : ↑cell

•How do we treat type names?
Get equivalent type expression for a type name (then use structural equivalence), or
Treat a type name as a basic type

Overloading of Functions and Operators

AN OVERLOADED OPERATOR may have different meanings depending upon its context.
Normally overloading is resolved by the types of the arguments,
but sometimes this is not possible and an expression can have a set of possible types.
Example 2 In the previous section we were resolving overloading of binary arithmetic operators by looking at the the types of the arguments. Indeed we had two possibles types, say $ \mathbb {Z}$ and $ \mathbb {R}$, with a natural coercion due to the inclusion
$\displaystyle \mbox{${\mathbb Z}$}$ $\displaystyle \subseteq$ $\displaystyle \mbox{${\mathbb R}$}$ (2)
But what could we do if we had the three types $ \mathbb {Z}$, $ \mbox{${\mathbb Z}$}$/p$ \mbox{${\mathbb Z}$}$ and $ \mbox{${\mathbb Z}$}$/m$ \mbox{${\mathbb Z}$}$ for two different integers m and p?
There is no natural coercion between $ \mbox{${\mathbb Z}$}$/p$ \mbox{${\mathbb Z}$}$ and $ \mbox{${\mathbb Z}$}$/m$ \mbox{${\mathbb Z}$}$.
So the type of an expression like 1 + 2 and consequently the signature of + may also depend on what is done with 1 + 2.
SET OF POSSIBLE TYPES FOR A SUBEXPRESSION. The first step in resolving the overloading of operators and functions occuring in an expression E' is to determine the possible types for E'.
For simplicity, we restrict here to unary functions.
We assign to each subexpression E of E' a synthesized attribute E.types which is the set of possible types for E.

These attributes can be computed by the following translation scheme.

E' $\longmapsto$ E	{ E'.types :=	E.types }
E $\longmapsto$ $\bf id$	{ E.types :=	lookup(id.entry) }
E $\longmapsto$ E1[E2]	{ E.types :=	{t | ($\exists$ s $\in$ E2.types) | (s $\rightarrow$ t) $\in$ E1.types} }

NARROWING THE SET OF POSSIBLE TYPES. The second step in resolving the overloading of operators and functions in the expression E' consists of
determining if a unique type can be assigned to each subexpression E of E' and
generating the code for evaluating each subexpression E of E'.
This is done by
assigning an inherited attribute E.unique to each subexpression E of E' such that
either E can be assigned a unique type and E.unique is this type,
or E cannot be assigned a unique type and E.unique is $\bf type\_error$.
assigning a synthesized attribute E.code which is the target code for evaluating E and
executing the following translation scheme (where the attributes E.types have already been computed).