An Overview of Compilation

Uday Khedker

(www.cse.iitb.ac.in/~uday)

Department of Computer Science and Engineering,

Indian Institute of Technology, Bombay

January 2025

Outline

Outline

- Introduction
- Compilation phases
- Compilation models
- Modern challenges
- Incremental construction of compilers
- Course plan
- Expectation management

Introduction to Compilation

Binding

Nothing is known except the problem

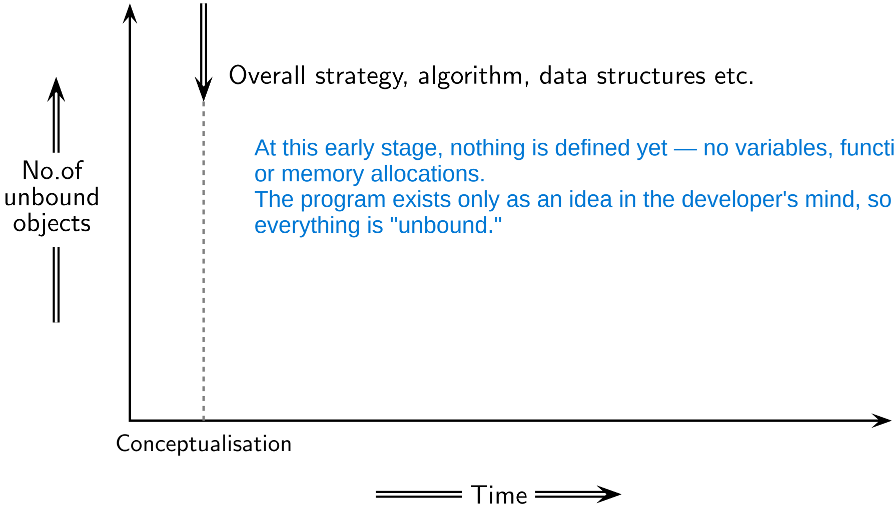
No.of unbound objects

Binding in the Compilation Process refers to the association of program elements (like variables, functions, or types) with their attributes (e.g., memory locations, data types, or values) during different stages of program execution or compilation.
Types of Binding:
1. Static Binding (Early Binding)
2. Dynamic Binding (Late Binding)

This diagram explains binding times in the compilation and execution process. The vertical axis represents the number of unbound objects, while the horizontal axis shows the stages in the lifecycle of a program. Let's break it down step-by-step:

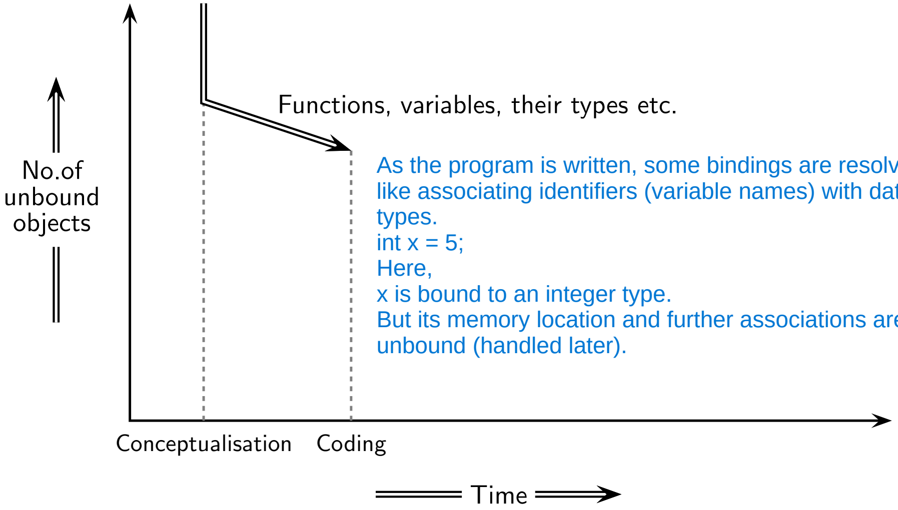
Time

Binding



At this early stage, nothing is defined yet — no variables, functions, or memory allocations.
The program exists only as an idea in the developer's mind, so everything is "unbound."

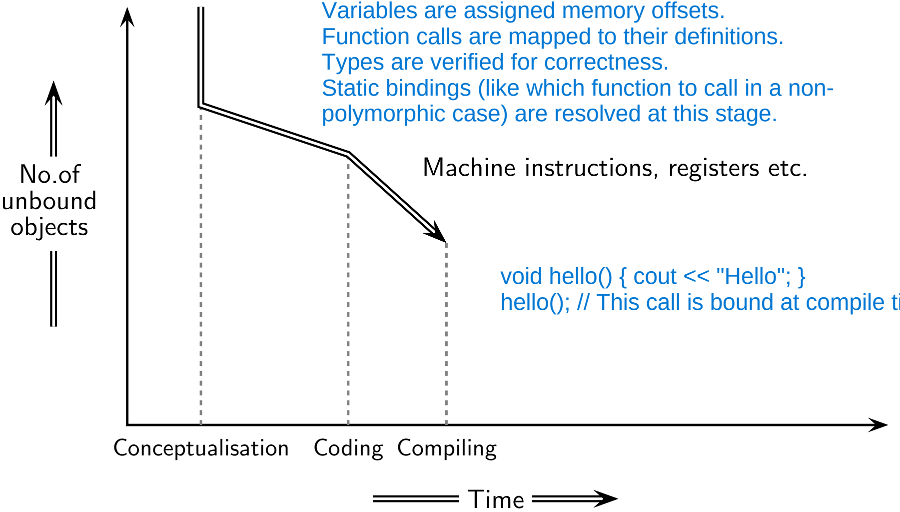
Binding



As the program is written, some bindings are resolved, like associating identifiers (variable names) with data types.
int x = 5;
Here,
x is bound to an integer type.
But its memory location and further associations are still unbound (handled later).

Binding

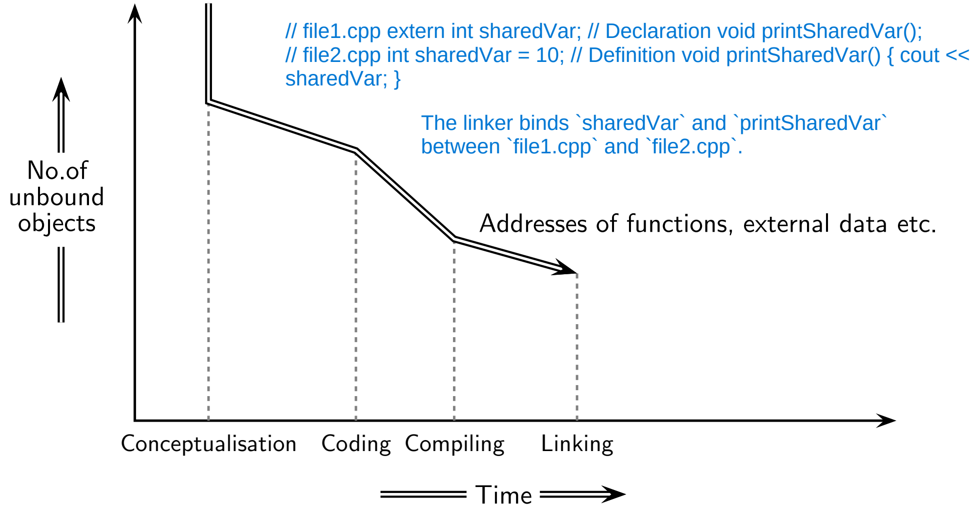
During compilation, many bindings are resolved:
Variables are assigned memory offsets.
Function calls are mapped to their definitions.
Types are verified for correctness.
Static bindings (like which function to call in a non-polymorphic case) are resolved at this stage.



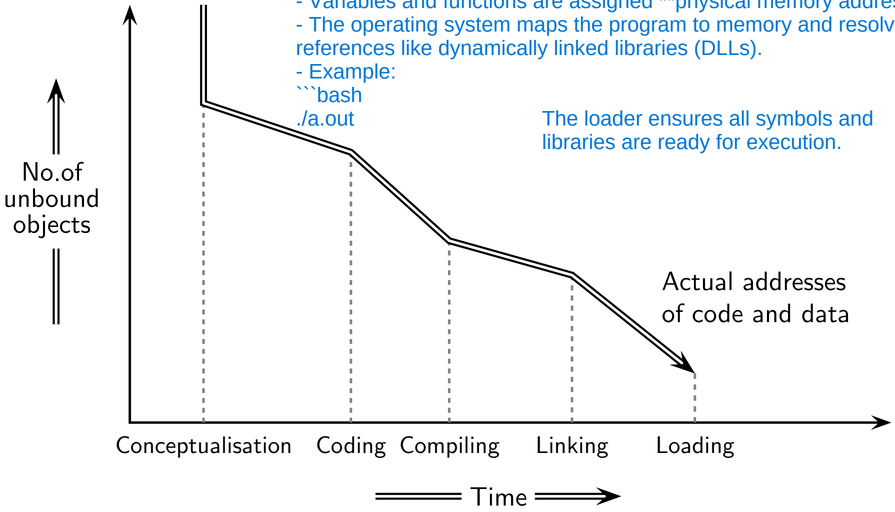

```
void hello() { cout << "Hello"; }
hello(); // This call is bound at compile time.
```

Binding

At this stage, external references (e.g., functions or variables defined in separate files) are resolved. The linker binds these references to their actual memory locations or addresses in the final executable.



Binding

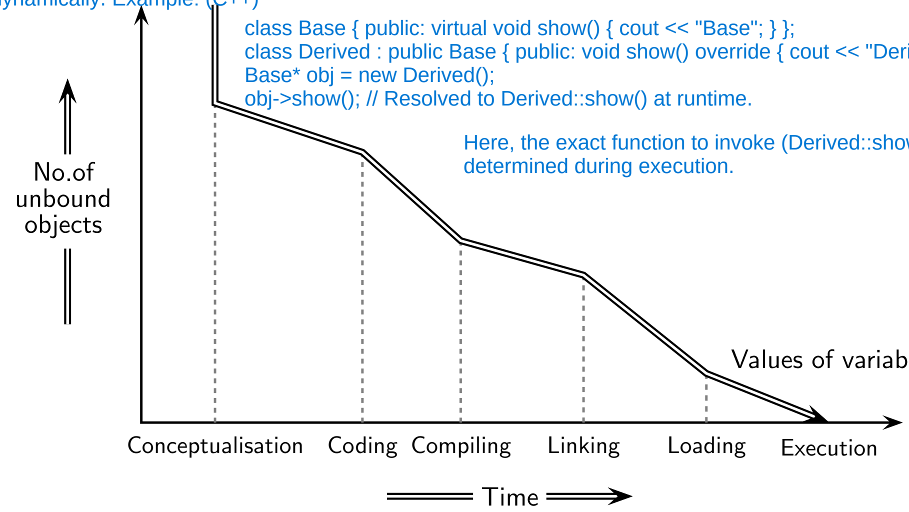
- When the program is loaded into memory for execution, some bindings are finalized:
- Variables and functions are assigned **physical memory addresses**.
- The operating system maps the program to memory and resolves references like dynamically linked libraries (DLLs).
- Example:
```bash
./a.out

The loader ensures all symbols and libraries are ready for execution.


```

Binding

During execution, some bindings (like dynamic binding) are resolved:
Decisions made by the program (e.g., which function to call in a polymorphic case) are handled dynamically. Example: (C++)

```
class Base { public: virtual void show() { cout << "Base"; } };
class Derived : public Base { public: void show() override { cout << "Derived";}};
Base* obj = new Derived();
obj->show(); // Resolved to Derived::show() at runtime.
```

Here, the exact function to invoke (Derived::show) is determined during execution.



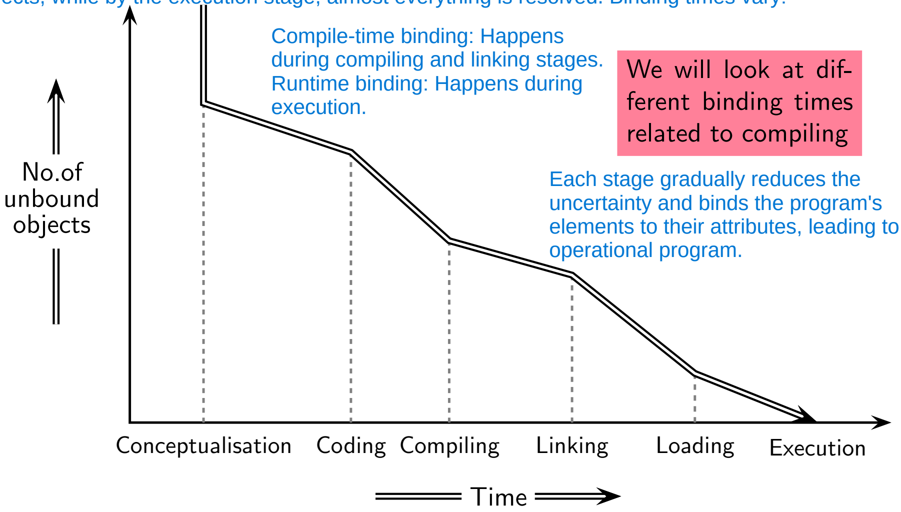
Binding

The diagram illustrates how the number of unbound objects decreases as the program progresses through stages. Early stages (conceptualization and coding) have the highest number of unbound objects, while by the execution stage, almost everything is resolved. Binding times vary:

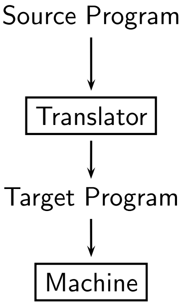
Compile-time binding: Happens during compiling and linking stages.
Runtime binding: Happens during execution.

We will look at different binding times related to compiling

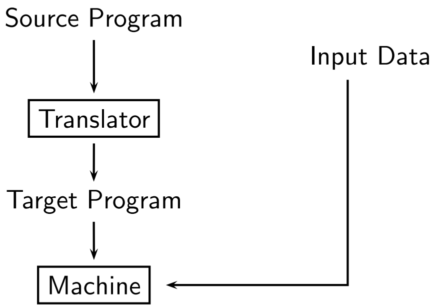
Each stage gradually reduces the uncertainty and binds the program's elements to their attributes, leading to a fully operational program.



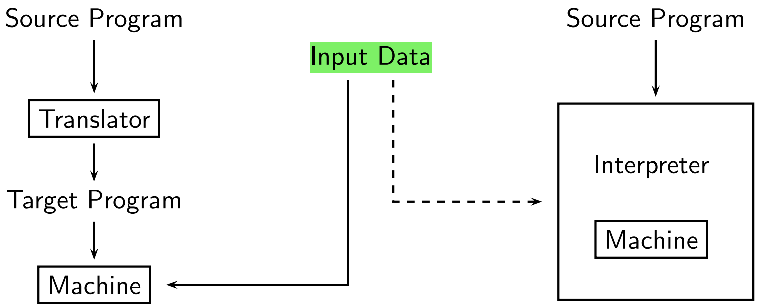
Implementation Mechanisms



Implementation Mechanisms



Implementation Mechanisms



Comparing the Implementation Mechanisms

Translation	=	Analysis + Synthesis
Interpretation	=	Analysis + Execution

Comparing the Implementation Mechanisms

$$\begin{aligned}\text{Translation} &= \text{Analysis} + \text{Synthesis}\\ \text{Interpretation} &= \text{Analysis} + \text{Execution}\end{aligned}$$

The input program is not required during translation (compilation) because a compiler converts the entire program into machine code (executable) in advance. Once compiled, the program can run independently without the source code.
In contrast, input is required during interpretation because an interpreter processes the program line-by-line or command-by-command while it runs. The interpreter reads and executes the source code directly.

Implementation mechanism	Input	Output	Separate execution	Input for the input program
Translation	Program	Equivalent program	Required	Not required
Interpretation	Program	The result of the Program	Not required	Required

The compiler converts input.c into input.exe. The executable can run without the source code.
The Python interpreter requires input.py at runtime to execute each line of code and take input dynamically.

Seeing the Difference Between Compilation and Interpretation

```
$ ./lp --help
Usage: lp [OPTION...]
  -c                    Compile the input into three address code and
                        print it
  -i                    Interpret the input and print result
  -?, --help            Give this help list
      --usage           Give a short usage message

$ ./lp -i
a = 10 + 20 * 30;
> a = 610

$ ./lp -c
a = 10 + 20 * 30;
The three address code generated for the input is
t0 = 20 * 30
t1 = 10 + t0
a = t1
```

Implementation Mechanisms as “Bridges”

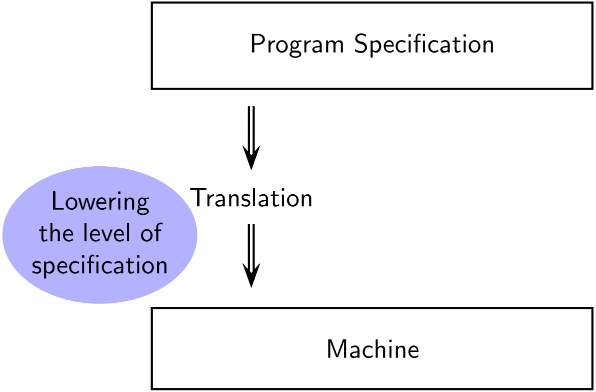
- “Gap” between the “levels” of program specification and execution

Program Specification

Machine

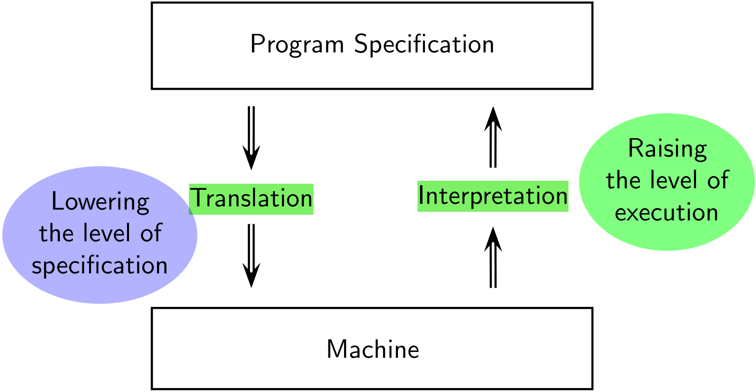
Implementation Mechanisms as “Bridges”

- “Gap” between the “levels” of program specification and execution



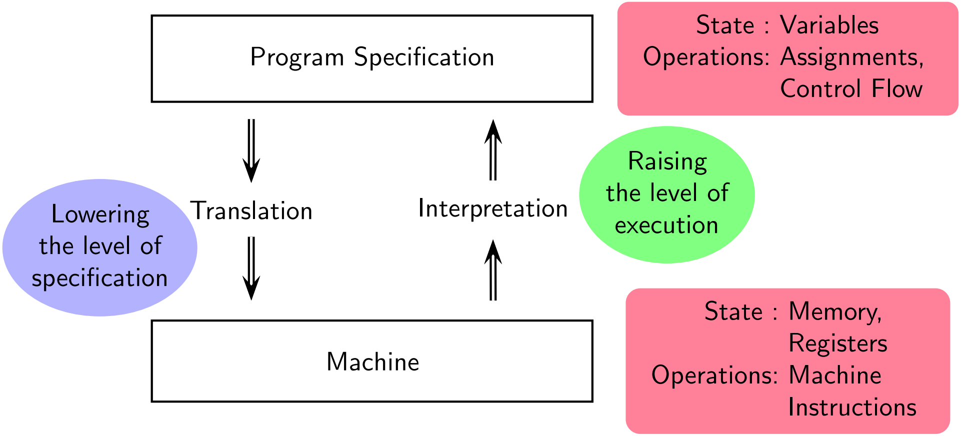
Implementation Mechanisms as “Bridges”

- “Gap” between the “levels” of program specification and execution



Implementation Mechanisms as "Bridges"

- "Gap" between the "levels" of program specification and execution



A Source Program in C++: High Level Abstraction

```
#include <iostream>
using namespace std;

int main()
{
    int n, fact=1;

    cout << "Enter the number: ";
    cin >> n;
    for (int i=n; i > 0; i--)
        fact = fact * i;

    cout << "The factorial of " << n << " is " << fact << endl;

    return 0;
}
```

Its Target Program: Low Level Abstraction (1)

```
f3 0f 1e fa 48 83 ec 08 48 8b 05 d9 2f 00 00 48 85 c0 74 02 ff d0 48 83 c4
08 c3 ff 35 5a 2f 00 00 f2 ff 25 5b 2f 00 00 0f 1f 00 f3 0f 1e fa 68 00 00
00 00 f2 e9 e1 ff ff ff 90 f3 0f 1e fa 68 01 00 00 00 f2 e9 d1 ff ff ff 90
f3 0f 1e fa 68 02 00 00 00 f2 e9 c1 ff ff ff 90 f3 0f 1e fa 68 03 00 00 00
f2 e9 b1 ff ff ff 90 f3 0f 1e fa 68 04 00 00 00 f2 e9 a1 ff ff ff 90 f3 0f
1e fa 68 05 00 00 00 f2 e9 91 ff ff ff 90 f3 0f 1e fa 68 06 00 00 00 f2 e9
81 ff ff ff 90 f3 0f 1e fa f2 ff 25 1d 2f 00 00 0f 1f 44 00 00 f3 0f 1e fa
f2 ff 25 d5 2e 00 00 0f 1f 44 00 00 f3 0f 1e fa f2 ff 25 cd 2e 00 00 0f 1f
44 00 00 f3 0f 1e fa f2 ff 25 c5 2e 00 00 0f 1f 44 00 00 f3 0f 1e fa f2 ff
25 bd 2e 00 00 0f 1f 44 00 00 f3 0f 1e fa f2 ff 25 b5 2e 00 00 0f 1f 44 00
00 f3 0f 1e fa f2 ff 25 ad 2e 00 00 0f 1f 44 00 00 f3 0f 1e fa f2 ff 25 a5
2e 00 00 0f 1f 44 00 00 f3 0f 1e fa 31 ed 49 89 d1 5e 48 89 e2 48 83 e4 f0
50 54 4c 8d 05 86 02 00 00 48 8d 0d 0f 02 00 00 48 8d 3d c1 00 00 00 ff 15
92 2e 00 00 f4 90 48 8d 3d b9 2e 00 00 48 8d 05 b2 2e 00 00 48 39 f8 74 15
48 8b 05 6e 2e 00 00 48 85 c0 74 09 ff e0 0f 1f 80 00 00 00 00 c3 0f 1f 80
00 00 00 00 48 8d 3d 89 2e 00 00 48 8d 35 82 2e 00 00 48 29 fe 48 89 f0 48
c1 ee 3f 48 c1 f8 03 48 01 c6 48 d1 fe 74 14 48 8b 05 45 2e 00 00 48 85 c0
74 08 ff e0 66 0f 1f 44 00 00 c3 0f 1f 80 00 00 00 00 f3 0f 1e fa 80 3d ad
30 00 00 00 75 2b 55 48 83 3d f2 2d 00 00 00 48 89 e5 74 0c 48 8b 3d 26 2e
00 00 e8 b9 fe ff ff e8 64 ff ff ff c6 05 85 30 00 00 01 5d c3 0f 1f 00 c3
```

Its Target Program: Low Level Abstraction (2)

```
0f 1f 80 00 00 ff ff e8 64 ff ff ff c6 05 85 30 00 00 01 5d c3 0f 1f 00 c3
0f 1f 80 00 00 00 00 f3 0f 1e fa e9 77 ff ff ff f3 0f 1e fa 55 48 89 e5 48
83 ec 20 64 48 8b 04 25 28 00 00 00 48 89 45 f8 31 c0 c7 45 f0 01 00 00 00
48 8d 35 d3 0d 00 00 48 8d 3d 07 2e 00 00 e8 92 fe ff ff 48 8d 45 ec 48 89
c6 48 8d 3d 14 2f 00 00 e8 5f fe ff ff 8b 45 ec 89 45 f4 83 7d f4 00 7e 10
8b 45 f0 0f af 45 f4 89 45 f0 83 6d f4 01 eb ea 48 8d 35 a4 0d 00 00 48 8d
3d c5 2d 00 00 e8 50 fe ff ff 48 89 c2 8b 45 ec 89 c6 48 89 d7 e8 80 fe ff
ff 48 8d 35 93 0d 00 00 48 89 c7 e8 31 fe ff ff 48 89 c2 8b 45 f0 89 c6 48
89 d7 e8 61 fe ff ff 48 89 c2 48 8b 05 17 2d 00 00 48 89 c6 48 89 d7 e8 1c
fe ff ff b8 00 00 00 00 48 8b 4d f8 64 48 33 0c 25 28 00 00 00 74 05 e8 13
fe ff ff c9 c3 f3 0f 1e fa 55 48 89 e5 48 83 ec 10 89 7d fc 89 75 f8 83 7d
fc 01 75 32 81 7d f8 ff ff 00 00 75 29 48 8d 3d 72 2f 00 00 e8 f4 fd ff ff
48 8d 15 f5 2c 00 00 48 8d 35 5f 2f 00 00 48 8b 05 d7 2c 00 00 48 89 c7 e8
97 fd ff ff 90 c9 c3 f3 0f 1e fa 55 48 89 e5 be ff ff 00 00 bf 01 00 00 00
e8 9c ff ff ff 5d c3 66 2e 0f 1f 84 00 00 00 00 00 90 f3 0f 1e fa 41 57 4c
8d 3d 03 2a 00 00 41 56 49 89 d6 41 55 49 89 f5 41 54 41 89 fc 55 48 8d 2d
fc 29 00 00 53 4c 29 fd 48 83 ec 08 e8 7f fc ff ff 48 c1 fd 03 74 1f 31 db
0f 1f 80 00 00 00 00 4c 89 f2 4c 89 ee 44 89 e7 41 ff 14 df 48 83 c3 01 48
39 dd 75 ea 48 83 c4 08 5b 5d 41 5c 41 5d 41 5e 41 5f c3 66 66 2e 0f 1f 84
00 00 00 00 00 f3 0f 1e fa c3 f3 0f 1e fa 48 83 ec 08 48 83 c4 08 c3
```

Commands to Obtain the Low Level Abstraction

- Write the program and name the file `fact-iterative.cc`
- `g++ fact-iterative.cc` produces the executable in `a.out` file
- `strip a.out` removes names from the executable `a.out`
- `file a.out` produces the following output

  ```
  a.out:  ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
  dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
  BuildID[sha1]=0c218bf025a20bc43339dfd15cec41adc1c13946, for
  GNU/Linux 3.2.0, stripped
  ```

- `objdump -d a.out` produces the hexadecimal form along with assembly program

High and Low Level Abstractions: Our View

Input C statement

```
a = b<10?b:c+5;
```

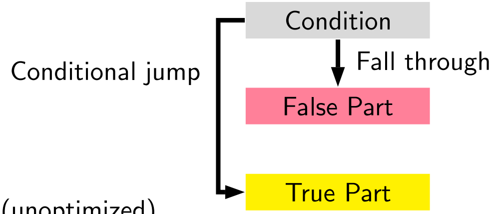
Spim assembly equivalent (unoptimized)

```
     lw     $v0, 4($fp)   ;     v0 <- b           # Is b smaller
     slti   $t1, $v0, 10  ;     t1 <- v0 < 10     # than 10?
     xori   $t2, $t1, 1   ;     t2 <- !t1
     bgtz   $t2, L0       ;     if t2 > 0 goto L0
     lw     $t3, 4($fp)   ;     t3 <- b           # YES
     b      L1            ;     goto L1
L0:  lw     $t4, 8($fp)   ;L0:  t4 <- c           # NO
     addi   $t3, $t4, 5   ;     t3 <- t4 + 5      # NO
L1:  sw     0($fp), $t3   ;L1:  a <- t3
```

High and Low Level Abstractions: Our View

Input C statement

```
a = b<10?b:c+5;
```



Spim assembly equivalent (unoptimized)

```
     lw    $v0, 4($fp)   ;     v0 <- b          # Is b smaller
     slti  $t1, $v0, 10  ;     t1 <- v0 < 10    # than 10?
     xori  $t2, $t1, 1   ;     t2 <- !t1
     bgtz  $t2, L0       ;     if t2 > 0 goto L0
     lw    $t3, 4($fp)   ;     t3 <- b          # YES
     b     L1            ;     goto L1
L0:  lw    $t4, 8($fp)   ;L0:  t4 <- c          # NO
     addi  $t3, $t4, 5   ;     t3 <- t4 + 5     # NO
L1:  sw    0($fp), $t3   ;L1:  a <- t3
```

High and Low Level Abstractions: Our View

NOT Condition

True Part

False Part

Input C statement

```
a = b<10?b:c+5;
```

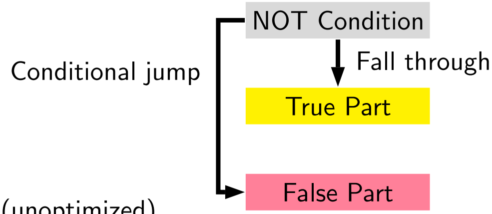
Spim assembly equivalent (unoptimized)

```
     lw    $v0, 4($fp)   ;    v0 <- b           # Is b smaller
     slti  $t1, $v0, 10  ;    t1 <- v0 < 10     # than 10?
     xori  $t2, $t1, 1   ;    t2 <- !t1
     bgtz  $t2, L0       ;    if t2 > 0 goto L0
     lw    $t3, 4($fp)   ;    t3 <- b           # YES
     b     L1            ;    goto L1
L0:  lw    $t4, 8($fp)   ;L0: t4 <- c           # NO
     addi  $t3, $t4, 5   ;    t3 <- t4 + 5      # NO
L1:  sw    0($fp), $t3   ;L1: a <- t3
```

High and Low Level Abstractions: Our View

Input C statement

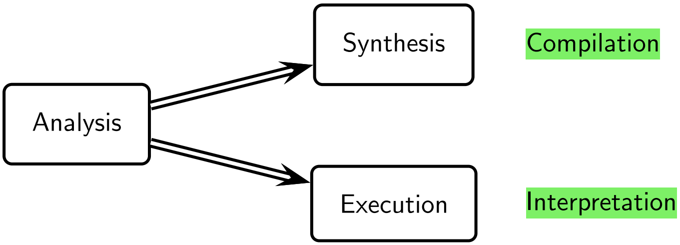
```
a = b<10?b:c+5;
```



Spim assembly equivalent (unoptimized)

```
      lw     $v0, 4($fp)   ;     v0 <- b             # Is b smaller
      slti   $t1, $v0, 10  ;     t1 <- v0 < 10       # than 10?
      xori   $t2, $t1, 1   ;     t2 <- !t1
      bgtz   $t2, L0       ;     if t2 > 0 goto L0
      lw     $t3, 4($fp)   ;     t3 <- b             # YES
      b      L1            ;     goto L1
L0:   lw     $t4, 8($fp)   ;L0:  t4 <- c             # NO
      addi   $t3, $t4, 5   ;     t3 <- t4 + 5        # NO
L1:   sw     0($fp), $t3   ;L1:  a <- t3
```

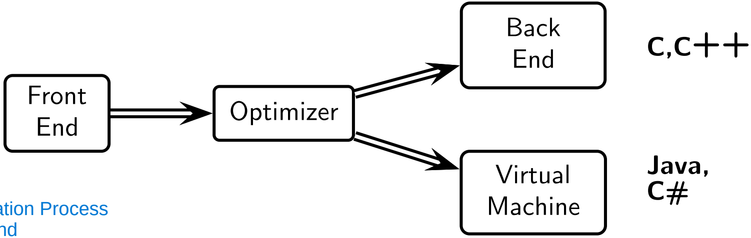
Language Implementation Models



Synthesis -> Compilation
This suggests that analysis (breaking down the code) leads to synthesis (constructing a machine-executable format), which happens through compilation (translating the entire source code into machine code before execution).
Execution -> Interpreter
This implies that analysis can also lead directly to execution via an interpreter, which translates and runs the code line (stmt. / expr.) by line without producing a separate compiled output.

Language Processor Models

phases of a compiler



Compilation Process
Front End
Analyzes the source code (lexical analysis, syntax analysis, and semantic analysis).
Converts the code into an intermediate representation (IR).
Optimizer
Improves the IR for better performance (e.g., eliminating redundant computations, loop optimizations).
Back End
Translates the optimized IR into machine code (for compiled languages like C, C++).
Virtual Machine
Instead of compiling directly to machine code, some languages (like Java, C#) compile to an intermediate bytecode.
This bytecode is executed by a Virtual Machine (JVM for Java, CLR for C#), enabling portability across platforms.

IIT Bombay
cs302: Implementation of Programming Languages

Topic:

Compilation Overview

Section:

Why Do We Need Compilers and Interpreters, Both?

t: Time

A: Analysis, O: Optimization, S: Synthesis, E: Execution, B: Bookkeeping

p: Program, c: Compiler usage, i: Interpreter usage, j: Number of executions

$$t_c(p,j) = t_c^A(p) + t_c^O(p) + t_c^S(p) + \left(t_c^E(p) \times j\right)$$

$$t_i(p,j) = \left(t_i^A(p) + t_i^B(p) + t_i^E(p)\right) \times j$$

t (Time): Represents the overall time taken for program processing.
A (Analysis): Examines and breaks down the source code (lexical, syntax, and semantic analysis).
O (Optimization): Improves code efficiency by reducing redundant computations and enhancing performance.
S (Synthesis): Constructs the final output, such as machine code or intermediate representation.
E (Execution): Runs the program, either through direct execution (compiled) or step-by-step interpretation.
B (Bookkeeping): Maintains metadata, symbol tables, and runtime information for program execution.
p (Program): The input source code that undergoes compilation or interpretation.
c (Compiler usage): Indicates whether a compiler is used to translate the program before execution.
i (Interpreter usage): Indicates whether an interpreter executes the program directly without full compilation.
j (Number of executions): Represents how many times the program is run, affecting optimization trade-

Why Do We Need Compilers and Interpreters, Both?

t: Time

A: Analysis, O: Optimization, S: Synthesis, E: Execution, B: Bookkeeping

p: Program, c: Compiler usage, i: Interpreter usage, j: Number of executions

compilation overheads

$$t_c(p,j) = \boxed{t_c^A(p) + t_c^O(p) + t_c^S(p)} + \left(t_c^E(p) \times j\right)$$

$$t_i(p,j) = \left(\boxed{t_i^A(p) + t_i^B(p)} + t_i^E(p)\right) \boxed{\times j}$$

interpretation overheads

Why Do We Need Compilers and Interpreters, Both?

t: Time
A: Analysis, O: Optimization, S: Synthesis, E: Execution, B: Bookkeeping
p: Program, c: Compiler usage, i: Interpreter usage, j: Number of executions

compilation overheads

$$t_c(p,j) = \boxed{t_c^A(p) + t_c^O(p) + t_c^S(p)} + \left(t_c^E(p) \times j\right)$$

$$t_i(p,j) = \left(\boxed{t_i^A(p) + t_i^B(p)} + t_i^E(p)\right) \boxed{\times j}$$

interpretation overheads

In general

- For large values of j, $t_c(p,j) \ll t_i(p,j)$
  Overheads of compilation are amortized over multiple executions

Why Do We Need Compilers and Interpreters, Both?

t: Time
A: Analysis, O: Optimization, S: Synthesis, E: Execution, B: Bookkeeping
p: Program, c: Compiler usage, i: Interpreter usage, j: Number of executions

$$t_c(p,j) = \underbrace{\boxed{t_c^A(p) + t_c^O(p) + t_c^S(p)}}_{\text{compilation overheads}} + \left(t_c^E(p) \times j\right)$$

$$t_i(p,j) = \left(\boxed{t_i^A(p) + t_i^B(p)} + t_i^E(p)\right) \boxed{\times j}$$

interpretation overheads

In general

- For large values of j, $t_c(p,j) \ll t_i(p,j)$
  Overheads of compilation are amortized over multiple executions
- For small values of j, $t_c(p,j) \gg t_i(p,j)$
  Overheads of interpretations are meaningful for infrequently executed jobs

Why Do We Need Compilers and Interpreters, Both?

t: Time
A: Analysis, O: Optimization, S: Synthesis, E: Execution, B: Bookkeeping
p: Program, c: Compiler usage, i: Interpreter usage, j: Number of executions

$$t_c(p,j) = \underbrace{t_c^A(p) + t_c^O(p) + t_c^S(p)}_{\text{compilation overheads}} + \left(t_c^E(p) \times j\right)$$

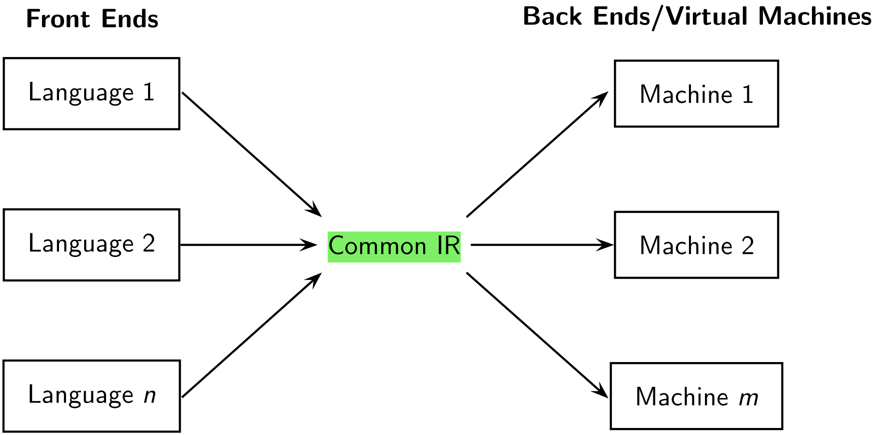
$$t_i(p,j) = \left(\boxed{t_i^A(p) + t_i^B(p)} + t_i^E(p)\right) \boxed{\times j}$$

interpretation overheads

In general

- For large values of j, $t_c(p,j) \ll t_i(p,j)$
  Overheads of compilation are amortized over multiple executions
- For small values of j, $t_c(p,j) \gg t_i(p,j)$
  Overheads of interpretations are meaningful for infrequently executed jobs
- For any value of $j > 0$, $\left(t_c^E(p) \times j\right) \ll t_i(p,j)$
  Unless interpreter identifies hot paths/procedures and compiles them with run time data

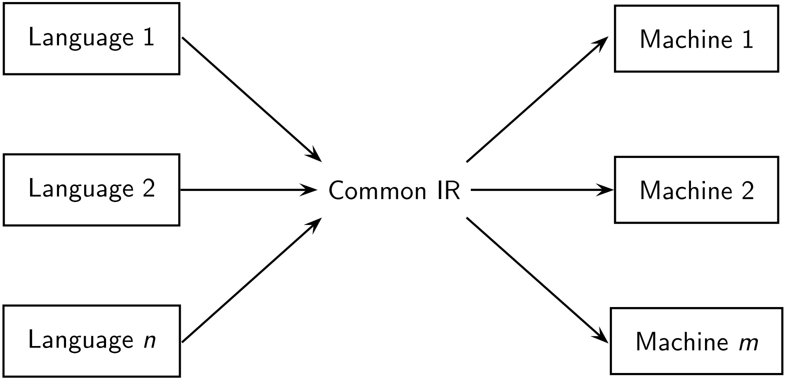
Reusability of Language Processor Modules



Reusability of Language Processor Modules

A module refers to a distinct component of the compiler, such as the front-end (responsible for syntax and semantics) or the back-end (responsible for code generation and optimization).
If you have m front-end modules (handling different source languages) and n back-end modules (targeting different machine architectures), you can mix and match them to create m × n unique compilers.

Front Ends **Back Ends/Virtual Machines**

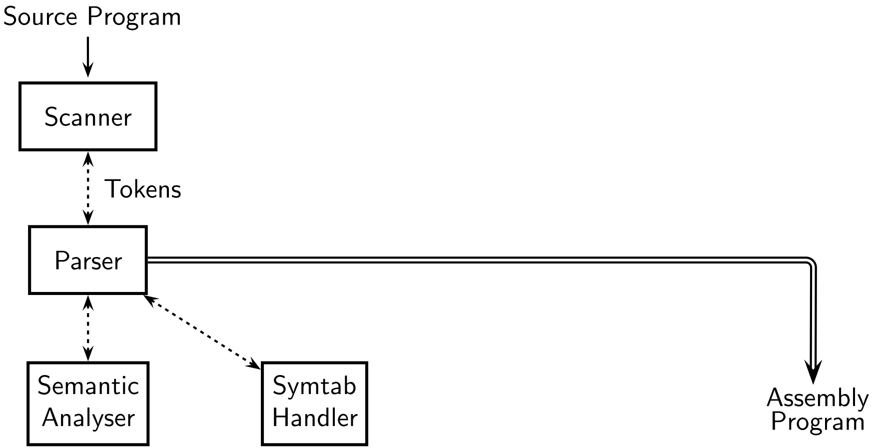


$m \times n$ compilers can be obtained from $m + n$ modules

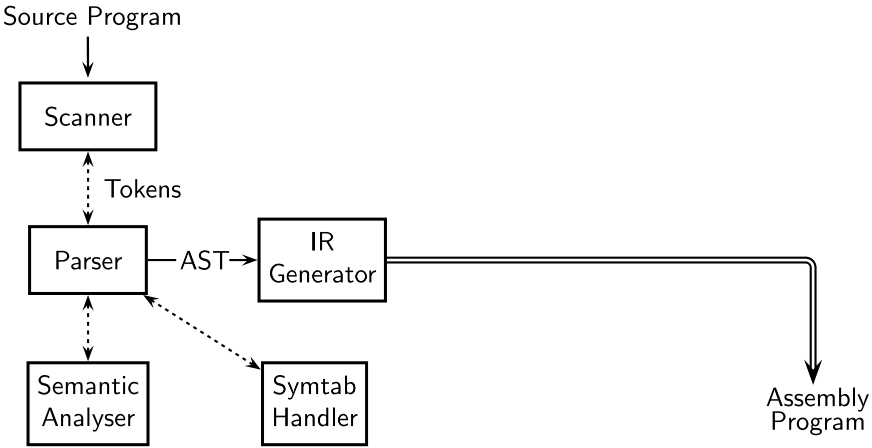
An Overview of Compilation Phases

The Structure of a Simple Compiler

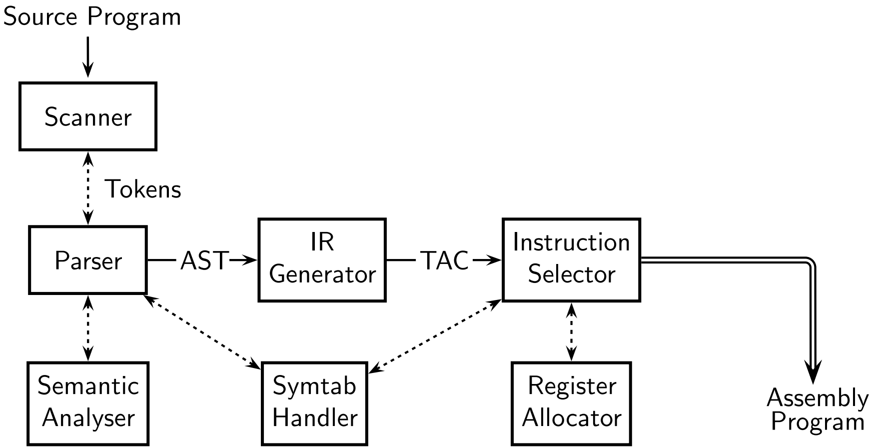
The Structure of a Simple Compiler



The Structure of a Simple Compiler

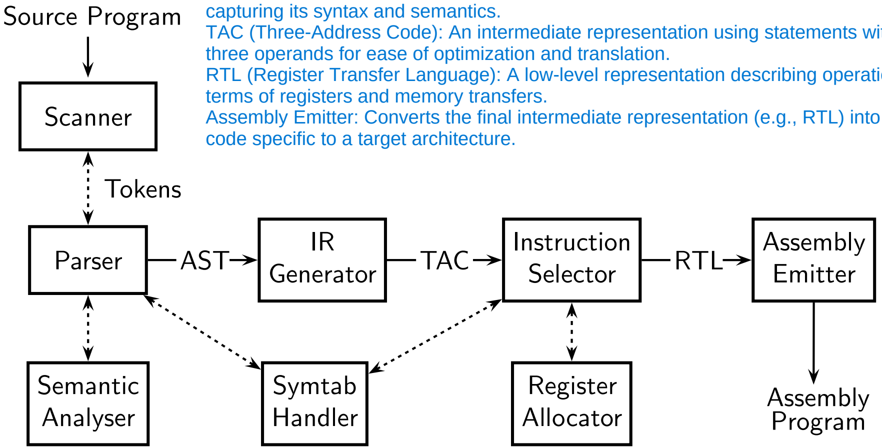


The Structure of a Simple Compiler



The Structure of a Simple Compiler

AST (Abstract Syntax Tree): A hierarchical tree representation of the source code structure, capturing its syntax and semantics.
TAC (Three-Address Code): An intermediate representation using statements with at most three operands for ease of optimization and translation.
RTL (Register Transfer Language): A low-level representation describing operations in terms of registers and memory transfers.
Assembly Emitter: Converts the final intermediate representation (e.g., RTL) into assembly code specific to a target architecture.



Translation Sequence in Our Compiler: Scanning and Parsing

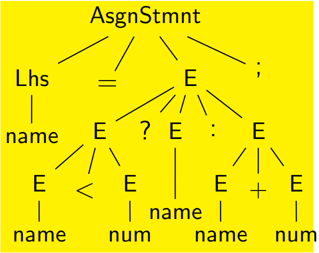
Input

```
a = b<10 ? b : c+5;
```

Translation Sequence in Our Compiler: Scanning and Parsing

Input

`a = b<10 ? b : c+5;`



Parse Tree

How the input is actually stored in the memory

`a␣=␣b␣<␣10␣?␣b␣:␣c␣+␣5␣;␣`

How we want to see it

$\boxed{\texttt{a}}$␣$\boxed{\texttt{=}}$␣$\boxed{\texttt{b}}$␣$\boxed{\texttt{<}}$␣$\boxed{\texttt{10}}$␣$\boxed{\texttt{?}}$␣$\boxed{\texttt{b}}$␣$\boxed{\texttt{:}}$␣$\boxed{\texttt{c}}$␣$\boxed{\texttt{+}}$␣$\boxed{\texttt{5}}$␣$\boxed{\texttt{;}}$␣

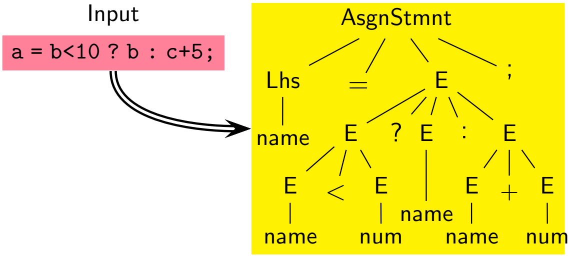
Issues:

- Grammar rules, terminals, non-terminals
- Order of application of grammar rules

  eg. is it `(a = b<10?)` followed by `(b:c)`?
- Values of terminal symbols

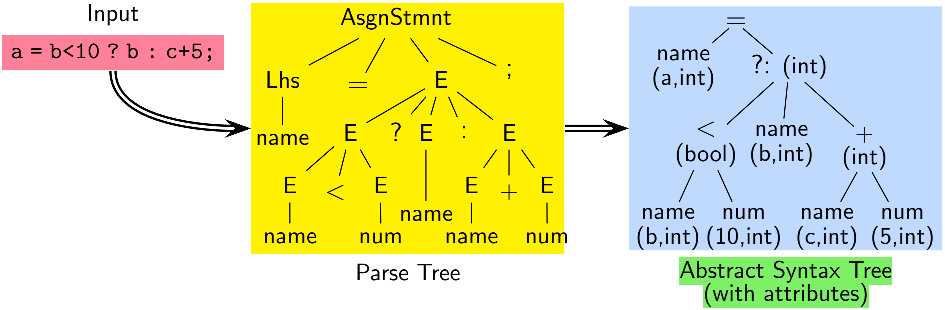
  eg. string "10" vs. integer number 10.

Translation Sequence in Our Compiler: Semantic Analysis



Parse Tree

Translation Sequence in Our Compiler: Semantic Analysis



Issues:

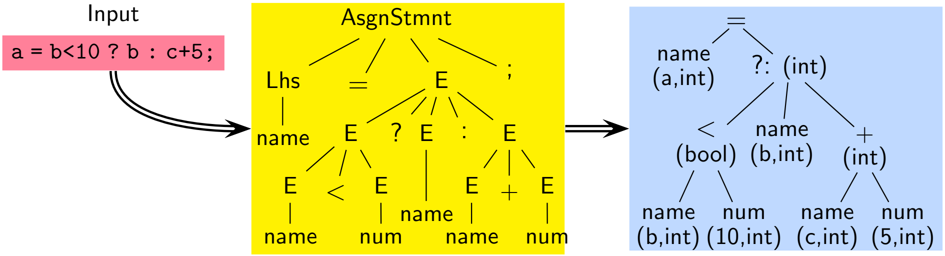
- Symbol tables

  Have variables been declared? What are their types? What is their scope?

- Type consistency of operators and operands

  The result of computing `b<10?` is bool and not int

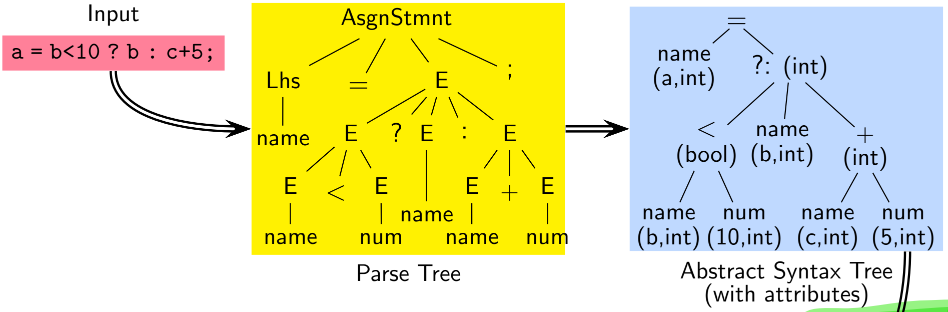
Translation Sequence in Our Compiler: IR Generation



Parse Tree

Abstract Syntax Tree
(with attributes)

Translation Sequence in Our Compiler: IR Generation



IR yha bn rhi h

TAC List

$T_1 = b < 10$
$T_2 = \neg T_1$
if T_2 goto L0
$T_3 = b$
goto L1:
L0: $T_3 = c + 5$
L1: $a = T_3$

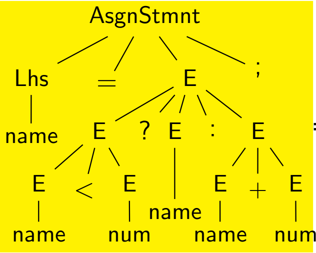
Issues:

- Convert to three address code (TAC)
  separating data and control flow
  Simplifies optimization
- Linearise control flow by flattening nested control constructs

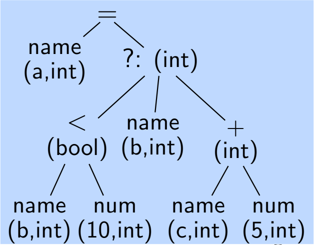
Translation Sequence in Our Compiler: Instruction Selection

Input

```
a = b<10 ? b : c+5;
```



Parse Tree



Abstract Syntax Tree
(with attributes)

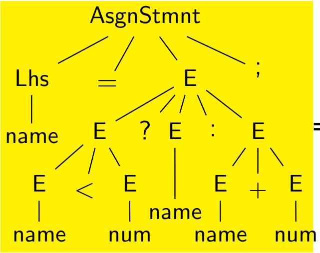
TAC List

$T_1 = b < 10$
$T_2 = \neg T_1$
if T_2 goto L0
$T_3 = b$
goto L1:
L0: $T_3 = c + 5$
L1: $a = T_3$

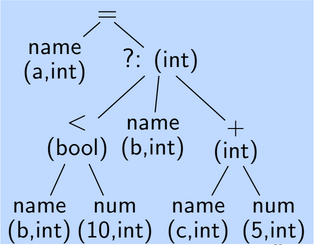
Translation Sequence in Our Compiler: Instruction Selection

Input

`a = b<10 ? b : c+5;`



Parse Tree



Abstract Syntax Tree
(with attributes)

TAC List

$T_1 = b < 10$
$T_2 = \neg T_1$
if T_2 goto L0
$T_3 = b$
goto L1:
L0: $T_3 = c + 5$
L1: $a = T_3$

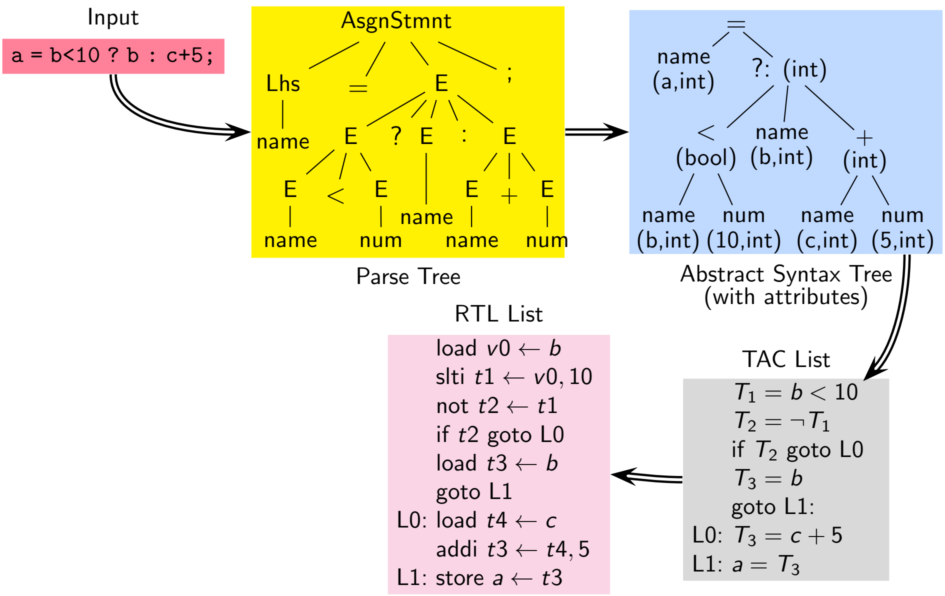
RTL List

load $v0 \leftarrow b$
slti $t1 \leftarrow v0, 10$
not $t2 \leftarrow t1$
if $t2$ goto L0
load $t3 \leftarrow b$
goto L1
L0: load $t4 \leftarrow c$
addi $t3 \leftarrow t4, 5$
L1: store $a \leftarrow t3$

Issues:

- Generate as few instructions as possible (list shown here is unoptimized)
- Use temporaries and local registers

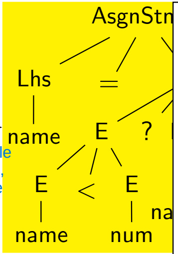
Translation Sequence in Our Compiler: Emitting Instructions



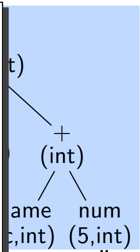
Translation Sequence in Our Compiler: Emitting Instructions

Input

```
a = b<10 ? b : c+5;
```



Parse T



tax Tree
(with attributes)

Issues:

- Offsets of variables in the stack frame
- Actual register numbers and assembly mnemonics
- Code to construct and discard activation records

Assembly Mnemonics: Human-readable symbolic instructions (e.g., MOV, ADD, JMP) that correspond to machine code operations.
Activation Records: Data structures in the stack that store function-related information like local variables, return addresses, and saved registers during function calls.

TAC List

$T_1 = b < 10$
$T_2 = \neg T_1$
if T_2 goto L0
$T_3 = b$
goto L1:
L0: $T_3 = c + 5$
L1: $a = T_3$

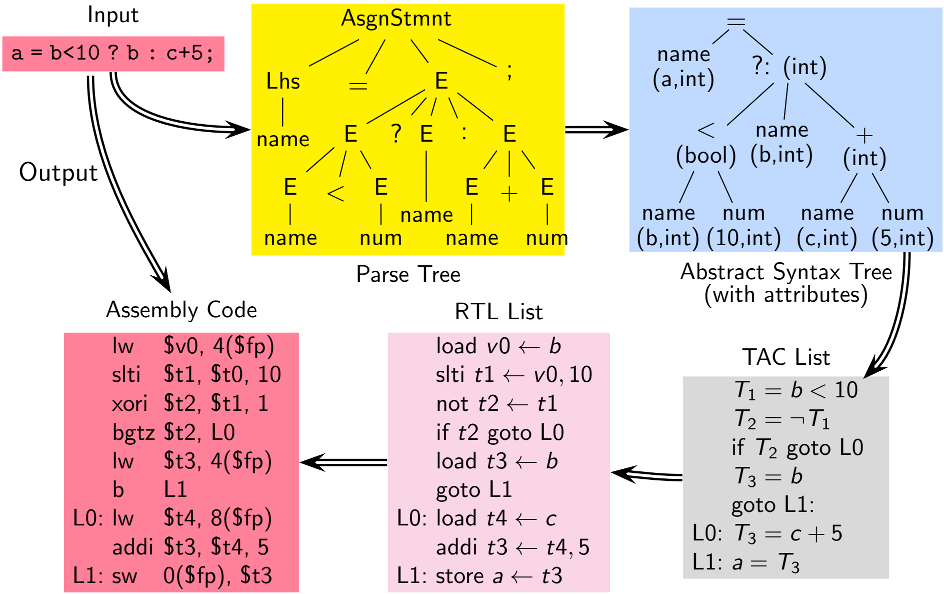
RTL List

load $v0 \leftarrow b$
slti $t1 \leftarrow v0, 10$
not $t2 \leftarrow t1$
if $t2$ goto L0
load $t3 \leftarrow b$
goto L1
L0: load $t4 \leftarrow c$
addi $t3 \leftarrow t4, 5$
L1: store $a \leftarrow t3$

Assembly Code

```
     lw   $v0, 4($fp)
     slti $t1, $t0, 10
     xori $t2, $t1, 1
     bgtz $t2, L0
     lw   $t3, 4($fp)
     b    L1
L0:  lw   $t4, 8($fp)
     addi $t3, $t4, 5
L1:  sw   0($fp), $t3
```

Translation Sequence in Our Compiler: Emitting Instructions



Observations

- A compiler bridges the gap between source program and target program

Observations

- A compiler bridges the gap between source program and target program
- Compilation involves gradual lowering of levels of the IR of an input program

Observations

- A compiler bridges the gap between source program and target program
- Compilation involves gradual lowering of levels of the IR of an input program
- The design of IRs is the most critical part of a compiler design
  - How many IRs should we have?
  - What are the details that each IR captures?

Observations

- A compiler bridges the gap between source program and target program
- Compilation involves gradual lowering of levels of the IR of an input program
- The design of IRs is the most critical part of a compiler design
  - How many IRs should we have?
  - What are the details that each IR captures?
- Practical compilers are desired to be retargetable
  $\Rightarrow$ Back ends should be generated from specifications

Why Is Compiler Construction a Relevant Course?

Even though very few people write compilers . . .

Why Is Compiler Construction a Relevant Course?

Even though very few people write compilers . . .

- Translation and interpretation are fundamental CS at a conceptual level
  - Stepwise refinement Vs. look up
  - Analytics Vs. Transactional software

Why Is Compiler Construction a Relevant Course?

Even though very few people write compilers . . .

- Translation and interpretation are fundamental CS at a conceptual level
  - Stepwise refinement Vs. look up
  - Analytics Vs. Transactional software
- Computer Science is all about building layers of abstractions and bridging the gaps between successive layers

Why Is Compiler Construction a Relevant Course?

Even though very few people write compilers . . .

- Translation and interpretation are fundamental CS at a conceptual level
  - Stepwise refinement Vs. look up
  - Analytics Vs. Transactional software
- Computer Science is all about building layers of abstractions and bridging the gaps between successive layers
- Knowing compilers internals makes a person a much better programmer

  Writing programs whose data is programs

Why Is Compiler Construction a Relevant Course?

Analytics vs. Transactional Software: Analytics software processes and analyzes data to extract insights (e.g., BI tools, data mining), whereas transactional software focuses on executing real-time operations like database transactions (e.g., banking systems, e-commerce).

Even though very few people write compilers . . .

- Translation and interpretation are fundamental CS at a conceptual level
  - Stepwise refinement Vs. look up
  - Analytics Vs. Transactional software

Stepwise Refinement vs. Look-up: Stepwise refinement involves incrementally breaking down a problem into smaller subproblems (top-down design), while look-up retrieves precomputed results or stored information without further decomposition.

- Computer Science is all about building layers of abstractions and bridging the gaps between successive layers
- Knowing compilers internals makes a person a much better programmer

  Writing programs whose data is programs
- The beauty and enormity of compiling lies in
  - Raising the level of abstraction and bridging the gap without performance penalties
  - Meeting the expectations of users with a wide variety of needs

Where Can I Use the Lessons Learnt in Compiler Design?

- Compilers for all languages exist, so what can I do with the technology?

Where Can I Use the Lessons Learnt in Compiler Design?

- Compilers for all languages exist, so what can I do with the technology?
- Compiler techniques and tools have many applications
  - Parsers for HTML in web browser
  - Interpreters for javascript/flash
  - Machine code generation for high level languages
  - Software testing
  - Program optimization
  - Detection of malicious code
  - Design of new computer architectures
    Hardware-software codesign!
  - Hardware synthesis: VHDL to RTL translation
  - Compiled simulation to simulate designs written in VHDL

Credits: Adapted from the slides of Prof. Y. N. Srikant for NPTEL course on compilers

The Beauty and Enormity of Compiling

- Bridging the rather large gap between high and low level languages
  - Creating several layers of abstractions with smaller gaps
  - A great example of divide and conquer or stepwise refinement
- Developing and maintaining a rather large code base of millions of lines

The Beauty and Enormity of Compiling

- Bridging the rather large gap between high and low level languages
  - Creating several layers of abstractions with smaller gaps
  - A great example of divide and conquer or stepwise refinement
- Developing and maintaining a rather large code base of millions of lines
- Writing programs that read programs and write programs maintaining the semantics

The Beauty and Enormity of Compiling

- Bridging the rather large gap between high and low level languages
  - Creating several layers of abstractions with smaller gaps
  - A great example of divide and conquer or stepwise refinement
- Developing and maintaining a rather large code base of millions of lines
- Writing programs that read programs and write programs maintaining the semantics
- Extensive use of tools to generate modules from declarative specifications
  "Higher" level than HLLs

The Beauty and Enormity of Compiling

- Bridging the rather large gap between high and low level languages
  - Creating several layers of abstractions with smaller gaps
  - A great example of divide and conquer or stepwise refinement
- Developing and maintaining a rather large code base of millions of lines
- Writing programs that read programs and write programs maintaining the semantics
- Extensive use of tools to generate modules from declarative specifications
  "Higher" level than HLLs

HLLs (High-Level Languages): Programming languages like C, Python, and Java that provide abstraction from machine code, making code more human-readable and portable.

- Handling every possible programs from an infinite set of possible programs

The Beauty and Enormity of Compiling

- Bridging the rather large gap between high and low level languages
    - Creating several layers of abstractions with smaller gaps
    - A great example of divide and conquer or stepwise refinement
- Developing and maintaining a rather large code base of millions of lines
- Writing programs that read programs and write programs maintaining the semantics
- Extensive use of tools to generate modules from declarative specifications
  "Higher" level than HLLs
- Handling every possible programs from an infinite set of possible programs
- Exploiting advanced features of rich computer architectures

The Beauty and Enormity of Compiling

- Bridging the rather large gap between high and low level languages
  - Creating several layers of abstractions with smaller gaps
  - A great example of divide and conquer or stepwise refinement
- Developing and maintaining a rather large code base of millions of lines
- Writing programs that read programs and write programs maintaining the semantics
- Extensive use of tools to generate modules from declarative specifications
  "Higher" level than HLLs
- Handling every possible programs from an infinite set of possible programs
- Exploiting advanced features of rich computer architectures
- Spanning both theory and practice (and everything in between) rather deeply
  Translating deep theory into general, efficient, and scalable, practice!

Modern Compilers Span Both Theory and Practice Deeply

Compiler design and implementation translates deep theory into general, efficient, and scalable, practice!

- Uses principles and techniques from many areas in Computer Science
  - The design and implementation of a compiler is a great application of software engineering
  - Makes practical application of deep theory and algorithms and rich data structures
  - Uses rich features of computer architecture

Translating Deep Theory into Affordable Practice

- Theory and algorithms
  - Mathematical logic: type inference and checking
  - Lattice theory: static analysis
  - Linear algebra: dependence analysis and loop parallelization
  - Probability theory: hot path optimization
  - Greedy algorithms: register allocation
  - Heuristic search: instruction scheduling
  - Graph algorithms: register allocation
  - Dynamic programming: instruction selection
  - Optimization techniques: instruction scheduling
  - Finite automata: lexical analysis
  - Pushdown automata: parsing
  - Fixed point algorithms: data-flow analysis

Credits: Adapted from the slides of Prof. Y. N. Srikant, IISc Bangalore

Translating Deep Theory into Affordable Practice

- Data structures
    - Sparse representations: scanner and parser tables
    - Stacks, lists, and arrays: Symbols tables
    - Trees: abstract syntax trees, expression trees
    - Graphs: control flow graphs, call graphs, data dependence graphs,
    - DAGs: Expression DAG
    - Representing machine details such as instruction sets, registers, etc.

Compilation Models

Compilation Models

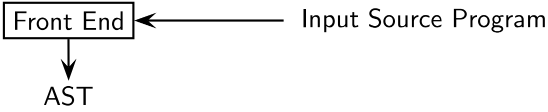
Aho Ullman Model

Davidson Fraser Model

Compilation Models

Aho Ullman Model

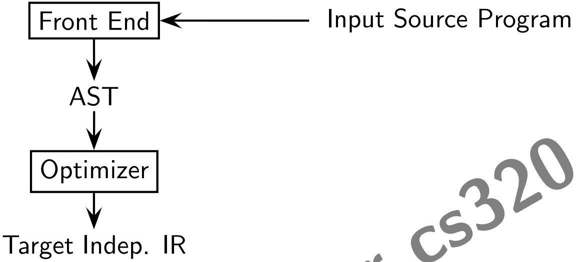
Davidson Fraser Model



Compilation Models

Aho Ullman Model

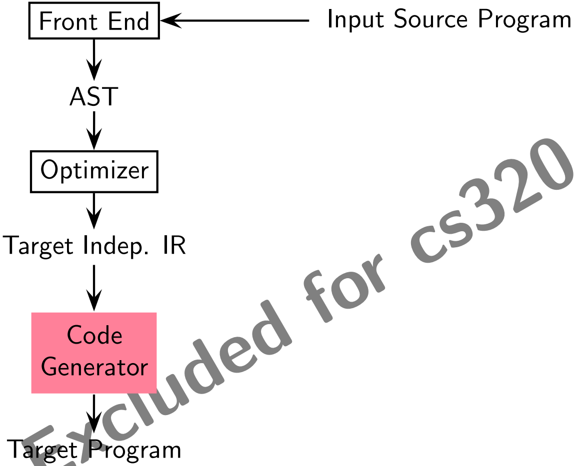
Davidson Fraser Model



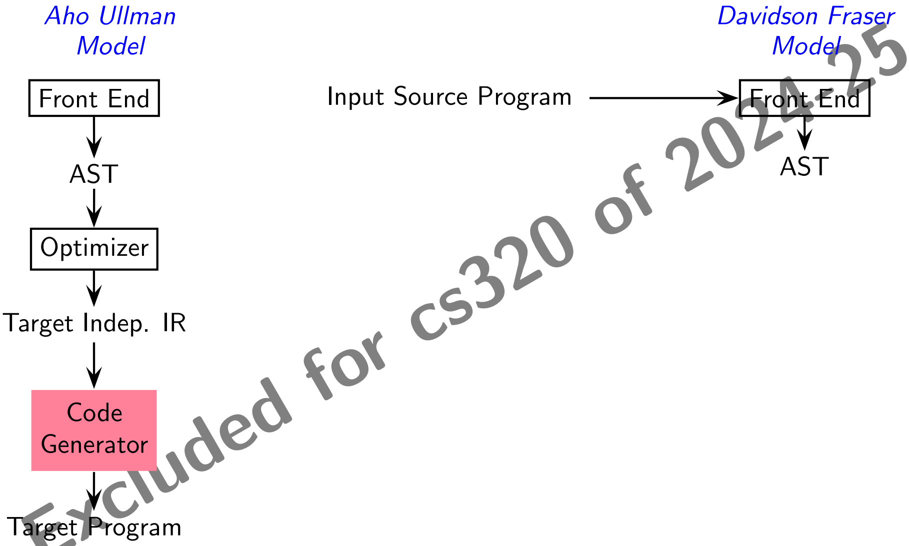
Compilation Models

Aho Ullman Model

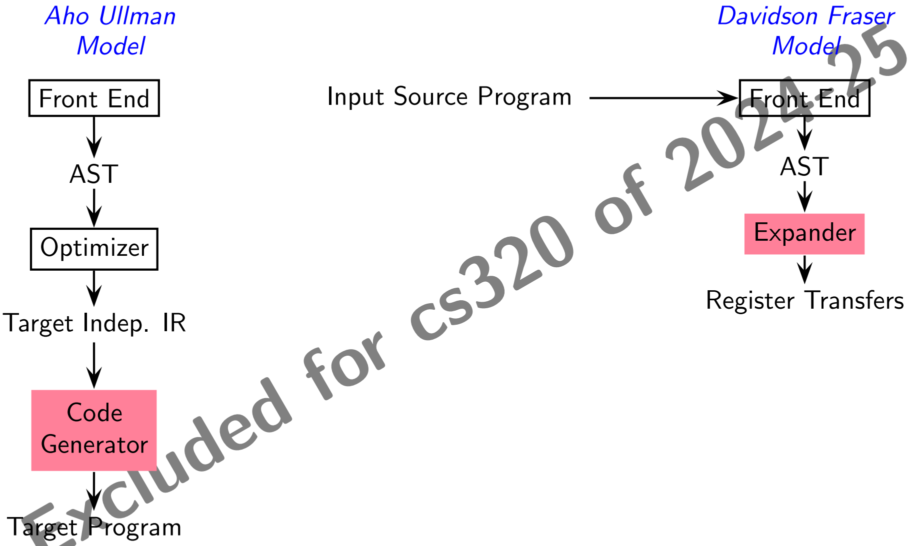
Davidson Fraser Model



Compilation Models



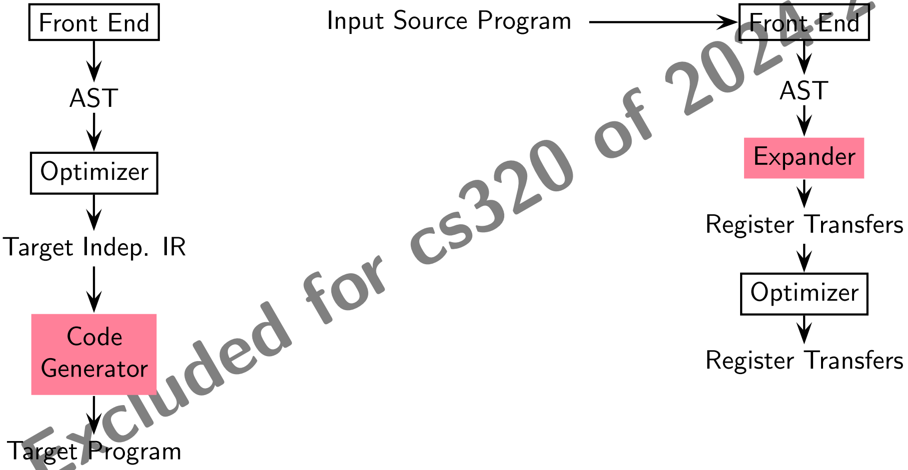
Compilation Models



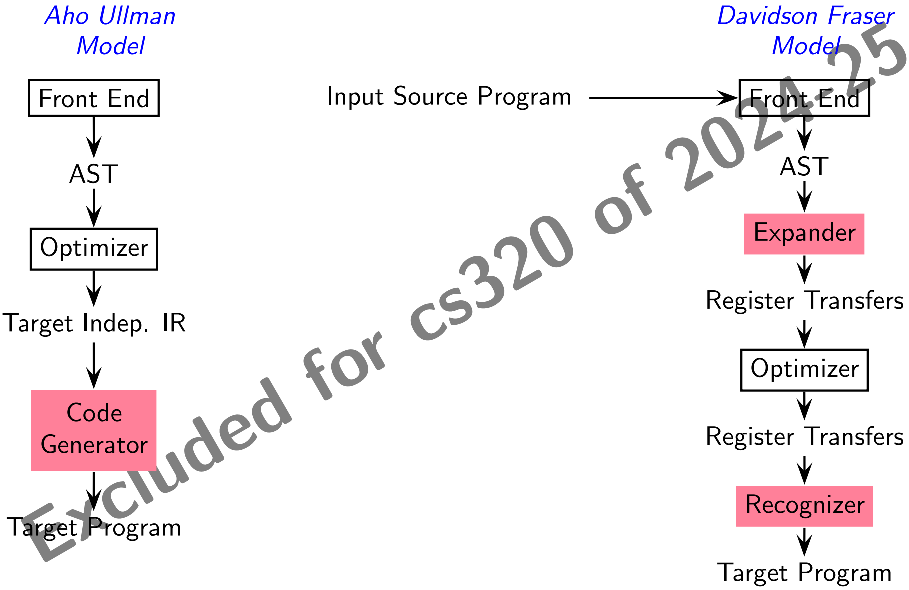
Compilation Models

Aho Ullman Model

Davidson Fraser Model



Compilation Models



Compilation Models

Aho Ullman Model

Front End → AST → Optimizer → Target Indep. IR → Code Generator → Target Program

Aho Ullman: Instruction selection

- over optimized IR using
- cost based tree tiling matching

Davidson Fraser: Instruction selection

- over AST using
- simple full tree matching based algorithms that generate
- naive code which is
  - target dependent, and is
  - optimized subsequently

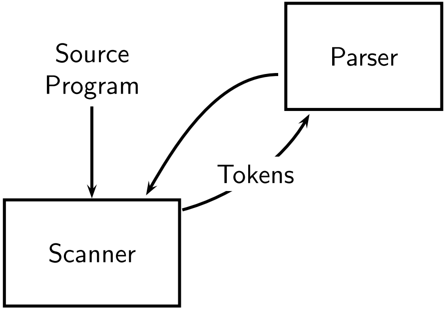
Davidson Fraser Model

Front End → AST → Expander → Register Transfers → Optimizer → Register Transfers → Recognizer → Target Program

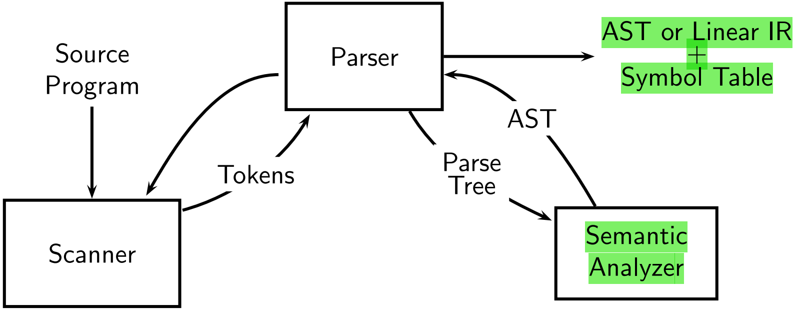
Typical Front Ends

Parser

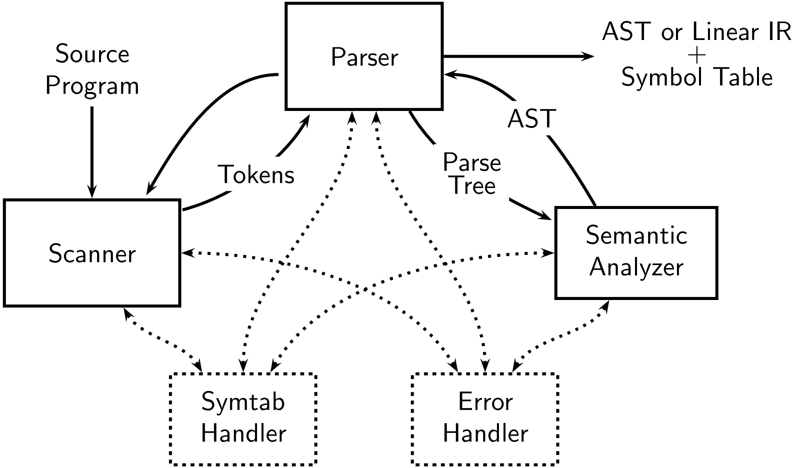
Typical Front Ends



Typical Front Ends

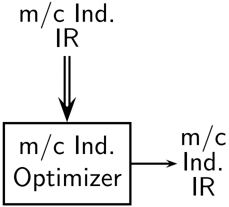


Typical Front Ends



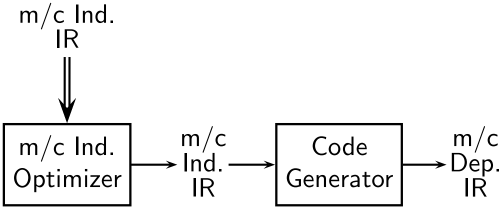
Typical Back Ends

Likely refers to "Machine Code Independence," meaning the ability of software to run on different machine architectures without modification.



- Compile time evaluations
- Eliminating redundant computations

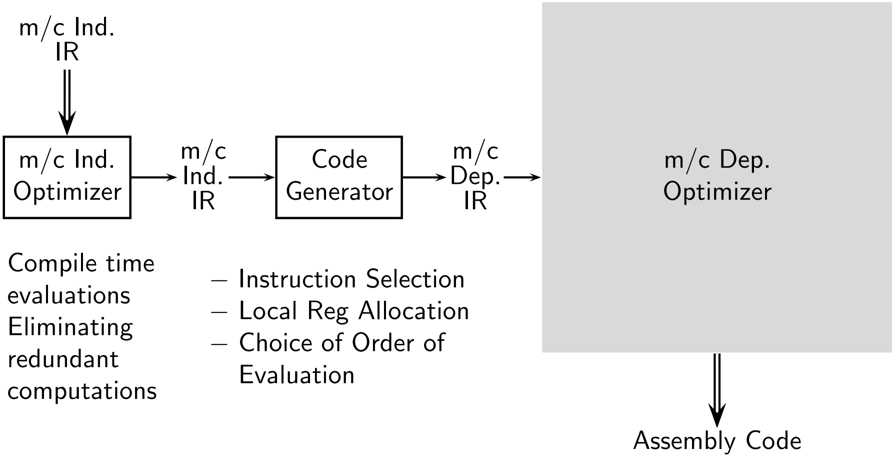
Typical Back Ends



– Compile time evaluations
– Eliminating redundant computations

– Instruction Selection
– Local Reg Allocation
– Choice of Order of Evaluation

Typical Back Ends

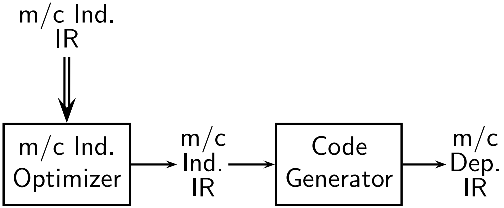


- Compile time evaluations
- Eliminating redundant computations

- Instruction Selection
- Local Reg Allocation
- Choice of Order of Evaluation

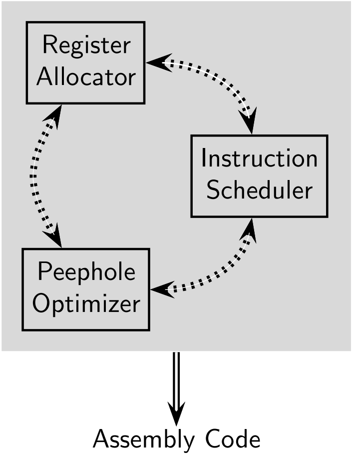
Typical Back Ends

A peephole optimizer is a type of optimization technique used in compilers that focuses on small, local optimizations on a small set of instructions (a "peephole") in the intermediate code or assembly. It looks for patterns or redundancies in a short sequence of instructions (typically 1 to 5 instructions) and replaces them with more efficient equivalents, improving performance without affecting the overall program structure.
For example, replacing MOV A, B; ADD A, C; with a simpler ADD B, C if A is not needed later.

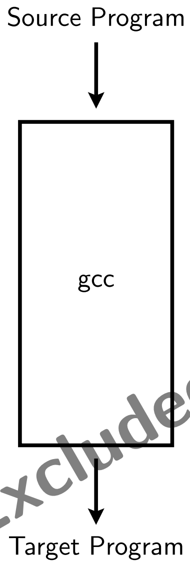


- Compile time evaluations
- Eliminating redundant computations

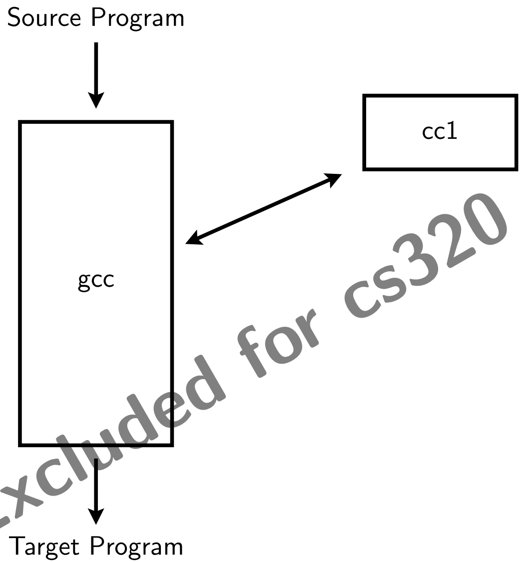
- Instruction Selection
- Local Reg Allocation
- Choice of Order of Evaluation



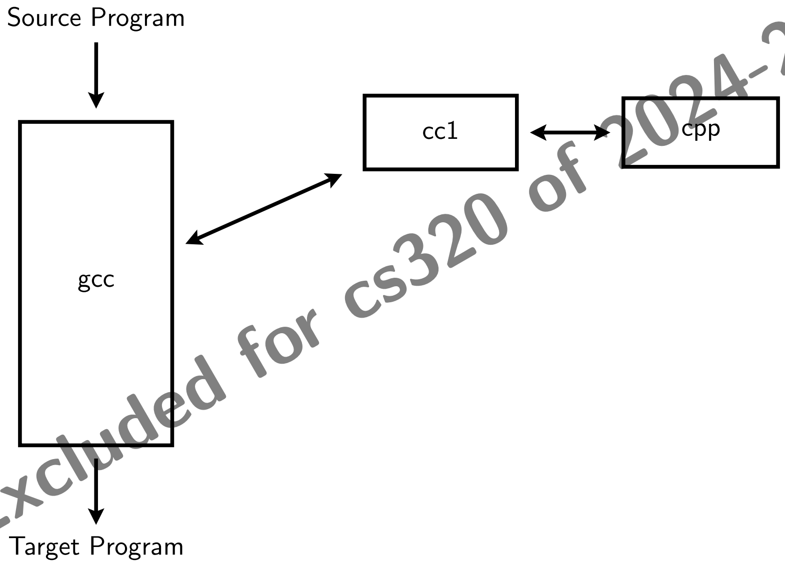
The GNU Tool Chain for C



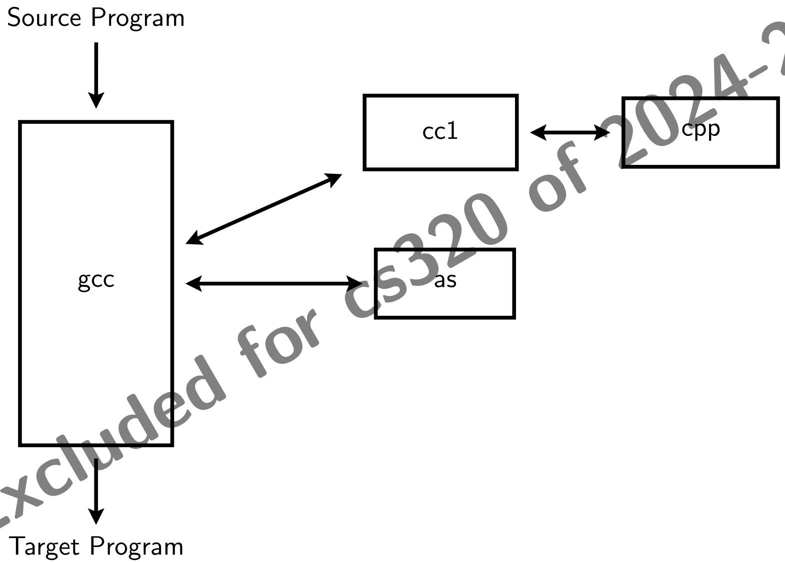
The GNU Tool Chain for C



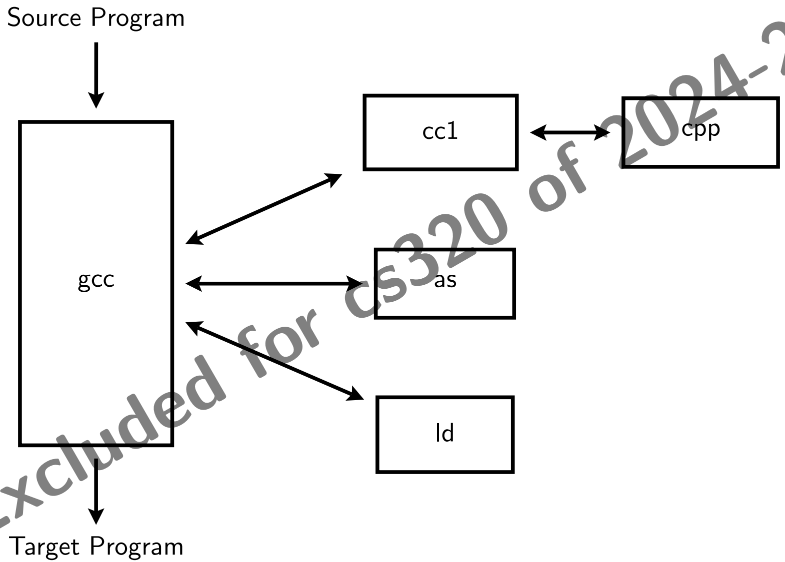
The GNU Tool Chain for C



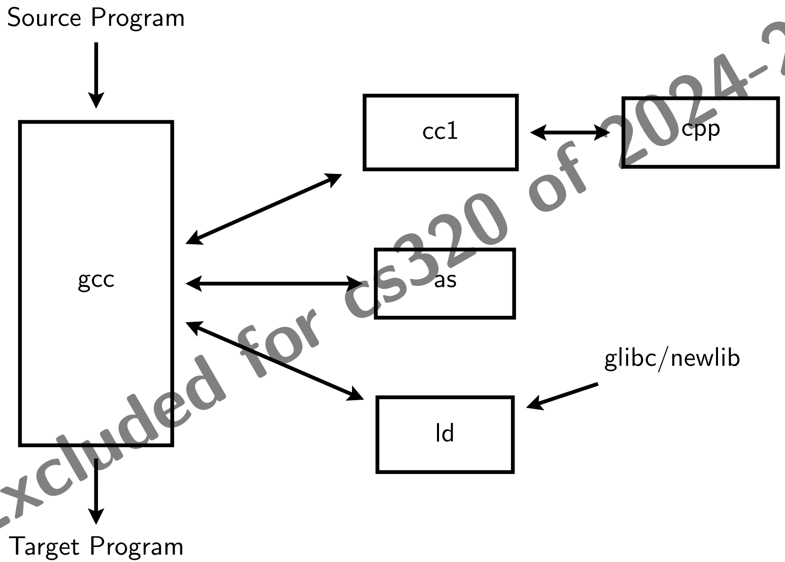
The GNU Tool Chain for C



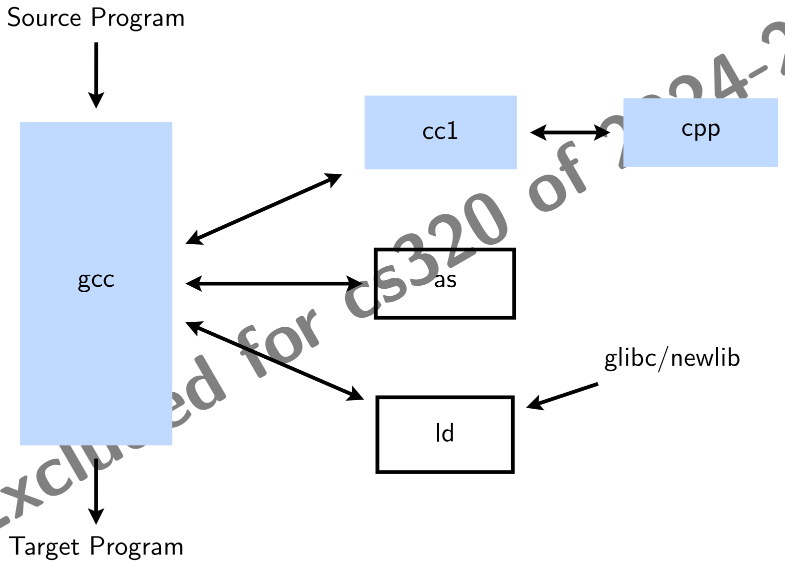
The GNU Tool Chain for C



The GNU Tool Chain for C



The GNU Tool Chain for C



IIT Bombay
cs302: Implementation of Programming Languages

Topic:
Compilation Overview

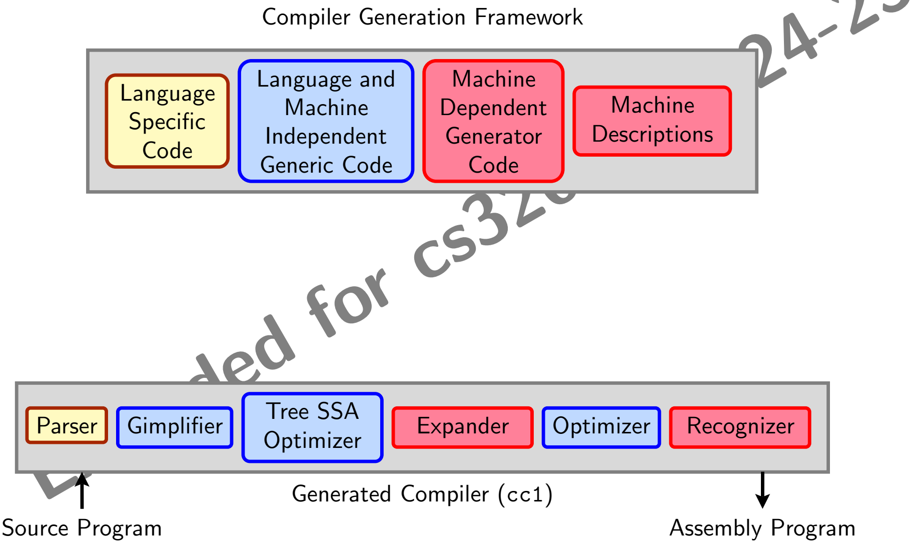
Section:

The Architecture of GCC

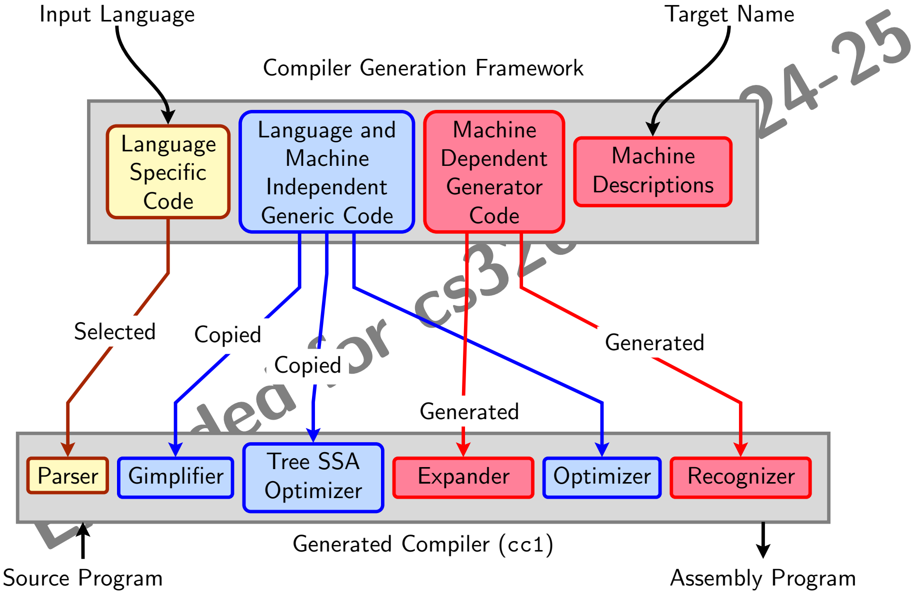
Compiler Generation Framework

Language Specific Code	Language and Machine Independent Generic Code	Machine Dependent Generator Code	Machine Descriptions

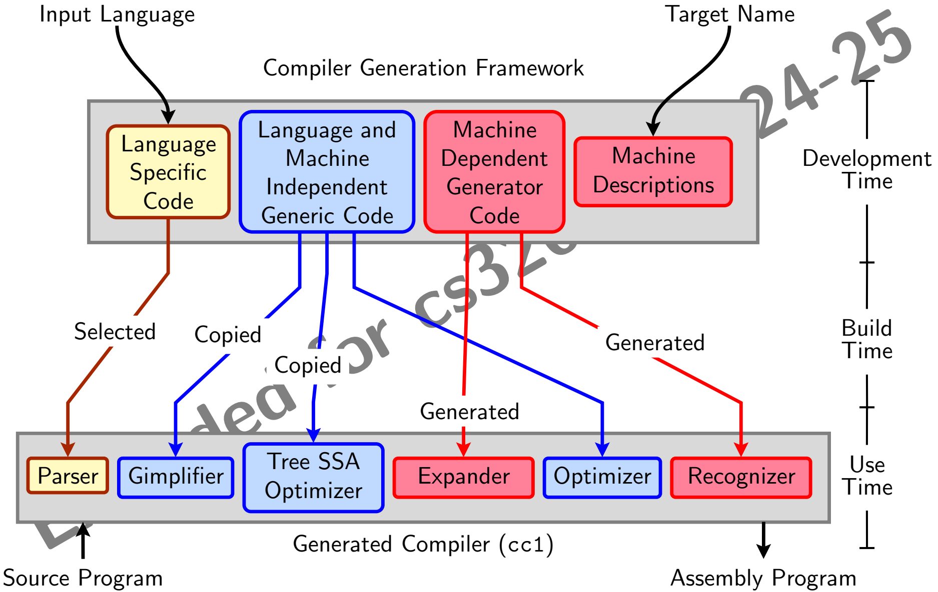
The Architecture of GCC



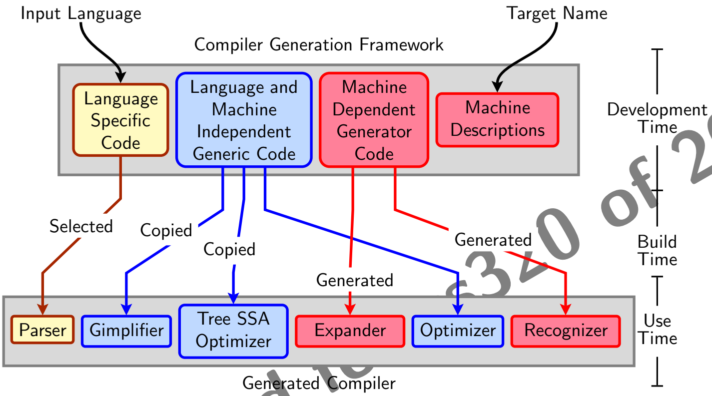
The Architecture of GCC



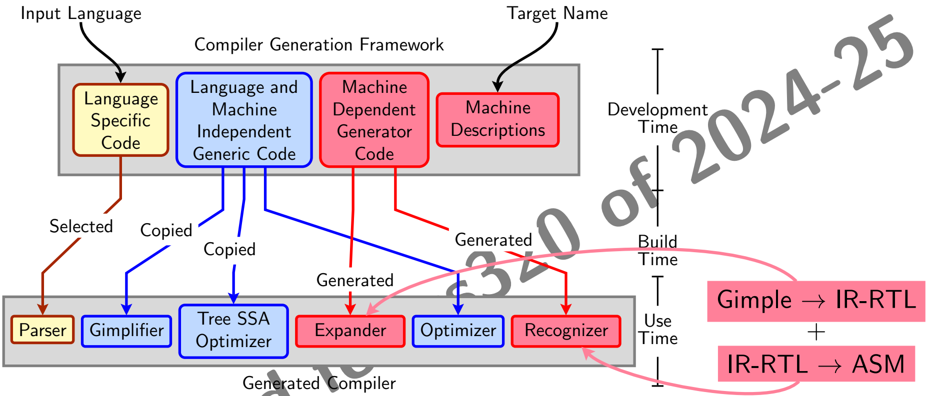
The Architecture of GCC



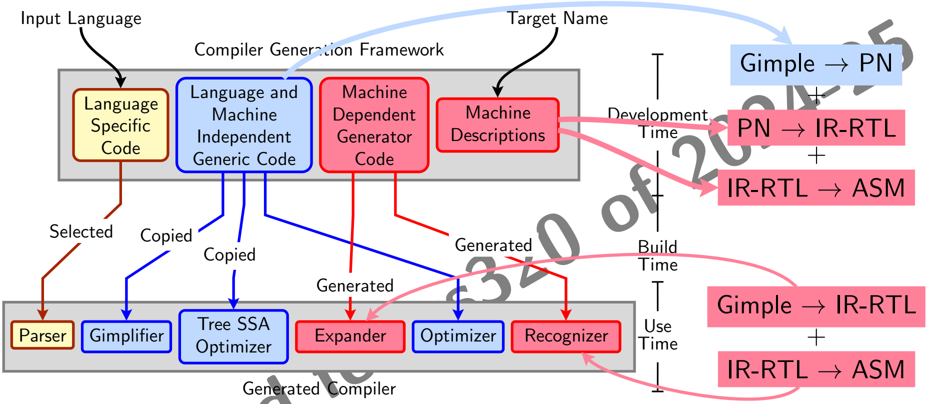
GCC Retargetability Mechanism



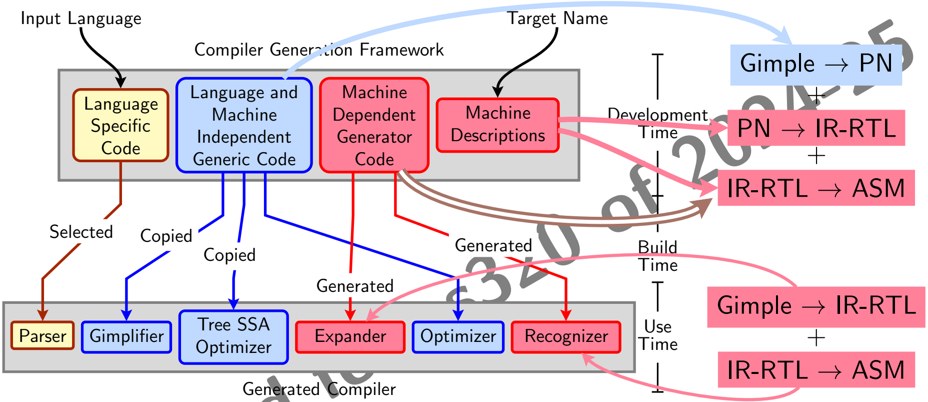
GCC Retargetability Mechanism



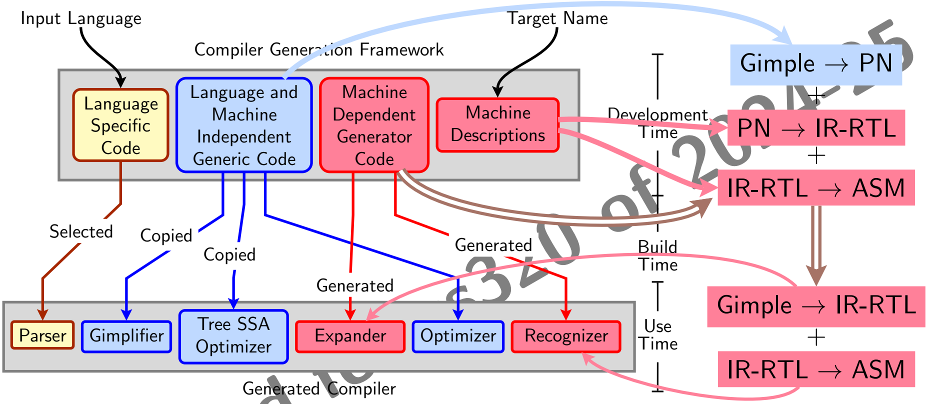
GCC Retargetability Mechanism



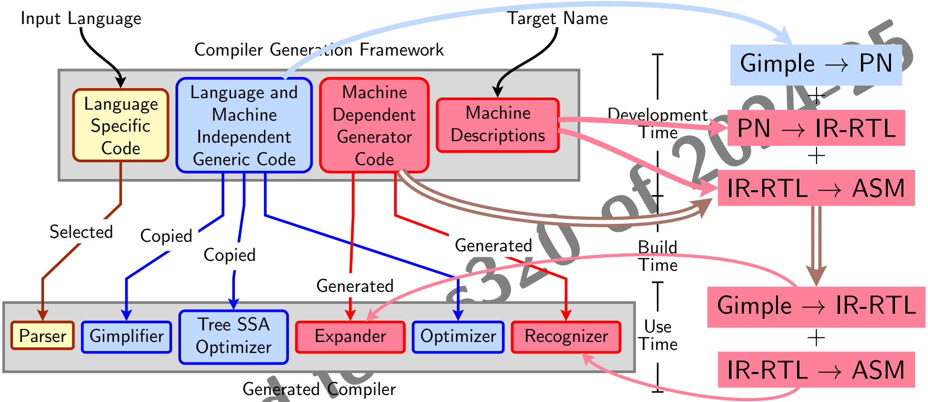
GCC Retargetability Mechanism



GCC Retargetability Mechanism



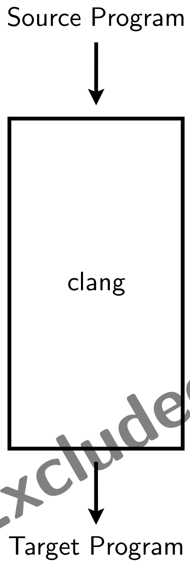
GCC Retargetability Mechanism



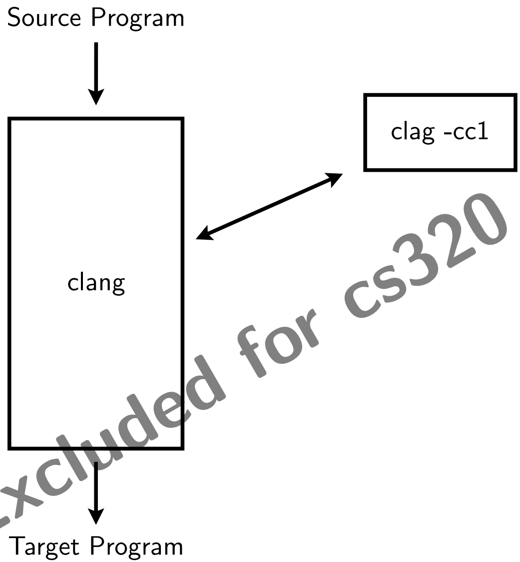
The generated compiler uses an adaptation of the Davidson Fraser model

- Generic expander and recognizer
- Machine specific information is isolated in data structures
- Generating a compiler involves generating these data structures

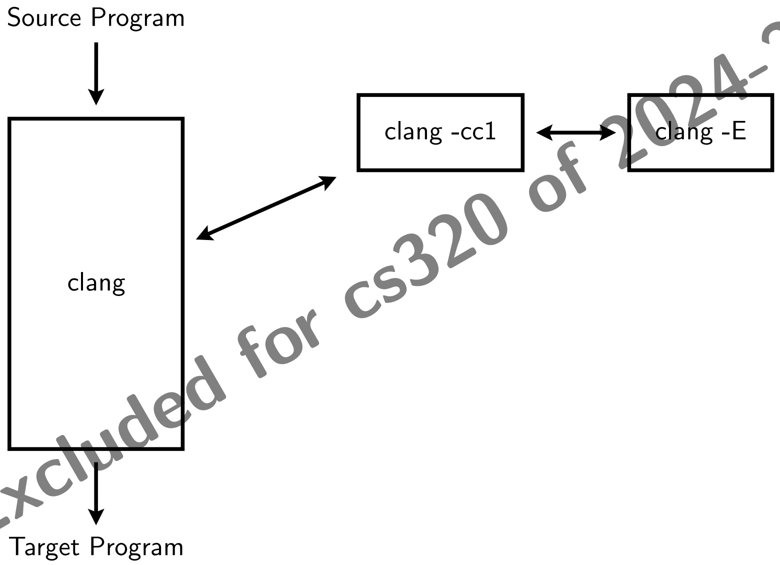
The LLVM Tool Chain for C



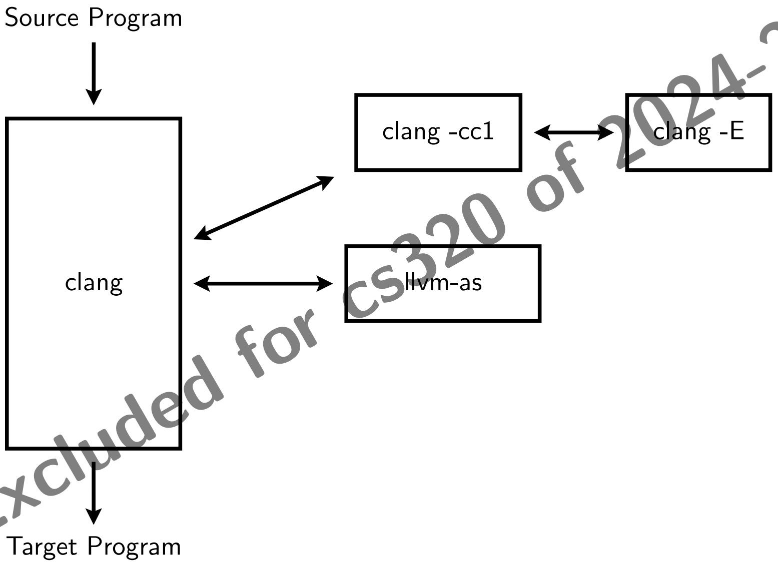
The LLVM Tool Chain for C



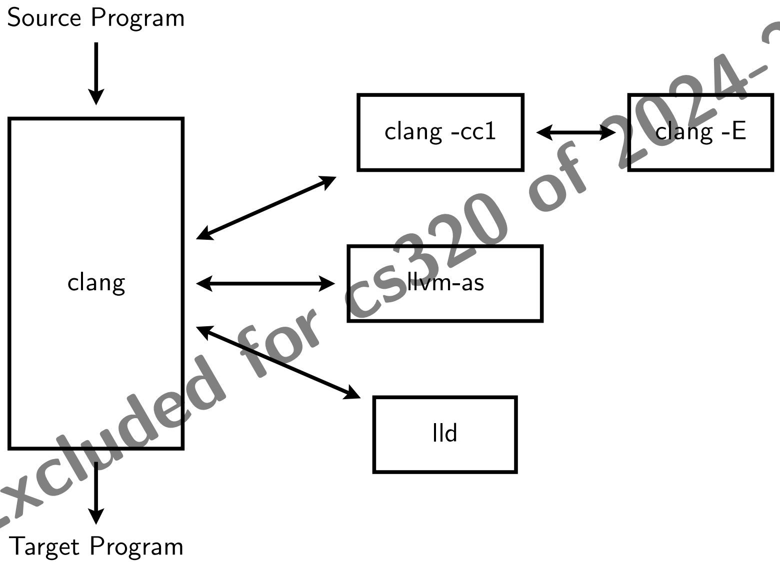
The LLVM Tool Chain for C



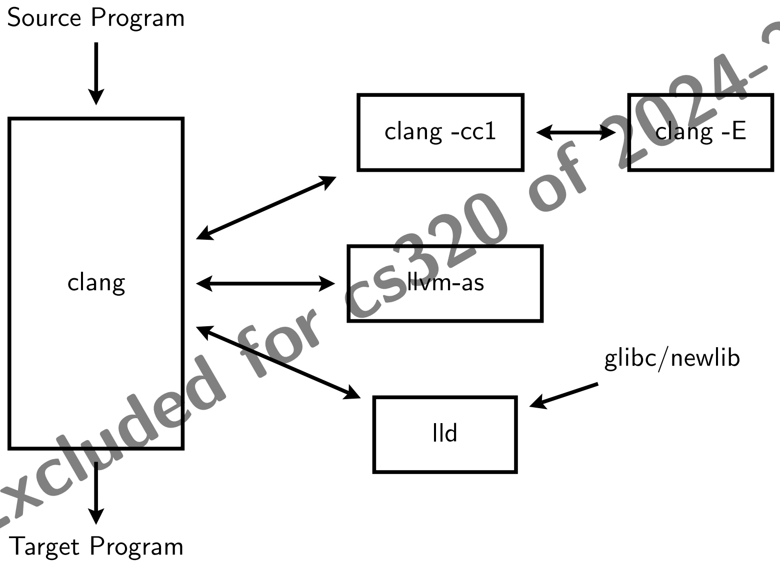
The LLVM Tool Chain for C



The LLVM Tool Chain for C



The LLVM Tool Chain for C

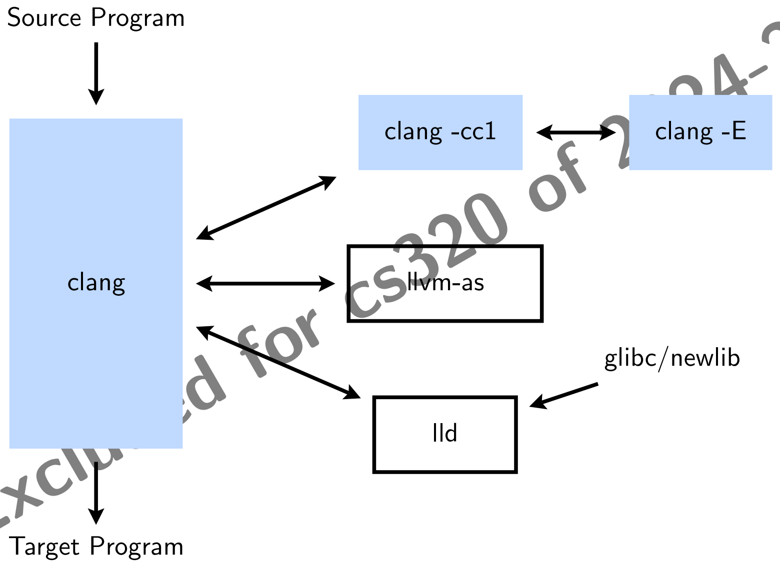


IIT Bombay
cs302: Implementation of Programming Languages

Topic:
Compilation Overview

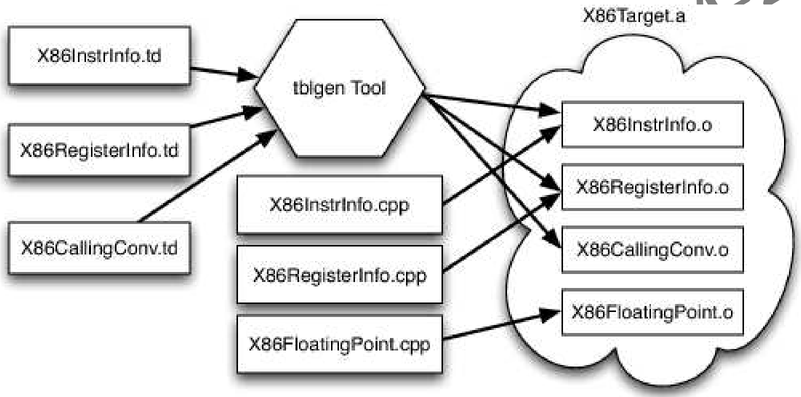
Section:

The LLVM Tool Chain for C



LLVM Retargetability Mechanism

Simplified x86 Target Definition



Reproduced from https://www.aosabook.org/en/llvm.html

Building a Compiler: Terminology

- The sources of a compiler are compiled (i.e. built) on *Build system*, denoted BS.
- The built compiler runs on the *Host system*, denoted HS.
- The compiler compiles code for the *Target system*, denoted TS.

The built compiler itself runs on HS and generates executables that run on TS.

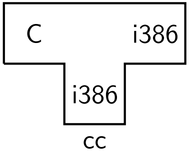
Variants of Compiler Builds

BS = HS = TS	Native Build
BS = HS ≠ TS	Cross Build
BS ≠ HS ≠ TS	Canadian Cross

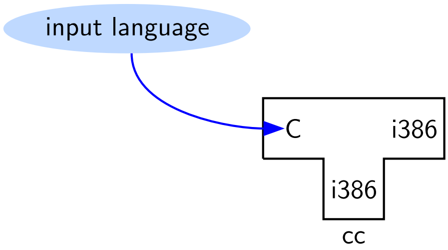
Example

Native i386: built on i386, hosted on i386, produces i386 code.
Sparc cross on i386: built on i386, hosted on i386, produces Sparc code.

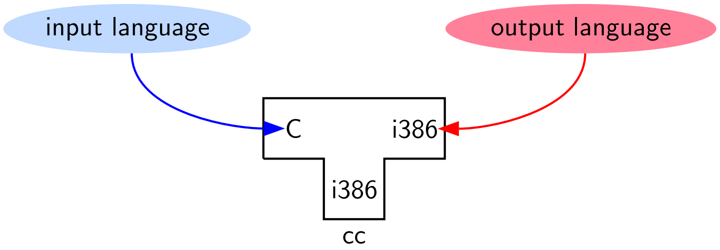
T Notation for a Compiler



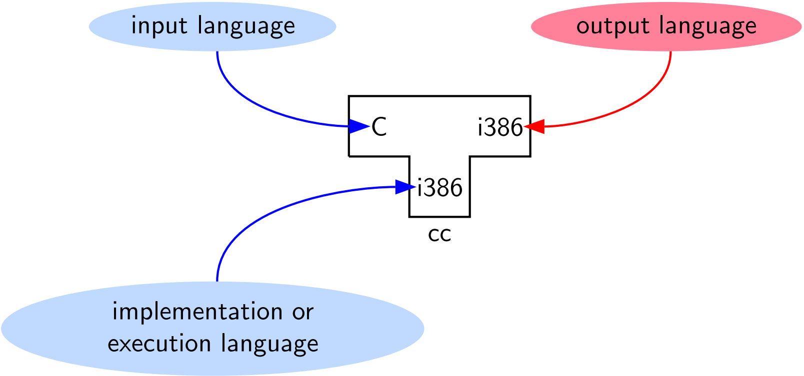
T Notation for a Compiler



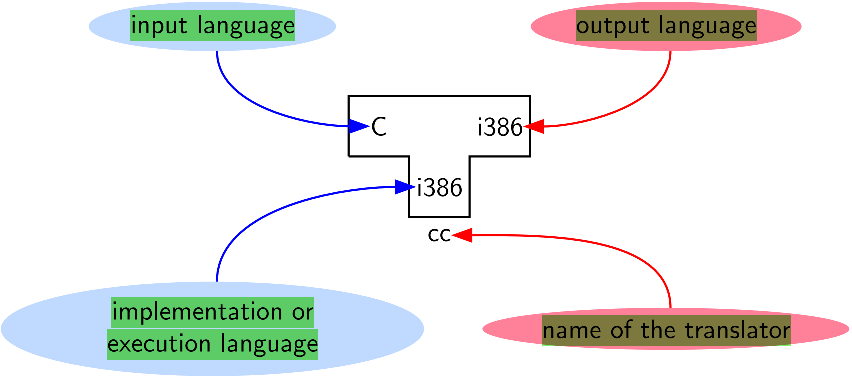
T Notation for a Compiler



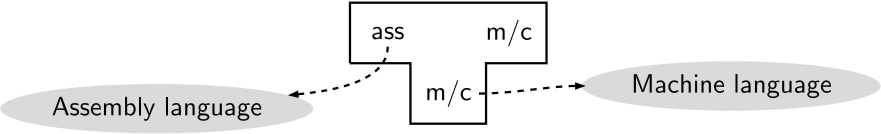
T Notation for a Compiler



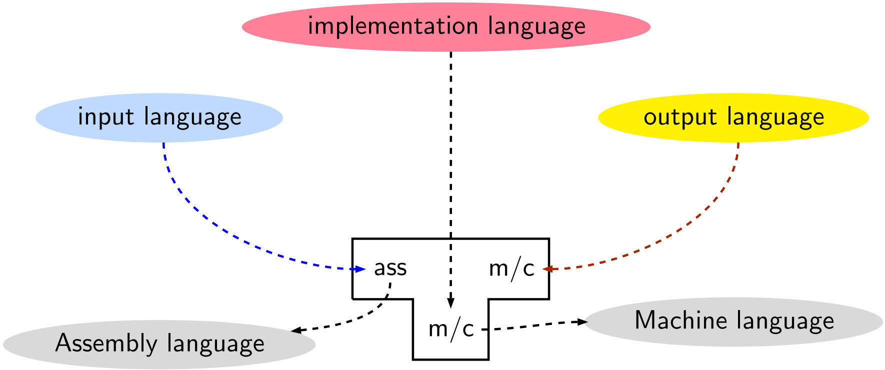
T Notation for a Compiler



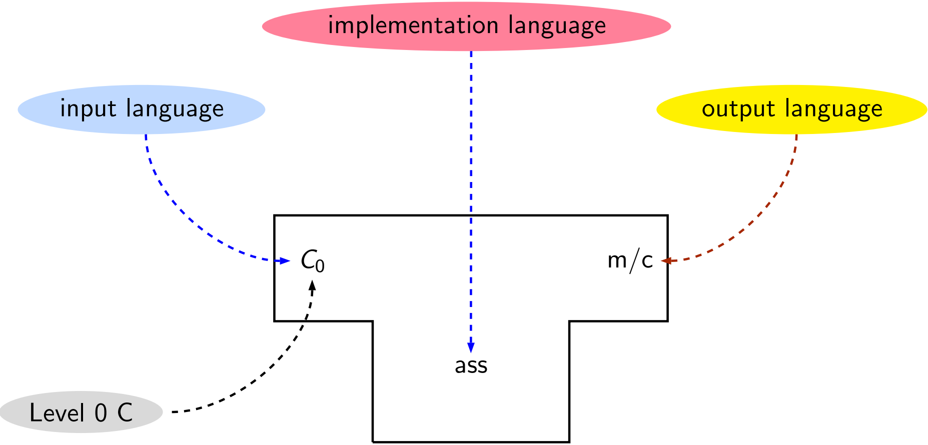
Bootstrapping: The Conventional View



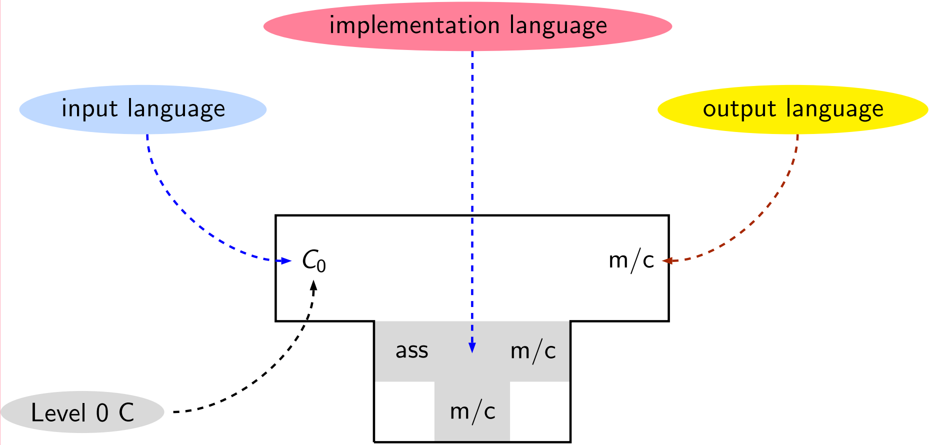
Bootstrapping: The Conventional View



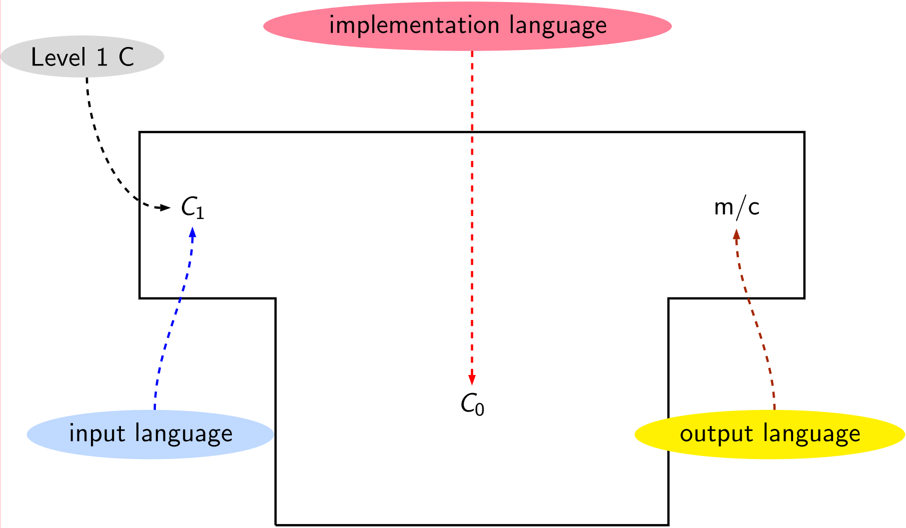
Bootstrapping: The Conventional View



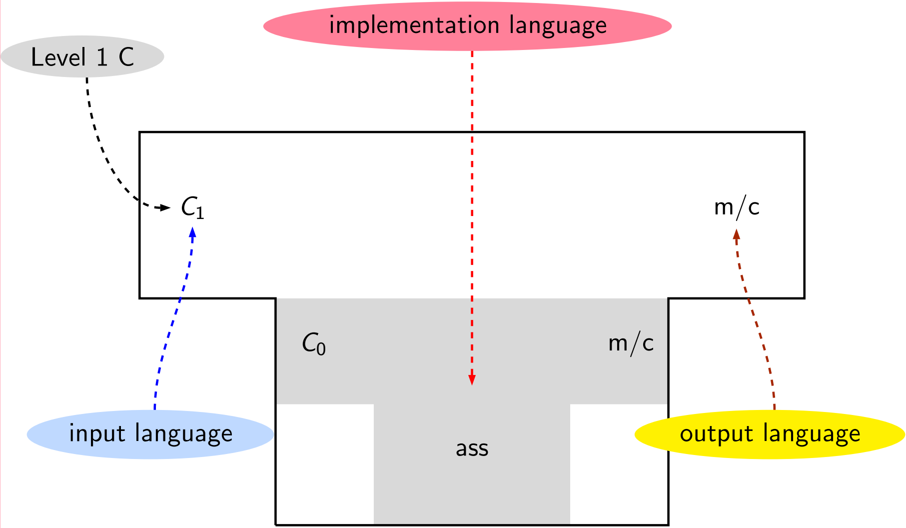
Bootstrapping: The Conventional View



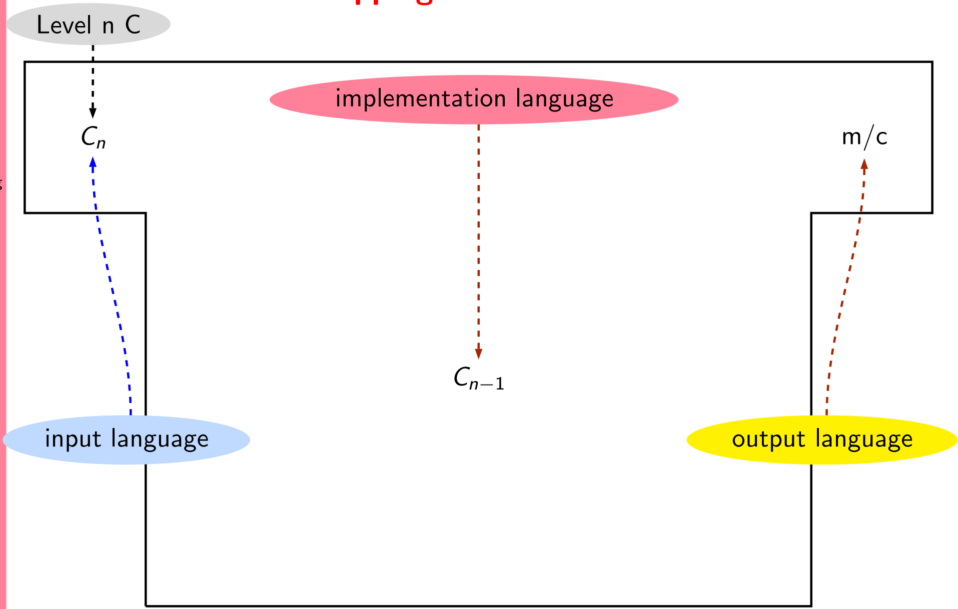
Bootstrapping: The Conventional View



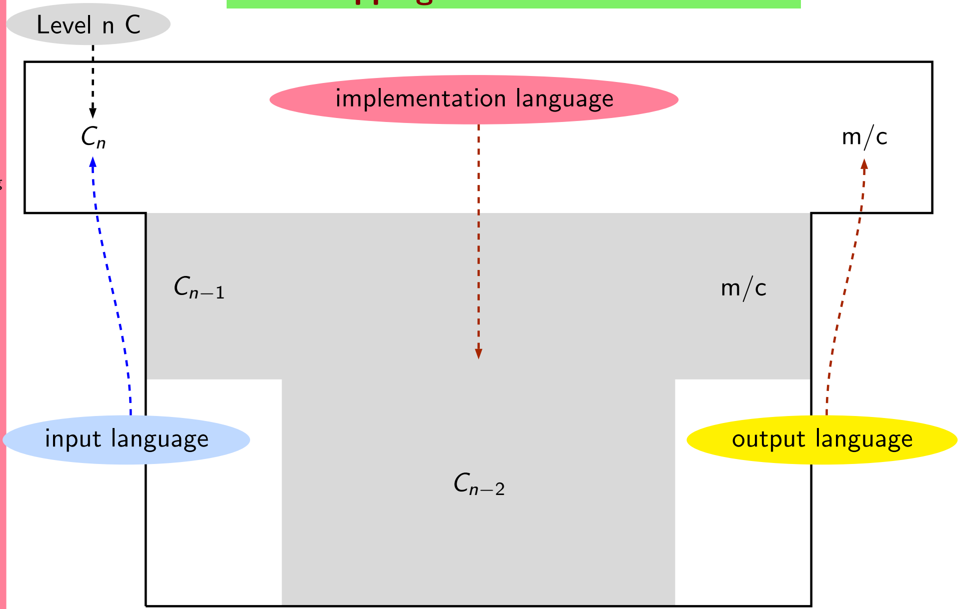
Bootstrapping: The Conventional View



Bootstrapping: The Conventional View



Bootstrapping: The Conventional View



Bootstrapping: GCC View

- Language need not change, but the compiler may change
  Compiler is improved, bugs are fixed and newer versions are released
- To build a new version of a compiler given a built old version:
  - Stage 1: Build the new compiler using the old compiler
  - Stage 2: Build another new compiler using compiler from stage 1
  - Stage 3: Build another new compiler using compiler from stage 2

    Stage 2 and stage 3 builds must result in identical compilers

$\Rightarrow$ Building cross compilers stops after Stage 1!

Modern Challenges

The Sources of New Challenges

- Languages have changed significantly
- Processors have changed significantly
- Problem sizes have changed significantly
- Expectations have changed significantly
- Analysis techniques have changed significantly

The Sources of New Challenges

- Languages have changed significantly
  - "The worst thing that has happened to Computer Science is C because it brought pointers with it." (Frances Allen, IITK, 2007)
- Processors have changed significantly
- Problem sizes have changed significantly
- Expectations have changed significantly
- Analysis techniques have changed significantly

The Sources of New Challenges

- Languages have changed significantly
  - "The worst thing that has happened to Computer Science is C because it brought pointers with it." (Frances Allen, IITK, 2007)
- Processors have changed significantly
  - GPUs, Many core processors, Embedded processors
- Problem sizes have changed significantly
- Expectations have changed significantly
- Analysis techniques have changed significantly

The Sources of New Challenges

- Languages have changed significantly
  - "The worst thing that has happened to Computer Science is C because it brought pointers with it." (Frances Allen, IITK, 2007)
- Processors have changed significantly
  - GPUs, Many core processors, Embedded processors
- Problem sizes have changed significantly
  - Programs running in millions of lines of code
- Expectations have changed significantly
- Analysis techniques have changed significantly

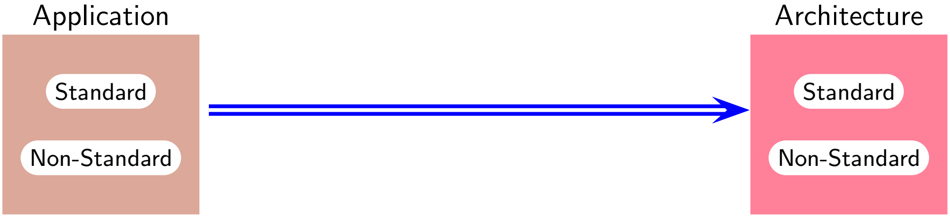
The Sources of New Challenges

- Languages have changed significantly
  - "The worst thing that has happened to Computer Science is C because it brought pointers with it." (Frances Allen, IITK, 2007)
- Processors have changed significantly
  - GPUs, Many core processors, Embedded processors
- Problem sizes have changed significantly
  - Programs running in millions of lines of code
- Expectations have changed significantly
  - Interprocedural analysis and optimization, validation, reverse engineering, parallelization
- Analysis techniques have changed significantly

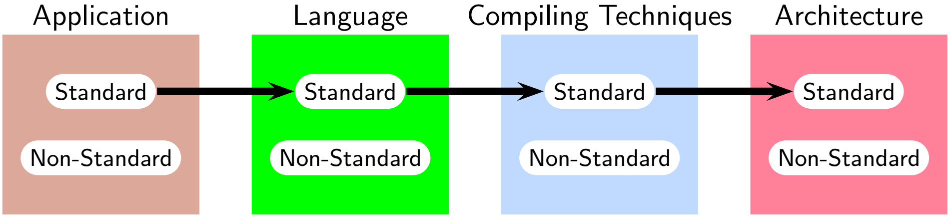
The Sources of New Challenges

- Languages have changed significantly
  - "The worst thing that has happened to Computer Science is C because it brought pointers with it." (Frances Allen, IITK, 2007)
- Processors have changed significantly
  - GPUs, Many core processors, Embedded processors
- Problem sizes have changed significantly
  - Programs running in millions of lines of code
- Expectations have changed significantly
  - Interprocedural analysis and optimization, validation, reverse engineering, parallelization
- Analysis techniques have changed significantly
  - Parsing, Data flow analysis, Parallelism Discovery, Heap Analysis

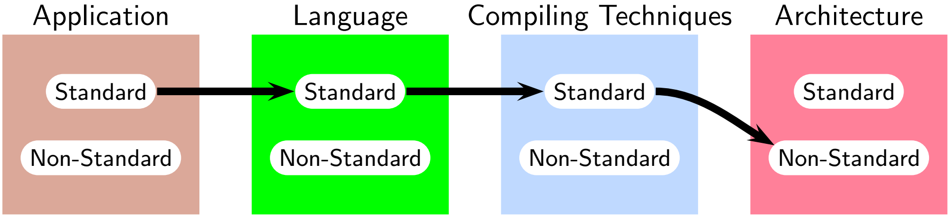
Compilation for Embedded Processors



Compilation for Embedded Processors

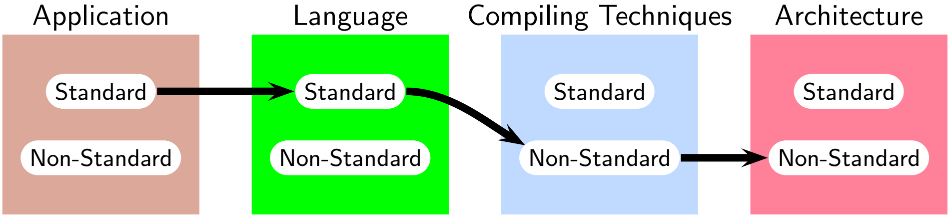


Compilation for Embedded Processors



- Special addressing modes (viz. on-chip addressable memory)
- Use of predicated instructions

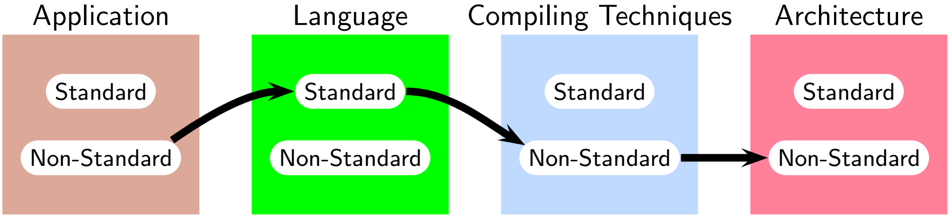
Compilation for Embedded Processors



- SIMD operations, Extracting ILP for VLIW
- Offset assignment, Array reference allocation

SIMD operations: Single Instruction, Multiple Data (SIMD) executes the same operation on multiple data elements in parallel, improving performance in vectorized computations.
Extracting ILP for VLIW: Identifies and schedules independent instructions in Very Long Instruction Word (VLIW) architectures to maximize parallel execution.
Offset assignment: Optimally assigns memory offsets to variables in register-based architectures to minimize address computation overhead.
Array reference allocation: Efficiently maps array references to memory or registers to optimize access patterns and reduce memory bottlenecks.

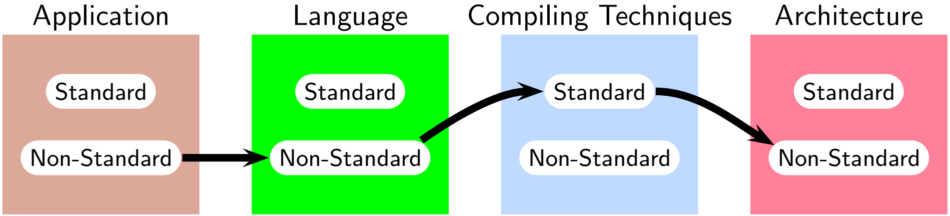
Compilation for Embedded Processors



- MACs, Special loop instructions

MACs (Multiply-Accumulate Operations): Perform multiplication and addition in a single instruction, commonly used in DSP and neural network computations.
Special Loop Instructions: Hardware-supported instructions that optimize loop execution by reducing overhead, such as loop unrolling or zero-overhead looping.

Compilation for Embedded Processors



- Setting arithmetic modes, circular addressing, special loop instructions

Modern Challenges: Design issues

- The IR interface

  What to export? What to hide?

  The most challenging component to design and implement in a compiler is the IR handler

- Retargetability

  Extending to the new version of a processor?

  Extending to a new processor?

Modern Challenges: Improving Performance of Programs

- Scaling analysis to large programs without losing precision
  - Interprocedural analysis
  - Pointer analysis

Modern Challenges: Improving Performance of Programs

- Scaling analysis to large programs without losing precision
  - Interprocedural analysis
  - Pointer analysis
- Increasing the precision of analysis
  - How to interleave different analyses to benefit from each other?
  - How to exclude infeasible interprocedural paths?

Modern Challenges: Improving Performance of Programs

- Scaling analysis to large programs without losing precision
  - Interprocedural analysis
  - Pointer analysis
- Increasing the precision of analysis
  - How to interleave different analyses to benefit from each other?
  - How to exclude infeasible interprocedural paths?
- Combining static and dynamic analysis
  - Using statically computed information for optimization at run time
  - Using run time information for improving optimizations in the next compilation
    (Profile guided optimization aka feedback driven optimization)

Modern Challenges: Improving Performance of Programs

- Scaling analysis to large programs without losing precision
  - Interprocedural analysis
  - Pointer analysis
- Increasing the precision of analysis
  - How to interleave different analyses to benefit from each other?
  - How to exclude infeasible interprocedural paths?
- Combining static and dynamic analysis
  - Using statically computed information for optimization at run time
  - Using run time information for improving optimizations in the next compilation
    (Profile guided optimization aka feedback driven optimization)
- Inventing more effective optimizations

Modern Challenges: Improving Performance of Programs

- Scaling analysis to large programs without losing precision
  - Interprocedural analysis
- I
- C
  …ne
  …xt
  (Profile guided optimization aka feedback driven optimization)
- Inventing more effective optimizations

Full Employment Guarantee Theorem for Compiler Writers

(https://en.wikipedia.org/wiki/Full_employment_theorem)

The notion of "best" compiler cannot exist and there is endless scope to keep improving

⇒ For every compiler, a better compiler can be written

Modern Challenges: Language Issues

- How to efficiently compile
    - Dynamic features such as closures, higher order functions (eg. eval in Javascript)
    - Exceptions
- What guarantees to give in the presence of undefined behaviour
    - Memory accesses such as array access out of bound
- Designing analyses for features supporting parallelism
    - Doall, Async, Threads, Synchronization, Fork/Join, Lock/Unlock, Mutex, Semaphores
    - Some features enable parallelism in a sequential language whereas some enforce sequentiality on essentially parallel execution
- Designing analyses for extracting parallelism

Modern Challenges: Target Machine Issues

How to exploit

- Pipelines? (Spectre/Meltdown attacks)
- Multiple execution units (pipelined)
- Cache hierarchy
- Parallel processing
  (Shared memory, distributed memory, message-passing)
- Vector operations
- VLIW and Superscalar instruction issue

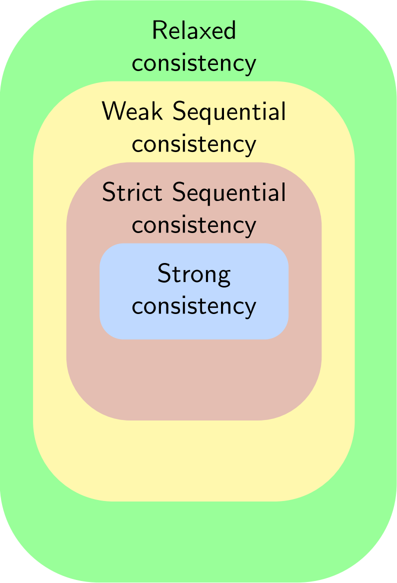
General strategy: Hardware software co-design

Modern Challenges: Target Machine Issues

The crux of the matter

- Hardware is parallel, (conventional) software is sequential
  - Software view of memory model: Strong consistency
    Every execution with the same input should give the same result
  - Hardware view of memory model: Sequential consistency
    Result should coincide with some interleaving of threads
    (Parallelism at the granularity of instructions in threads)
  - Modern architectures gives weak consistency
    (Parallelism at the granularity of pipeline units of instructions, e.g., load/store buffering)
- Software view is stable, hardware is disruptive

Classes of Memory Models



Model	Data dependence across threads	Data dependence within threads
SC	Preserved	Preserved
SSC	Not preserved	Preserved
WSC	Not preserved	Not preserved (no reordering but writes may not be available)
RC	Not preserved	Not preserved (reordering of instructions)

Architecture Feature Influencing Programming Language

A concurrent program

Initially $X = Y = 0$

$a = X$	$\|\|$	$b = Y$
$Y = 1$		if(b) $X = 1$

- Variables a and b are thread-local variables
- Variables X and Y are shared global variables

Architecture Feature Influencing Programming Language

A concurrent program

Initially $X = Y = 0$

Thread 1		Thread 2
$a = X$	$\parallel$	$b = Y$
$Y = 1$		$\text{if}(b)\ X = 1$

- Variables a and b are thread-local variables
- Variables X and Y are shared global variables

Sequential Consistency preserves program order

$a = X$	$b = Y$	$b = Y$	$b = Y$	$a = X$	$a = X$
$Y = 1$	$b?\ X = 1$	$a = X$	$a = X$	$b = Y$	$b = Y$
$b = Y$	$a = X$	$b?\ X = 1$	$Y = 1$	$b?\ X = 1$	$Y = 1$
$b?\ X = 1$	$Y = 1$	$Y = 1$	$b?\ X = 1$	$Y = 1$	$b?\ X = 1$
$a = 0, b = 1$	$a = b = 0$	$a = b = 0$	$a = b = 0$	$a = b = 0$	$a = b = 0$

Architecture Feature Influencing Programming Language

A concurrent program

To see the difference between sequential consistency and strict consistency, consider the following interleaving of two threads

$a = 0$
$a = 1$
print a
print a

- Strict sequential consistency. Result of writes are available instantaneously

  Both prints of a must read the value 1

  $\Rightarrow$ Only one result is possible: 1, 1

- Weak sequential consistency. Result of writes may be available with a delay

  The first print of a may read 0

  $\Rightarrow$ Both 0, 1 and 1,1 are possible

Architecture Feature Influencing Programming Language

A concurrent program

Initially $X = Y = 0$

$a = X$	$\|\|$	$b = Y$
$Y = 1$		$\text{if}(b)\; X = 1$

- Variables a and b are thread-local variables
- Variables X and Y are shared global variables

Relaxed Memory Consistency allows violating program order

$Y = 1$
$b = Y$
$b?\; X = 1$
$a = X$
$a = b = 1$

- Order of assignments in the first thread can be interchanged
  No thread-local data dependence
- Supported by out-of-order execution in processors restricted to a local view of the threads
- Being pushed in C standard in spite of the fact that it is difficult to understand for a programmer

Architecture Feature Influencing Programming Language

A concurrent program

Initially $X = Y = 0$

$a = X$	$\|\|$	$b = Y$
$Y = 1$		if(b …

- Variables a and b are thread-local variables
- … global variables

Relaxed … m order

$Y = 1$
$b = Y$
$b?\ X = 1$
$a = X$
$a = b = 1$

- … be interchanged
- Supported by out-of-order execution in processors restricted to a local view of the threads
- Being pushed in C standard in spite of the fact that it is difficult to understand for a programmer

Why is this useful?

Out of order execution offers more opportunities of keeping the pipeline full, thereby increasing the throughput

Modern Challenges: Providing Guarantees

- Correctness of optimizations
  - Hard even for machine independent optimizations
  - Verification of a production optimizing compiler is a pipe dream
    Requires proving the correctness of translation of ALL programs
  - Compiler validation is more realistic, and yet not achieved fully
    Allows proving the correctness of translation of A program
- Interference with Security
  - Optimizations disrupt memory view
    Correctness is defined in terms of useful states
    Clearing stack location by writing all zeros is dead code
  - Optimizations also disrupt timing estimates

Compiler Verification

Formalize and verify the following diagram for every source program P

$$
\begin{tikzcd}
\mathbf{P} \arrow[rr] \arrow[d] & & \mathit{comp}(\mathbf{P}) \arrow[d] \\
\mathit{sMean}(\mathbf{P}) & & \mathit{tMean}(\mathit{comp}(\mathbf{P})) \arrow[ll, "\text{abstract}"']
\end{tikzcd}
$$

comp represents the transformation performed by

- a compiler (harder problem), or
- a model of the compiler (easier)

  Is the model faithful?

Credits: Adapted from the slides of Prof. Amitabha Sanyal, IIT Bombay

Compiler Verification

Formalize and verify the following diagram for every source program P

smean (Simple Mean): The arithmetic mean, calculated as the sum of values divided by the total count.
tmean (Trimmed Mean): A mean calculated after removing a certain percentage of the highest and lowest values to reduce the effect of outliers.

```
y = 5
x = y + 1
```

```
li $t0, 5
sw $t0, 4($fp)
lw $t0, 4($fp)
addi $t0, 1
sw $t0, 8($fp)
```

$$
\begin{tikzcd}
\mathbf{P} \arrow[r] \arrow[d] & \mathit{comp}(\mathbf{P}) \arrow[d] \\
\mathit{sMean}(\mathbf{P}) & \mathit{tMean}(\mathit{comp}(\mathbf{P})) \arrow[l, "\text{abstract}"']
\end{tikzcd}
$$

```
y = 5
x = 6
```

```
M[4($fp)] = 5
M[8($fp)] = 6
```

comp represents the transformation performed by

- a compiler (harder problem), or
- a model of the compiler (easier)

  Is the model faithful?

Credits: Adapted from the slides of Prof. Amitabha Sanyal, IIT Bombay

Difficulties in Compiler Verification

- Complexity
  - Requires reasoning about actual compiler implementation.
  - Requires reasoning about the behaviour of the compiler for an infinite number of programs and their translations.
- Automation - unlikely
- Proof reuse?

Credits: Adapted from the slides of Prof. Amitabha Sanyal, IIT Bombay

Translation Validation

Formalize and verify the following diagram for a given source program P

$$\begin{tikzcd}
\mathbf{P} \arrow[r] \arrow[d] & \mathit{comp}(\mathbf{P}) \arrow[d] \\
\mathit{sMean}(\mathbf{P}) & \mathit{tMean}(\mathit{comp}(\mathbf{P})) \arrow[l, "\text{abstract}"']
\end{tikzcd}$$

comp represents the transformation performed by

- a compiler (harder problem), or
- a model of the compiler (easier)

  Is the model faithful?

Credits: Adapted from the slides of Prof. Amitabha Sanyal, IIT Bombay

Translation Validation

Formalize and verify the following diagram for a given source program P

y = 5
x = y + 1

$$\begin{tikzcd} \mathbf{P} \arrow[rr] \arrow[d] & & \mathit{comp}(\mathbf{P}) \arrow[d] \\ \mathit{sMean}(\mathbf{P}) & & \mathit{tMean}(\mathit{comp}(\mathbf{P})) \arrow[ll, "\text{abstract}"] \end{tikzcd}$$

```
li $t0, 5
sw $t0, 4($fp)
lw $t0, 4($fp)
addi $t0, 1
sw $t0, 8($fp)
```

y = 5
x = 6

M[4($fp)] = 5
M[8($fp)] = 6

comp represents the transformation performed by

- a compiler (harder problem), or
- a model of the compiler (easier)

  Is the model faithful?

Credits: Adapted from the slides of Prof. Amitabha Sanyal, IIT Bombay

Translation Validation

- Less complex
  - Involves reasoning about a given pair of programs
  - The compiler can be made to provide information to help verification.
- Automation - likely.

Credits: Adapted from the slides of Prof. Amitabha Sanyal, IIT Bombay

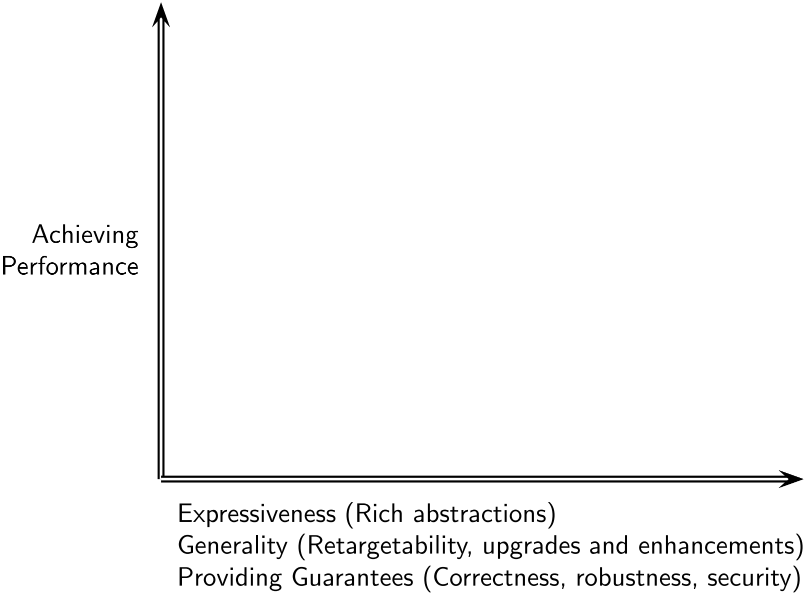
Modern Challenges: New Expectations

- New application domains bringing new challenges
- What are the underlying abstractions of the domains that should become first class citizens in a programming language?
  - Language design and compilers for machine learning algorithms?
  - Language design and compilers for streaming applications?
- Can machine learning algorithms help compilers create new optimizations?
  - Can human ingenuity in design of novel algorithms be replaced by machine learning?
    Need explainability for guaranteeing soundness of new optimizations
    Known cost based optimizations have a better chance with machine learning
  - Can compilers learn from the programs they have compiled and become "better" over time?

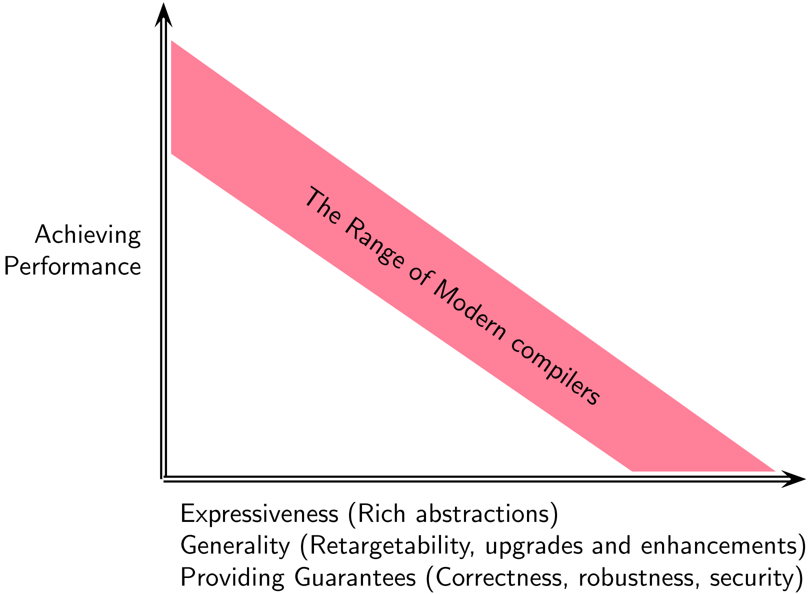
Modern Challenges: New non-PL Technologies

- Can we use LLMs to scale program analysis and optimizations?
  - It may be possible to use LLMs to analyse smaller chunks of code
  - Some light-weight analysis can be used to connect the results of analysis of two different chunks of code

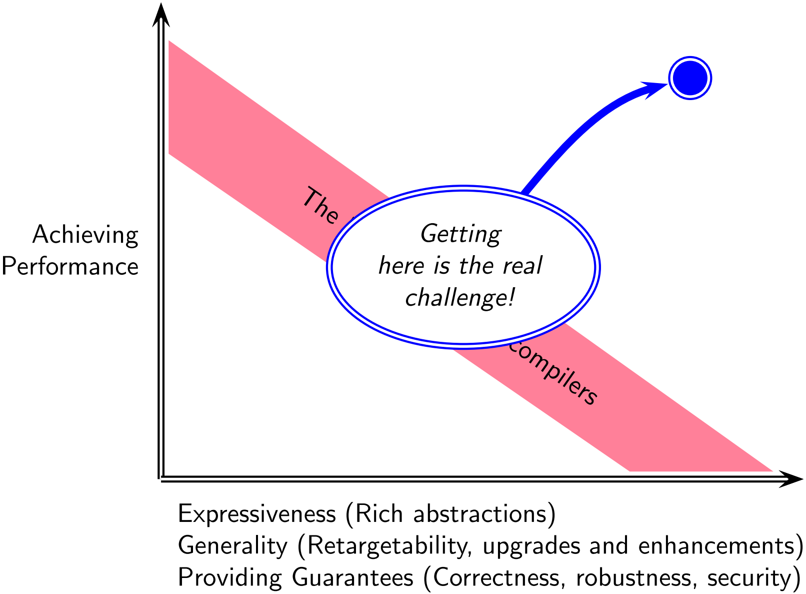
The Moral of the Story



The Moral of the Story

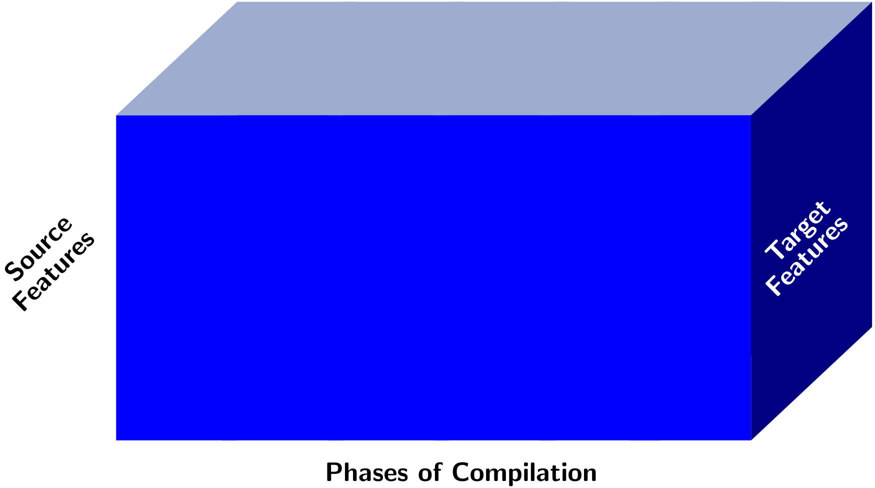


The Moral of the Story

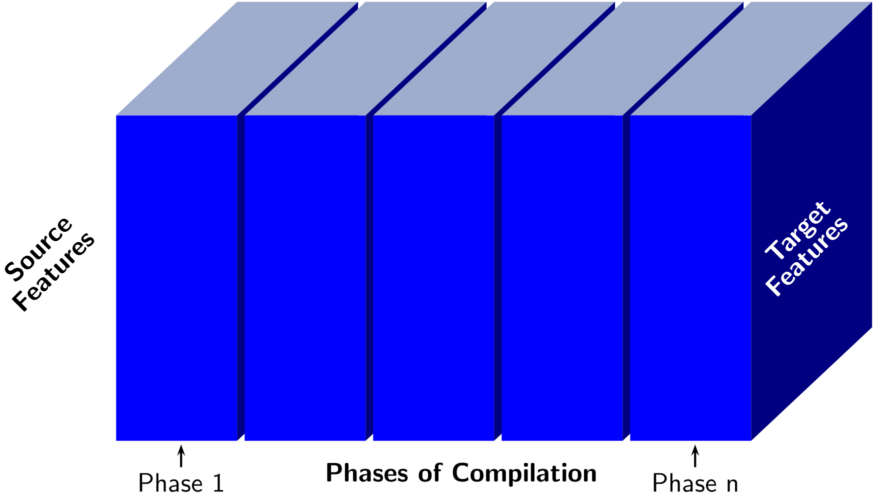


Incremental Construction of Compilers

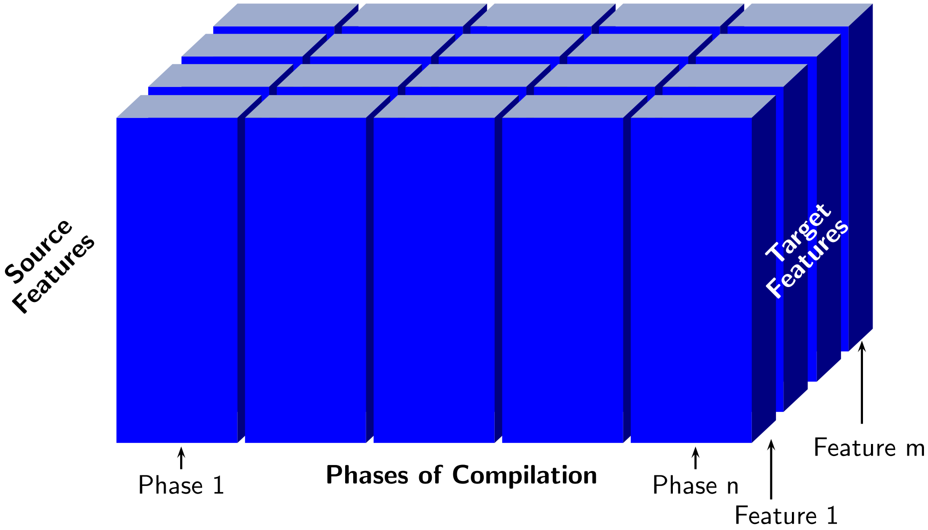
In Search of Modularity in Compilation



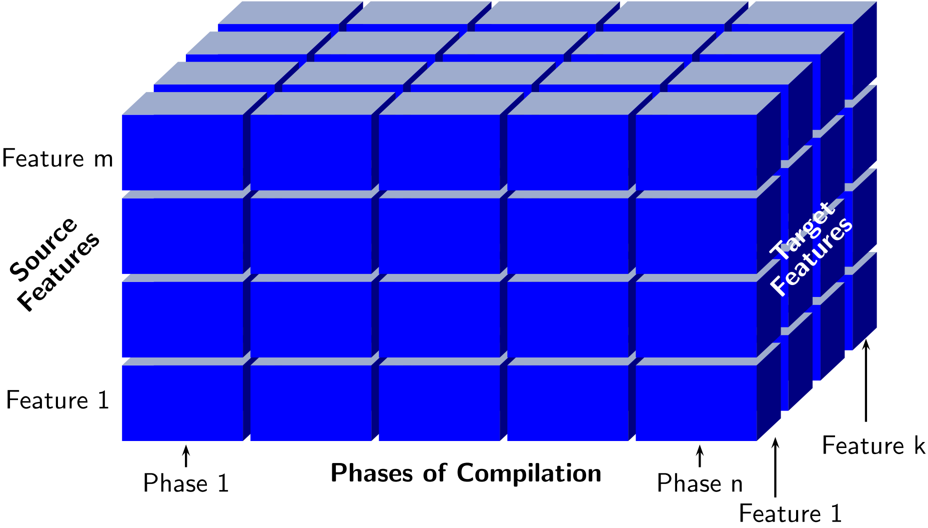
In Search of Modularity in Compilation



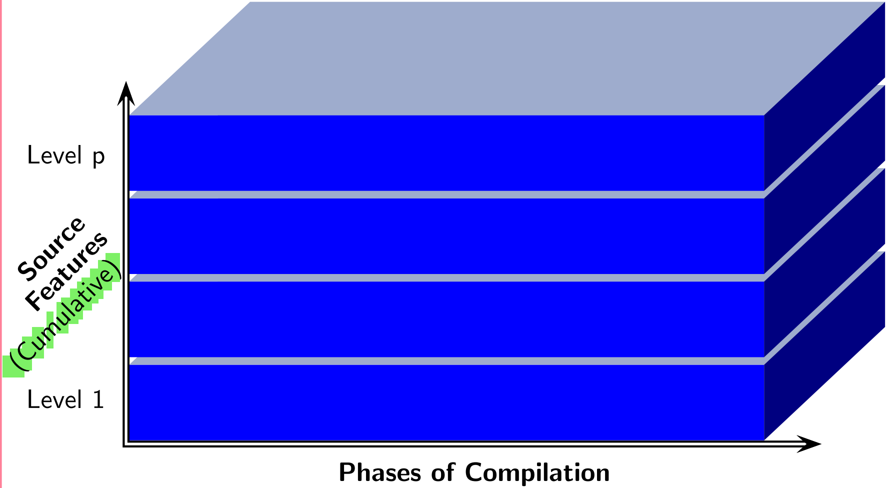
In Search of Modularity in Compilation



In Search of Modularity in Compilation



In Search of Modularity in Compilation



Language Increments

Assignments with simple RHS

Level 1

Language Increments

Arithmetic Expressions

Assignments with simple RHS

Level 1

Level 2

Language Increments

Comparison and Logical Expressions

Arithmetic Expressions

Assignments with simple RHS

Level 1

Level 2

Level 3

Language Increments

Control Structures

Comparison and Logical Expressions

Arithmetic Expressions

Assignments with simple RHS

Level 1

Level 2

Level 3

Level 4

Language Increments

Function Calls (including recursion)

Control Structures

Comparison and Logical Expressions

Arithmetic Expressions

Assignments with simple RHS

Level 1

Level 2

Level 3

Level 4

Level 5

Language Increments

Pointers and Arrays

Function Calls (including recursion)

Control Structures

Comparison and Logical Expressions

Arithmetic Expressions

Assignments with simple RHS

Level 1

Level 2

Level 3

Level 4

Level 5

Level 6

Language Increments

Pointers and Arrays

Function Calls (including recursion)

Control Structures

Comparison and Logical Expressions

Arithmetic Expressions

Classes, structures, and unions will be included in subsequent years

Level 3

Level 4

Level 5

Level 6

Course Plan

cs320 Coverage and Pedagogy

- Coverage
  - Scanning, Parsing, Static Semantics, Runtime Support, Code Generation
  - Code Optimization, Register Allocation (may be omitted)
- Pedagogy
  - Lectures
  - Slides will be made available on moodle
  - Asynchronous discussions on moodle discussion forum
  - Tutorial problems on moodle
- Evaluation
  - Two quizzes, mid sem, end sem, and class participation

cs306 Coverage

- Coverage
  - Incremental construction of SCLP
  - Reference implementation with some test cases will be provided
- Pedagogy
  - Five assignments common to all students
    - Implementation of additional features may fetch bonus credit
    - Independent projects may be allowed to replace some of the assignments
  - Roughly two weeks per submission
  - Work do be done in a personalized VM in the lab machines or on your laptop
  - Groups of two
- Evaluation will be by running diff on the output
  - Standard file names and directory names must be used
  - It will not be possible to entertain violations
  - Use your creativity inside your code, not in file names, Makefile commands and program output

SCLP Increments

Pointers and Arrays

Function Calls

Control Structures

Comparison and Logical Expressions

Arithmetic Expressions

Assignments with simple RHS

Level 1

Level 2

Level 3

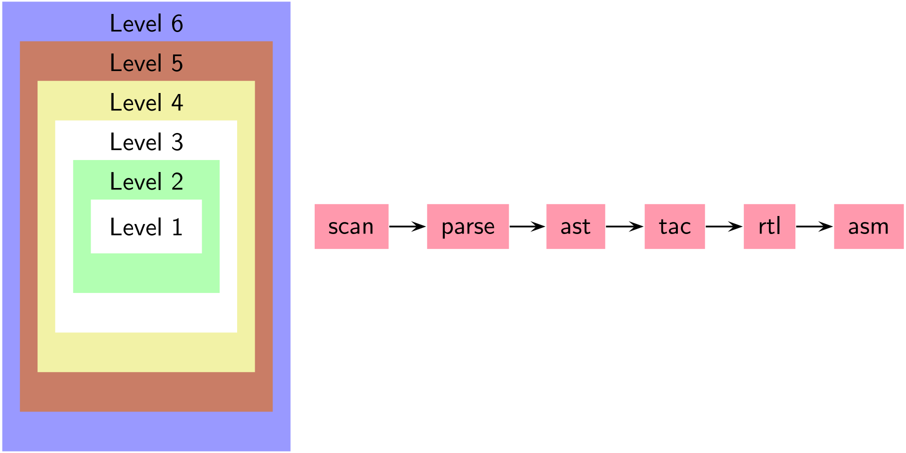
Level 4

Level 5

Level 6

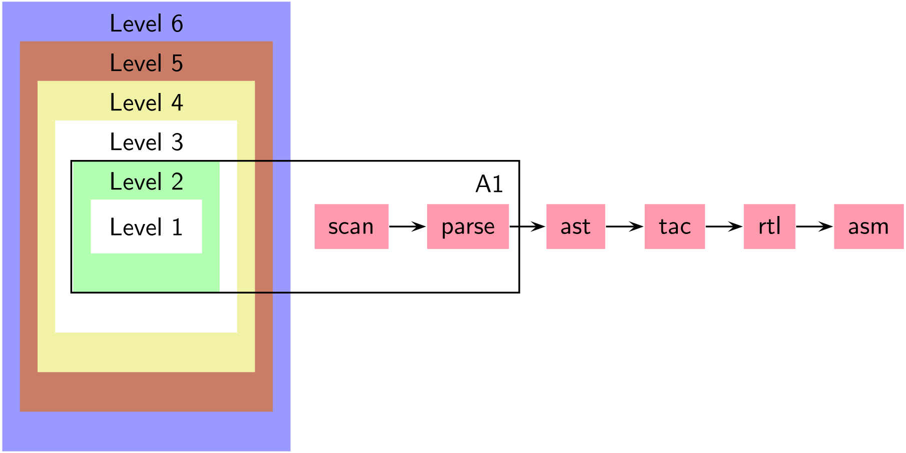
Proposed Assignment Plan: Incremental Construction

A series of assignments; each assignment builds on the previous assignment



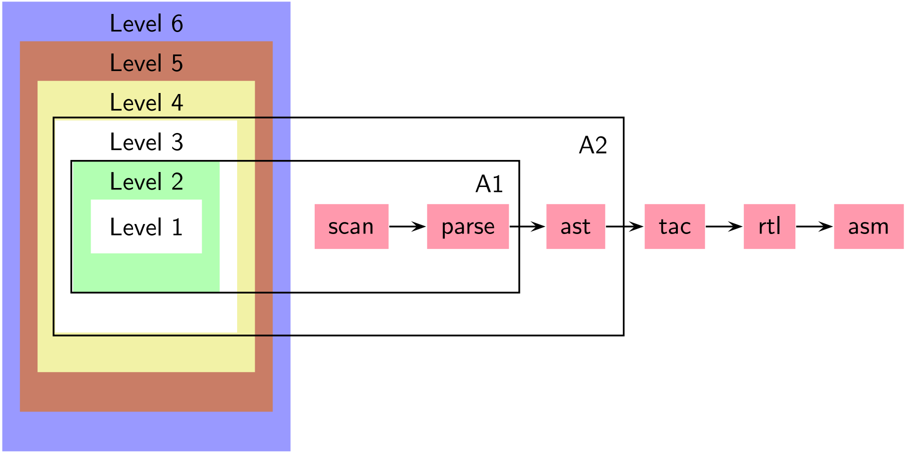
Proposed Assignment Plan: Incremental Construction

A series of assignments; each assignment builds on the previous assignment



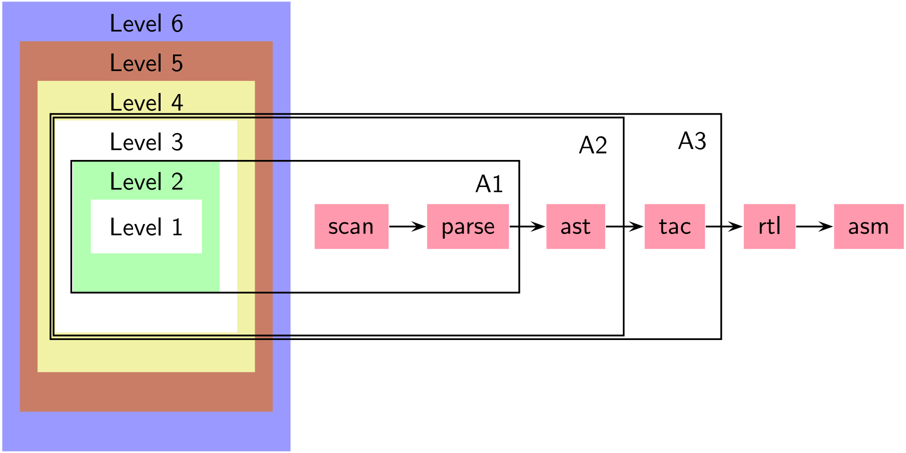
Proposed Assignment Plan: Incremental Construction

A series of assignments; each assignment builds on the previous assignment



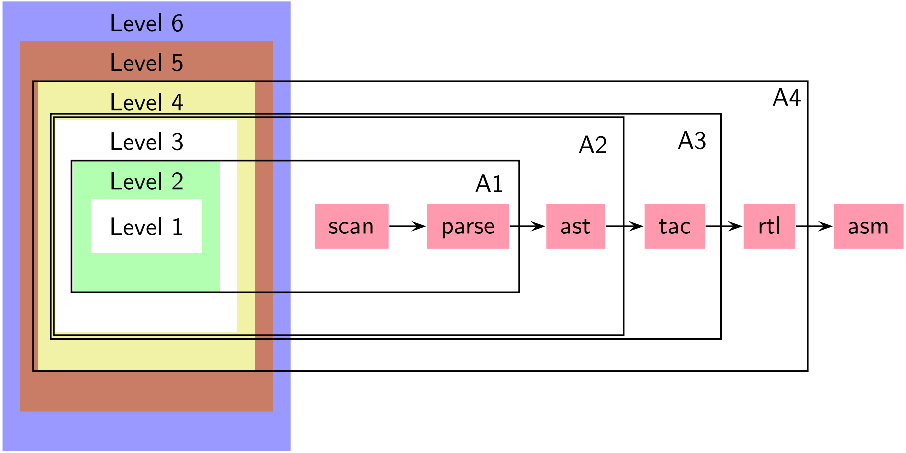
Proposed Assignment Plan: Incremental Construction

A series of assignments; each assignment builds on the previous assignment



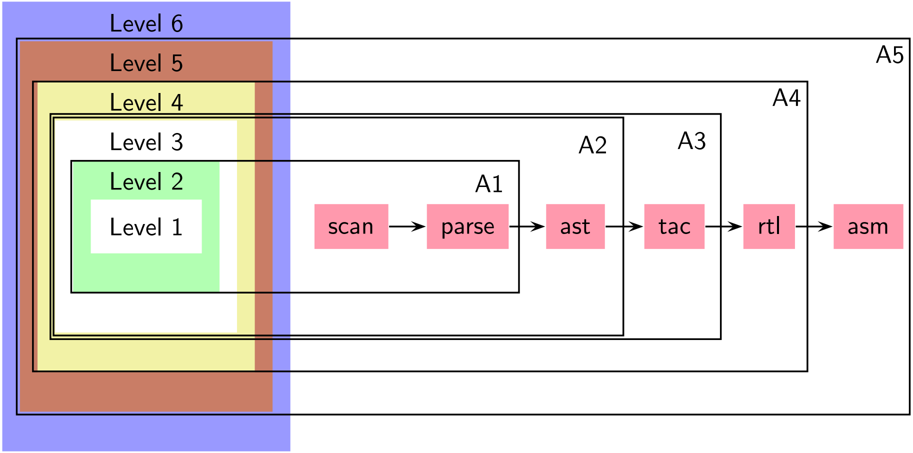
Proposed Assignment Plan: Incremental Construction

A series of assignments; each assignment builds on the previous assignment



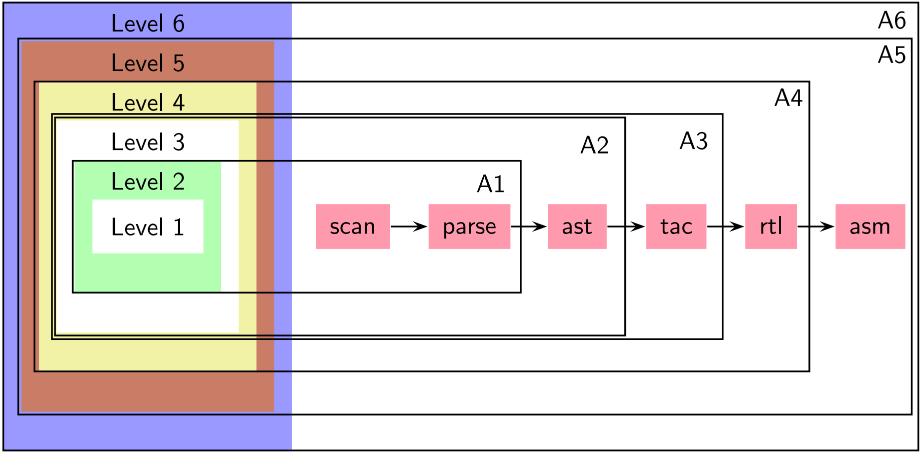
Proposed Assignment Plan: Incremental Construction

A series of assignments; each assignment builds on the previous assignment



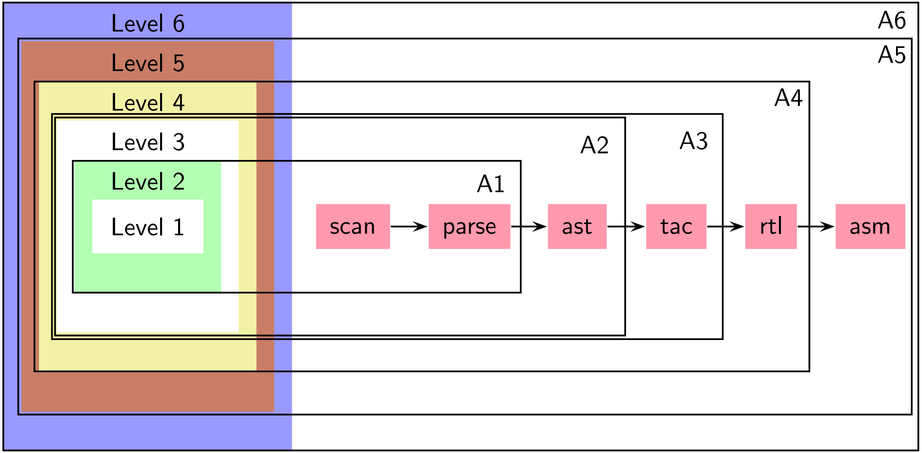
Proposed Assignment Plan: Incremental Construction

A series of assignments; each assignment builds on the previous assignment



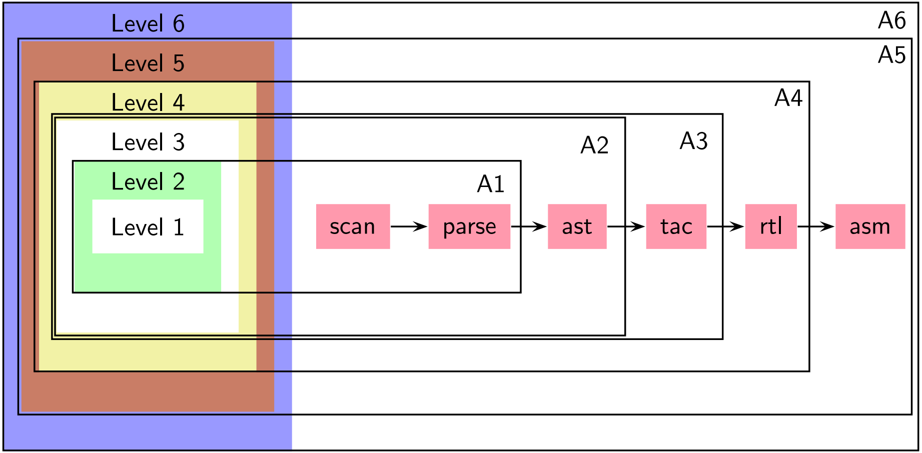
Proposed Assignment Plan: Incremental Construction

A series of assignments; each assignment builds on the previous assignment



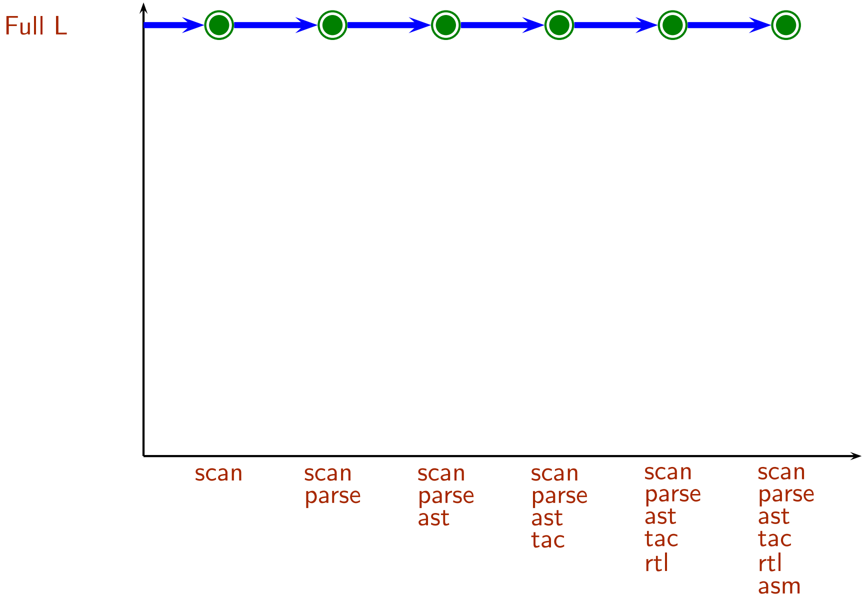
Proposed Assignment Plan: Incremental Construction

A series of assignments; each assignment builds on the previous assignment

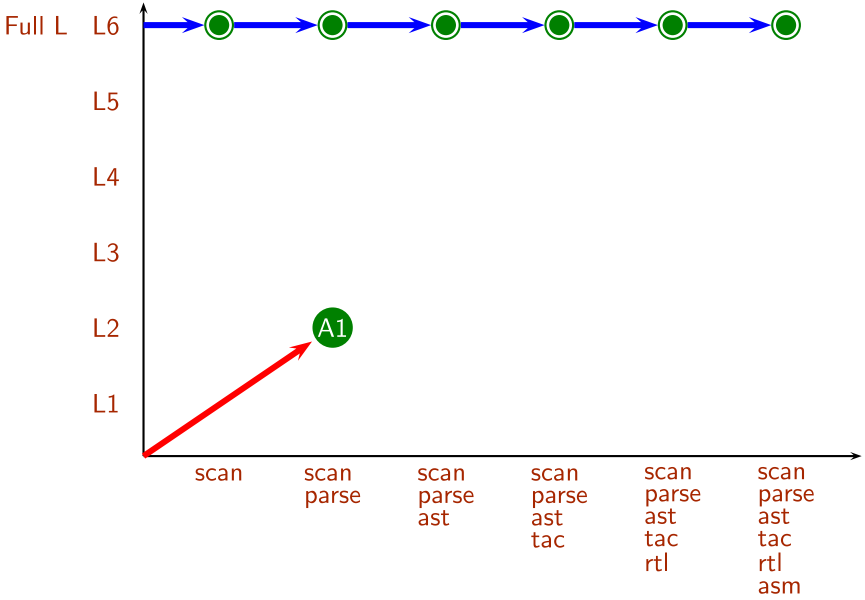


A6 is optional

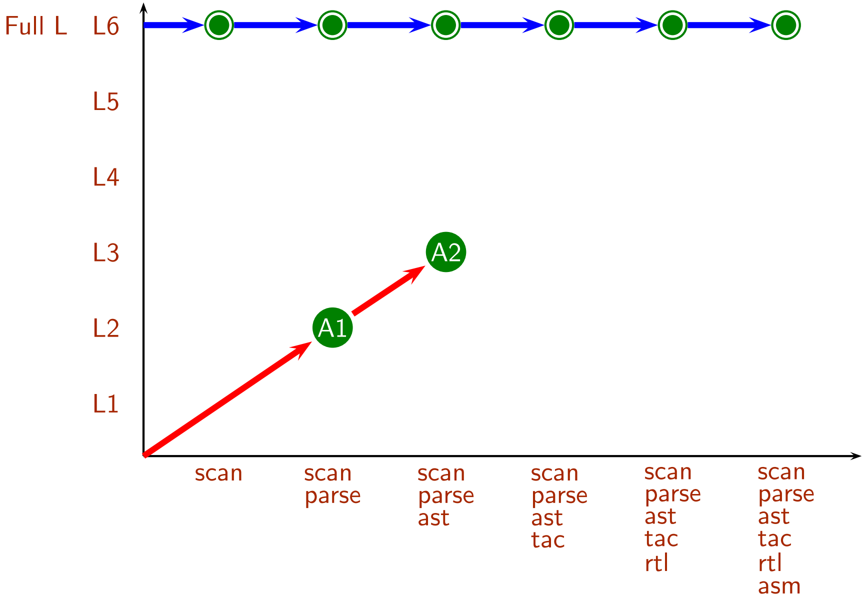
Conventional Increments Vs. Proposed Increments



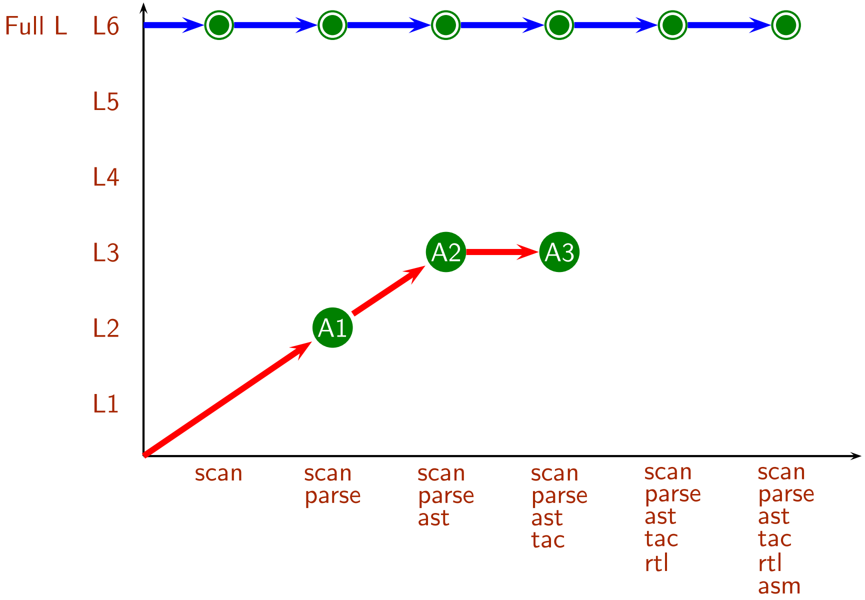
Conventional Increments Vs. Proposed Increments



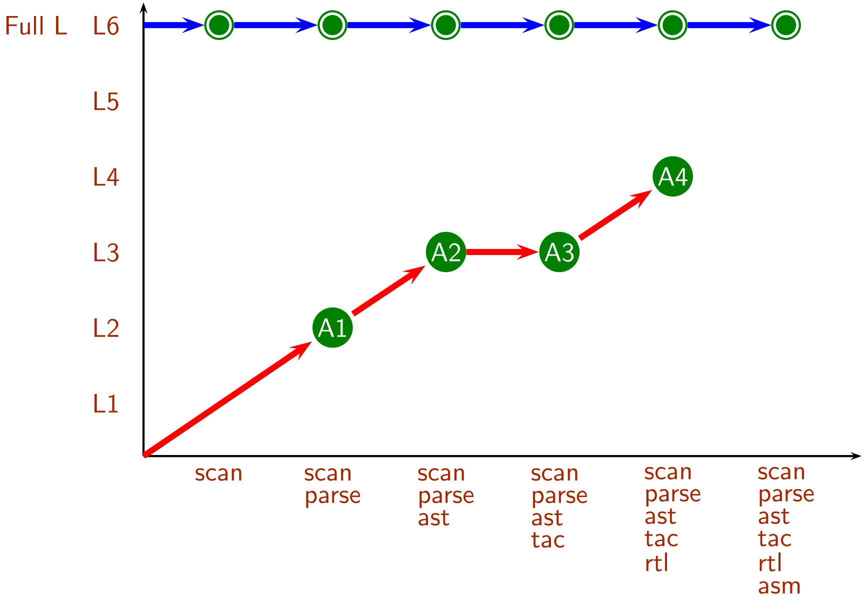
Conventional Increments Vs. Proposed Increments



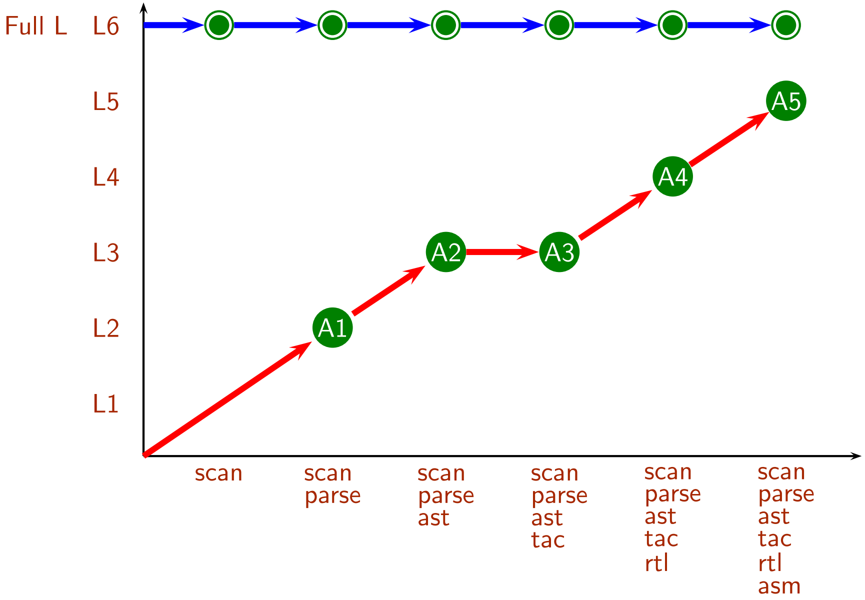
Conventional Increments Vs. Proposed Increments



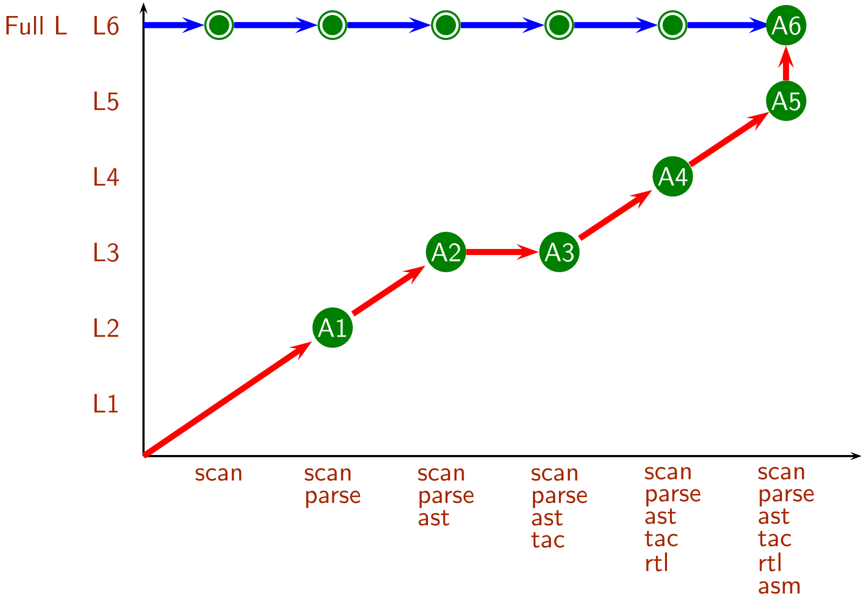
Conventional Increments Vs. Proposed Increments



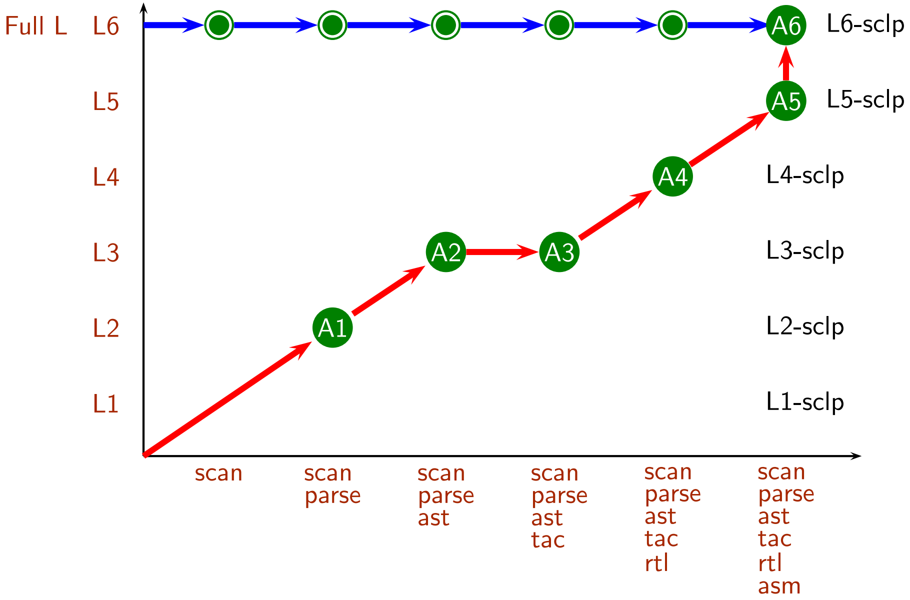
Conventional Increments Vs. Proposed Increments



Conventional Increments Vs. Proposed Increments



Conventional Increments Vs. Proposed Increments



Evaluation Plan and Grading

- cs320

  Mid-Sem (30%) and End-Sem (40%)

  Quizzes (20%)

  Class Participation (10%)

  A viva may be factored in later

- cs306

  Five assignments to be done on coffre (90%)

  Class Participation (10%)

  A viva may be factored in later

- Additional opportunities for students to pass the course

Expectation Management

FAQ (1)

- Q. Can I form a group of 3 for cs306 assignments?

  A. The credit of each student will be proportionately reduced
- Q. Can I form a group of 1 for cs306 assignments?

  A. No, I do not want to increase the number of groups
- Q. Why can't I take my code out of coffre server even at the end of the course?

  Your take away from the course is your learning and not your code. Besides, we have had students publishing their code on public repositories which conflicts with the goals of the course for subsequent batches

FAQ (3)

- Q. I sent a message to you on MS Teams but did not get any response!

  A. I don't use MS Teams

- Q. I sent a message to you on Moodle but did not get any response!

  A. Post queries on moodle forums so they get tracked by multiple people

- Q. What are your office hours?

  A. In my experience, office hours have not been used by students in past. So instead of blocking my time with no takers, I prefer to have the freedom to schedule other activities/meetings

  However, if many students feel that they would like to use my time in office hours, I don't mind blocking my time

  Contact your CRs and I will coordinate with them

FAQ (4)

- Q. How do we communicate with you outside of the class?

  A. Talk me just after the class, minimize the email communication

  For issues that are likely to be of common interest, please write on moodle discussion forum

  For issues of personal interests rather than common interests, send an email to uday@cse.iitb.ac.in with a copy to nisha@cse.iitb.ac.in

  The subject header of every email sent to me must contain the word "cs320", "cs302", "cs306", or "cs316"; otherwise the mail may be mis-classified and may not be attended to for a long time

FAQ (5)

- Q. Is attendance compulsory?

  A. Yes, if you need my time and attention. No, otherwise.

  If your attendance is less than 80%, your emails, questions, queries, grievances about evaluation and marks, will be ignored totally. If you need to talk to me for *anything* you must attend the classes.

  Both attendance and participation is highly desirable. Participation can be in the form of asking questions in the class, discussions in meetings, on moodle forums, pointing out bugs in the reference implementation, responding to the posts made by others etc.

  In the absence of any such participation, I will have to use attendance as a proxy

- Q. Why do you insist on students attending and participating? Why can't we learn independently?

  A. Studying is like evaluating expressions that have lots of free variables. Regular participation ensures that these variables are bound to right values

  Familiarity with the notations and conventions used in the class simplifies understanding and evaluation and makes grades consistent and reliable

Things that I Discourage and Disapprove

- Asking questions only in the last two days before exams

  I will not provide clarifications two days before the exam

- Direct communication with the TAs about policies, evaluations, requests for extensions, or marks

  All communication must be with me. TAs are not authorized to respond to you directly on these matters without my explicit permission to do so

- Expecting instantaneous responses, making a request at the last moment

  Email responses will be provided in batch mode. And there may be no response to some emails. Please do not remind me

- Trying to push your luck by assuming that requests for concessions can only give benefit but no harm

  I will listen sympathetically, and may explain the rationale behind my decision but will not allow persuasions nor will try to convince you

- Feigning ignorance about the policies that have been described clearly in the class, slides, emails, or moodle announcements

  I have no sympathy for such students—sorry, bad luck!

Further Expectation Management

- Things that I like and strongly encourage
  - Asking questions, reporting bugs in reference implementation
  - Free, frank, and respectful communication
- Thing that I detest and despise
  - Cheating
  - Grade Litigation

Cheating

- Collaboration in learning or discussions about your code are fine
  But your answers in exam and your code in submit MUST BE YOURS
- No compromises on it
- Cheating is also a way of denying yourself an opportunity of learning
- Would you advice your younger siblings, nephews, nieces, and your children to copy?
- If you still want to compromise on your integrity, don't even think of doing so in cs320 and cs306

Pre-empting Cheating

- cs320 exams will be designed in a manner that attending the lectures would be sufficient to pass them
- Getting high marks in two quizzes and mid-sem will guarantee passing even if you get a zero in the end-sem
- Getting full marks in first two assignments in cs306 will guarantee passing even if you do not submit the remaining assignments
- Your lab work will be in coffre VM
    - Flexibility of doing work at your convenience
    - Can be used on multiple devices
    - You are free to access internet while you work on your programming
    - All programming and submissions within coffre
    - All submissions will undergo through plagiarism checking
- Additional opportunities will be provided to students to pass the courses

Hopefully, I will make it harder for you to fail

Grade Litigation

- Grade Litigation. Requests for accepting incomplete or inaccurate answers, requests to condone errors in answers or implementations, seeking waivers motivated by improving chances of grades

  Persistent demands in the disguise of requests with a sense of entitlement
- Grade litigation is easy to handle for small classes but not for large classes
- My take motivated by consistency and reliability of grades in large classes
  - Marks are given for rigorous demonstration of knowledge *and* the skill of demonstrating the knowledge
  - Most students rely on knowledge but ignore the skill part and end up writing vague or incomplete answers
  - You cannot claim marks for indirectly showing that you possess the knowledge to solve a question You need to demonstrate it directly by showing
    - the skill of using the right knowledge, and
    - the diligence of solving the questions using the notations and conventions used in the class

Grade Litigation

- Practically, grade litigation in large classes amounts to a Denial of Service Attack

  Takes the time and energy away from
  - The students who need and deserve my time and attention
  - Improving my lectures, explanation, and course material
  - Designing interesting exams
- I do not take this denial of service attack, kindly

Pre-empting The Grade Litigation for Large Classes: cs320

- By design, the answers will be objective and precise and not subjective
- Answers will be subdivided into parts and there will be no partial marks other than the subdivision by parts
- All answers will be published and corrections, if any, as well as possible valid variations, will be invited before finalizing the marks
- Corrections and acceptable variations in the answers will be published and no requests for consideration of any other answers will be entertained
- Evaluated answer sheets will be made available for you to scan them and go through them at leisure to ensure that the evaluation is consistent with the published answers
- A grievance form will be floated and valid grievances will be addressed (no discussions over email/phone, please)

Pre-empting The Grade Litigation for Large Classes: cs306

- A reference implementation of the compiler with test cases will be provided; you can run it on your own test cases to understand its behaviour
- You can submit the assignments late for reduced credit. Exact details are provided in moodle.
- You will have a total four late days of submission without late penalty through out the semester. You can use them for any assignment. Exact details are provided on moodle
- Full grammar and class hierarchies used in the reference implementation will be provided; you are free to use you own grammar and classes but the output must match the output of the reference implementation
- The scripts and the test cases used for evaluation will be published
- A grievance form will be floated and valid grievances will be addressed (no discussions over email/phone, please)

Are you ready for the fun?