## *Lexical Error Recovery:*

- Deleting an extraneous character
- Inserting a missing character
- Replacing an incorrect character by a correct character
- Transposing two adjacent characters(such as , fi→if)

**Why are Grammars to formally describe Languages Important ?**

Why are grammars important ?

1.Precise, easy-to-understand representations

2.Compiler-writing tools can take grammar and generate a compiler

3.Allow language to be evolved (new statements, changes to statements, etc.) Languages are not static, but are constantly upgraded to add new features or fix "old" ones

**Why study lexical and syntax analyzers?**

✓ Lexical and syntax analysis not just used in compiler design:
  - program listing formatters
  - programs that compute complexity
  - programs that analyze and react to configuration files

**Syntax Error Handling:**

1) **Lexical errors:**
   ✓ Include misspellings of identifiers, keywords, or operators
   ✓ Example: the use of an identifier elipseSize instead of ellipseSize – and missing quotes around text intended as a string.
2) **Syntactic errors:**
   ✓ Omission, wrong order of tokens
   ✓ include misplaced semicolons or extra or missing braces; that is, "{" or "}."
3) **Semantic errors:**
   ✓ Incompatible types
   ✓ include type mismatches between operators and operands.
   ✓ An example is a return statement in a Java method with result type void.
4) **Logical errors:**
   ✓ anything from incorrect reasoning on the part of the programming.
   ✓ Infinite loop / recursive call

## Error handler goals

- ✓ Report the presence of errors clearly and accurately
- ✓ Recover from each error quickly enough to detect subsequent errors
- ✓ Add minimal overhead to the processing of correct programs

## Error-recover strategies:

1) **Panic mode recovery**
   - ✓ Discard input symbol one at a time until one of designated set of synchronization tokens is found
   - ✓ The synchronizing tokens are usually delimiters, such as semicolon or }
   - ✓ **Advantages:**
     simple ⇒ suited to 1 error per statement
   - ✓ **Problems:**
     skip input ⇒ miss declaration – causing more errors
     ⇒ miss errors in skipped material

2) **Phrase level recovery**
   - ✓ Replacing a prefix of remaining input by some string that allows the parser to continue
   - ✓ Local correction on input is to replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon.
   - ✓ Not suited to all situations
   - ✓ Used in conjunction with panic mode to allow less input to be skipped

3) **Error productions**
   - ✓ Augment the grammar with productions that generate the erroneous constructs
   - ✓ Example: add a rule for
     := in C assignment statements
     Report error but continue compile
   - ✓ Self correction + diagnostic messages

4) **Global correction**
   - ✓ Choosing minimal sequence of changes to obtain a globally least-cost correction
   - ✓ Adding / deleting / replacing symbols
   - ✓ Costly - key issues

"**Why use regular expressions to define the lexical syntax of a language?**"

**Answer:** There are several reasons:

1 . Separating the syntactic structure of a language into lexical and nonlexical parts provides a convenient way of modularizing the front end of a compiler into two manageable-sized components.

2. The lexical rules of a language are frequently quite simple, and to describe them we do not need a notation as powerful as grammars.

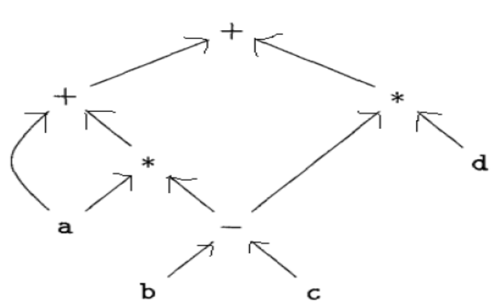
3. Regular expressions generally provide a more concise and easier-to-understand notation for tokens than grammars.

4. More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.

- A *Directed Acyclic Graph* (**DAG**) for an expression identifies the common sub-expressions (sub-expressions that occur more than once) of the expression.
- **DAG** is a tool that
  - ✓ depicts the structure of basic blocks,
  - ✓ helps to see the flow of values flowing among the basic blocks,
  - ✓ ***offers optimization***.

**Example-2:** Three-address code is a linearized representation of a syntax tree or a DAG.

a + a * (b - c) + (b - c) * d




DAG

$$
\begin{aligned}
t_1 &= b - c \\
t_2 &= a * t_1 \\
t_3 &= a + t_2 \\
t_4 &= t_1 * d \\
t_5 &= t_3 + t_4
\end{aligned}
$$

Three Address Code

# Quadruples…

**Advantage –**

- Easy to rearrange code for global optimization.
- One can quickly access value of temporary variables using symbol table.

**Disadvantage –**

- Contain lot of temporaries.
- Temporary variable creation increases time and space complexity.

# Triples…

**Disadvantage –**

- Temporaries are implicit and difficult to rearrange code.
- It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

# Indirect Triples…

## Advantages

- With indirect triples, an optimizing compiler can move an instruction by reordering the instruction list, without affecting the triples themselves.

## Issues in the design of a code generator:

1) Input to the Code Generator.
2) Target Programs
3) Memory Management
4) Instruction Selection
5) Register Allocation
6) Choice of Evaluation Order
7) Approaches of Code Generation

### Basic Blocks:

A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without half or possibility of branching except at the end.

### Flow Graphs:

- ✓ The flow-of-control information to the set of basic blocks making up a program by constructing a directed graph called a flow graph.
- ✓ The nodes of the flow graph are the basic block.
- ✓ One node is distinguished as initial; it is the block whose leader is the first statement.

## Static Allocation:

- Storage is allocated at compile time
- Static storage has fixed allocation that does not change during program execution
- As bindings do not change at runtime, no runtime support is required
- At compile time, compiler can fill the address at which the target code can find the data it operates on
- FORTRAN uses the static allocation strategy

### Limitations

- Size of data objects should be known at compile time
- Recursion is not supported
- Data structures cannot be created at runtime

## Stack Allocation

- Stack allocation manages the runtime storage as a stack, i.e., control stack
- Activation records are pushed and popped as activation begins and end respectively
- Locals are always bound to fresh storage in each activation, because a new activation is onto a stack when a call is made
- Values of locals are deleted as activation ends
- The data structure can be created dynamically for stack allocation

### Limitations

- Values of locals cannot be retained once activation ends
- The memory addressing can be done using pointers and indexed registers
- This type of allocation is slower than static allocation

## Heap allocation

- Storage can be allocated and deallocated in any order
- If the values of non-local variables must be retained even after the activation record then such a retaining is not possible by stack allocation
- It is used for retaining of local variables
- The heap allocation allocates the continuous block of memory when required for storage of activation records. This allocated memory can be deallocated when activation ends
- Free space can be further reused by heap manager
- It supports for recursion and data structures can be created at runtime

### Limitation

- Heap manages overhead.

| S.No. | Static Allocation | Stack Allocation |
|---|---|---|
| 1. | Static Allocation does not makes data structures and objects dynamically. | Stack allocation makes data structures and objects dynamically. |
| 2. | In static allocation, allocation of all data objects is performed at compile time. | While in stack allocation, allocation of data objects is performed at run time. |
| 3. | It does not support recursive procedures. | It supports recursive procedures. |
| 4. | Static allocation is not able to manage the allocation of memory at run time. | Stack allocation use stack to manage the allocation of memory at run time. |
| 5. | In static allocation, at compile time the data object names are fixed. | In stack allocation, index and registers performs the memory addressing. |
| 6. | This strategy is easy and simple in implementing. | This strategy is slower than static allocation. |

| S.No. | Static Allocation | Heap Allocation |
|---|---|---|
| 1. | Static allocation allocates memory on the basis of size of data objects. | Heap allocation make use of heap for managing the allocation of memory at run time. |
| 2. | In static allocation, there is no possibility of creation of dynamic data structures and objects. | In heap allocation, dynamic data structures and objects are created. |
| 3. | In static allocation, the names of the data objects are fixed with storage for addressing. | Heap allocation allocates contiguous block of memory to data objects. |
| 4. | Static allocation is simple, but not efficient memory management technique. | Heap allocation does memory management in efficient way. |
| 5. | Static allocation strategy is faster in accessing data as compared to heap allocation. | While heap allocation is slow in accessing as there is chance of creation of holes in reusing the free space. |
| 6. | Static allocation is inexpensive, it is easy to implement. | While heap allocation is comparatively expensive. |