Explicit type conversion (int)x
≈ cast

Implicit type conversion char y = 'a'
// convert y to a value of 97
int x = 10;
float z = x + 1.0 ; //x is implicitly convert to float

(3.14159 + 5) is capability type

(int)95.7 type conversion

in OCamel, unification ... → inference

in C, declare an array w structelem → Ortho

C++, use of virtual func → Dyn, type

Best describe OCaml, Dyn... Structural

Strongly name S/ equi

[|1; 2; 3|] → an array w 3 int ele 1, 2 & 3

In OCaml, !y is the value (content) accessed by the ref variable y

Option types programmer to specify a value valid/invalid → (T)

- C++ denotational and abstraction-based → (T)
- Programing language highly ortho .. less ortho → (F)
- Real number type, scalar type (F)
- Large number coercionss between decrease ease of use and understandability → (T)
- Functional language make extensive use of side effects → (F)
- Functional language manipulated Same mechanism manipulate data (T)
- lambda calculus C++ Java (T)
- Most functional langue do NOT support (F)
- Garbage collection essential feature → (T)

Why does OCaml provide separate
→ To prevent potential errors arising from coercions (implicit type conversions)
- To help with type inference

Briefly explain physical >< structural
- Physical equality implies eq that the values being compared are the Same object intance in memory

- Structural equ. implies that each element of the values being compared recursively has the same value, even if they are not the same object instances in memory

* Name >< Structural:
- Name type equivalence is simila to physical value equivalence: for two variables to have equivalence types under name type equivalence, the declarations of those variables must reference the same type.
- Structural type equivalence name is similar to structural value equivalence: two variables are considered to have equivalent types if each element of the types in there declarations are equivalent, when compared recursively.

Ada Celcius ↔ Fahrenheit:
The use of derived types can catch certain common errors such as mixing values of different physical units, as

int + int = int ; int + double = double

:= operator to assign to references
! deferences to get out the contents

- LL parser top-down → (T) LR comes LL (T)
- Stage reads a stream of charakters a stream of token → Scanner
- Stage determines the meaning of a program, errors and declaration before use → Sem
- Grammar ambigous, type grammar → None of LL, LR, LALR, SLR
- Grammar recursive descent → LL
- If a grammar left-recursive → LR, LALR, SLR
- Module 2 - Ada → "end marker"
- Front end → Parser
- High level intermediate form → Abstract
- On modern machines, assembly lang > compiler (F)
- BNF enables Algol-60 → (T)
- Operator precedence "higher" → (F)
- Scanner peek (T)
- Epsilon LR > LL (F)
- Top down A → α T∈First(α) → (T)
- S-attributed synthesized, inherited → (F)
- recursive desent incorp semantic routine (F)
- frontend GCC RTL (F)
- machine dependent number of reg (F)
- which is NOT regular expression generate tokens of a programing language → recursion
- DO Loop in FORTRAN → Variable need not Spaces ...
- LL top-down → (T)

- array allocation → locally static / runtime dyn
- Smart pointers $\xrightarrow{C++}$ reference counts
- int my_arr [10][10] → 100 integers (T)
- multidimensional array allocated (F)

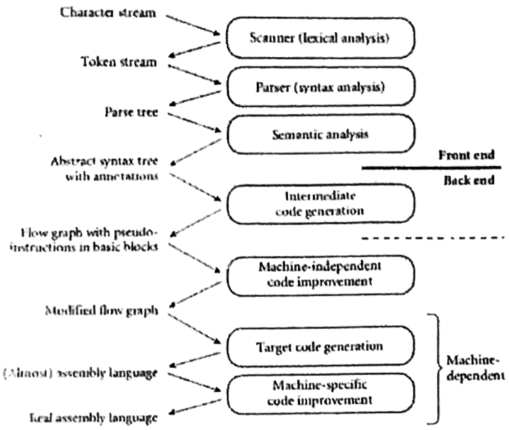
1T 2T 3F 4F 5T 6F 7T 8T 9F 10 T

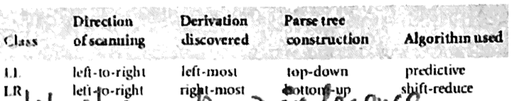
1a 2c 3c 4b 5a 6bf 7a 8c

- Stage reads stream of tokens and produces a parse tree? Parser
- Declaration before use? Semantic
- ambiguous grammar → None of LL, LR, LALR, SLR
- if a grammar can be implemented recursive descent → LL
- contains right-recursive prod → LL, → LL, LR, LALR, SLR
- C++ dangling else → closest unmatched
- "back end" Machine dep/indep code gen / code improve stage
- parser CFG → PDA
- Epsilon → (T)
- modern compilers > assem (T)
- T ∈ FIRST(α) → (T)
- FORTRAN → (F)
- L-Attribute → (T)
- Operator precedence "lower" → (T)
- Scanner "peak" → (T)
- functional & logic language "read loop" (T)
- recursive descent semantic attributes (F)
- Bottom up parsers push "shift" → (T)

- In Ada, the programmer may use a pragma, drawback
→ Because the resulting packed record structure layout in memory may result in Some fields being non-aligned to word boundaries, access to those fields may require multi-instruction sequences, thus reducing performance.

Exam
- Stage traverses AST, low high level → code generation
- Stage reads stream of token → Parser
- Bottom up parser → LR, LALR, SLR
- right recursive → LL, LR, LALR, SLR
- ambigous → LR, LALR, SLR
- Back end → (c) (d) machine dep
- 3.14 + 5 → compa low level: Assem + RTL
- dangling else c++ → closest
- reinterprete_cast <int> (95.7)
→ type conversion

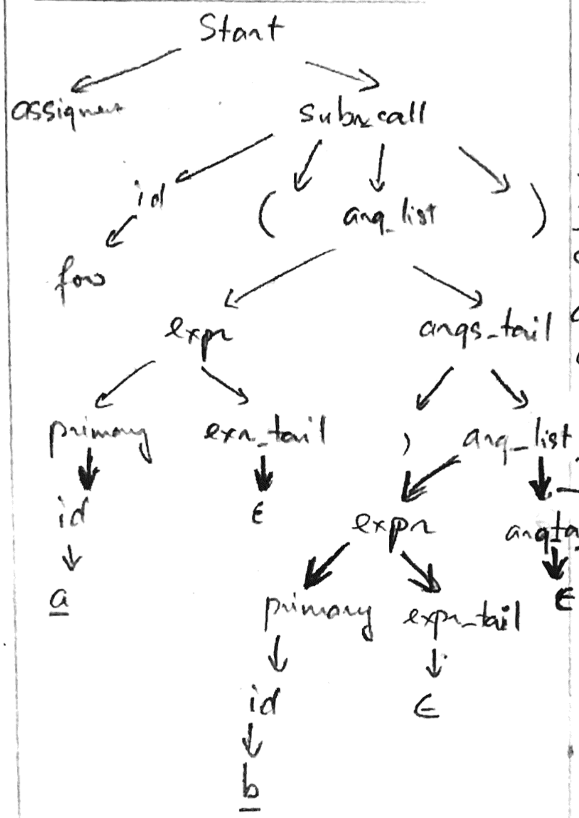

Character stream
Scanner (lexical analysis)
Token stream
Parser (syntax analysis)
Parse tree
Semantic analysis
Abstract syntax tree with annotations
Front end
Back end
Intermediate code generation
Flow graph with pseudo-instructions in basic blocks
Machine-independent code improvement
Modified flow graph
Target code generation
(Almost) assembly language
Machine-specific code improvement
Real assembly language
Machine-dependent



Class	Direction of scanning	Derivation discovered	Parse tree construction	Algorithm used
LL	left-to-right	left-most	top-down	predictive
LR	left-to-right	right-most	bottom-up	shift-reduce

- lot pluss a b → inference
- OCaml, recordin pair → Ortho
- C: int min(int a, int b){ } → Dyn.
- template <typename T> → Poly
- Best describe C++ → Static struc / Strong_Struc
- (1,2,3) → tuple
- f [1;2;3] → 6
- double (*a[n])() → function
- mark sweep → locks keys
- Kleene → T
- Lower (closer to the leaf nodes) → T
- LR(n) LL(n) by n → F
- Epsilon LL > LR → T
- L-attribu: synthetic → T
- Bottom up parser semantic → F
- Most frontend GCC AST → T
- machine dependent unlimited → T
- OCaml val f : 'a → T
- C denotational abstract → T
- high orthogonal flexible → F
- real number scalar, not dis → T
- C large number coercion → T
- "Pure" do not allow side → T
- Logic Prolog
- Lambda expression C++ → T
- Ada widely use → F
- Most func lang. allocate / dea → T
- char *my-array[10] ⇒ T
- a multidimen, complicated address → F

- S and T
  A and B) structural eq.
- A, B, C, D because lex.



- Code generation involves using a model compiler to generate a low_level representation of the model using existing programming language and platform
- Model intepretation relies on the existence of a virtual machine able to directly read and run the model.

S occupies N to N+2 (N: address, even) $\boxed{240}$
c: N+2 to N+3
t N+4 to N+6
d N+6 to N+7
r: N+8 to N+16
i: N+16 to N+20

So we need at least 20 bytes for this structure. It is not dividable by 8 ⇒ choose 24

Hence: 24 × 10 = 240 bytes for the array

Code generation includes utilizing a model compiler to produce a low level act of model utilizing the programming languagage with its platform whereas model intepretation depends on the presence of a virtual machine that can read directly.

In case of interpretation, the actua source code is often generally changed to few intermediate codes and further process with an interpreter which translates the machine readable code into particul machine code.