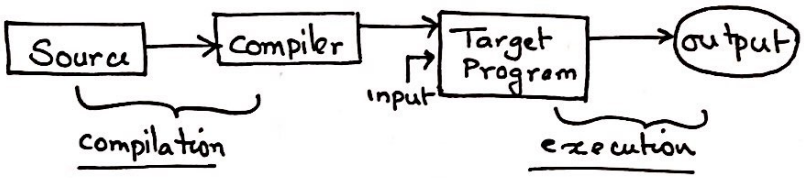
Compiler Design

→ For a program to be run, it must first be translated into a form in which it can be executed by the machine. This translation is done by a system software called the compiler.

→ The compiler reads the program in one language and translates it to another. From source to target language.

→ The compiler also reports errors it encounters during the translation

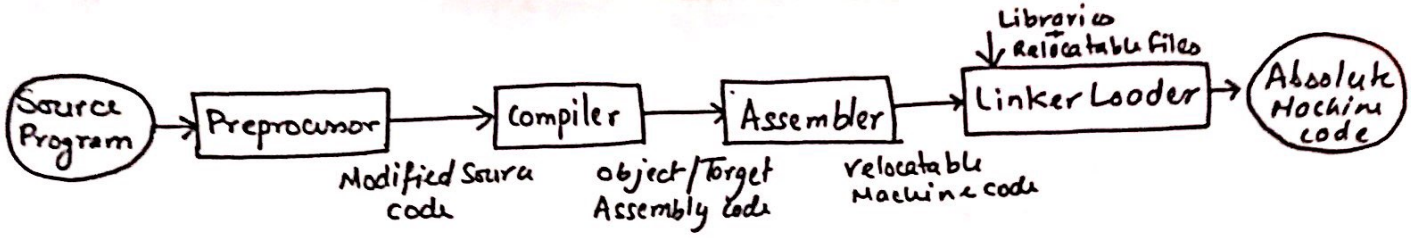


→ Apart from compilers, there are interpreters which instead of translating to a target program directly execute the code in the source itself.

→ Languages like Java use a hybrid system which first translate to an intermidiate language and then this is executed by an interpreter.

— Language Translation System

1. The source is sometimes first collected by the preprocessor. This resolves any preprocessor directives and Macros. Both the input and ouput are high level languages.
2. The compiler produces assembly code. This is easy to produce and debug. The output is called the object / target assembly code.
3. The assembler produces the relocatable machine code. This is in binary
4. The linker handles linking of different components and libraries as well as external
5. The loader loads all the files into memory for execution



Structure of Compiler

The Compiler has two Functional Parts.

— Analyzer

> This breaks up the input source into its constituent sequences Pieces and imposes a grammatical structure on them. This is used to create an intermediate representation of the source.

> The analyzer also alerts the user with errors (semantic/sytactic) through informative error messages.

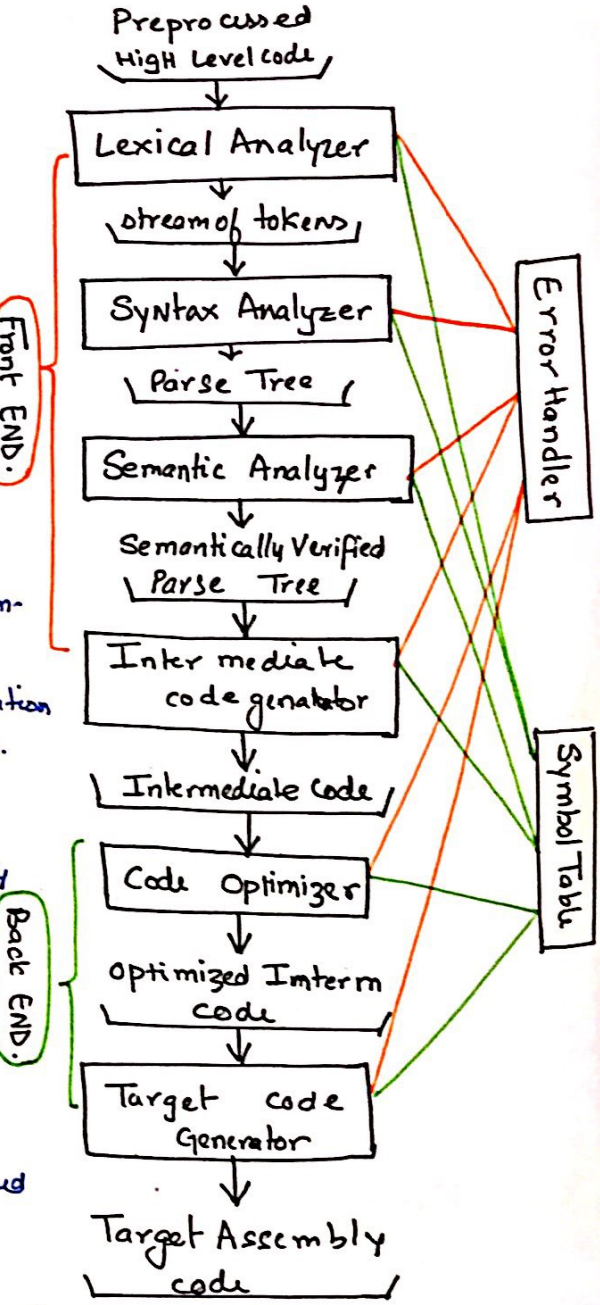
> This also collects and stores information about the soura in a symbol table.

— Synthesizer

> This constructs the desired target Assembly code from the intermediate code.

> This may also include some form of machine dependant/independant optimizers. These are for improving code quality.

> This uses the shared Error Handler and symbol tables.



The Error Handler and symbol table is share by all phases.

Phases of Compiler

— Lexical Analyzer

> Reads the stream of characters making up the soura program and groups them into meaningful sequences called Lexemes.

> The output of the L.A is a sequesce of tokens. <token-name, attr>

— Syntax Analyzer

> This is considered to be the main part of the compiler. It builds the parse tree using a set of grammatical rules particular to that language.

> The output of the Syn.A is the parse tree.

— Semantic Analyzer

> This uses the parse tree and the symbol table to check for semantic consistency with language definition.

> The type information is either stored in the tree or the table.

> It also performs type checking to check for matching operands. and if needed do type conversion i.e coercion.

> The output is the semantically verified parse tree.

— Intermediate Code Generation

> This is an explicit low-level machine like code generator which uses the output of the Sem.A and Symbol table.

> This code must be easy to produce & translate to machine.

> They can be of the following type : 1) 3 address code, 2 addrs

— Code Optimization

> This is done to improve the running time of the program without slowing down the compilation much.

> This removes redundant / duplicates instructions, type conversion etc.

> Compiler that do this are called optimizing compilers.

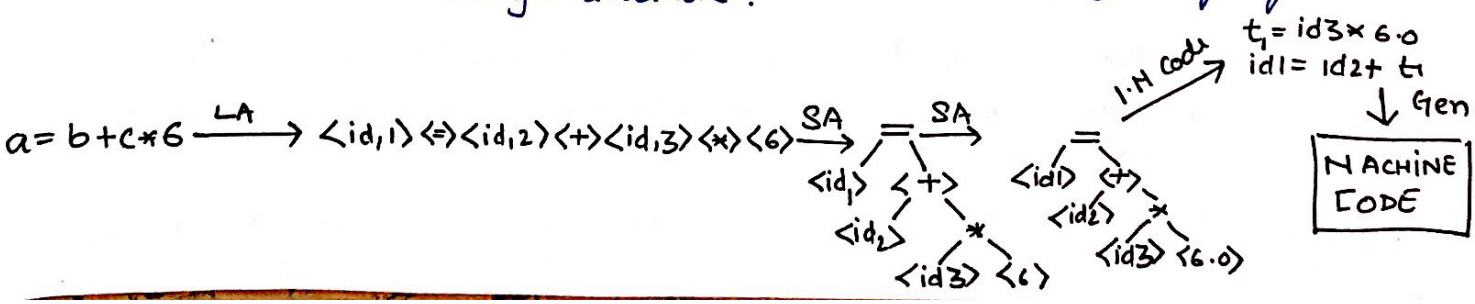
— Code Generation

> Maps the intermediate [optimized] code to machine language.

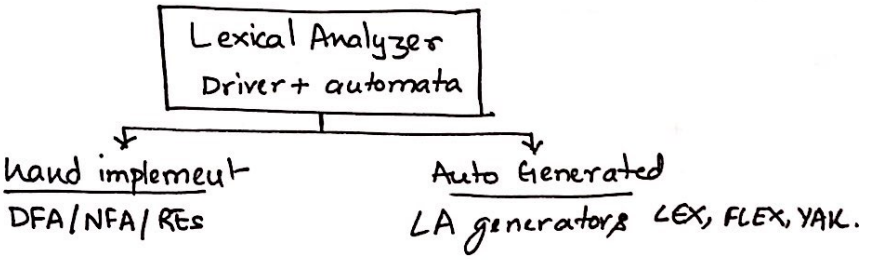
> The mapping can be to other non machine codes too.

> In case of machine code, memory locations, registers etc are selected and defined.

> The most essential part is to judiciously assign registers and storage allocation.



Lexical Analyzer (Scanner)



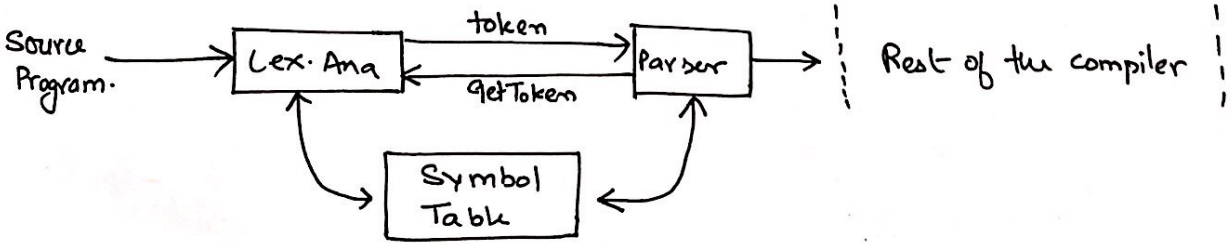
— Role of the L.A

> Read the input program, group the characters into lexemes and produce a sequence of tokens of each lexem.
When a new lexeme containing an identifier is encountered, an entry must be made into the symboltable. [L A]

> Stripping of comments and white spaces. [scanning]

> Correlating errors with the source program. some times a copy of the source is made with the errors messages inserted at the appropriate places.

The LA and Syn.A often interact with each other. This is implemented by the parser calling the LA with the getToken command. The LA Then reads in the next chars, matches the lexeme & produces a token which it gives to the parser.



— LA as an independant component

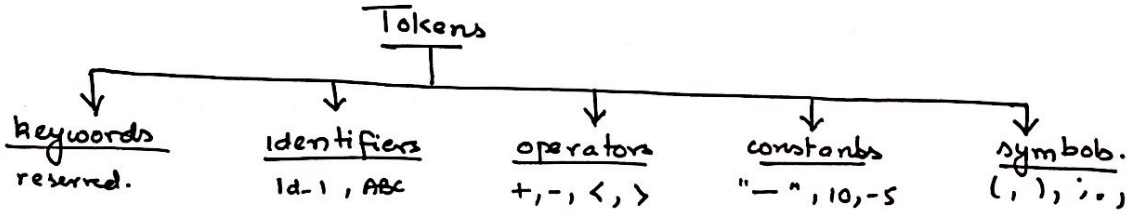
1. Simplifies over all design. The LA deals with token structure and scanning while the parser deals with syntactic structure. a parser that does both will be very complex to create & maintain.
2. Improves Compiler Efficiency. Reading is a very time consuming part. Having an independant L.A allows us to apply specialized tasks and buffering strategies.
3. Due to the inherent advantage of modularity, makes changes easier.

4. Improves the portability of the system.
5. Allows for the consolidation of device specific issues to the LA.

➤ **Token** : A token is a pair consisting of a token name and an optional attribute value. The name is abstract & reflects the type of lexical unit.

➤ **Pattern** : A pattern is the description of the form that the lexeme of a token may take. These are used to match the lexeme with the type

➤ **Lexeme** : A sequence of characters in the source program that matches the pattern for a token and is identified by the lex. A as an instance of that token.



Attributes of Tokens

> When more than one lexeme matches a pattern the L.A provides additional info to the rest of the compiler.

> The LA provides the token name and attribute, the token name influences the parser while the attribut influences the translation.

> This information i.e the lexeme, type, location [for error reporting] is maintained in the symbol table.

The handling to white spaces is a serious implementational challenge sometimes a keyword can be mistook as an id or something else e.g.

eg> DO 5 I = 1.25
can be DO5I as an Id
or DO 5 I as a do while

Token Representation

< token, attributes> eg> <add_op>, <id, pointer to id,>
<literal, string value "My appriciation>
<keyword, if value keyword>

Lexical Errors

> The LA on its own cannot tell errors in the code & thus requires some aid in this task.

> Errors arise when the lexical analyzer is unable to proceed because none of the pattern of the token matches any of the remaing input.

> The simplest strategy (panic mode) is to delete successive characters from the remaing input until the LA can find a well formed token.

> other techniques include

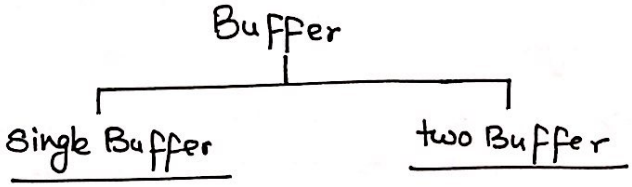
1. delete one char.
2. insert any missing char.
3. replace a char.
4. Transpose two adjecent.

> The simplest transform strategy is to check if the prefix of the input can be transformed to a valid lexeme in one transform.

> The general approach is to find the minimum transforms.

Input Buffering

> To be sure that we have the right lexeme, we often have to look one or more characters beyond the next lexeme.



– Single Buffer

> The Buffer is usually of the size of one disk block i.e 4096 b.

> On one system call, N chars are read into the buffer at once and if there's < N chars a special eof is added.

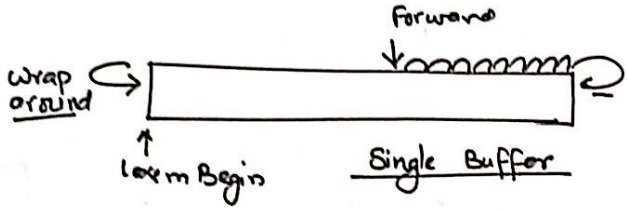
> Two pointers i.e

lexemeBegin : marks the start of the current lexeme.

Forward : scans ahead till the match is found.

> Once the lexeme is determind forward is set to its right. Then it is recorded and reflected in the ST. Then the LexemeBegin is set to the immediate next char of forward.

> One major drawback of single Buffer is if the lexeme is longer than the block, wrap around occcur.



— Two Buffer

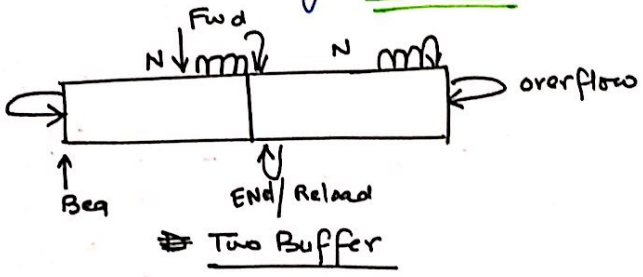
> This involveds two buffers that are alternatly ~~over~~reloaded. Each Block is N i.e one disk block.

> When the Forward is advanced, we check if we haven't reached the buffer end. if so the other buffer must be reloaded, & the fwd is moved to the start of it.

> This suffers from the same wrap around drawback.

> Additionally there's two checks each time forward is moved
1) to check buffer end
2) to check char read.

This can be resolved using sentinels.



For very long sequences like strings, they are broken up and treated as combinatoins.

Token Recognition

> There is a set of patterns (defined by regular exp) for each type of token which are used to match the lexeme.

> These patterns are converted to transition diagrams (DFA/NFA) The start state can be analogous to LexemeBegin and each state transition can be analogous to moving the forward.

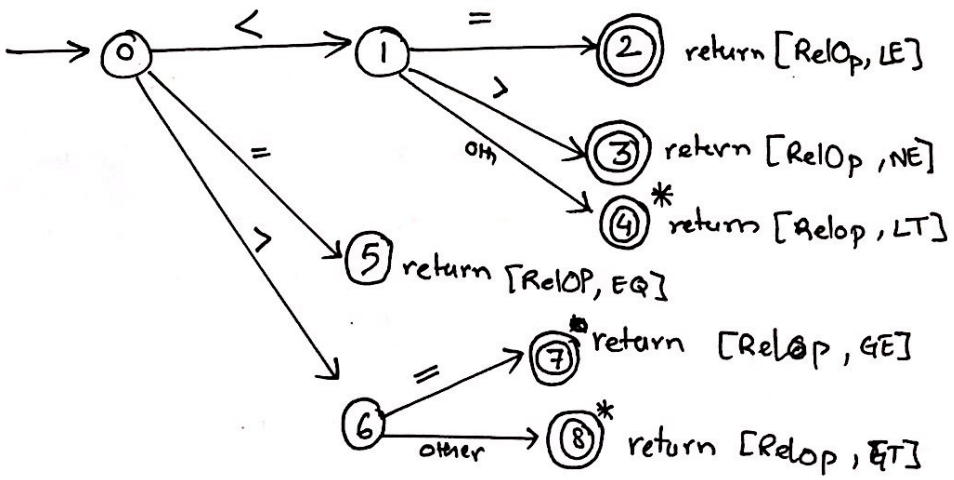
> The basic conventions for such transition diagrams are:

1) Some states are marked as accepting/final. If there is an action to be taken, it is attached to it.
2) If it is needed to retract the fwd pointer i.e the lexeme does not include the transition symbol, we place a *.
3) The TD always starts with the start state.

> We start with one TD and keep moving to the next till we can find a match. If no match is found error recovery is invoked

> Frequently used tokens have their TD higher in the matching scheme.

Eg> Relational Op



Keyword and Identifier Recognition

> The task of Identifier and keyword Recognition is problematic because interms of pattern they're indistinguishable.

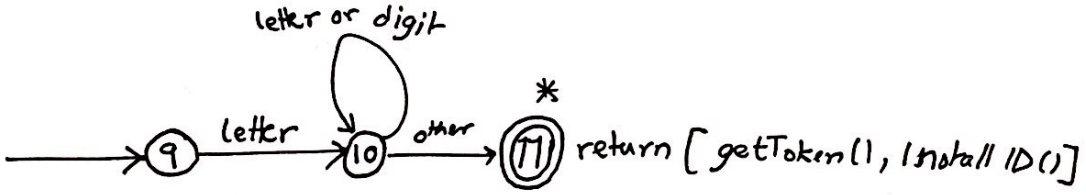
> There are two possible ways to solve this.

1. Use the symbol table intelligently

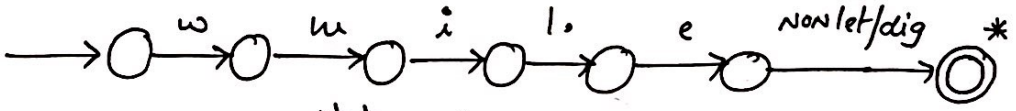
   Install all keywords into the symbol table initially. When an ID is found do a call to InstallID() to place it in the table if its not already there.
   GetToken() examines the symbol table entry for the lexeme and returns its type i.e KW/ID.

2. Make a seperate TD for keywords

   Build a seperate TD for Each keyword. This TD has higher precedence over ID when the lexeme matches both ID & KW.

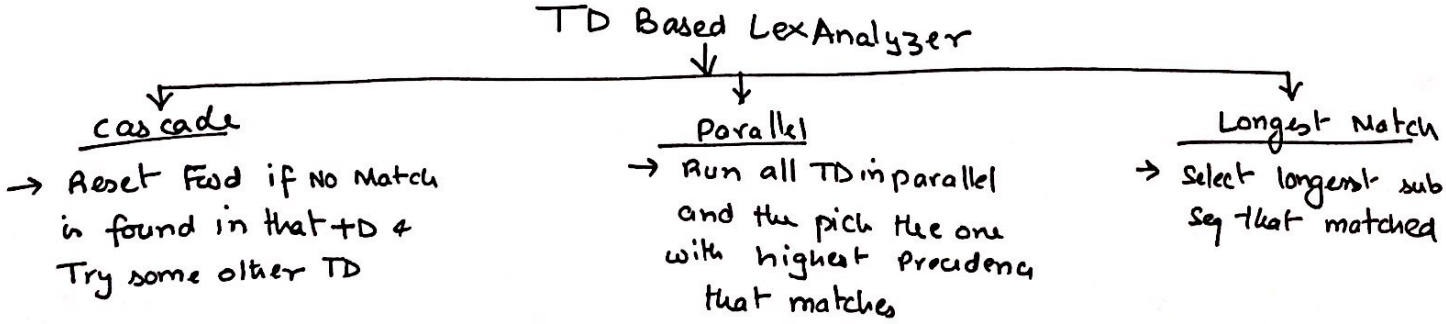


strategy 1.



Strategy 2.

> The main benifit of using 2. over 1. is it allows to easily add or modify the keywords instead of having to redefine the Symbol Table.



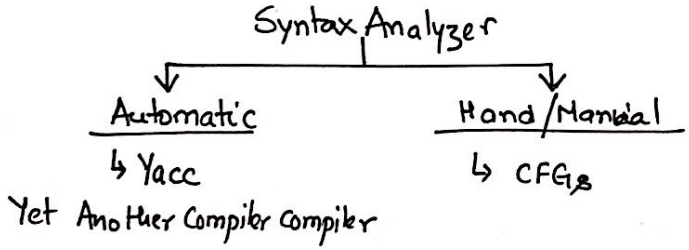
Syntax Analysis [Parsing]

> The syntax analyzer takes in a sequence of tokens and generates a parse tree.

> The syntactical constructs of a language are specified using Context Free Grammars.

Grammar Based Parsers

1. The grammar provides a specific & easy-to-understand specification
2. For some grammars, we can automatically generate parsers. This also allows us to uncover missed ambiguities of the Language.
3. The structed method of using grammar helps ease the translating table
4. Grammars allow the language to evolve & accomodate new constructs.



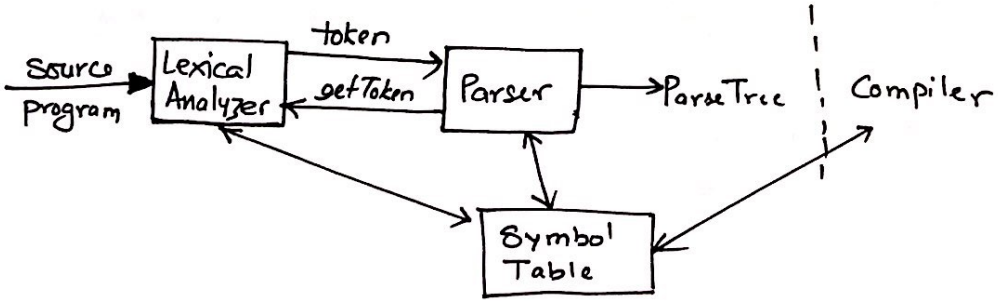
The Role of Parser

> Obtain a string of tokens from the L.A and verify if it can be generated by the Grammar by generating the parse tree

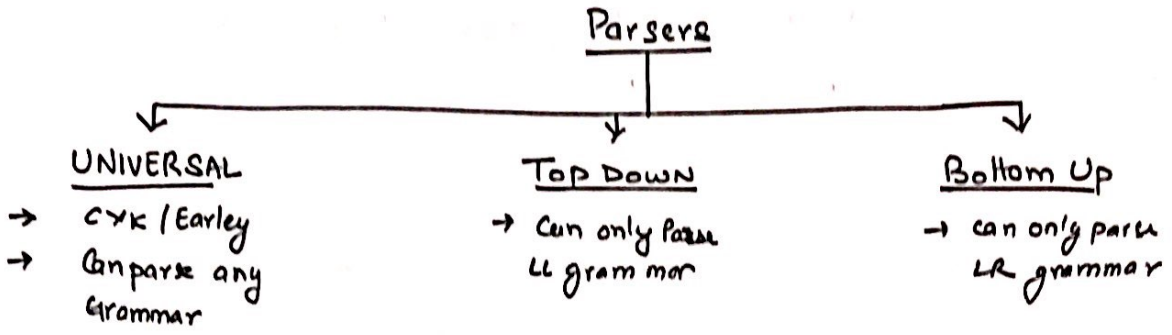
> To report any errors in an intelligible manner.

> Recover from commonly occcuring errors and resume the task.

> In essence, this is the most important part of the compiler.



Types of Parsers



— Universal Parsers

- These use methods like those of the Cocke – Younger – Kasami algorithm or the Earley's algorithm and can parse any type of grammar.
- These are proven to be highly inefficient in production use and are thus rarely used.

— Top Down Parsers

- Just as the name suggests, the parser work from the root to the leaves scanning the input left to Right, one symbol at a time
- These only work (efficiently) with LL grammars i.e L-R read | Left Most derivation.

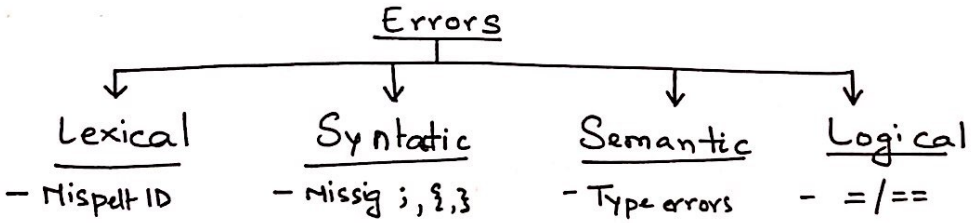
— Bottom-Up Parsers

- Similar to Top-Down parser but start with the leaf and work to the top.
- These only work efficiently on LR grammar i.e L-R read / Right Most derivation.

Larger LR Grammar are generated automatically in most case.

Apart from this parsers also collect info on tokens, do type checking and other analysis in syntax & semantics.

Syntax Error Handling -



Error Types

— **Lexical**
- These include mispelt ID, KW or operators as well as missing quotes arround strings.

— **Syntactic**
- These include misplaced, Nissing, extra ";" or braces. else without ifs, case without switch etc.

— **Semantic**
- These include type mismatch errors, return type matching.

— **Logical**
- These can be incorrect reasoning by the coder, using assignment instead of comparison, equal op.

> Most LL and LR Parsers detect errors as soon as they occur i.e they exhibit the variable prefix property.

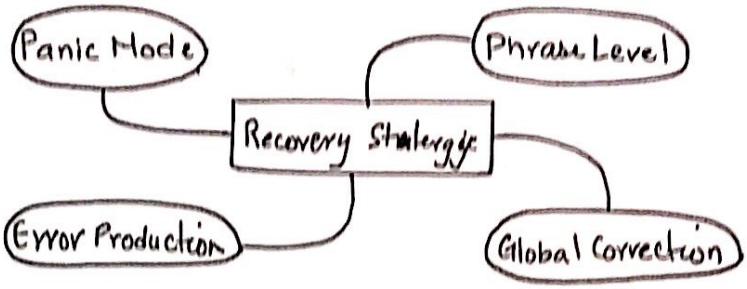
"Report an error if the prefix of the input cannot be completed to form a string in that language."

Role of the Error Handler

1. Report the presence of errors clearly and accurately
2. Recover from error quickly to detect further errors.
3. Add a minimal overhead to the parsing phase.

> The most common error reporting method it to show the offending line with a pointer to the error.

Error Recovery Strategies



> The simplest method is to quit and display an informative message. if the compiler is rerun, additional errors may pop-up and there may be a case when an avalanch of spurious errors occur in this case the compiler must prevent this.

→ **Panic Mode Recovery**

- On finding an error, discard the error one at a time until a set of designated synchronizing token is found (';', '}'). The role of these symbols is always clear and ambigious.
- This has a tendency to skip a considerable amount of errors this is simple and is guaranteed never to go into an ∞ loop.

⇒ **Phrase Level Recovery**

- Perform local correction on the ~~rest~~ remaining input i.e replace/ remove / add a ')', ';', . This choice is left to the designer.
- This can correct any input string but misses errors that occured befor the point of detection
- Can cause error avalanche.

≡ **Error Productions**

- Anticipate common errors and add productions that generate those errors.
- whenever one of these error production is used the parser can generate the appropriate ~~the~~ error diagonistic.
- This is the most commonly used strategy.

四 **Global Correction**

- Select the minimal no of change to be done to the input for it to be error free
- These are very costly methods and can cause unintended behavior.
- input x is changed to least cost y using G.

Context Free Grammar (Review)

— **Terminals** : They're the basic symbols which make up the strings They're shown using small letters or in bold.

⇉ **Non Terminals**: They're the sytactic variables that denote strings They're shown using capital letters.

☰ **Start State** : This is a specied Non Terminal and is the starting point of any string.

🕮 **Productions** : They specify the manner in which NT and T are combined

$$S \longrightarrow \alpha \qquad [\alpha \text{ can be NT, T, or a mix}]$$

i) $S \xrightarrow{*} \alpha$

S in 0 or More derivations derives the sentental α.

ii) $S \xrightarrow{+} \beta$

S in one or more derivatim derives the sentental β.

Parse Trees as Derivation

> A parse tree is the graphical representation of a derivation detailing out the order in which the productions are applied.
> Each internal node is the application of a Production
> Each Leaf node is a terminal. The tree is read left to right.
> The main challenge in building the parse tree is selecting the production to expand first and then which production.

Ambiguity in Grammars

" A grammar that produces more than one parse tree for a sentence is called ambigious "

OR

" An ambigious grammar is one which produces more than one LMD or RMD for the same sentence "

> It is desirable for the grammar to be unambigious as the compiler has no method for determining which parse tree to select. The only exception to this is the operator precidence parser.
> An ambigious grammar can be unambiguated using "disambiguating rulls which discard wrong P Trees.

$$AG \xrightarrow{\text{disambi Rules}} UG$$

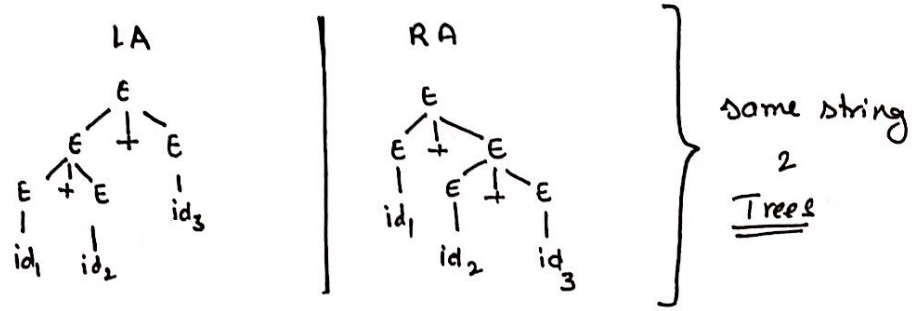
Why does Ambiguity Exist?

→ Ambiguity in associativity -

> operators can be left/ Right associative. The parse tree for both are very diffrent bet sententially same.

> In most languages +, −, *, ÷ are left associative and ^ and = are right associative.

eg $id_1 + id_2 + id_3$

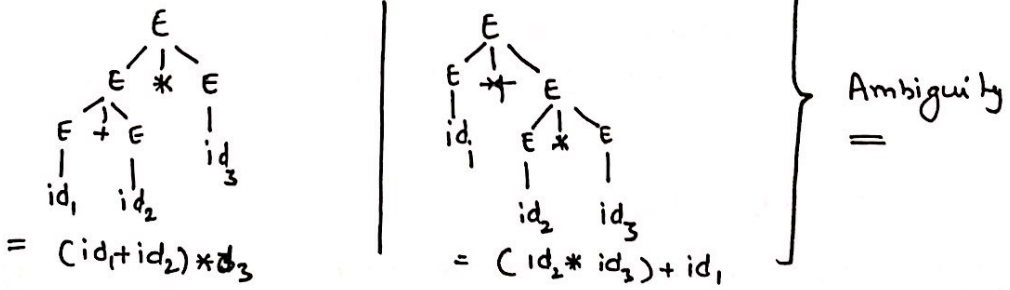


⇒ Ambiguity in presedence

> operators have a scale of presendence i.e if two are to simultaniously occur which should be evaluated first.

> Ideally the operators with least presedence must be evaluated first.

eg $id_1 + id_2 * id_3$

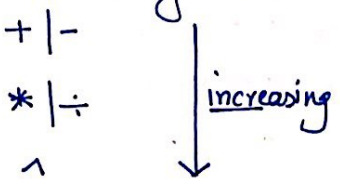


Associativity is handled using recursion.

> Left Asso ⇒ LR

> Right Asso ⇒ RR

Presedence is handled using levels.



Writing Grammar for a Parser

> CFGs are capable of describing most of the syntax but cannot handle task like declaration testing. This is done by consequent Phases.

Why Have a Grammar Based Parser ?

> As a grammar can also be used to describe regular expressions. why is there a need of operation the lexical & syntactic unit into RE and CFG based units ?

→ Modularizes the Front End.

⇒ Lexical Rules are very simple and using complex grammars to describe them is wasteful.

⇒ RE are more concise and easy-to understand notation for tokens.

RE based LAs can be automatically constructed.

"Ideal Grammar for Parser is Unambigious, deterministic and be free of states which might cause ∞ loops"
(esp for top-down parsing)

1 Eliminating Ambiguity

> An ambiguous grammar can be made unambigious using a set of disambiguating rules.

> For operators associativity, it can be resolved using Recursion.
(Top Down parsers go into ∞ loops with LR ∴ they're Iteration changed to RR)

> For operator precedence, it can be resolved by using levels with low precedence at the top.

> For conditionals "Match elses with closest unmatched thens"

> These unambiguating rules can be built directly into the grammar but are not in practice.

Eg $E \to E+E \mid E*E \mid id$

Ambigious

$$\left.\begin{array}{l} E \to TE' \\ E' \to +TE' \mid \epsilon \\ T \to FT' \\ T' \to *FT' \mid \epsilon \\ F \to id \end{array}\right\} \text{ unambigious.}$$

2. Eliminating Non Determinism

> If a grammar shows the common prefix problem then it suffers from non determinism.

$$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3 \ldots | \alpha\beta_n$$

> In the above production, there is no way of telling which production is to be selected without having to look ahead.

> The common prefix problem is solved by "Left Factoring".

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 | \beta_2$$

Algorithm : Left Factor (G)

1. For each NT in grammar say A, find the longest common prefix common to 2 or more productions α
2. if $\alpha \neq \epsilon$

   Replace $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \ldots | \alpha\beta_n | \gamma$

   with

   $A \rightarrow \alpha A' | \gamma$

   $A' \rightarrow \beta_1 | \beta_2 | \ldots | \beta_n$
3. Repeat till No NT exist showing the Common Prefix Problem.
4. Return Left Factored G.

> Non-Determinism and ambiguity are very different and solving for one doesn't solve for the other

> With ambiguity we know the solution but here we don't.

> This is only a problem for parsers that cannot back track but ambiguity is a problem for all.

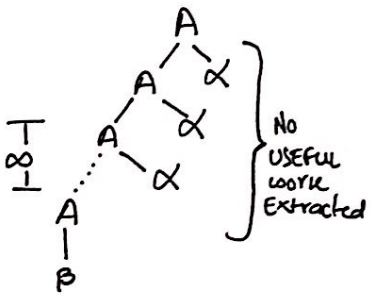
3. Eliminating Left Recursion

> A grammar shows Left Recursion if for some A in G there is a production of the type $A \overset{+}{\Longrightarrow} A\alpha$.

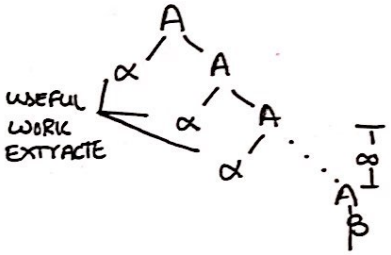
> Top down parsers enter an ∞ loop with such grammar.

$A \rightarrow A\alpha \mid \beta$

LR



$A \rightarrow \alpha A \mid \beta$



> Even if the parser enters an ∞ loop in RR there is still useful work being done thus there'll arise a case when it breaks.

> This Recursion can be immediate as in

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n$$

or be non immediate as in

$$A \rightarrow B \mid C$$
$$C \rightarrow A\alpha \mid id$$

> Immediate and non immediate recursion are handle by the following algorithm

Algorithm : Eliminate LR (G)

1. Arrang NT as $A_1, A_2 \ldots A_n$.
2. for each i from 1 to n {
3. for each j from 1 to i-1 {
4. replace all $A_i \rightarrow A_j\gamma$ by $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$ where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are current A_j Prods.
5. }
6. Eliminate all immediate LR among A_i productions.
7. }

← RR-G

> The above algorithm have two main requirements

1. No cycles . [$A \overset{+}{\Rightarrow} A$]
2. No ϵ productions . [$A \rightarrow \epsilon$]

> Immediate recursion is removed as follows

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

$$= A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$$

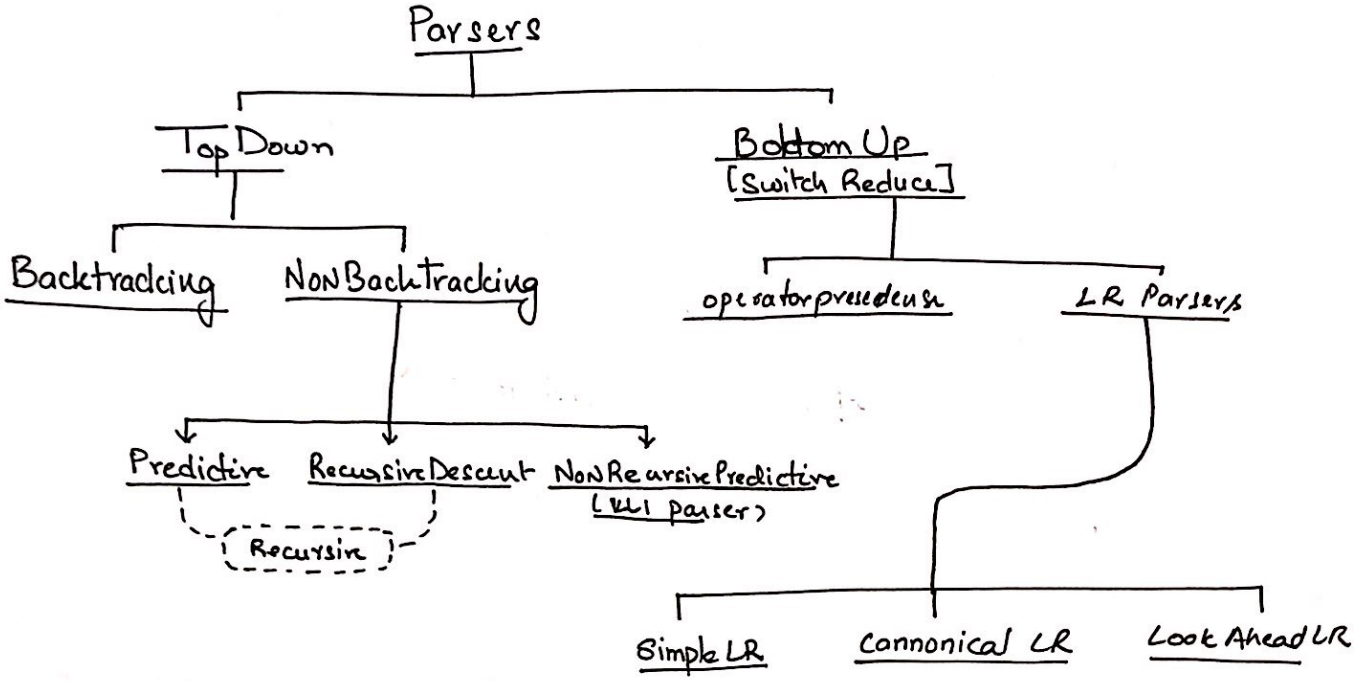
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A' \mid \epsilon$$

> There can be ϵ in the grammar if and only if it doesn't induce the recursion.

$$A \rightarrow A\alpha \mid \beta$$

$$= A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \epsilon$$

Classification of Parsers



Top Down

$S \Rightarrow aABe \Rightarrow aAbcBe \overset{*}{\Rightarrow} abbcde$

Bottom Up

$abbcde \Rightarrow aAbcde \Rightarrow aAde \overset{*}{\Rightarrow} S$

Top down Parsers

- This can be defined as the task of constructing a parse tree for the input starting from the root and creating the nodes in pre order.
- It is basically finding the leftmost derivation.
- A predictive parser can choose the correct A production by looking ahead a fixed number of symbols.
- Top down, predictive parsers only work on unambiguous, deterministic, right recursive grammars i.e LL(k) ~~parse~~
  k here is the look ahead.

— Recursive Descent Parsing

> This contains a set of procedures, one for each NT.
> The execution begins with the start symbol & halts when the entire string is scanned.
> General algorithms may require backtracking but normally they're not used.
> The main point when dealing with recursive descent parsing is that when using LR grammar even with backtracking they can go into an ∞ loop.

First and Follow

> First and follow allow us to determine which production to apply based on the next input symbol.
> First and follow are very useful when we're building the parsing table.
> In panic mode recovery, the tokens defined by follow can be the synchronizing tokens.

→ First(α)

> First(α) where α is a string of grammar symbols is the set of of terminals that begin strings derived from α and if $\alpha \overset{*}{\Rightarrow} \epsilon$ then ϵ is in first(α).

Compute First(α):

1. if α is a terminal then First(X) = {X}
2. if $\alpha \to \epsilon$ is a production add ϵ to first(α).
3. if X is a NT & $X \to Y_1 Y_2 \cdots Y_k$ for $k \geq 1$ then place a in first(X) if a is in First(Y_1) and ϵ is in all of First(Y_1), ..., first(Y_{k-1}) i.e $Y_1 \ldots Y_{i-1} \overset{*}{\Rightarrow} \epsilon$. if ϵ is in First(Y_j) $\forall j = 1, 2 \ldots k$ add ϵ to first(X)

> The First of a string like $X_1 X_2 \ldots X_n$ is first of (X_1) if $X \overset{*}{\Rightarrow} \epsilon$ is not a production otherwise it'll also have first (X_2) and so on.

⇉ Follow (α)

> Follow(A) for a NT A are the set of terminals that can appear immediately to the right of A in some sentential form. i.e a set of terminals a such that there exists $S \overset{*}{\Rightarrow} \alpha A a \beta$.

> If A is the right most symbol in a sentence then $ is in follow(A).

Compute Follow (A) :

1. Place $ in follow(S). S is the start symbol.
2. If there is a production $A \to \alpha B \beta$ then everything in first(β) except ϵ is in follow(B).
3. If there is a production $A \to \alpha B$ or $A \to \alpha B \beta$ and first(β) contains ϵ then everything in Follow(A) is in Follow(B).

LL(1) Grammars

> The first L stands for Left-to-Right Reading of input.
The second L stands for Left most derivation.
The (1) means a Look ahead of 1.

> A grammar is LL(1) iff whenever $A \to \alpha | \beta$ are two distinct productions then

- ⇉ For no T 'a' do both α and β derive string starting with a
- ⇉ at most one of α or β can derive empty.
- ⇉ if $\beta \Rightarrow \epsilon$ then α does not derive any str begening with a T in Follow(A). & v.v.

> Basically in LL(1) grammar with $A \to \alpha | \beta$ production

- ⇉ First(α) and First(β) are disjoint.
- ⇉ if $\epsilon \in$ First(B)
  First(A) ∩ Follow(A) = ϕ.
- ⇉ if $\epsilon \in$ first(A)
  First(B) ∩ Follow(B) = ϕ.

— Building The Parsing Table

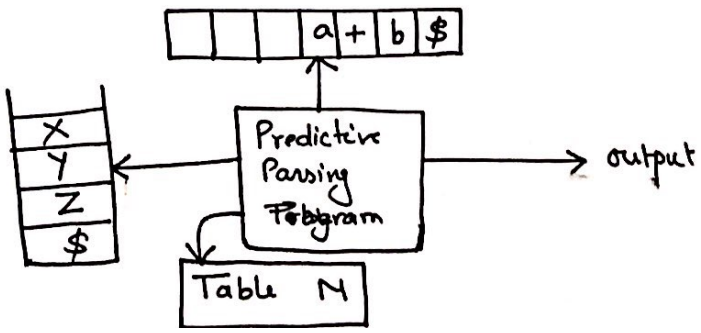
```
Construct_Parse_Table(G):
1. For each terminal a in First  A→α in G
2.      For each terminal a in First(α) add A→α to M[A,α]
3.      if ε is in First(α), For each terminal b in Follow(A)
        add A→α to M[A,b].
        If ε is in First(α) and $ in follow(A) add
        A→α to M[A,$].
4. Mark each undefined entry in M as error.
```

AMBIGIOUS GRAMMAR WILL HAVE MULTIPLE ENTRY IN M[A,α].

Non Recursive Predictive Parser (LL(1) Parser)

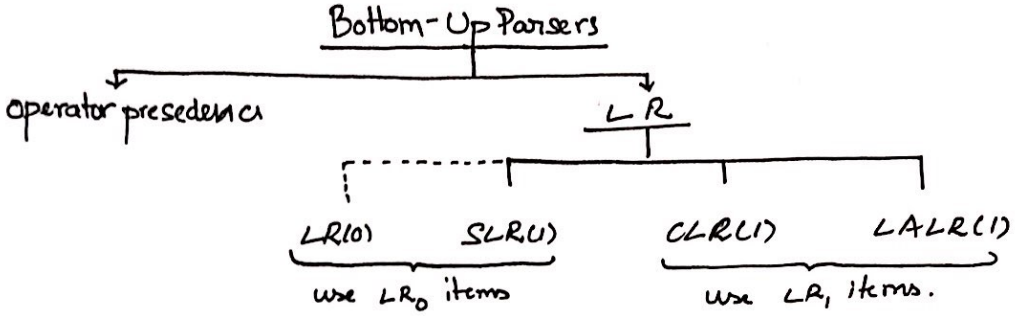
- This type of parser can be built by maintaing a stack explicitly.
- These are driven by the parsing table M.
- The parser is controlled by the symbol X on top of the stack if X is a NT then M[X,a] is used otherwise the T is matched.

```
N_R_Pr_Parser(w, M):
1.  let a be the starting (first) symbol of w.
2.  let X be at the top of stack.
3.  while (X≠$)
4.      if (X==a) Pop & let a be the next symbol in w.
5.      elif (M[X,a] is an error) error()
6.      else if (M[X,a] = X→Y1Y2..Yk)
7.              output X→Y1Y2...Yk
8.              Pop
9.              Push Yk Yk-1 ... Y1
10.     let X be the top of stk.
```



Bottom-Up Parsers

- Construction of a parse tree for an input string beginning at the leaves and working up towards the root.
- It follows a left to Right scan, Rightmost derivation in reverse.



Reductions

- Bottom-up parsing is a process of reducing a string w to the start symbol of a grammar.
- At each reduction step a specific substring matching the body part of a production is replaced by the nonterminal at the head of the production.
- A reduction is the reverse of a derivation step.

Bottom up thus constructs a derivation in reverse.

Handles

" A handle is a substring that matches the body of a production & whose reduction represents one step along the reverse RMD. "

Formally if $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha A w \overset{*}{\underset{rm}{\Rightarrow}} \alpha\beta w$ then $A \to \beta$ in the position following α is a handle.

If the grammar is unambiguous then every right sentential form of the grammar has exactly one handle.

The Reverse RMD can be obtained using handle pruning i.e start with a string of terminals w, if w is a sentence. let $w = \gamma_n$; γ_n is the n^{th} right sentential form of some der^n.
To get the derr locate handle β_n in γ_n and reduce.

Shift-Reduce Parsing (LR(k) Grammars only)

- A stack holds the grammar symbols
- The buffer holds the string.
- $ marks both the bottom of stack and end of an input.
- Initial config.

Stack	buffer
$	w$

- Successful parse config

Stack	buffer
$	$

- Scanning is done left to right and at each step the parser can put none or more symbols into the stack until it can reduce with β.
- This is repeated till there is an error or it is successfully parsed.

Operations

1. Shift : shift/push next input symbol onto the top of stack
2. Reduce : Right end of handle must be at top and then locate the left end and replace with LHS.
3. Accept : Announce successful parse
4. Error : Discover syntax error and call error handler.

" The handle will always eventually appear at the top of the stack, never inside. "

The main issues in design are what to reduce with and when to reduce.

Conflicts occur when we know we have a handle but the stack and input are insufficient to produce a decision for what to reduce with.

LR(k) Parsing -

L means Left to right scanning of the input buffer.

R means Right most derivation done in reverse.

k means the look-ahead size for getting a parsing decision.

They are table driven and work on LR grammars

LR Grammar: A grammar for which an LR parser can recognize a handle in Right Sentential form.

Advantages:

- LR parsers can be made to recognize virtually all programming constructs for which a CFG can be made
- LR parsing is the most general non backtracking Shift Reduce parsing.
- LR parser can detect errors as soon as they occur.
- The LR grammar class is a proper superset of LL grammar class.

Drawback: Way too much work. use Yacc instead.

Items: An Item is a grammar production which has a dot at some place in it

Eg $A \to \alpha$ has $A \to \cdot\alpha$ and $A \to \alpha\cdot$ as items.

Items indicate howmuch of a production we have seen at any point.

Augmented Grammar

An augmented grammar G' is a grammar with a production as $S' \to S$ added to the original grammar G.

The use of it is to show the end of parsing i.e when $S' \to S$ is about to be reduced.

Closure

If I is an item of G then the closure of I is the item constructed from I by:

1) add every item of I to closure(I).
2) if $A \to \alpha \cdot B\beta \in$ closure(I) and $B \to \gamma$ exists then add $B \to \cdot\gamma$ to closure(I).

Closure means at some point we expect to see a string derived from it.

LR Parsing Algorithm

- An LR parser consists of an input, an output, a driver program and a parsing table.
- Only the table varies b/w LR parsers.
- The parse driver shifts between states while parsing.

Structure of LR Parsing Table

The table has two parts, A parse Action part and a GOTO part.

Action takes in a state i and a terminal a and gives one of the following outputs.

1. Shift j : shift the input a to top of stack and push in j
2. Reduce $A \to \beta$: Reduces β and leaves A.
3. Accept : Accept the input and stop.
4. Error : discovery of error.

GOTO is for a set of items and if $GOTO[I_i, A] = I_j$ map state i to NT A to a state j.

```
LR-Parsing(w, LR_Table):
1.  put S0 onto stack
2.  let a be first symbol of w$;
3.  while(1) {
4.      let s be state on top of stack;
5.      if Action[s,a] = shift t {
6.
7.                  push t onto stack;
8.                  a = next_char();
9.      } else if (Action[s,a] = Reduce A→β) {
10.                 Pop |β| symbols;
11.                 let t be top of stack;
12.                 push GOTO(t,A) onto stack;
        } else if (Action[s,a] = accept) break;
13. } else call errorhandler
```

Construction of SLR Parsetable

- This begins with LR(0) items and the LR(0) automata.
- Get G' from the given grammar G.
- Get the cannonical collection of items of G'
- Get the goto function.

Build SLR Table (G')

1. Construct $C = \{I_0, I_1, \ldots, I_n\}$ the collection of set of LR(0) items for G'.
2. state i is constructed from I_i the actions are:
   - 2.1. if $[A \to \alpha . a\beta] \in I_i$ and GOTO$(I_i, a) = I_j$ set action $[i, a]$ to shift j.
   - 2.2. if $[A \to \alpha .] \in I_i$ set action $[i, a]$ to reduce $A \to \alpha$ $\forall a \in$ follow(A).
   - 2.3. if $[S' \to S] \in I_i$ set Action $[i, \$]$ to accept.
3. The goto are made for state i and NTA as:
   if GOTO$[I_i, A] = I_j$; GOTO$[i, A] = j$.
4. Mark all undefined entried as error.
5. initial state is the state for $[S' \to S]$.

Drawback of SLR

- The method for filling in reduce steps is stupid and should never be used.

Eg

$$S \to L = R \mid R$$
$$L \to *R \mid id$$
$$R \to L$$

$I_0 =$ $S' \to .S$, $S \to .L = R$, $S \to .R$, $L \to .*R$, $L \to .id$, $R \to .L$

$I_1 =$ $S' \to S.$

$I_2 =$ $S \to L. = R$, $R \to L.$

1) Action $[2, =] =$ shift to 6
2) Action $[2, =] =$ Reduce $R \to L$

} conflict.

Not all unambiguous grammars are SLR(1).

Variable Prefix

- The prefixes of right sentential forms that can appear on the stack of a LR parser
- A variable Prefix is a prefix of a right-sentential form that does not continue past the right end of the rightmost handle of that sentential.
- It is a central theorem of LR parsing that the set of valid item for a variable prefix γ is exactly the set of items reached from the initial state along the path labeled γ in the LR(0) automata.

Canonical LR Parser

- Makes full use of look ahead symbols
- Makes use of a large set of items called LR(1) items.

- LR(1) items

- By splitting states we can arrange to have each state of an LR Parser to indicate exactly which input can follows a handle α.
- The information is encorporated as lookahead.

$$[A \rightarrow \alpha \cdot \beta, a]$$

$A \rightarrow \alpha\beta$ is a production

a is a terminal or \$.

- Lookahead has no effect when β is not ε but $[A \rightarrow \alpha \cdot, a]$ calls for a reduction only when input is a.
- LR(1) item $[A \rightarrow \alpha \cdot \beta, a]$ is a valid variable prefix γ if

$$\exists \; S \overset{*}{\Rightarrow} \delta A w \overset{*}{\Rightarrow} \delta \alpha \beta w$$

1) $\gamma = \delta\alpha$
2) a is first symbol of w or w is ε and $a = \$$.

+ Generating LR(1) Items

- Use a modified closure logic.

```
closure (I):
  repeat :
    foreach item [A→α·Bβ,a] in I
      for each production B→γ in G'
        for each terminal b in first(βa):
          add [B→·γ, b] to I;
  Until no more items are added.
```

```
GOTO (I,X):
  INIT J as empty set
  for each item [A→α·Xβ,a]∈I:
    add [A→αX·β,a] to J
  return closure (J)
```

Building The CLR(1) Table

Build CLR Table (G'):

1. construct $C' = \{I_0, I_1, \ldots I_n\}$ the set of LR(1) items.
2. State i is constructed from I_i; the actions are:
   - 2.1 if $[A \to \alpha \cdot a\beta, b] \in I_i$ and $GOTO(I_i, a) = I_j$ set $Action[i, a] = shift\ j$
   - 2.2 if $[A \to \alpha \cdot, a] \in I_i$ and $A \neq S'$ then set $Action[i, a]$ to reduce $A \to \alpha$.
   - 2.3 if $[S' \to S, \$] \in I_i$ set $Action[i, \$]$ to accept.
3. Goto are constructed from i $\forall$ NT A st: if $GOTO(I_i, A) = I_j$; $GOTO(i, A) = j$.
4. All blank entries are errors
5. The init state is $[S' \to .S, \$]$

The number of states in CLR(1) $\geq$ SLR(1) always.

LALR

Lookahead LR Parsers

- Formed by merging the states of a CLR(1) parser where NO conflicts arise,
- a conflict can be a Reduce-Reduce conflict eg:

$I_1 =$ { $A \to a \cdot, c$; $B \to b \cdot, d$ } $I_2 =$ { $A \to a \cdot, d$; $B \to b \cdot, c$ }

No conflict

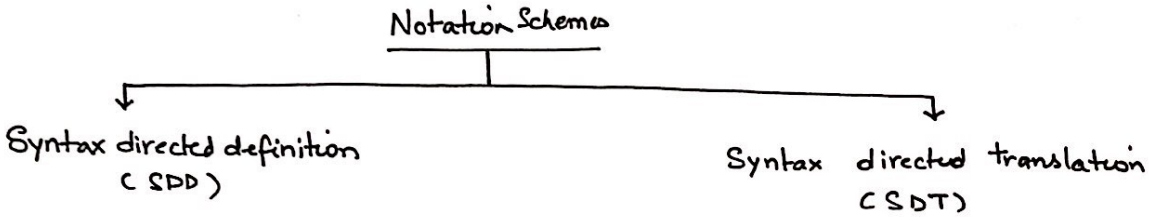
$I_{12} =$ { $A \to a \cdot, c/d$; $B \to b \cdot, c/d$ } R/R conflict.

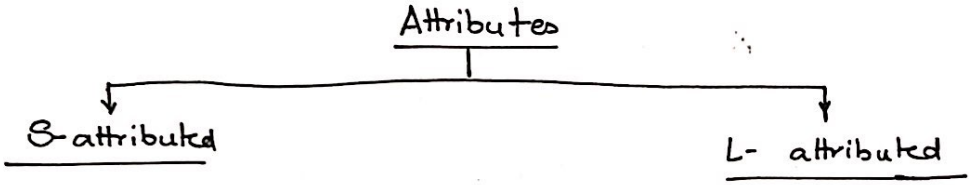
- The number of states in an LALR parser are always same as that in SLR parser.
- The idea of creating the LALR table is to construct the set of LR(1) items and if no conflicts arise, merge them, the ones with identical LR(0) cores.

All LALR grammars are CLR but all CLR grammars may not be LALR.

Semantic Analysis

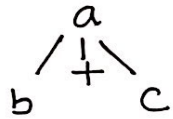
- Can associate info with a construct by adding attributes to the grammar.
- Done by augmenting Semantic Rules to the CFG.
- The semantic analyzer ensures the program conforms to a certain semantic.
- Semantic Analysis can capture things like names, scopes, types etc which syntax cannot.
- The semantic rules are attached to the productions.





each symbol has at least one associated attributes but depending on implementation exceptions might exist.

S attributes: Value of these depends only on the children (synthesized from).



a depends on [b, c]

These are evaluated in a bottom up manner.

L attributes : The value for these depends on the parent of the value or its left sibling.

These can be evaluated with both top down and bottom up parsers.

Semantic Analysis is not an independent process, it is done along with parsing.

Syntax Directed Definition (SDD)

- A SDD is a augmented CFG in which each grammar symbol, terminal or nonterminal has an associated set of attributes.
- SDD are abstract specifications i.e It only tells the way in which the attributes relate but does not cover anything about when they are computed.
- The rules set up a dependency graph which can be used to derive an order to execute the rules.
- The parse tree containing values at each node is called the annotated parse tree.

Formally: A SDD is a generalization of a context free grammar where each production $A \to \alpha$ is associated with a set of semantic rules of the type:

$$X := f(a_1, a_2 \dots a_k)$$

1. X is a synthesized attribute of A and $a_1, a_2, \dots a_n \in \alpha$
2. X is an inherited attribute of one of the grammar symbols in α and $a_1, a_2, \dots a_n \in A$ or α.

A SDD is S-attributed if every grammar symbol has synthesized attributes.

Inherited values are useful as they express dependencies in the context in which they appear.

Eg> 1. Synthesized Values Only

$E \to E_1 + T$	{ E.val := E_1.val + T.val }
$E \to E_1 - T$	{ E.val := E_1.val - T.val }
$E \to T$	{ E.val := T.val }
$T \to T_1 * F$	{ T.val := T_1.val * F.val }
$T \to T_1 / F$	{ T.val := T_1.val / F.val }
$T \to F$	{ T.val := F.val }
$F \to (E)$	{ F.val := E.val }
$F \to$ num	{ F.val := num.lexval }

2. Using Both Inherited and Synthesized

$D \to T$ list	{ list.type := T.type } // sibling
list $\to$ $list_1$, id	{ $list_1$.type := list.type } // parent
	{ id.type := list.type }
list $\to$ id	{ id.typ := list.type }
$T \to$ int	{ T.type := integer }
$T \to$ Float	{ T.type := float }

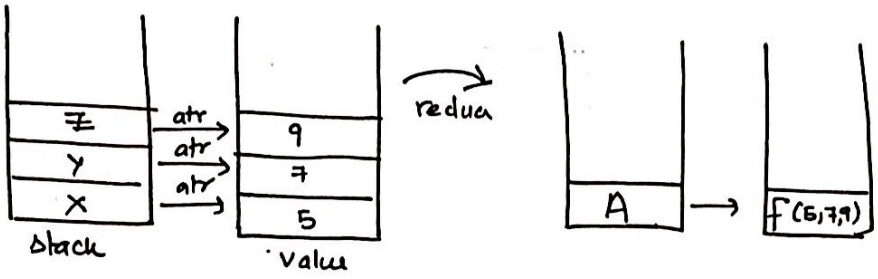
Every S attributed value is allways L attributed.

S-attributed Bottom-Up Parser

- only have synthesized values
- Uses a second stack to hold the attributes. called the value stack.
- if the stack contains the symbol A at i then the value stack at i has its attributes.
- If there is no attribute defined the stack entry is undefined too. (like operators)
- The value of new synthesized attribute can computed from the attributes appearing on the stack for that grammar symbol.

  Eg if there are K terms on RHS of the production then new top after reduction of value stack is [Top - K + 1]

$A \rightarrow XYZ$



L-attributed Definitions

- S-attributed value dont allow for inherited values and are too restrictive.

Formally: A SDD is said to be L attributed if for every $A \rightarrow X_1 X_2 .. X_n$ each inherited value of X_j depends only on:

1. The attributes of X_k where $k \leq j-1$
2. The inherited attributes of A.

- These can be implemented using LL(1) parsers for some grammars and using LR(1) parsers ~~using~~ for a larger grammar set.

Translation Schemes (SDT)

- A Translation scheme is a refinement of a SDD where the semantic actions are embedded within the right size of a production to specify the exact spot / point in time the actions are executed.
- A translation scheme generates an output for each sentence x generated by the grammar by executing the actions in the order they appear during the DFS of the parse tree.

Eg

$E \to E+T$	{ print ('+') }
$E \to T$	
$T \to 0$	{ print ('0') }
$T \to 1$	{ print ('1') }
⋮	

→ remove Left Recn →

$E \to TE'$	
$E' \to +T$	{ print ('+') } $\mid \epsilon$
$T \to 0$	{ print ('0') }
$T \to 1$	{ print ('1') }
⋮	

The actions are shown using dotted links in the parse tree.

- The restrictions of L-attributed values ensure that the attribute values are available.

Guidelines for designing a Translation Scheme

1. if there are only synthesized attributes, we can create an action consisting of an assignment for each semantic rule & place it at the end of it (production)
2. if there are both synthesized and Inherited values.
   - Compute an inherited attribute for a symbol on the RHS of a prod in an action before that symbol.
   - No action should refer to a synthesized attribute to the right of it.
   - Compute synth attr for non terminals only after computing all the attributes it references. This action can be placed at the end of RHS.

L-Attributed Bottom Up Parser

- To accomodate embedded actions we introduce marker symbols. in place of the action and add another Production for the marker.

Eg $A \to X_1 \{ACT\} X_2$

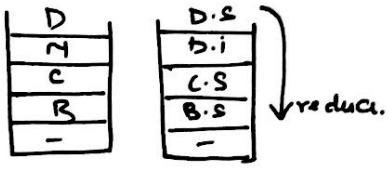
becomes

$A \to X_1 M, X_2$

$M_2 \to \epsilon \{ACT\}$

- Now any inherited value will be on the corresponding stack level of the marker symbol on the value stack.

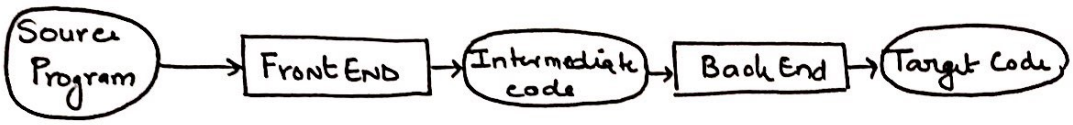
Eg $\underset{M}{A} \mid \begin{matrix} A.s \\ A.i \end{matrix}$ or $A \to BCMD$, $M \to \varepsilon \mid Ad$



- The remaining parsing follows that of SDD.

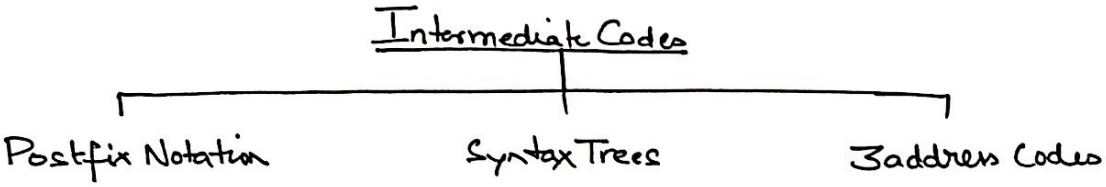
Intermediate Code Generation

- It is not always possible to directly generate target code in one pass of the compiler.
  - Features like delayed declaration may not be possible to handle in just one pass.
  - There may not be enough memory.
  - Single pass may not be able to generate highly efficient code.



Advantages of Machine Independent-IC

- We can create compilers for different machines by only creating seperate backends.
- We can create compilers for different languages by only making new front ends.
- We can apply machine independent optimn to the IC and make it st it is easier to translate.



To select a type of IC we need to consider:

1. ease of conversion from source to IC.
2. ease of conversion and optimization to target code.

Syntax Trees

- A syntax tree is a condensed form of the parse tree which is used to represent the syntactic structure of the language constructs. operators and keywords are at the interior nodes while numbers and identifiers are at the leaves.

Construction of Syntax Trees

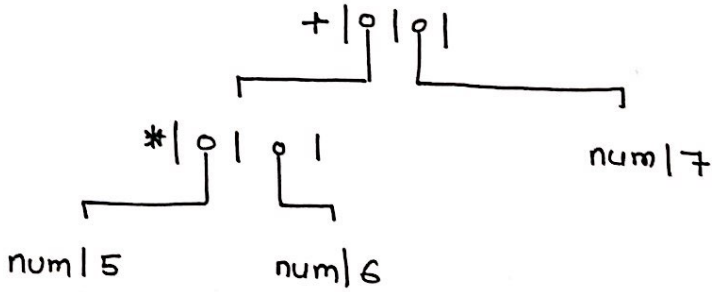
1. Create a leaf node for all constants and identifiers.
2. create subtrees for all subexpressions.
3. operators and the root of the subtrees with operands as children.

- It uses 3 functions
  1. CidLeaf(ident, sptr) : This takes in the id and the pointer to symbol table and returns a pointer to the newly made leaf node.
  2. CnumLeaf (num, value): This takes in the token number and its value and return a pointer to the new node.
  3. CopNode (oper, first, second) : This takes in the token oper and the two operands which may be other subtrees. and returns a new node.

Eg:

$E \to E_1 + T$	{ E.np := copnode (+ , E_1.np, T.np) }
$E \to T$	{ E.np := T.np }
$T \to T_1 * F$	{ T.np := copnode (* , T_1.np , F.np }
$T \to F$	{ T.np := F.np }
$F \to (E)$	{ F.np := Ep }
$F \to id$	{ F.np := cidleaf (ident, sptr) }
$F \to num$	{ F.np := cnumleaf (num, val) } .

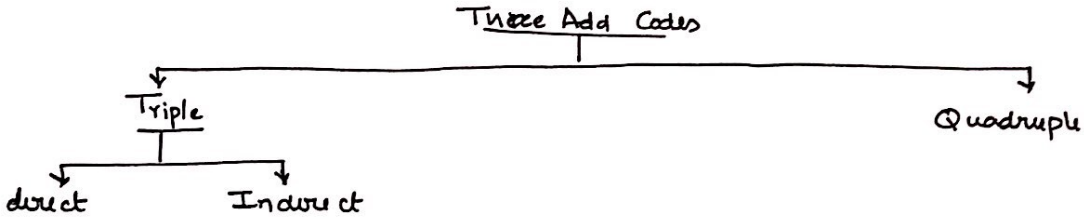
5 * 6 + 7



Add : $S \to id := E$ { id.np := cidleaf (ident, sptr) }
{ S.np := cnode (:=, id.np, E.np) }

To get SDD for assignment operation.

Three Address Codes



- Each statement at most has 3 addresses ; two for operands and one for the result

$$a := b \ op \ c$$

Common 3address Codes

- a := b op c ,
- a := op b
- a := b
- goto x
- if a relop b Goto x
- if a goto x
- param a call p,n return x
- a := b[i] or b[i] := a
- a := &b or a := *b or *a := b.

- These are implemented as records having fields for operator and two operands.

Triple : 3 records i.e operator, oprand1, oprand 2.
operands are either pointers to the symbol table or pointers back in the table of triples.
Eg a := a+b*c+c*d*e

Tiple	opr	op1	op2
0	*	b	c
1	*	c	d
2	*	(1)	e
3	+	a	(0)
4	+	(3)	(2)
5	:=	x	(4)

} Table of triples

- Triples make Identification of identical subexpressions easier.
- Table can be maintained and statements may be represented as a pointer to triples in order. This is Indirect Triple.

Advantage: makes optimization of cases where statements are moved around better.

Quadruple: 4 records i.e operator, oprand1, operand2 and result.

op1, op2, and result are references to symbol table entries.

- This use of result allows for greater flexibility for optimizing by making movement of statements even easier.

opr	op1	op2	res
*	b	c	t_1
*	c	d	t_2
*	t_2	e	t_3
+	t_1	a	t_4
+	t_3	t_4	t_5
:=	t_5		x.

} list of temporaries.

Generation of three Address Code

This is done by writing Semantic Rules.

It uses the following functions:

1) GenTemp(): returns a sequence of temporaries.
2) Gen( ): generates the 3 address statement and add it to the list of outp
3) Lookup(id.entry): gives the entry for id if it exists or otherwise gives Nil.

Eg) Assignment operation

$S \rightarrow id := E$ { id.name := lookup(id.entry) if id.name != nil: gen(id.nam := E.name) }

$E \rightarrow E_1 + E_2$ { E.name = gettemp gen(E.name ':=' E_1.name '+' E_2.name }

$E \rightarrow id$ { id.name = lookup(id.entry) if id.name != NIL E.name . id.name }

Type Conversion

- AKA coercion (done automatically by the compiler).

Eg X := y * 3 + y * r

(float) (int) (float) (int)

t_1 = int2float(y)
t_2 = 3 * t_1
t_3 = y * r
t_4 = INT2FLT(t_3)
t_5 = t_4 * t_2
X = t_5.

Assignment Op

$S \rightarrow id := E$

```
if lookup(id.entry) = NIL : Error
if id.T = INT and E.T = INT
        gen( id.name ':=' E.name)
        S.type := void
else if id.T = Float and E.T = Float
        gen(id.name ':=' E.name)
else if id.T = F and E.T = int
        t := floatGentemp()
        gen(t ':= int2float E.name)
        gen(id.name '=' t)
```

$E \rightarrow E_1 + E_2$

```
E1.T = INT and E2.T = INT
        E.T = INT
        E.name := INTGENTEMP()
        gen(E.name ':=' E1.name 'INT+' E2.name
else if E1.T = Float and E2.T = float
        E.T = float
        E.name = floatgentemt()
        gen(E.name ':=' E1.name 'float+' E2.name)
else if E1.T = float and E2.T = ~~float~~ INT
        E.T = float
        E.name = floatGenTemp()
        gen(t ':=' int2float E2.name)
        gen(E.name ':=' E1.name 'float+' t.name)
else if E1.T = INT and E2.T = float.
        E.T = float
        E.name := floatGenTemp()
        gen(t ':=' int2float E1.name)
        gen(E.name '=' t.name 'float+' E2.name)
else
        E.Type := E-error
```

Boolean Expressions

- They can be either as true/ false or using numbers 1 and 0.
- Using Numeric Representation

$E \to E_1 \| E_2$	E.name := gentemp gen (E.name ':=' E_1.name '||' E_2.name) ;
$E \to E_1 \&\& E_2$	E.name := gentemp gen (E.name ':=' E1.name '&&' E_2.name);
$E \to$ true	E.name := gentemp ~~E.name~~ gen(E.name ":=" 1)
$E \to$ false	E.name := gentemp gen (E.name ':='false)
$E \to E_1$ relop E_2	E.name := gentemp. gen (if E_1.name repo.op E_2.name goto nextlabel+3); gen (E.name :=0) gen (goto next lable +2) gen (E.name != (1));

- Using Bodean Expression and Control flow

- We use the function genlabel which generates a new label.
- each symbol has a .f and .t values which are jump to labels.

$E \to E_1 \| E_2$

$E_1.t := E.t$
$E_1.f := genlabel$
$E_2.t := E.t$
$E_2.f := E.f$
$E.code := E_1.code \| gen(E_1.f ':') \| E_2.code.$

$E \to E_1 \&\& E_2$

$E_1.t := genlabel$
$E_1.f = E.f$
$E_2.t = E.t$
$E_2.f = E.f$
$E.code := E_1.code \| gen(E_1.t\ 'i') \| E_2.code.$

$E \to E_1$ relop E_2 { E.code = gen ('if' E_1.name relop.op E_2.name 'goto' E.t)|| gen (goto E.f) }.

Control Flow Constructs

$S \rightarrow$ if E then S
| if E then S else S
| while E do S
| Begin SL end | id - E

$SL \rightarrow SL ; S \mid S$

$E \rightarrow$ Arth exp with bools.

The following instruction is in the label S.follow.

Expression like $S \rightarrow iEtS$; S is done if E is true.

$S \rightarrow iEtS$

$E.t := genlabel$
$E.f := S.follow$
$S_1.follow := S.follow$
$S.code := E.code \| gen(E.t ':') \| S_1.code.$

$S \rightarrow iEtS_1eS_2$

$E.t := genlabel$
$E.f := genlabel$
$S_1.follow := S.follow$
$S_2.follow := S.follow$
$S.code := E.code \| gen(E_1.t ':') \| S_1.code \| gen('Goto', S.follow) \| gen(E.f ':') \| S_2.code.$

$S \rightarrow$ while E do S_1

$S.start := genlabel$
$E.t := genlabel$
$E.f := S.follow$
$S_1.follow := S.start.$
$S.code := gen(S.start ':') \| E.code \| gen(E.t ':') \| S_1.code \| gen('goto', S.start)$

Type Analysis And Type Checking -

- Static type checking ensures that certain type of checks like uniqueness, flow and control are performed and errors reported.
- The static type checker needs to detect and report type mismatch errors.

Type System

- The type system keep track of types of variable and of all expressions.
- Type system determine if a program is well behaved.

Execution Errors

Trapped Errors
→ Cause the execution to stop immediately
% by 0.

Untrapped Errors
→ go unnoticed and later cause arbitrary behavior
eg bound checks missing

A program fragment is safe if there are no trapped errors. Typed languages can force good behavior.

- Type system is a collection of rules for assigning type expressions.
- If a language can guarantee if it's compiler ~~can~~ accepts a program with no errors then it is called strongly typed.
- The source language defines the notion of types and the associated rules.

Types

Basic
- INT
- float
- char

Constructed
- array
- pointer
- structs.

Type Expressions

This is a way to express the type of a language construct.

1. Basic : int, char, float
2. terror : special basic type to deal with errors.

3. Null : special basic type to show no type

4. name : type defs and aliases

5. Type Constructor : arrays, products, structs, pointers, function.

5.1 Arrays: if array elements are type T and T is a type then array (I,T) is a type

5.2 Product : The cartesian product $T_1 \times T_2$

5.3 Structs : The constructor struct applied to the product of the type of field.

5.4 Pointers : Pointer(T) is a type expression if T is.

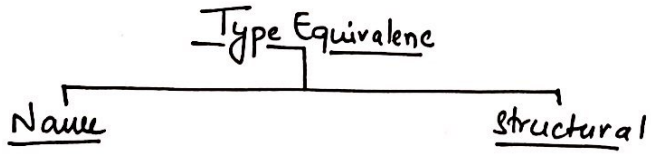
5.5 function : Function is a mapping from D to R types $D \rightarrow R$.

eg array(0...99, float) x int → float

Type expressions may be Trees or DAGs.

Basic Type Checker

- Type checker checks for type compatibility by checking if two expressions are equivalent.



Name Equivalence

Two exper are name equivalent iff they are identical. i.e Their type names match and the type names are treated as distinct.

Structural Equivalence

Two expressions are struct eq if they represent structurally equivalent types.

```
Set eq (a, b) {
    if a, b are same basic type
        ← true;
    elif a = array(a1, a2) and b = array(b1, b2)
        return seteq(a1, b1) and seteq(a2, b2)
    elif a = a1 x a2 and b = b1 x b2
        return seteq(a1, b2) and set-eq(b1, a2)
    elif a = pointer(a1) and b = Pointer(b1)
        return set-eq(b1, a1)
    elif a = a1 → a2 and b = b1 → b2
        return seteq(a1, b1) and seteq(a2, b2)
    else return no
}
```

For grammar 9 :

$P \rightarrow D ; S$

$D \rightarrow D ; D \mid TL$

$T \rightarrow int \mid Float \mid char$

$L \rightarrow L[num] \mid *L \mid id$

$S \rightarrow id := E \mid if(E) S ; \mid while(E) S \mid S_1 ; S_2$

$E \rightarrow literal \mid num \mid id \mid E \% E \mid E\ binop\ E \mid *E \mid E[E] \mid E(E).$

1) Type expression name record.

$D \rightarrow TL \quad \{ L.T = T.T_y \}$

$T \rightarrow int \quad \{ T.T_y = int \}$

$T \rightarrow float \quad \{ T.T_y = Flt \}$

$T \rightarrow char \quad \{ T.T_y = char \}$

$L \rightarrow id \quad \{ addtype(id.entry, L.type) \}$ // associate type in ST //

$L \rightarrow *L_1 \quad \{ L.T_y := pointer(L.typ) \}$ //

$L \rightarrow L[num] \quad \{ L.T_y = array(0 \ldots NUM, L.T_y) \}$ //

Inherited

2) Expressions

$E \rightarrow literal \quad \{ E.T = char \}$

$E \rightarrow num \quad \{ E.T = int \}$

$E \rightarrow id \quad \{ E.T = check(id.entry) \}$

$E \rightarrow E_1\ binop\ E_2 \quad \{$ i if i else f if f/f else error $\}$

$E \rightarrow E_1 \% E_2 \quad \{$ i if i, i else t_err $\}$

$E \rightarrow *E_1 \quad \{$ t if $E.T_y = pointer(t) \}$

$E \rightarrow E_1[E_2] \quad \{$ t if $E_2.T_y = INT$ and $E_1.T_y = array(int, t) \}$

$E \rightarrow E_1(E_2) \quad \{$ t if $E_2.type = S$ and $E_1.T = S \rightarrow t \}$

3) Statement checking

$S \rightarrow id := E \quad \{ S.T_y := void$ if $id.T_y = E.T_y$ else t_err $\}$

$S \rightarrow if(E) S_1 \quad \{ S.T_y := S_1.T_y$ if $E.T_y = int$ else terr $\}$

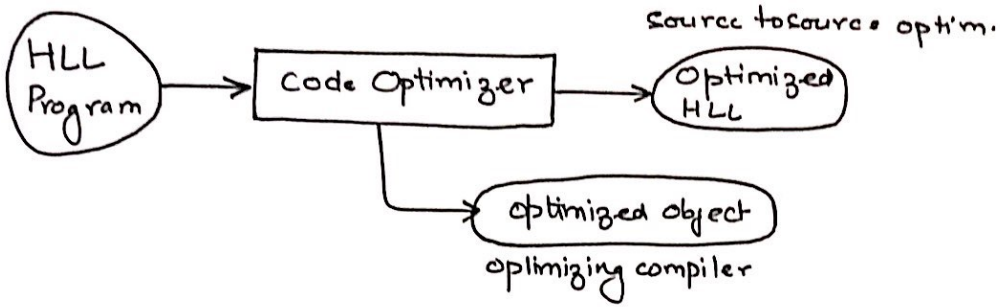
$S \rightarrow while(E) S_1 \quad \{ S.T_y = S_1.T_y$ if $E.T_y = int$ else terr $\}$

$S \rightarrow S_1 ; S_2 \quad \{ S.T_y = void$ if $S_1.T_y = void$ and $S_2.T_y = void$ else terr $\}$.

Code Optimization

- Technique to obtain more efficient object code than what would be obtained by the code generator.

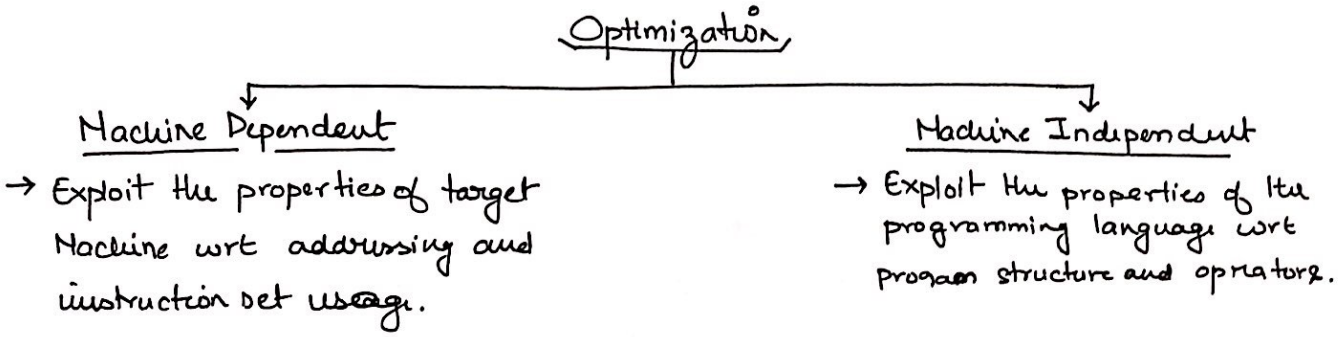
Goals: Must ensure the output is semantically same as the input
: Must not alter any algorithms used in the source program.



- Source-to-source produces code that is portable and can be executed on any machine that has that compiler.
- Optimizing compilers have the advantage of being able to do array reference and register optimization.

Levels of Optimization

1. At source level by the coder by changing loops or transforming loops.
2. At the ICG level by the compiler by improving loops and address calculations.
3. At the target code level by keeping heavily used values in registers.



Machine Dependent Optimization

- Allocation of scarce resources to achieve high efficiency in execution.
- Use of intermediate instructions, incrementation, indexing, indirection & vector ops.
- Intermixing data with instructions.

Machine Independent Optimization

- Based on the mathematical properties of a sequence of source statements.
- optimization essentially means analysing the overall purpose of statement to find an equivalent sequence.

Optimizing Transform Properties

1. Preserve the meaning of the program.
2. Speed up programs by a measurable amount.
3. reduce size of program.
4. The time and resource spent optimizing must be worthwhile.

1. Common Subexpression Elimination

- Expressions which have already been computed need not be evaluated again if the value in the variables hasn't changed.

Eg

```
a := b**c;
  :
d := b**c +x-y;
```

We can eliminate the 2nd b**c from the exp if No intervening statement has changed it's value.

```
t1 := b**c;
a := t1;
  :
d = t1+x-y;
```

Typically; two expressions are common if :

1. They are lexically equivalent.
2. They evaluate to identical values. i.e No assignment for any operands exists between the two expression.
3. The value of operands must Not change even due to a procedure calls.

```
a := b**c;
b := 10;
  :
d = b**c+x-y;
```

} Not reducable {

```
a := b**c;
x := b
  :
d = x**c+x-y
```

2. Compile Time Evaluation

- Some run time actions can be shifted to compile time to reduce re running of code sections.
- Expressions with constant operands can be evaluated at compile time and replaced by a single value this is called folding.

```
A = 2*(22.0/7.0)*r;
```

can be optimized to

```
A := (6.28...)*r;
```

- Folding is done by most compilers during array addresses.

```
x := 12.4;
:
y := x/2.3;
```

- Between x and y no operation changes x ∴ we can instead have

```
y := 12.4/2.3
```

done during compile time, this is called constant propagation.

- Constant propagation means using the const value assigned rather than variable reference. This enhances the scope of folding.
- This is only possible when the variable is assigned the same value along all paths reaching the point of use.
- Compile Time Evaluation can also eliminate identity operations like 1*a, 0+b.

3. Variable Propagation

- Replacing the use of one variable by another if it has been assigned the same rhs variable along all paths reaching its use and none of the variable is modified after this use.

```
c = a**b
x = a
:
d = x**b+x-y  ⇒  d = a**b+x-y  ⇒ folding possible now.
```

4. Dead Code Elimination

- If the value assigned to a variable is not used beyond a point in code then the variable is said to be dead at that point.
- Any assignment made to a dead variable are waste and can be eliminated.
- A block of code is said to be dead if it computes values which are dead.
- Assignment to a var results in dead code if another assignment is made befor it is used.
- Variable propagation tells assigment operations making them dead.

```
c = a**b
x = a   ↷ propagate
 ⋮
d = x**b+x-y
```

⇓

```
c = a**b  ← common
x = a     ← dead code
 ⋮
d = a**b+x-y
```

⟹

```
t1 = a**b
c = t1
 ⋮
d = t1+x-y
```

5. Code Movement

- Reduces evaluation frequency of expressions. This is done by moving the evaluation to other parts of the program.

Eg

```
if a<10
    b = x²-y²
else
    b=5
a = x²-y²*10;
```

- x^2-y^2 is evaluated twice which can be optimized to.

```
if a<10
    t = x²-y²   ← common + Repeat
    b = t
else
    t = x²-y²
    b = 5;
a = t*10;
```

⟹

```
t = x²-y²
if a <10
    b = t
else
    b=5
a = t*10;
```

Reduced to partial Redundency ⟶ moved code.

This is called Code Hoisting i.e moving code at it occurs befor in the og code.

This also reduces space usage.

This is significatly useful when the code is hoisted out of loops when they have invariant code in them.

6. Strength Reduction

This is like frequency reduction but instead of reducing the number i.e fuquency of execqation we reduce the computational complexity of the operations.

By strength Reduction we mean replacing high strength op with low ones.

Eg replacing * with + when possible.

```
i=1                                  i=1
while (i<10):      can be transformed to:    t=4
                                     while (i<10):
    Y = i*4                              y=t;
                                         t=t+4;
```

This is normally done for induction variable + light st operators.

° Induction variable : a var of the type $i := i \pm$ const ".

This reduces over all running time.

It is unfavorable to do strength Reduction to floats.

7. Loop Test Replacement

- we can replace a loop termination test wrt one variable with another equivalent variable.
- following S.R the loop induction variable is useless ∴ replaced by tmp.

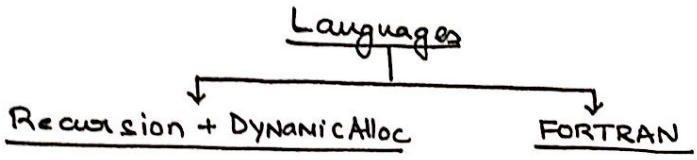
```
i=1                                  t=4
t=4                                  loop  y=t;
loop : y=t          ⟶                      t=t+4
       i=i+1                               if t<40 goto loop
       t=t+4
       if i<10 goto loop
```

Memory Allocation

- To bind data items to memory locations. i.e To collect data names & info and determine size of instructions.
- To allocate data we must know:
  - lenght of data item.
  - Type of data item.
  - dimension if any
  - Scope if any.

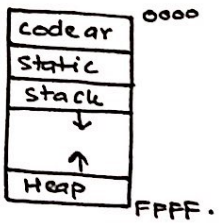
Features Affecting Malloc

1) Recursion : need to maintain various states.
2) Param Passing method.
3) Local Names :
4) Non Local Names.
5) Dynamic data structure.



Subdivision of run time Storages

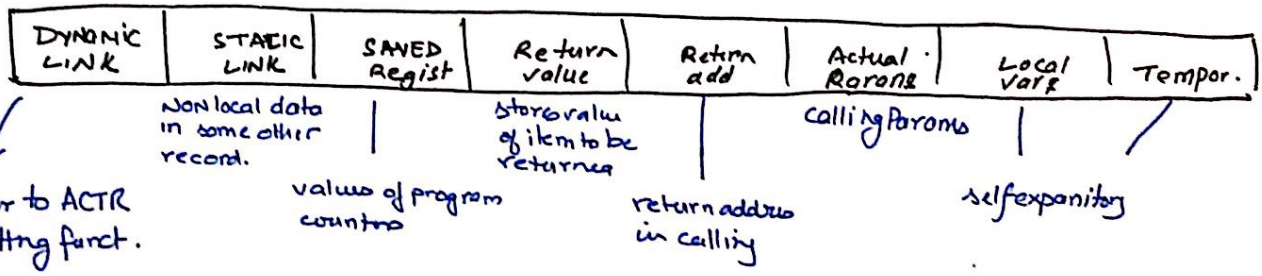
1. code area : holds the generated target code
2. Static area : holds the absolute addresses of determined items
3. Stack area : data objects or procedures.
4. Heap area : data items created at run time



Allocation Strategies

- Activation Record : a conceptual aggregate of data that contains all the info req for a single execution of procedure.
- The record is kept on the stack and new one is pushed for a call and popped for return

Contents



Static Allocation

- Vars remain permanently allocated irrespective of accessibility.
- Names are bound to storage as the program is compiled.
- Local names are retained across various activation records.

- The amount of storage is determined by
  1) elementary datatype → integral no of bytes.
  2) contigious blocks → arrays / strings.
- It is possible to fill address at which code can find data it needs.

Limitations

- Memory is bound for subprograms which may be mutually exclusive.
- Memory is bound for subprogram which are never called.
- Cannot recurse.
- No run time alloc.
- All data item sizes must be known.

Stack Allocation

- Storage is organized as a stack. An ACTR is pushed when a new procedure is run.
- The storage free on popping is available of reuse.
- Mutually exclusive blocks no longer bind space.
- Activation records contain storage for the local vars of that call. which are lost when it is popped.

Advantages:

1. supports recursion
2. Supports runtime malloc
3. allows for array declarations like A(I,J).

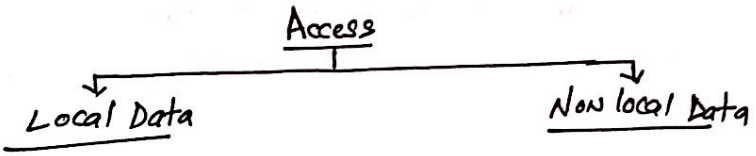
Drawbacks:

Memory addressing is via pointers and index regs which slows stuff down

Heap Allocation

- Allows retention of local values when the ACTR is popped. The programmar explicit control over the alloc / dealloc.
- Alloc is done from a large free area and access is through pointers.
- The space after a dealloc is not reused rather garbage collection is done.
- Heap alloc is useful for cases which are not LIFO or which are asynch.
- Heap alloc is more general but not as efficient as stack.

Memory Allocation In Block Structure Language :



Local Data

- Local data may be referred within the record using an offset wrt the base of the record.
- There is a pointer to the start of the record : Base Pointer.
- The BP always it at the Base of Record in execution.
- When a new procedure is entered, space is allocated and BP is assigned current value of top. & then Top is incremented.

New Procedure : *DL = BP
BP = Top
Top = Top + |actr|

*DL → DynamicLink.

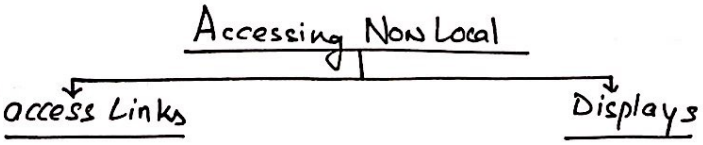
Procedure return : Top = BP
BP = *DL.

Non Local Data

- value of non local data cannot accessible with just offsets.
- " A Block is a statement containing its own data declarations enclosed b/w Delimiters.

The names that can be refered to at any point in Blocked language using static scoping :

1. Locally declared
2. Names of enclosing procedures
3. Name declared more closely to the block overide the other.



- To access non locals we use the static link pointer which points to the activation record of the most closely enclosing block.
- The non local vars are accesses through these pointers starting from base of ACTR

To setup static links:

1. Level of main is 0.
2. if level of enclosing is l then level of enclosed is $l+1$.

- Static link is traversed $l - l_2 + 1$ times starting from ACTR of calling.
- Static link of called is ~~to~~ set to base of new actR.

- Using static links is costly if large static block nesting exists.

To improve efficiency we use an array (called the display) of pointers to the activation records.

The array is such that the act of x at level i is at display[i].

- To setup display[i] to point to the base of current act at level i. The old display[i] is saved within act of the procedure.
- current value of display is saved in the new record & display[i] now points to it.
- Display is reset to old value at the end of execution.

- Display is updated whenever new activation occurs & whenever control returns from a new activation.
- Using Display we can reference all non locals at same cost.

Symbol Table Organization

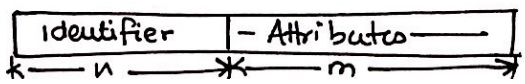
- It is a table containing all names used in the source program and their associated values.
- The table is looked up every time a name is encounted in the source program.
- It has two phases, building where symbols and associated values are inserted and a lookup/ reference phase where values/ symbols are search.
- Records are deleted when the symbol goes out of scope.

Desired Capabilities

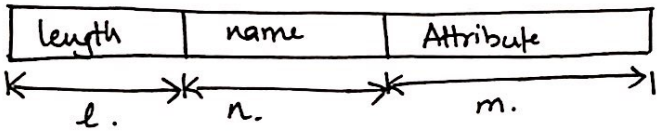
1. provide entry and retrival of symbol.
2. provide a mechasism for search.
3. Be able to delete elements which are no longer needed.
4. Provide Fast lookup by keeping the table on memory.
5. It should be able to grow as per the needs.
6. It must be able to handle duplicate entries. i.e support scopes.
7. Must be designed in a manner that some 3rd party can manage the table data structures and modules.

Elementary Symbol Table Org.

- Entries are stored as consequtive seq of words in memory. The nature of info depends on the name, this can be made uniform by moving some info about the name out.
- When the maximum allowed size of a id is know a fixed length entry can be used, where n unit are reserved for the ID name and m for the attributes.

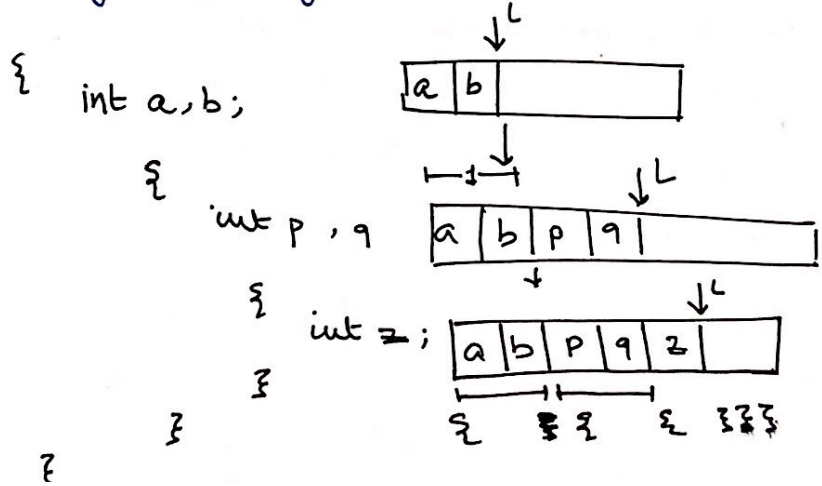


- Using a fixed length strategy has following advantages :
  1. All entries are alike and can be swaped.
  2. Table length can be fixed / estimated
  3. Position of an entry can easily be calculated.
- when the maximum size of identifier is not know, a variable length entry is used.
- Units of the block are kept for the length of the ID.
- Use of variable length entries restricts the table to linear lookup strategies.



1. Linear Search Organization

- Linear list of records with each record corresponding to only one symbol.
- The symbols are stored in order of arrival.
- The search if from last to first entry.
- Scope rules are inherently handled due to the entry scheme, since levels are closer to the end.
- For very large programs, the search can take a very long time



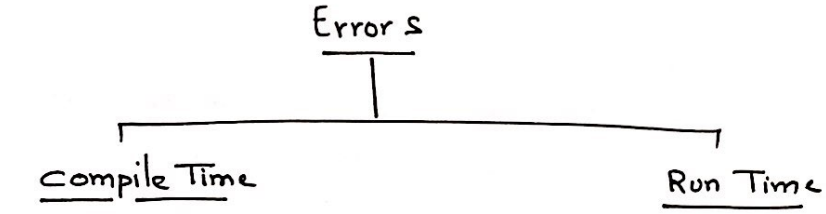
2. Binary Search Organization

- Search time is reduced by storing the entries sorted in a BST.
- The records are stored in groups of starting symbols.
- The lookup is reduced to O(log(n)) time
- The insert and delete are fairly complex tasks which requires moving of the table.
- This is best for static capacity tables.

Error Handler

Goals of the Error Handler

- To detect errors during various phases of the compiler
- To report the errors clearly providing sufficient info on the cause and Nature
- To recover from errors in order to continue processing
- To correct common errors [difficult]
- To Not slow down the compiler.



- Lexical
- Syntactic
- Semantic

Lexical Errors

- Difficult to locate due to nature of the LA.
- Possible sources :
  - Extra char.
  - Incorrect char.
  - Missing char.
  - Transposed chars.
  - Max ID_len exceeded.
- One method is to delete successive chars from remaining input till something matches — Panic mode.
- Pass the char to the parser and let it decide.
- Error Reporting must be:
  - clear and unambigious
  - avoiding duplications.
  - highlight the source of error
  - be as close as possible to the error.

- Must be interms of soura program entities
- Error caused by recovery must be different from a genuine error.
- If a correction is made, it should be notified to the user.

- **Syntax Errors** :
  - Most common type of error.
  - Identified when tokens disobey the grammatical rules of the language.
  - Table driven parsers have explicit entries for errors.
  - Tables have a viable prefix rule i.e detect error as soon as there's a mismatch.
  - Reporting :
    - Info on the token
    - 1. Examine entries in each state the errors
      2. Break routines e_i for state i
      3. Set of valid tokes for i.
      4. e_i first tells the line, the token & then the expected token.
    - e_i can be same for 2 states.

- Recovery
  - suspend normal parsing.
  - change the error configuration.
  - resume normal parsing.

1. Panic Mode : Attain a state from which the parsing can safely resume.
   - Discard input symbols one-by-one until a sync token is found. The sync need not be the end of the ~~token~~ statement.
   - Syncs are normally delimiters.
   - Examines at the point of error.
   - Guarantees it'll work and not loop only
   - Skips a lot of code (which itself may have errors)

2. Phase Level Recovery : perfor local correction on remaining input.
   - delete a source symbol.
   - Insert a synthetic symbol
   - Replace source symbol with synthetic symbol.
- present new string which'll bypass the error.
- Can go into ∞ loops if the incorrect symbol is used
- Connot be used when cases where ~~of~~ error occurs much before detection.

3. Error Productions : Encorporate Production for most common type of errors.
when an error occurs, these productions are used to create the error promt & parsing continues.

4. Global Correction : Minimizes the level of modifications at the global level when multiple possible modif exist. i.e min dist recovery.
Normally Parsers go for simpler and faster logics rather than minimizing cost.

- **Top-Down Parsers**
  - 2 cases, terminal at top != next symbol or NT at Top and input symbol have a blank entry.
  - Use sync set. this has symbols from First and Follow.
  - do panic mode or parse the stack and create list of syncs till the point of error & then unstack till ~~top~~ ~~==~~ some symbol is at top.
- **Bottom-Up Parsers**
  - Use inserting synthetic symbols.
  - For panic mode, scan down the stack till until s with goto A. discard input till symbol which matches A is found.
  - OR fill blanks with routines.

Semantic Errors

- Due to violation of rules of accessibility i.e type error, arr index.
- Can be local / Global in scope.
- Not always possible to detect.
- Can be immediate or delayed.

Immediate : can be detected while processing the erroreous line.

$A = B++C.$

- Not that easy to recover / correct.

Delayed : Not detected when the line is processed but when the line takes effect.

Eg: Missig labels, Invalid Trasfer C outside of loop to inside)
The error are detected when the destination is encountered.
Intersecting DO's / invalid nesting.

• Runtime Errors

- Due to violation of semantics of machine computation.
  eg underflows / overflows.
- Detected by machine H/W. this passed into to run time control routine & then it generates the error. Some times recovery is done using assumptions.
- undefined values → Flag bits
- Dimension overrun → Bound checks.
- I/O device → IOCS.