UNIT-III

Semantics: Syntax directed translation, S-attributed and L-attributed grammars, Intermediate code – abstract syntax tree, translation of simple statements and control flow statements.
Context Sensitive features – Chomsky hierarchy of languages and recognizers. Type checking, type conversions, equivalence of type expressions, overloading of functions and operations.

Syntax directed translation

In syntax directed translation, along with the grammar, we associate some informal notations and these notations are called as semantic rules.
Therefore, we can say that

Grammar + semantic rule = SDT (syntax directed translation)

- In syntax directed translation, every non-terminal can get one or more than one attribute or sometimes 0 attribute depending on the type of the attribute. The value of these attributes is evaluated by the semantic rules associated with the production rule.
- In the semantic rule, attribute is VAL and an attribute may hold anything like a string, a number, a memory location and a complex record
- In Syntax directed translation, whenever a construct encounters in the programming language then it is translated according to the semantic rules define in that particular programming language.

Example

Production	Semantic Rules
E → E + T	E.val := E.val + T.val
E → T	E.val := T.val
T → T * F	T.val := T.val + F.val
T → F	T.val := F.val
F → (F)	F.val := F.val
F → num	F.val := num.lexval

E.val is one of the attributes of E.
num.lexval is the attribute returned by the lexical analyzer.

Syntax directed translation scheme

- The Syntax directed translation scheme is a context -free grammar.
- The syntax directed translation scheme is used to evaluate the order of semantic rules.
- In translation scheme, the semantic rules are embedded within the right side of the productions.
- The position at which an action is to be executed is shown by enclosed between braces. It is written within the right side of the production.

Example

Production	Semantic Rules
S → E $	{ printE.VAL }
E → E + E	{E.VAL := E.VAL + E.VAL }
E → E * E	{E.VAL := E.VAL * E.VAL }
E → (E)	{E.VAL := E.VAL }
E → I	{E.VAL := I.VAL }
I → I digit	{I.VAL := 10 * I.VAL + LEXVAL }
I → digit	{ I.VAL:= LEXVAL}

Implementation of Syntax directed translation

Syntax direct translation is implemented by constructing a parse tree and performing the actions in a left to right depth first order.

SDT is implementing by parse the input and produce a parse tree as a result.

Example

Production	**Semantic Rules**
S → E $	{ printE.VAL }
E → E + E	{E.VAL := E.VAL + E.VAL }
E → E * E	{E.VAL := E.VAL * E.VAL }
E → (E)	{E.VAL := E.VAL }
E → I	{E.VAL := I.VAL }
I → I digit	{I.VAL := 10 * I.VAL + LEXVAL }
I → digit	{ I.VAL:= LEXVAL}

Parse tree for SDT:

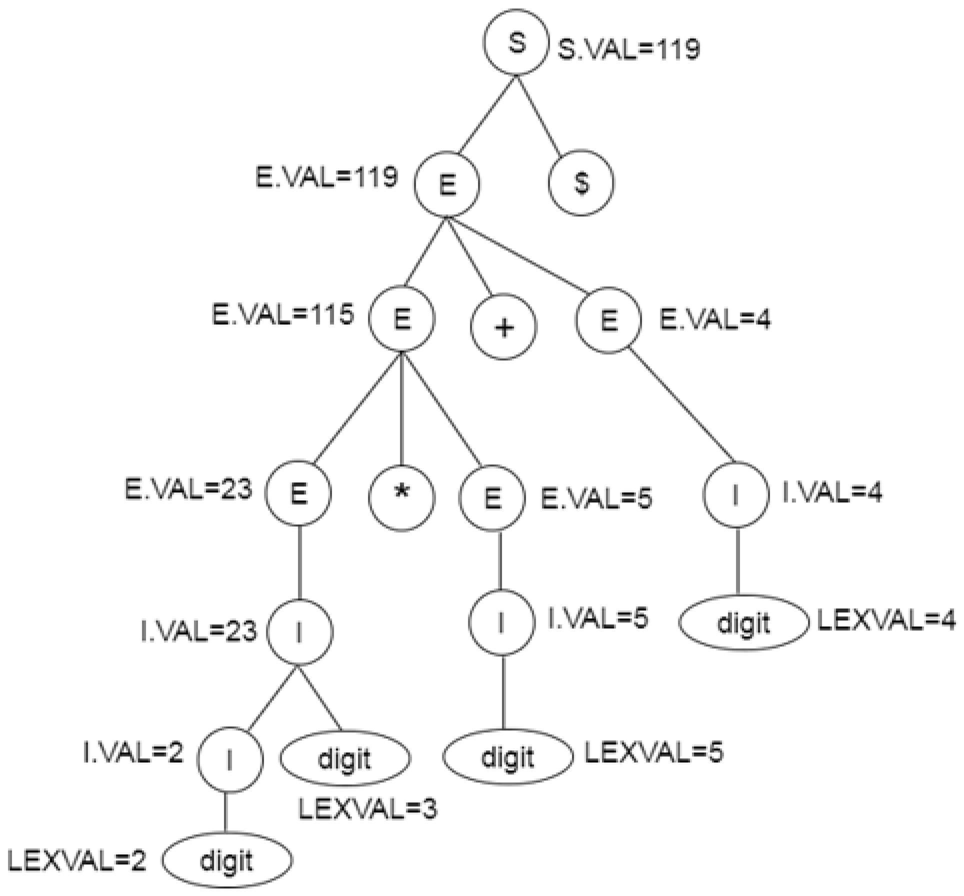


Fig: Parse tree

S-attributed and L-attributed grammars

Before coming up to S-attributed and L-attributed SDTs, here is a brief intro to Synthesized or Inherited attributes

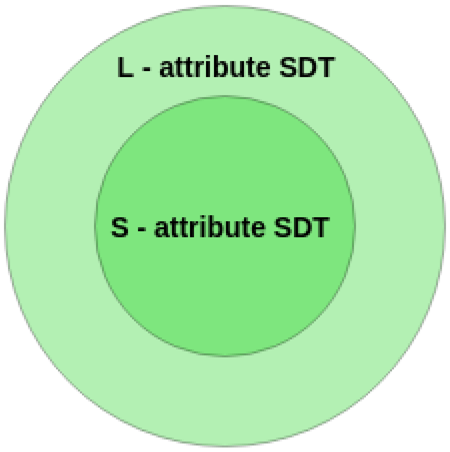
Types of attributes –
Attributes may be of two types – Synthesized or Inherited.
Synthesized attributes –A Synthesized attribute is an attribute of the non-terminal on the left-hand side of a production. Synthesized attributes represent information that is being passed up the parse tree. The attribute can take value only from its children (Variables in the RHS of the production).
For eg. let's say A -> BC is a production of a grammar, and A's attribute is dependent on B's attributes or C's attributes then it will be synthesized attribute.
Inherited attributes - An attribute of a nonterminal on the right-hand side of a production is called an inherited attribute. The attribute can take value either from its parent or from its siblings (variables in the LHS or RHS of the production).
For example, let's say A -> BC is a production of a grammar and B's attribute is dependent on A's attributes or C's attributes then it will be inherited attribute.
Now, let's discuss about S-attributed and L-attributed SDT.
S-attributed SDT:
- If an SDT uses only synthesized attributes, it is called as S-attributed SDT.
- S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.
- Semantic actions are placed in rightmost place of RHS.

L-attributed SDT:
- If an SDT uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from left siblings only, it is called as L-attributed SDT.
- Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.
- Semantic actions are placed anywhere in RHS.

For example,

A -> XYZ {Y.S = A.S, Y.S = X.S, Y.S = Z.S}

is not an L-attributed grammar since Y.S = A.S and Y.S = X.S are allowed but Y.S = Z.S violates the L-attributed SDT definition as attributed is inheriting the value from its right sibling.
Note – If a definition is S-attributed, then it is also L-attributed but **NOT** vice-versa.



Example – Consider the given below SDT.
P1: S -> MN {S.val= M.val + N.val}
P2: M -> PQ {M.val = P.val * Q.val and P.val =Q.val}
Select the correct option.
A. Both P1 and P2 are S attributed.
B. P1 is S attributed and P2 is L-attributed.
C. P1 is L attributed but P2 is not L-attributed.
D. None of the above

S.No.	S-attributed definition	L-attributed definition
1.	It uses synthesized attributes.	It uses synthesized and inherited attributes.
2.	Semantics actions are placed at right end of production.	Semantics actions are placed at anywhere on RHS.
3.	S-attributes can be evaluated during parsing.	L-attributes are evaluated by traversing the parse tree in depth first, left to right.

Intermediate code

A source code can directly be translated into its target machine code, then why at all we need to translate the source code into an intermediate code, which is then translated to its target code? Let us see the reasons why we need an intermediate code.

- If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
- Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.
- The second part of compiler, synthesis, is changed according to the target machine.
- It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

Intermediate Representation

Intermediate codes can be represented in a variety of ways and they have their own benefits.

- **High Level IR** - High-level intermediate code representation is very close to the source language itself. They can be easily generated from the source code and we can easily apply code modifications to enhance performance. But for target machine optimization, it is less preferred.
- **Low Level IR** - This one is close to the target machine, which makes it suitable for register and memory allocation, instruction set selection, etc. It is good for machine-dependent optimizations.

Intermediate code can be either language specific (e.g., Byte Code for Java) or language independent (three-address code).

Three-Address Code

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code.

For example:

```
a = b + c * d;
```

The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

```
r1 = c * d;
r2 = b + r1;
a = r2
```

r being used as registers in the target program.

A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms : quadruples and triples.

Quadruples

Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. The above example is represented below in quadruples format:

Op	arg_1	arg_2	result
*	c	d	r1
+	b	r1	r2
+	r2	r1	r3
=	r3		a

Triples

Each instruction in triples presentation has three fields : op, arg1, and arg2.The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.

Op	arg_1	arg_2
*	c	d
+	b	(0)
+	(1)	(0)
=	(2)	

Triples face the problem of code immovability while optimization, as the results are positional and changing the order or position of an expression may cause problems.

Indirect Triples

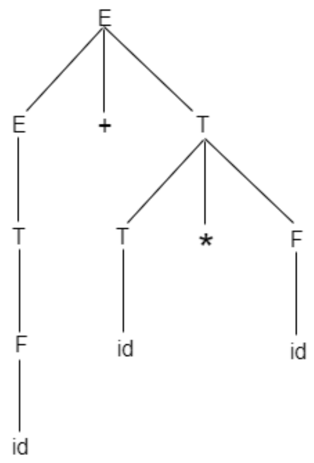
This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

Declarations

A variable or procedure has to be declared before it can be used. Declaration involves allocation of space in memory and entry of type and name in the symbol table. A program may be coded and designed keeping the target machine structure in mind, but it may not always be possible to accurately convert a source code to its target language.

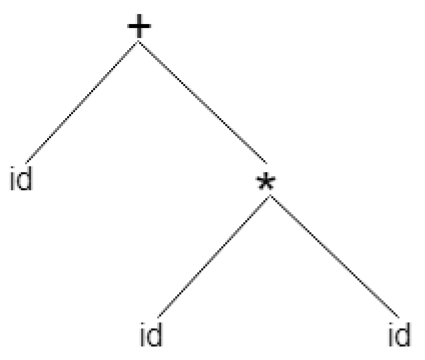
Abstract syntax tree

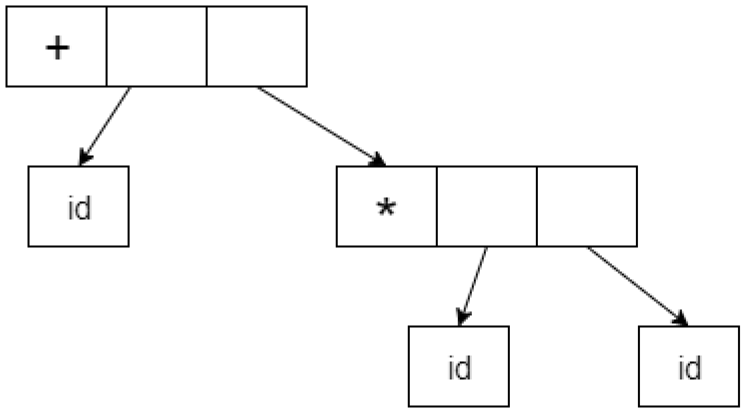
When you create a parse tree then it contains more details than actually needed. So, it is very difficult to compiler to parse the parse tree. Take the following parse tree as an example:



- In the parse tree, most of the leaf nodes are single child to their parent nodes.
- In the syntax tree, we can eliminate this extra information.
- Syntax tree is a variant of parse tree. In the syntax tree, interior nodes are operators and leaves are operands.
- Syntax tree is usually used when represent a program in a tree structure.

A sentence **id + id * id** would have the following syntax tree:





Abstract syntax tree can be represented as:

Abstract syntax trees are important data structures in a compiler. It contains the least unnecessary information.

Abstract syntax trees are more compact than a parse tree and can be easily used by a compiler.

Translation of simple statements and control flow statements

Control statements are the statements that change the flow of execution of statements.

Consider the Grammar

S → if E then S1

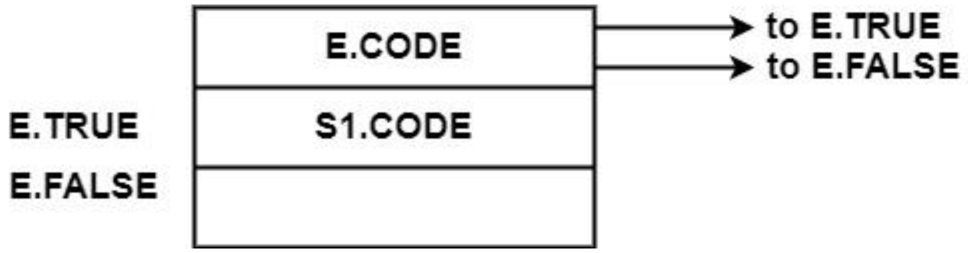
|if E then S1 else S2

|while E do S1

In this grammar, E is the Boolean expression depending upon which S1 or S2 will be executed.

Following representation shows the order of execution of an instruction of if-then, ifthen-else, & while do.

- ***S* → if E then *S1***



E.CODE & S.CODE are a sequence of statements which generate three address code.

E.TRUE is the label to which control flow if E is true.

E.FALSE is the label to which control flow if E is false.

The code for E generates a jump to E.TRUE if E is true and a jump to S.NEXT if E is false.

∴ E.FALSE=S.NEXT in the following table.

In the following table, a new label is allocated to E.TRUE.

When S1.CODE will be executed, and the control will be jumped to statement following S, i.e., to S1.NEXT.

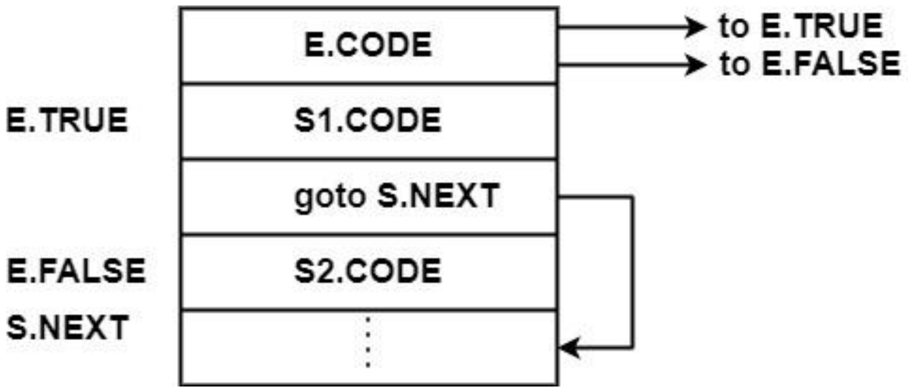
∴ S1. NEXT = S. NEXT.

Syntax Directed Translation for "If E then S1."

Production	Semantic Rule

S → if E then S1	E. TRUE = newlabel;
	E. FALSE = S. NEXT;
	S1. NEXT = S. NEXT;
	S. CODE = E. CODE | |
	GEN (E. TRUE '− ') | | S1. CODE

- **S → If E then S1 else S2**

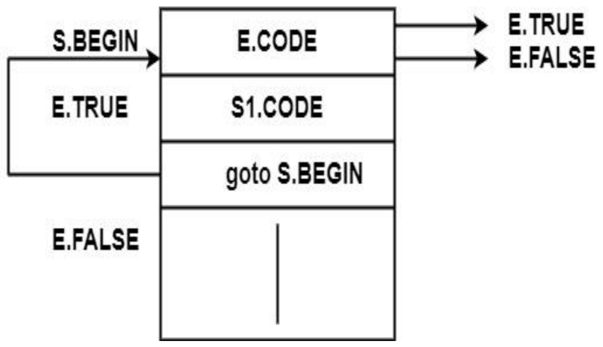


If E is true, control will go to E.TRUE, i.e., S1.CODE will be executed and after that S.NEXT appears after S1.CODE.
If E.CODE will be false, then S2.CODE will be executed.
Initially, both E.TRUE & E.FALSE are taken as new labels. Hen S1.CODE at label E.TRUE is executed, control will jump to S.NEXT.
Therefore, after S1, control will jump to the next statement of complete statement S.
S1.NEXT=S.NEXT
Similarly, after S2.CODE, the next statement of S will be executed.
∴ S2.NEXT=S.NEXT
Syntax Directed Translation for "If E then S1 else S2."

Production	Semantic Rule

S → if E then S1 else S2	E. TRUE = newlabel;
	E. FALSE = newlabel;
	S1. NEXT = S. NEXT;
	S2. NEXT = S. NEXT;
	S. CODE = E. CODE | | GEN (E. TRUE '− ') | | S1. CODE
	GEN(goto S. NEXT) | |
	GEN (E. FALSE −) | | S2. CODE

S → while E do S1
Another important control statement is while E do S1, i.e., statement S1 will be executed till Expression E is true. Control will arrive out of the loop as the expression E will become false.
A Label S. BEGIN is created which points to the first instruction for E. Label E. TRUE is attached with the first instruction for S1. If E is true, control will jump to the label E. TRUE & S1. CODE will be executed. If E is false, control will jump to E. FALSE. After S1. CODE, again control will jump to S. BEGIN, which will again check E. CODE for true or false.
∴ S1. NEXT = S. BEGIN



If E. CODE is false, control will jump to E. FALSE, which causes the next statement after S to be executed.

∴ E. FALSE = S. NEXT

Syntax Directed Translation for " → **while E do S1** "

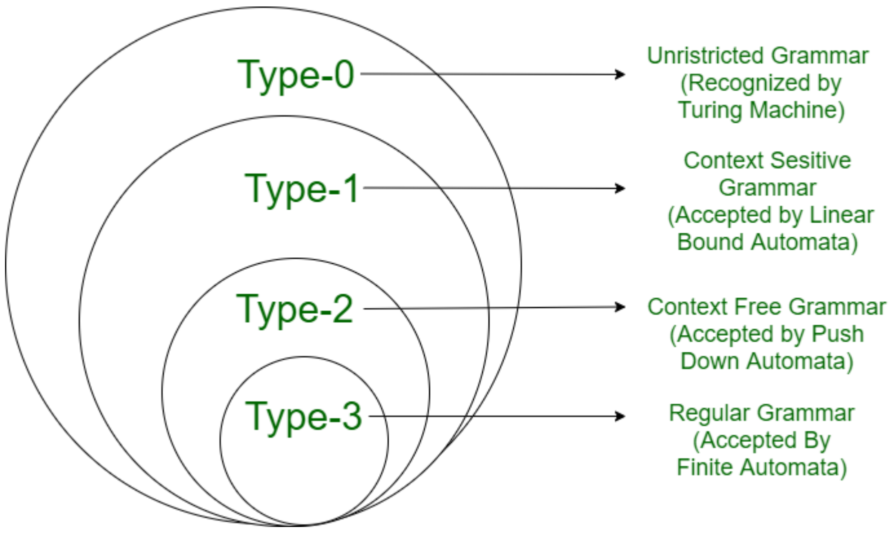
Production	**Semantic Rule**

S → while E do S1	S. BEGIN = newlabel;
	E. TRUE = newlabel;
	E. FALSE = S. NEXT;
	S1. NEXT = S. BEGIN;
	S. CODE = GEN(S. BEGIN '− ') | |
	E. CODE | | GEN(E. TRUE '− ')| |
	S1. CODE | | GEN('goto' S. BEGIN)

Chomsky hierarchy of languages and recognizers

According to Chomsky hierarchy, grammar is divided into 4 types as follows:

1. Type 0 is known as unrestricted grammar.
2. Type 1 is known as context-sensitive grammar.
3. Type 2 is known as a context-free grammar.
4. Type 3 Regular Grammar.


Type-0
Type-1
Type-2
Type-3
Unristricted Grammar (Recognized by Turing Machine)
Context Sesitive Grammar (Accepted by Linear Bound Automata)
Context Free Grammar (Accepted by Push Down Automata)
Regular Grammar (Accepted By Finite Automata)

Type 0: Unrestricted Grammar:

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

Grammar Production in the form of <strong> α ->β </strong> where

\alpha is (V + T)* V (V + T)*
V : Variables
T : Terminals.
β is (V + T)*.

In type 0 there must be at least one variable on the 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 context-sensitive languages. The language generated by the grammar is recognized by the Linear Bound Automata

In Type 1

- First of all Type 1 grammar should be Type 0.
- Grammar Production in the form of

α->β

|\alpha |<=|\beta |

That is the count of symbol in α is less than or equal to β

Also β ∈ $(V + T)^+$

i.e. β can not be ε

For Example:

S --> AB
AB --> abc
B --> b

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

- First of all, it should be Type 1.
- The left-hand side of production can have only one variable and there is no restriction on β |\alpha | = 1.

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 the most restricted form of grammar.

Type 3 should be in the given form only :

V --> VT / T (left-regular grammar)

(or)

V --> TV /T (right-regular grammar)

For example:

S --> a

The above form is called strictly regular grammar.
There is another form of regular grammar called extended regular grammar. In this form:

V --> VT* / T*. (extended left-regular grammar)
(or)
V --> T*V /T* (extended right-regular grammar)

For example :

S --> ab.

Type checking

Type checking is the process of verifying and enforcing constraints of types in values. A compiler must check that the source program should follow the syntactic and semantic conventions of the source language and it should also check the type rules of the language. It allows the programmer to limit what types may be used in certain circumstances and assigns types to values. The type-checker determines whether these values are used appropriately or not.

It checks the type of objects and reports a type error in the case of a violation, and incorrect types are corrected. Whatever the compiler we use, while it is compiling the program, it has to follow the type rules of the language. Every language has its own set of type rules for the language. We know that the information about data types is maintained and computed by the compiler.

The information about data types like INTEGER, FLOAT, CHARACTER, and all the other data types is maintained and computed by the compiler. The compiler contains modules, where the type checker is a module of a compiler and its task is type checking.

Conversion - Conversion from one type to another type is known as **implicit** if it is to be done automatically by the compiler. Implicit type conversions are also called **Coercion** and coercion is limited in many languages.

Example: An integer may be converted to a real but real is not converted to an integer.
Conversion is said to be **Explicit** if the programmer writes something to do the Conversion.

Tasks:

1. has to allow "Indexing is only on an array"
2. has to check the range of data types used
3. INTEGER (int) has a range of -32,768 to +32767
4. FLOAT has a range of 1.2E-38 to 3.4E+38.

Types of Type Checking:

There are two kinds of type checking:

1. Static Type Checking.
2. Dynamic Type Checking.

Static Type Checking: Static type checking is defined as type checking performed at compile time. It checks the type variables at compile time, which means the type of the variable is known at the compile time. It generally examines the program text during the translation of the program. Using the type rules of a system, a compiler can infer from the source text that a function (fun) will be applied to an operand (a) of the right type each time the expression fun(a) is evaluated.

The Benefits of Static Type Checking:

- Runtime Error Protection.

- It catches syntactic errors like spurious words or extra punctuation.
- It catches wrong names like Math and Predefined Naming.
- Detects incorrect argument types.
- It catches the wrong number of arguments.
- It catches wrong return types, like return “70”, from a function that’s declared to return an int.

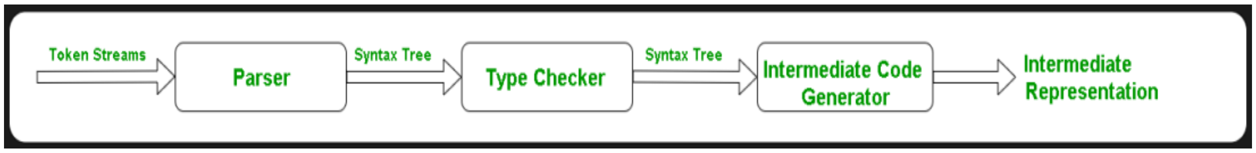
Dynamic Type Checking: Dynamic Type Checking is defined as the type checking being done at run time. In Dynamic Type Checking, types are associated with values, not variables. Implementations of dynamically type-checked languages runtime objects are generally associated with each other through a type tag, which is a reference to a type containing its type information. Dynamic typing is more flexible. A static type system always restricts what can be conveniently expressed. Dynamic typing results in more compact programs since it is more flexible and does not require types to be spelled out. Programming with a static type system often requires more design and implementation effort.

Languages like Pascal and C have static type checking. Type checking is used to check the correctness of the program before its execution. The main purpose of type-checking is to check the correctness and data type assignments and type-casting of the data types , whether it is syntactically correct or not before its execution.
Static Type-Checking is also used to determine the amount of memory needed to store the variable.

The design of the type-checker depends on:

- Syntactic Structure of language constructs.
- The Expressions of languages.
- The rules for assigning types to constructs (semantic rules).

The Position of the Type checker in the Compiler:



Type checking in Compiler

The token streams from the lexical analyzer are passed to the PARSER. The PARSER will generate a syntax tree. When a program (source code) is converted into a syntax tree, the type-checker plays a Crucial Role. So, by seeing the syntax tree, you can tell whether each data type is handling the correct variable or not. The Type-Checker will check and if any modifications are present, then it will modify. It produces a syntax tree, and after that, INTERMEDIATE CODE Generation is done.

Overloading:

An Overloading symbol is one that has different operations depending on its context.

Overloading is of two types:

1. Operator Overloading
2. Function Overloading

Operator Overloading: In Mathematics, the arithmetic expression "x+y" has the addition operator '+' is overloaded because '+' in "x+y" have different operators when 'x' and 'y' are integers, complex numbers, reals, and Matrices.

Example: In Ada, the parentheses '()' are overloaded, the ith element of the expression A(i) of an Array A has a different meaning such as a 'call to function 'A' with argument 'i' or an explicit conversion of expression i to type 'A'. In most languages the arithmetic operators are overloaded.

Function Overloading: The Type Checker resolves the Function Overloading based on types of arguments and Numbers.

Example:

```
E-->E1(E2)
            {
              E.type:= if E2.type = s
              E1.type = s -->t then t
                          else type_error
            }
```

Equivalence of type expressions

Type Checking Rules usually have the form

```
        if two type expressions are equivalent
        then return a given type
        else return type_error
```

Key Ideas The central issue is then that we have to define when two given type expressions are equivalent.

- The main difficulty arises from the fact that most modern languages allow the naming of user-defined types.
- For instance, in C and C++ this is achieved by the typedef statement.
- When checking equivalence of named types, we have two possibilities.

Name equivalence.

Treat named types as basic types. Therefore two type expressions are name equivalent if and only if they are identical, that is if they can be represented by the same syntax tree, with the same labels.

Structural equivalence.

Replace the named types by their definitions and recursively check the substituted trees.

Structural Equivalence If type expressions are built from basic types and constructors (without type names, that is in our example, when using products instead of records), structural equivalence of types can be decided easily.

- For instance, to check whether the constructed types array(n1,T1) and array(n2,T2) are equivalent
  - we can check that the integer values n1 and n2 are equal and recursively check that T1 and T2 are equivalent,
  - or we can be less restrictive and check only that T1 and T2 are equivalent.
- Compilers use representations for type expressions (trees or dags) that allow type equivalence to be tested quickly.

Recursive types In PASCAL a linked list is usually defined as follows.

```
          type link = ^ cell;
```

```
cell = record
      info: type;
      next: link;
    end;
```

The corresponding type graph has a cycle. So to decide structural equivalence of two types represented by graphs PASCAL compilers put a mark on each visited node (in order not to visit a node twice). In C, a linked list is usually defined as follows.

```
struct cell {
   int info;
   struct cell *next;
};
```

To avoid cyclic graphs, C compilers

- require type names to be declared before they are used, except for pointers to records.
- use structural equivalence except for records for which they use name equivalence.

Overloading of functions and operations

- Overloaded Symbol: one that has different meanings depending on its context
- Example: Addition operator +
- Resolving (operator identification): overloading is resolved when a unique meaning is determined.
- Context: it is not always possible to resolvev overloading by looking only the arguments of a function
    - Set of possible types
    - Context (inherited attribute) necessary
- function "*" (i, j: integer) return complex;
- function "*" (x, y: complex) return complex;
    - Has the following types:
        - fcn(tuple(integer, integer), integer)
        - fcn(tuple(integer, integer), complex)
        - fcn(tuple(complex, complex), complex)
            - int i, j;
            - k = i * j;

Narrowing Types

$E' \rightarrow E$	{E'.types = E. types E.unique = if E'.types = {t} then t else error}
$E \rightarrow id$	{E.types = lookup(id.entry)}
$E \rightarrow E_1(E_2)$	{E.types = {s' | ∃ s ∈ E2 .types and s s' → ∈ E1 .types}
	t = E.unique
	S = {s | s ∈ E2 .types and S t → ∈E1 .types}
	E2 .unique = if S ={s} then S else error
	E1 .unique = if S = {s} then S t else error

Polymorphic Type checking Translation Scheme

E → E1 (E2)	{ p := mkleaf(newtypevar);
	unify (E1 .type, mknode(' ', E → 2 .type, p);
	E.type = p}
E → E1 , E2	{E.type := mknode('x', E1 .type, E2 .type); }
E → id	{ E.type := fresh (id.type) }