ACM Summer School on Compilers for AI/ML

Uday Khedker

(www.cse.iitb.ac.in/˜uday)

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

January 2024

Outline

Outline

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

Introduction to the School

Coverage: First Week

Day	Topic	Instructor
Monday	Introduction to the school Introduction to compilation Inrtoduction to scanning using lex	Uday Khedker Abhijat Vichare
Tuesday	Introduction to parsing using yacc A compiler and interpreter source	Uday Khedker
Wednesday	Scanning	Manas Thakur
Thursday	Parsing	Jyothi Vedurada
Friday	Semantic analysis	Jyothi Vedurada
Saturday	Compiling function calls	Swati Jaisawl

Coverage: Additional Demos in the First Week

Day	Demo	Instructor
Monday	tcc compiler for AIDSL	Soumik Kumar Basu
Friday	Compiler Explorer	Dhruv Chawla
Friday	GCC IRs	Prathmesh Kulkarni
Friday	LLVM IR	Supriya Bhide

Coverage: Second Week

Day	Lecture Topics	Instructor
Monday	Execution Environment	Girish Bharambe
Tuesday	Execution Environment	Girish Bharambe
Wednesday	Visit to NVIDIA	
Thursday	MLIR, Polyhydral Analysis and Optimization	Uday Reddy
Friday	History of compiling Modern challenges Concluding session	Uday Khedker Abhijat Vichare Ramana Radhakrishnan

Schedule

Time	Activity
09:30 to 11:00	Lecture (and tutorials, as needed)
11:00 to 11:20	Tea break
11:20 to 12:50	Lecture (and tutorials, as needed)
12:50 to 14:00	Lunch break
14:00 to 15:30	Lab (or lecture, as needed)
15:30 to 15:50	Tea break
15:50 to 17:10	Lab (or lecture, as needed)

Pedagogy

- Journey from practice to theory
- You will be given a language, a compiler for its subset, and you will start extending it

  Theory will follow on a need basis
- Plenty of practical work

Teaching Assistants

- Atharva Badve, NVIDIA
- Dhruv Chawla, NVDIA
- Prachi Godbole, NVIDIA
- Prathamesh Kulkarni, NVIDIA
- Soumik Kumar Basu, IIT Hyderabad
- Subhranil Mukherjee, NVIDIA
- Supriya Bhide, IIT Bombay

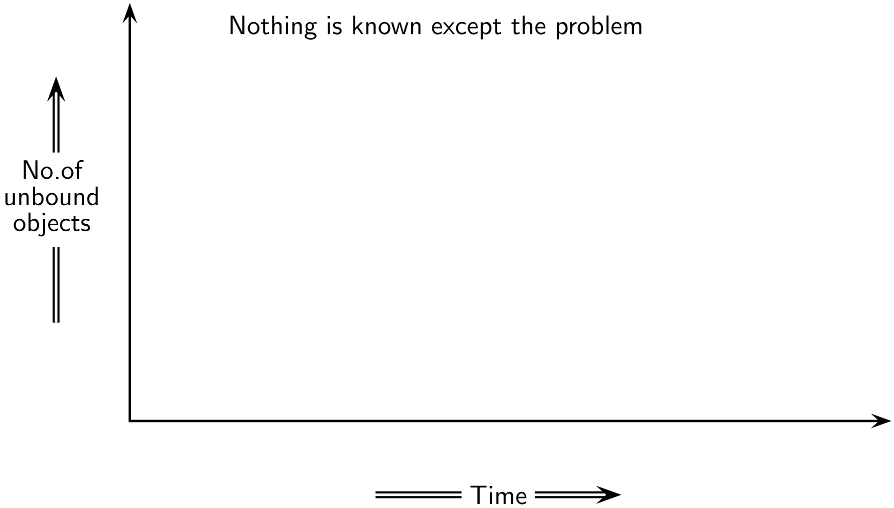
Plan for Monday and Tuesday

- Monday morning
  - A journey from C source program to assembly program (Uday Khedker)
  - A journey from Assembly program to execution on the hardware to obtain results (Abhijat Vichare)
- Monday afternoon
  - Introduction to AIDSL and tcc (Soumik Kumar Basu)
  - Introduction to base code (ioc) (Uday Khedker)
  - Introduction to scanning using lex (Uday Khedker)
  - Lab experiements with scanner of ioc You and the TAs ;-)
- Tuesday morning
  - An overview of shift reduce parsing (Uday Khedker)
  - Inroduction to parsing using yacc (Uday Khedker)
- Tuesday afternoon

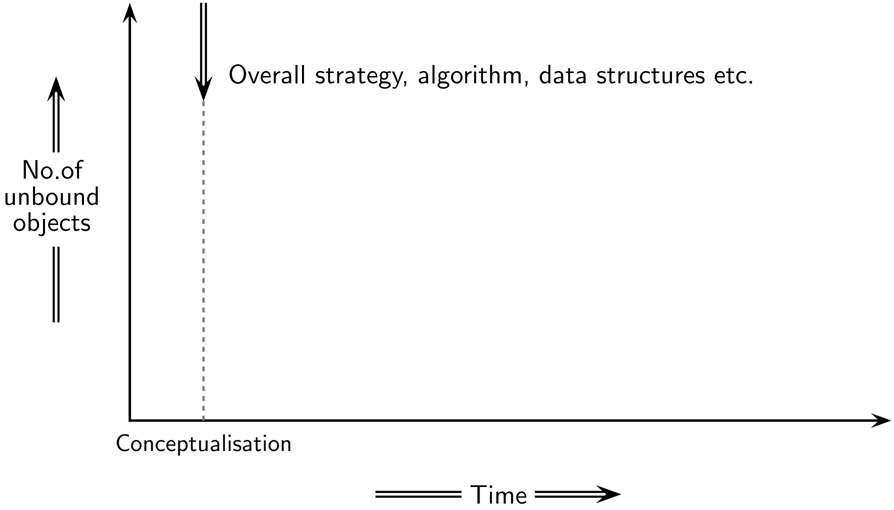
  Lab exercises to enhance ioc to include the features of tcc You and the TAs ;-)

Introduction to Compilation

Binding


Nothing is known except the problem
No.of unbound objects
Time

Binding



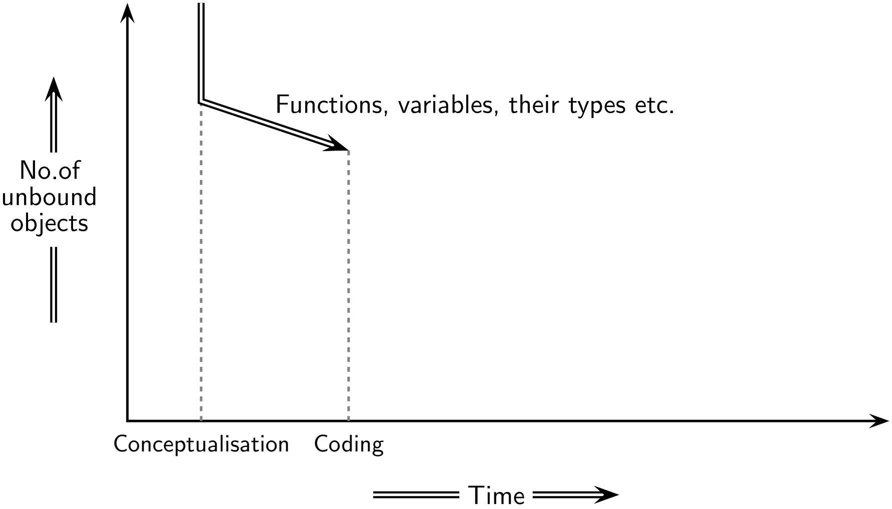
Overall strategy, algorithm, data structures etc.

No.of unbound objects

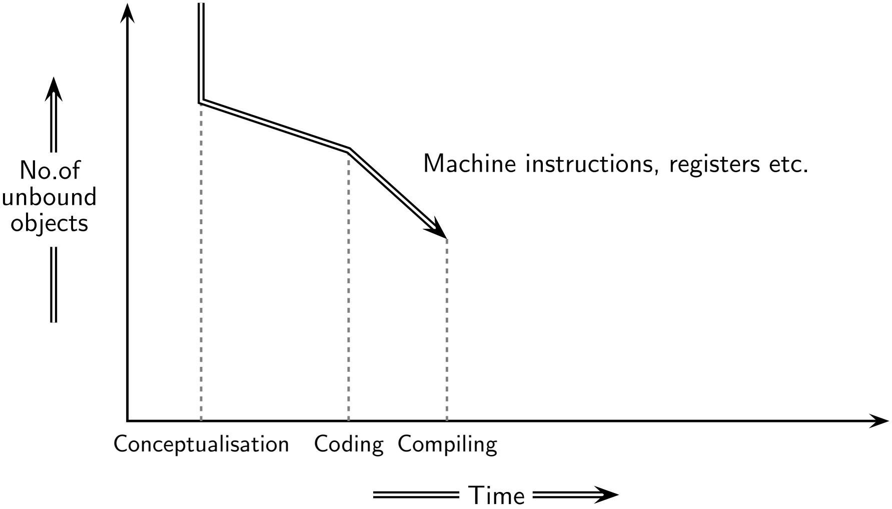
Conceptualisation

Time

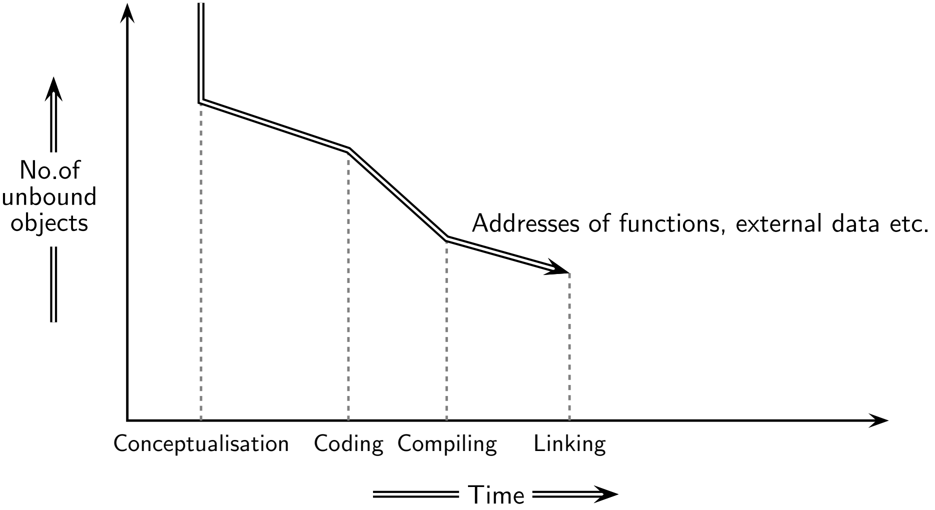
Binding


No.of unbound objects
Functions, variables, their types etc.
Conceptualisation
Coding
Time

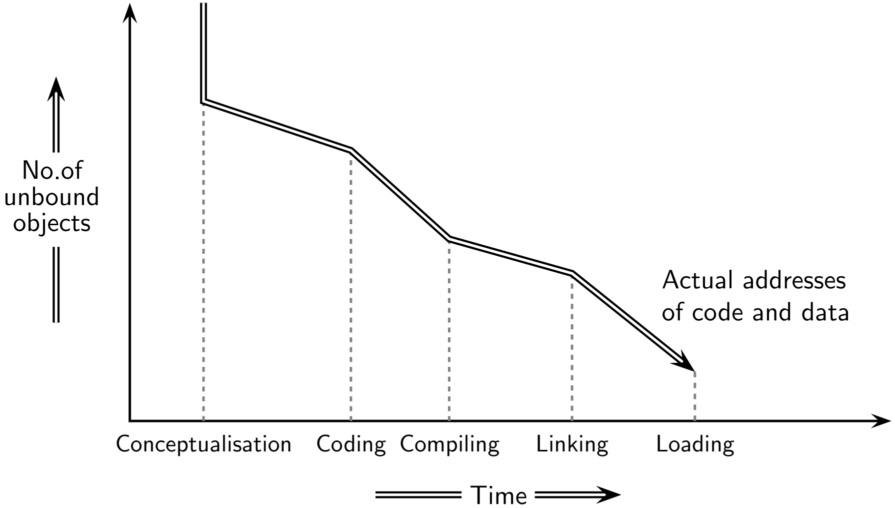
Binding


No.of unbound objects
Machine instructions, registers etc.
Conceptualisation
Coding
Compiling
Time

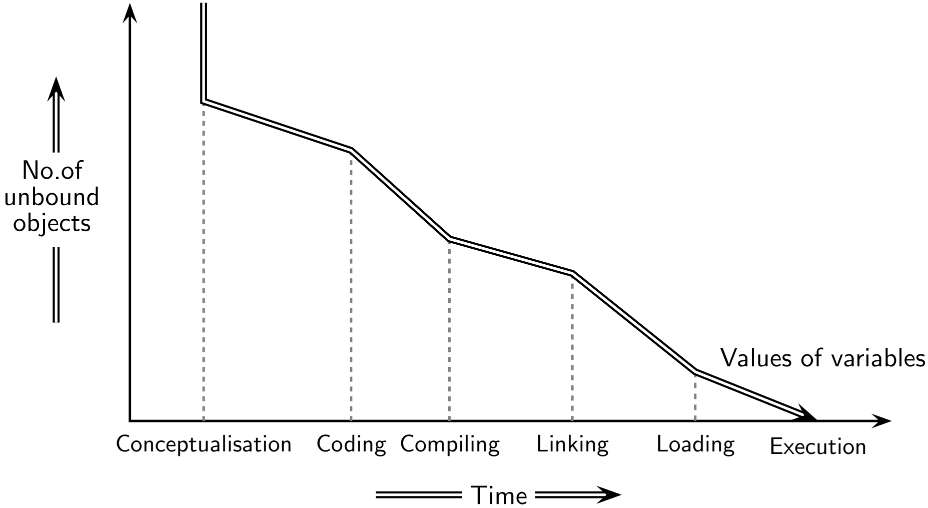
Binding


No.of unbound objects
Addresses of functions, external data etc.
Conceptualisation
Coding
Compiling
Linking
Time

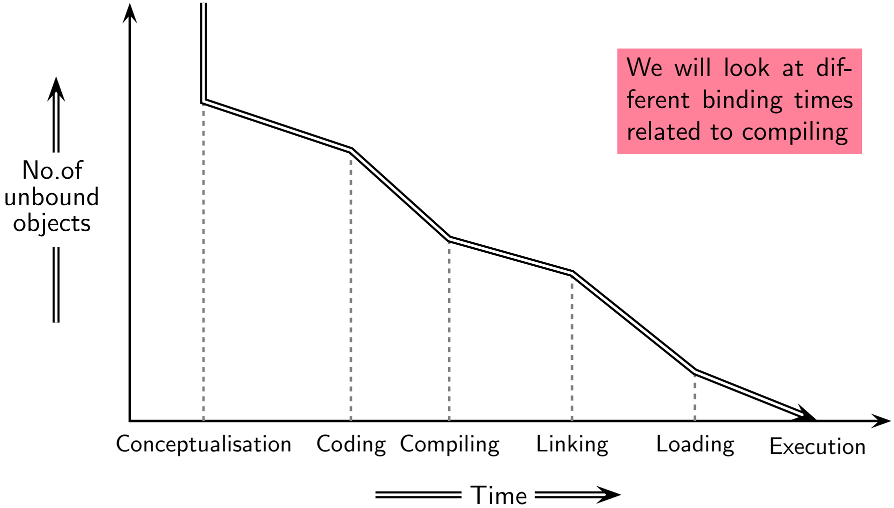
Binding


No.of unbound objects
Actual addresses of code and data
Conceptualisation
Coding
Compiling
Linking
Loading
Time

Binding

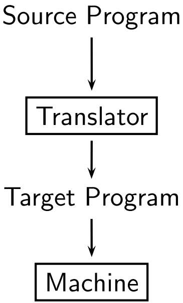

No.of unbound objects
Values of variables
Conceptualisation
Coding
Compiling
Linking
Loading
Execution
Time

Binding

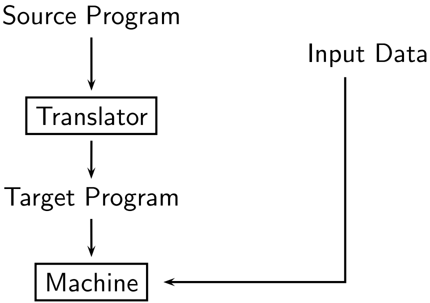


We will look at different binding times related to compiling

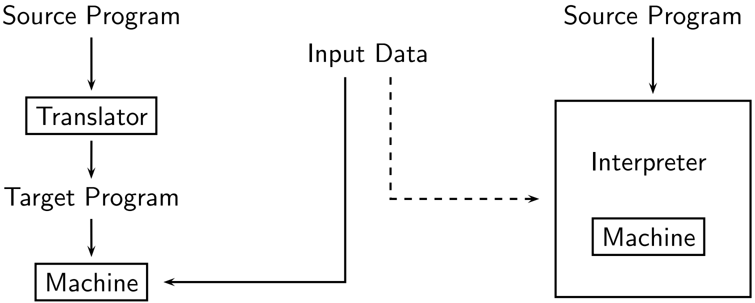
Implementation Mechanisms



Implementation Mechanisms



Implementation Mechanisms



Comparing the Implementation Mechanisms

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

Comparing the Implementation Mechanisms

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

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

Seeing the Difference Between Compilation and Interpretation

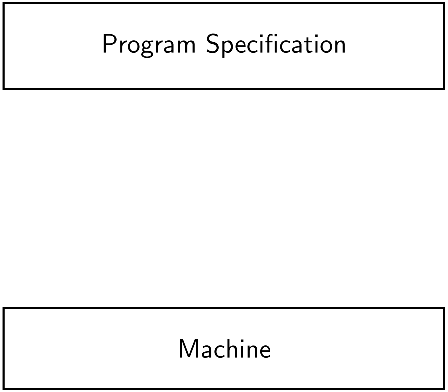
```
$ ./lp --help
Usage: lp [OPTION...]
  -c                   Compile the input into three address code and
                       print it
  -i                   Interpret the input and print result
  -?, --help           Give this help list
      --usage          Give a short usage message

$ ./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

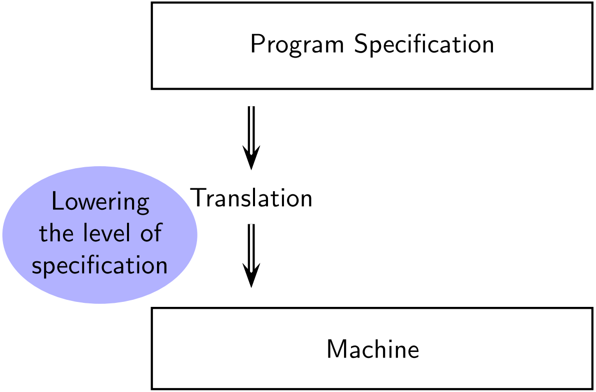


Program Specification

Machine

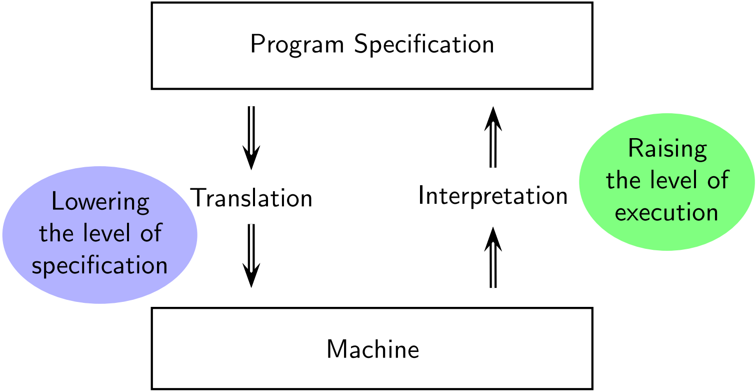
Implementation Mechanisms as “Bridges”

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



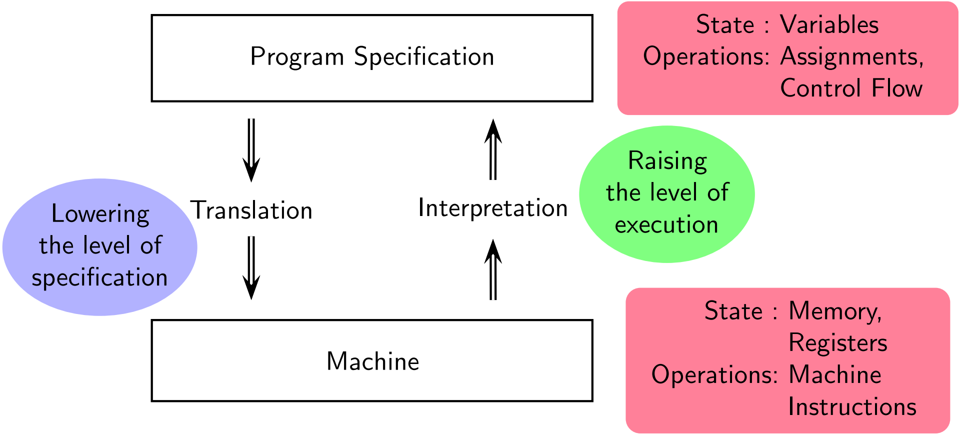
Implementation Mechanisms as "Bridges"

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


Program Specification
Translation
Interpretation
Lowering the level of specification
Raising the level of execution
Machine

Implementation Mechanisms as “Bridges”

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



Program Specification

State : Variables
Operations: Assignments, Control Flow

Translation

Lowering the level of specification

Interpretation

Raising the level of execution

Machine

State : Memory, Registers
Operations: Machine Instructions

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;
```

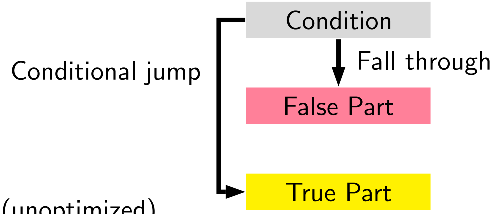
Spim assembly equivalent (unoptimized)

```
    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;
```

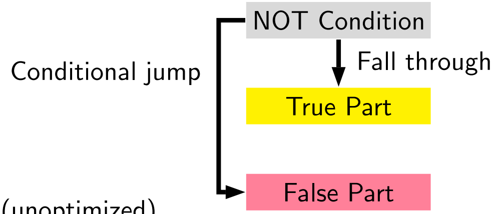
Spim assembly equivalent (unoptimized)

```
    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

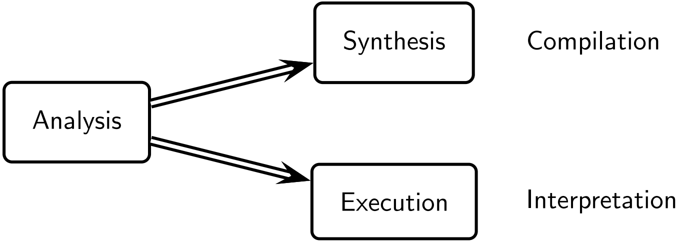
```
a = b<10?b:c+5;
```



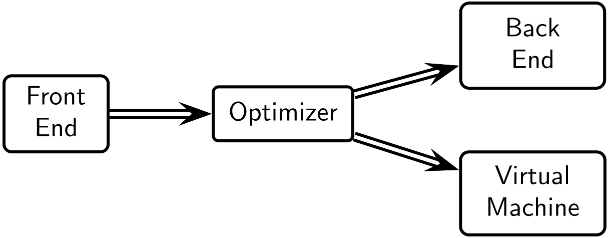
Spim assembly equivalent (unoptimized)

```
    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



Language Processor Models



C,C++

Java, C#

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

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

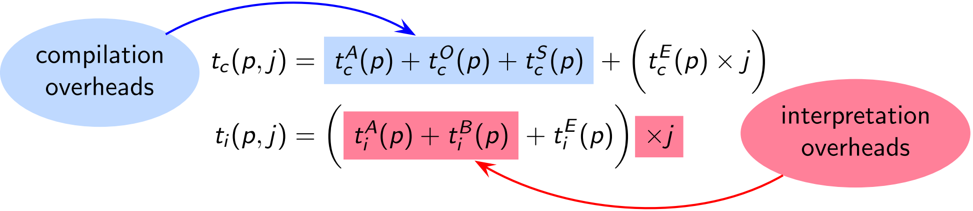
$$t_i(p,j) = \left(t_i^A(p) + t_i^B(p) + t_i^E(p) \right) \times j$$

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

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

$$t_i(p,j) = \left(\underbrace{t_i^A(p) + t_i^B(p)}_{\text{interpretation overheads}} + t_i^E(p)\right) \times j$$

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

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

- For small values of j, $t_c(p,j) \gg t_i(p,j)$

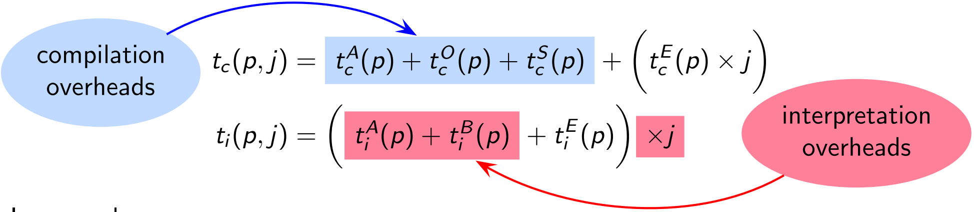
  Overheads of interpretations are meaningful for infrequently executed jobs

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



compilation overheads

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

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

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

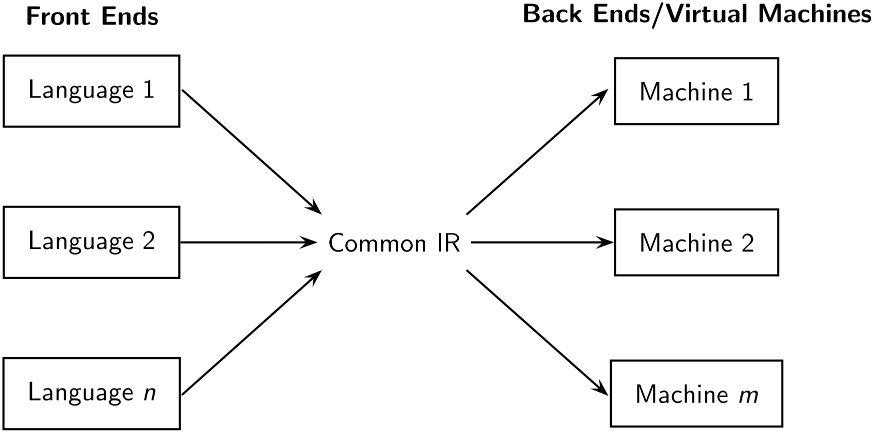
- For small values of j, $t_c(p,j) \gg t_i(p,j)$

  Overheads of interpretations are meaningful for infrequently executed jobs

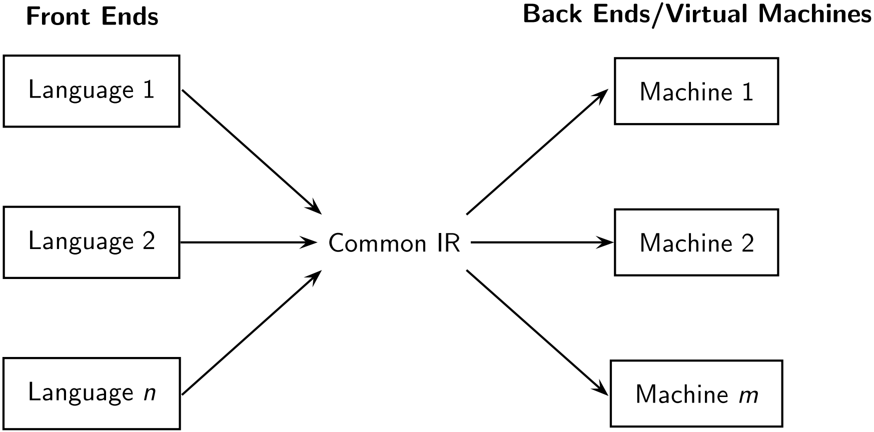
- For any value of $j > 0$, $\left(t_c^E(p) \times j\right) \ll t_i(p,j)$

  Overheads of compilation are meaningful for jobs with large execution times

Reusability of Language Processor Modules



Reusability of Language Processor Modules



Front Ends

Back Ends/Virtual Machines

Language 1

Language 2

Language *n*

Common IR

Machine 1

Machine 2

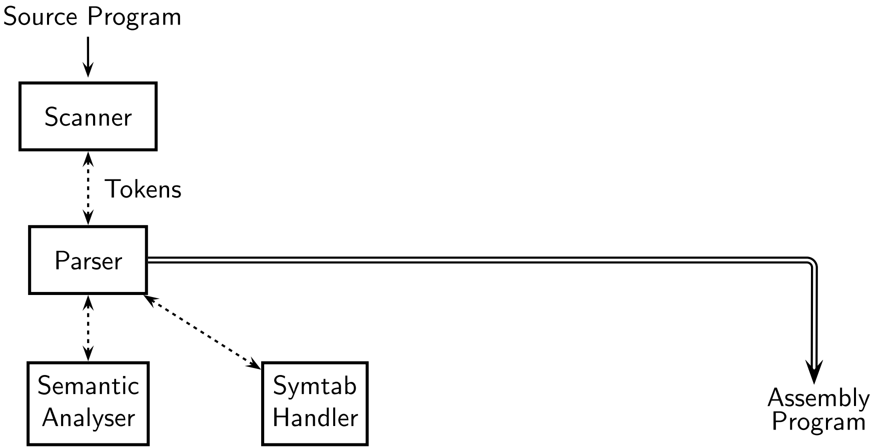
Machine *m*

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

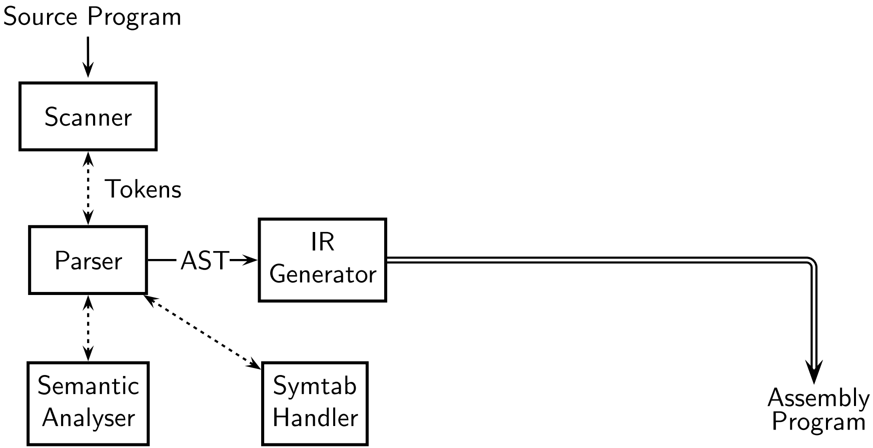
An Overview of Compilation Phases

The Structure of a Simple Compiler

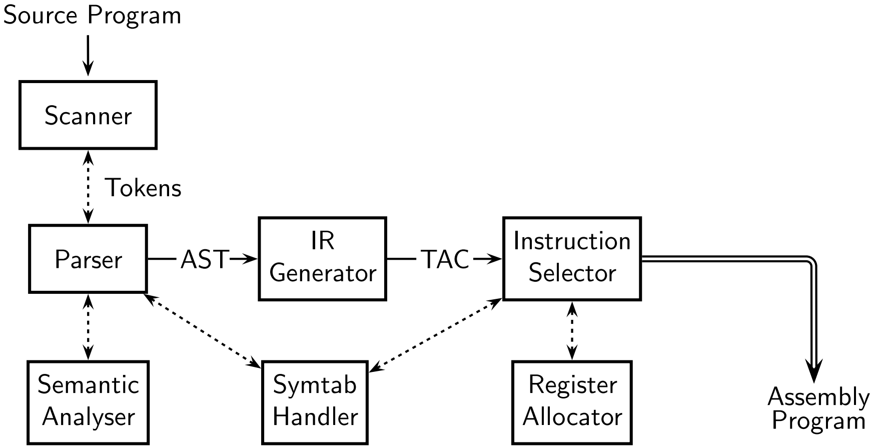
The Structure of a Simple Compiler



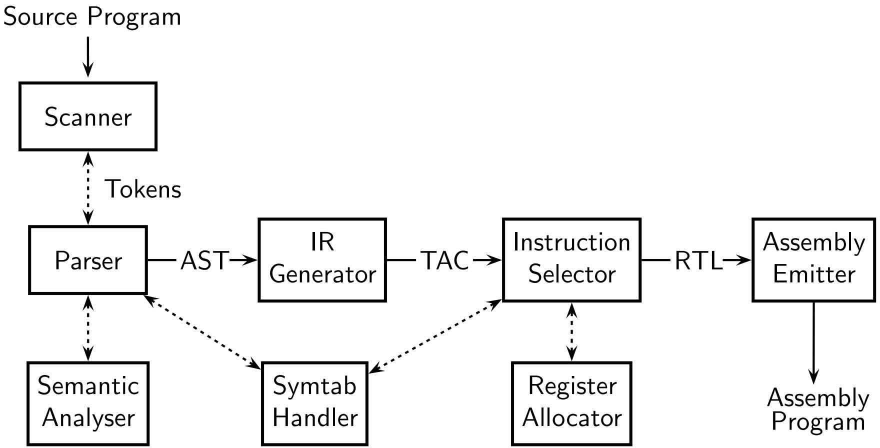
The Structure of a Simple Compiler



The Structure of a Simple Compiler



The Structure of a Simple Compiler

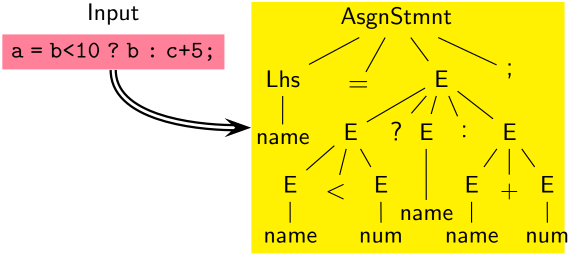


Translation Sequence in Our Compiler: Scanning and Parsing

Input

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

Translation Sequence in Our Compiler: Scanning and Parsing



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

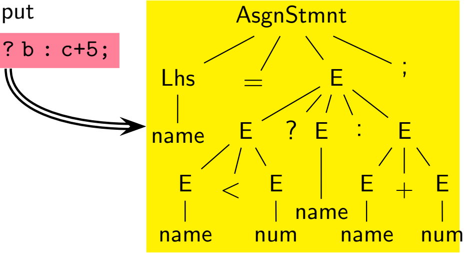
  eg. is it `(a = b<10?)` followed by `(b:c)`?
- Values of terminal symbols

  eg. string “10” vs. integer number 10.

Translation Sequence in Our Compiler: Semantic Analysis

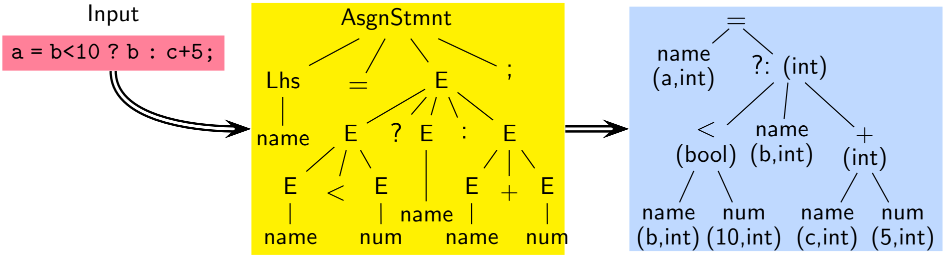
Input

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



Parse Tree

Translation Sequence in Our Compiler: Semantic Analysis



Parse Tree

Abstract Syntax Tree (with attributes)

Issues:

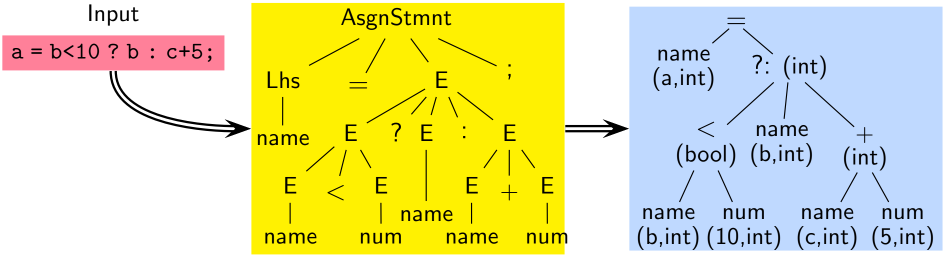
- Symbol tables

  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



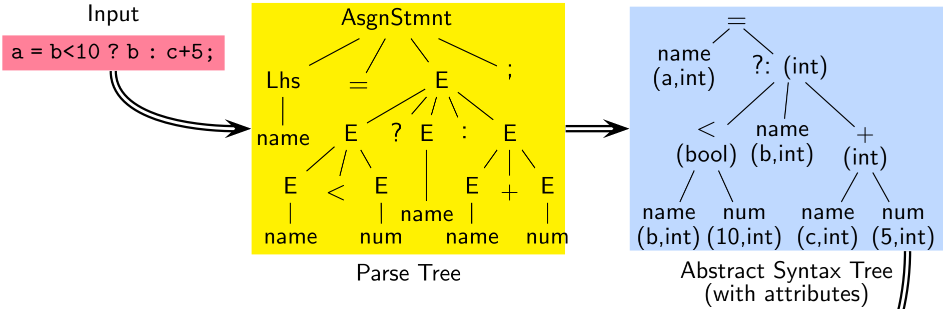
Parse Tree

Abstract Syntax Tree
(with attributes)

Translation Sequence in Our Compiler: IR Generation

Input

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



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: $a = T_3$

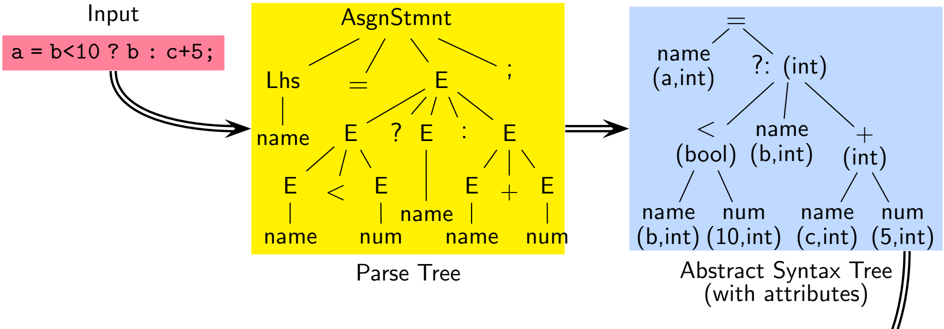
Issues:

- 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



TAC List

$T_1 = b < 10$

$T_2 = \neg T_1$

if T_2 goto L0

$T_3 = b$

goto L1:

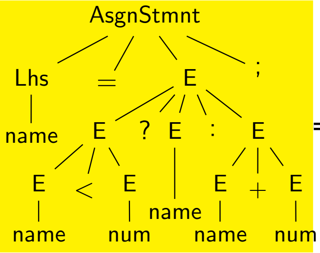
L0: $T_3 = c + 5$

L1: $a = T_3$

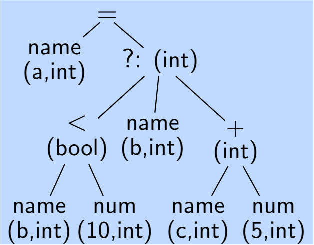
Translation Sequence in Our Compiler: Instruction Selection

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 $a \leftarrow t3$

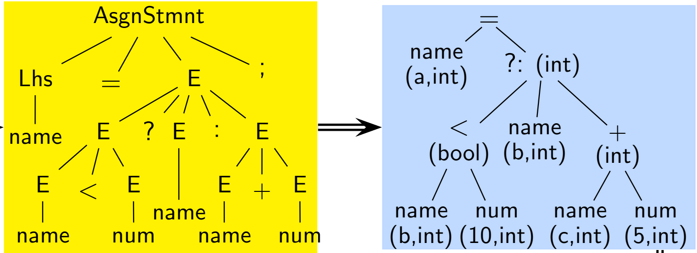
Issues:

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

Translation Sequence in Our Compiler: Emitting Instructions

Input

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



Parse Tree

Abstract Syntax Tree (with attributes)

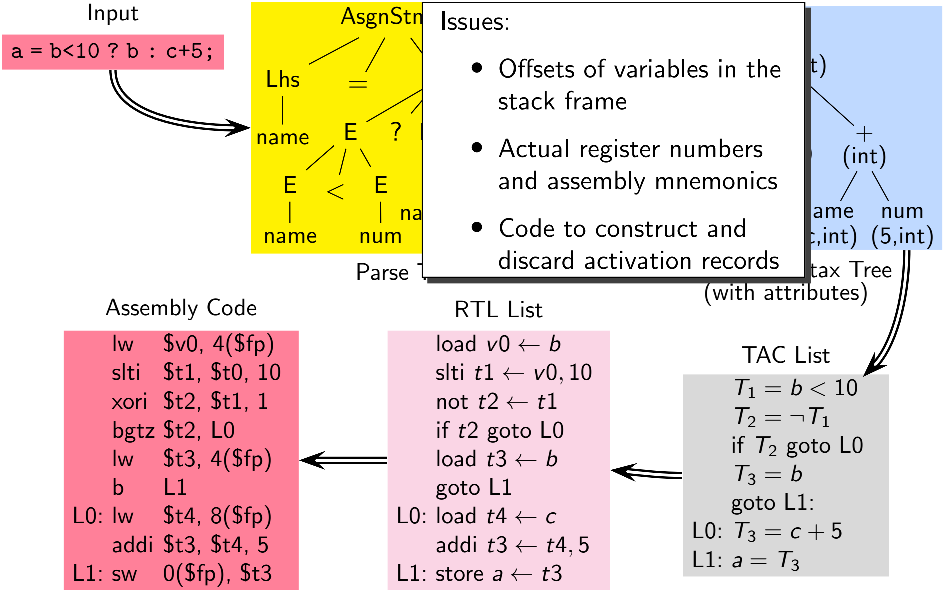
TAC List

$T_1 = b < 10$
$T_2 = \neg T_1$
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 $a \leftarrow t3$

Translation Sequence in Our Compiler: Emitting Instructions



Issues:

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

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: $a = T_3$

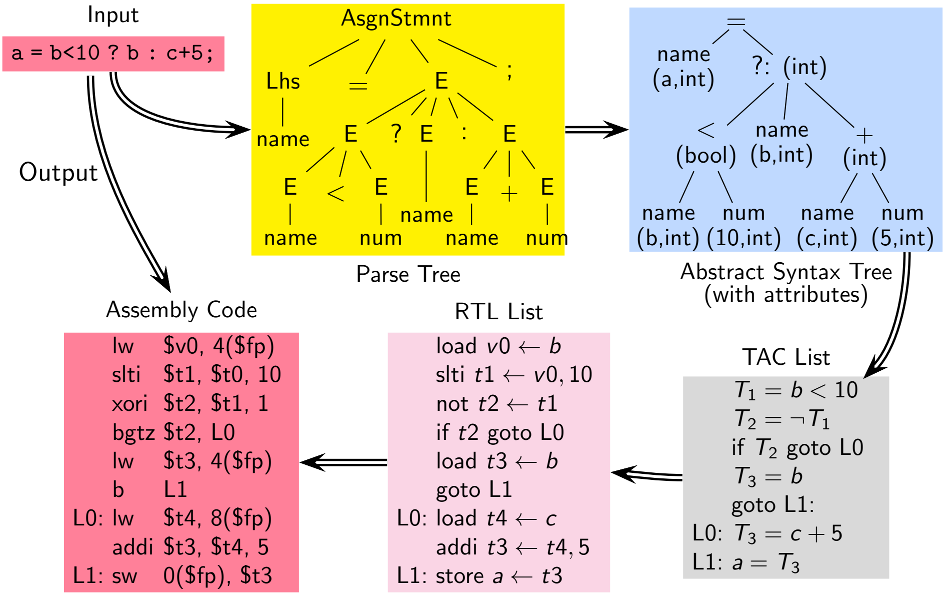
RTL List

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



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

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
```

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

  ⇒ 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?

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
- 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
- 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

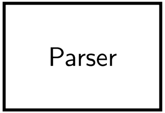
Credits: Adapted from the slides of Prof. Y. N. Srikant, IISc Bangalore

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.

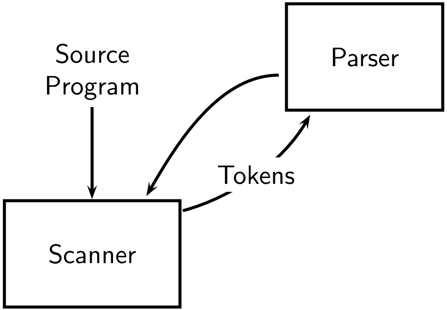
Compilation Models

Typical Front Ends

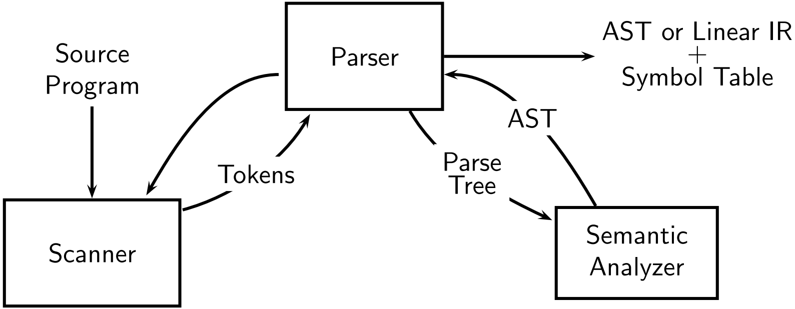


Parser

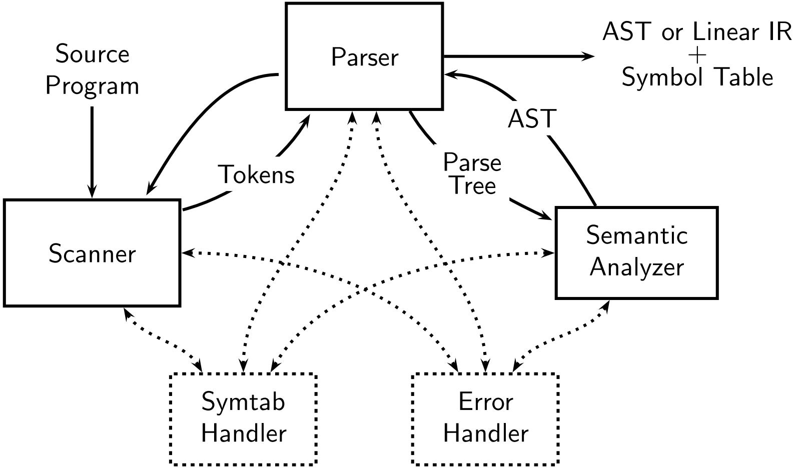
Typical Front Ends



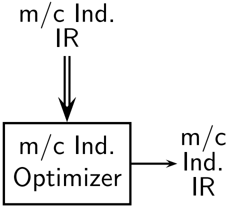
Typical Front Ends



Typical Front Ends

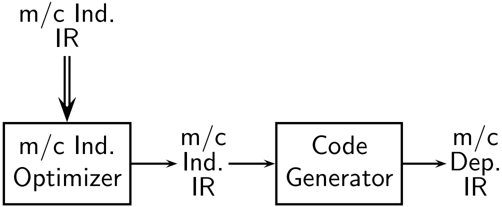


Typical Back Ends



- Compile time evaluations
- Eliminating redundant computations

Typical Back Ends



m/c Ind. IR

m/c Ind. Optimizer

m/c Ind. IR

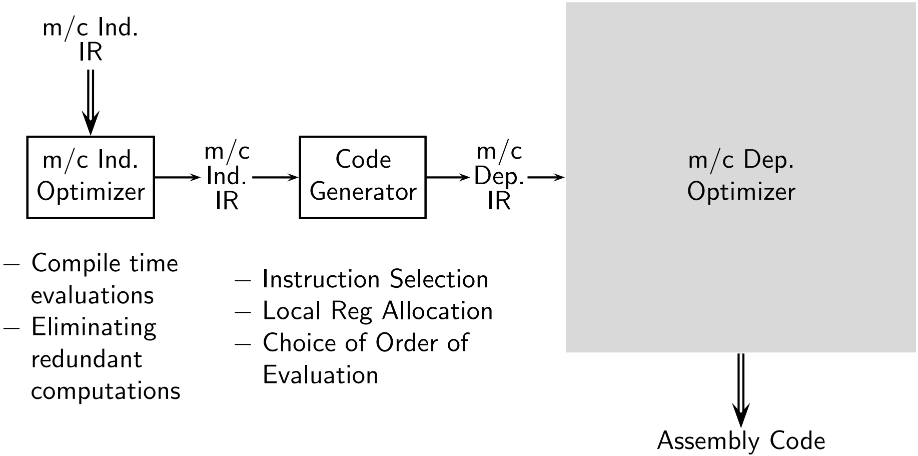
Code Generator

m/c Dep. IR

- Compile time evaluations
- Eliminating redundant computations

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

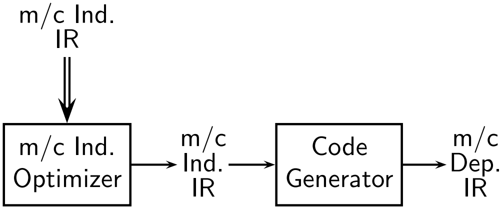
Typical Back Ends



- Compile time evaluations
- Eliminating redundant computations

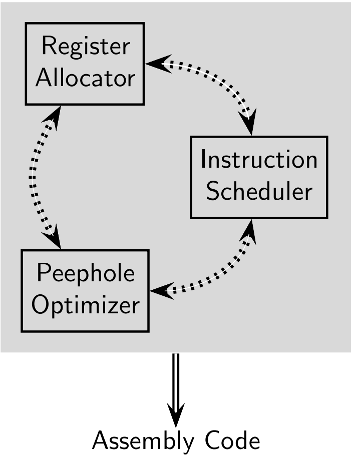
- Instruction Selection
- Local Reg Allocation
- Choice of Order of Evaluation

Typical Back Ends

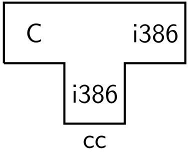


- Compile time evaluations
- Eliminating redundant computations

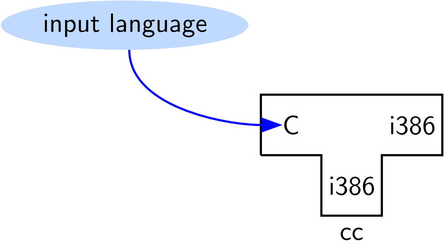
- Instruction Selection
- Local Reg Allocation
- Choice of Order of Evaluation



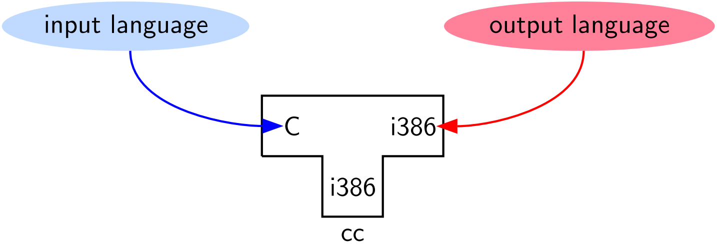
T Notation for a Compiler



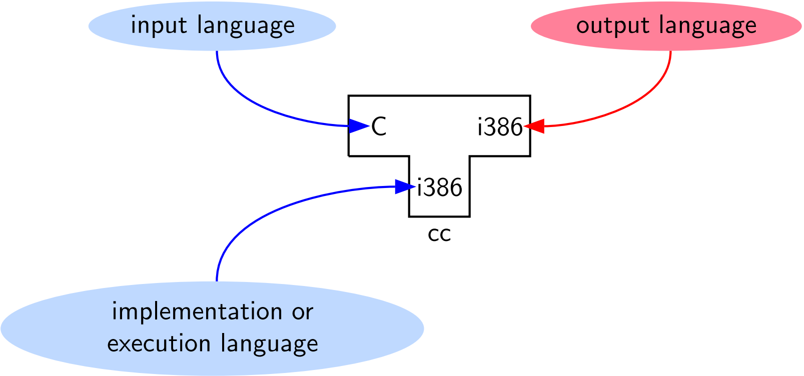
T Notation for a Compiler



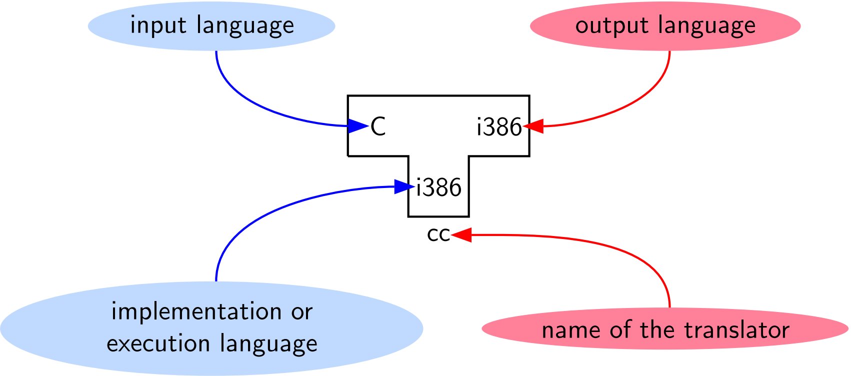
T Notation for a Compiler



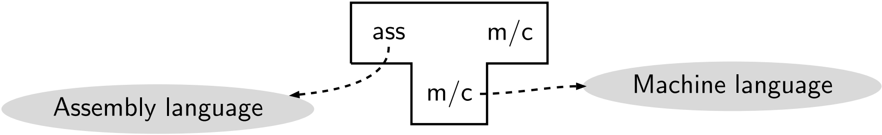
T Notation for a Compiler



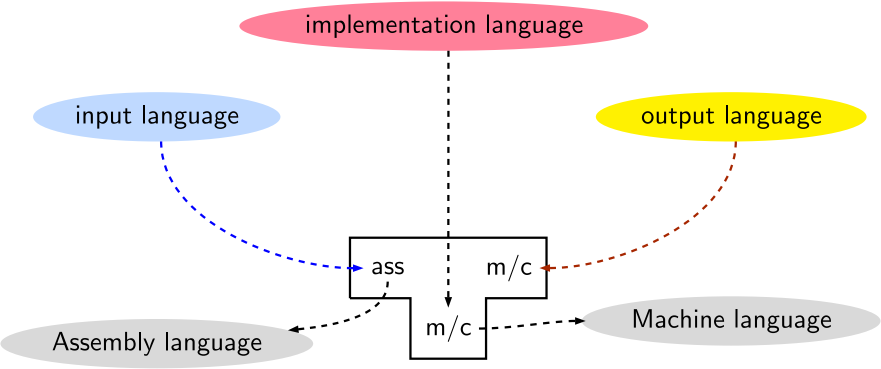
T Notation for a Compiler



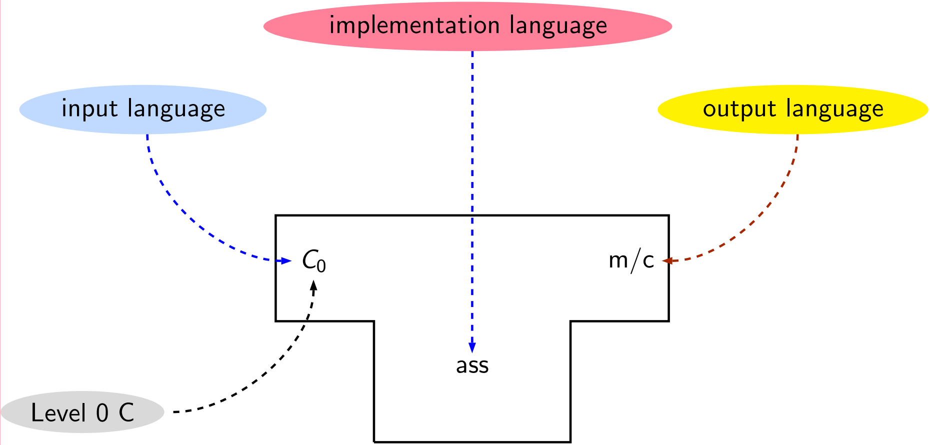
Bootstrapping: The Conventional View



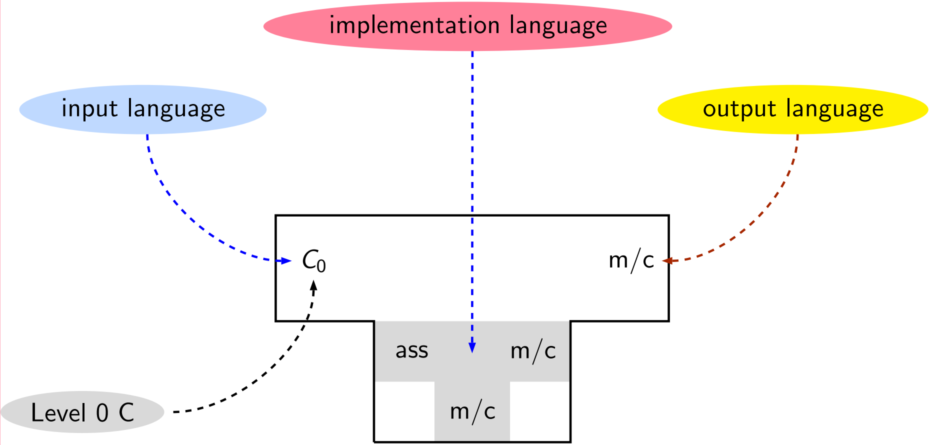
Bootstrapping: The Conventional View



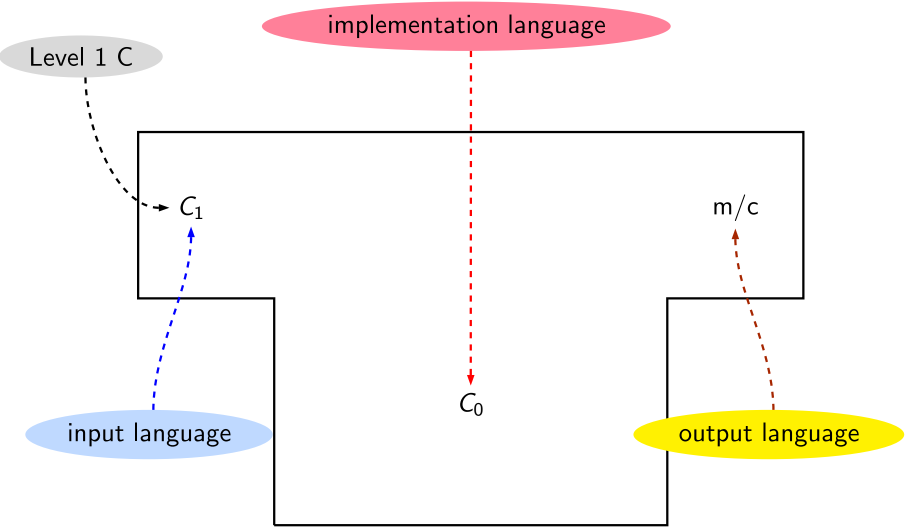
Bootstrapping: The Conventional View



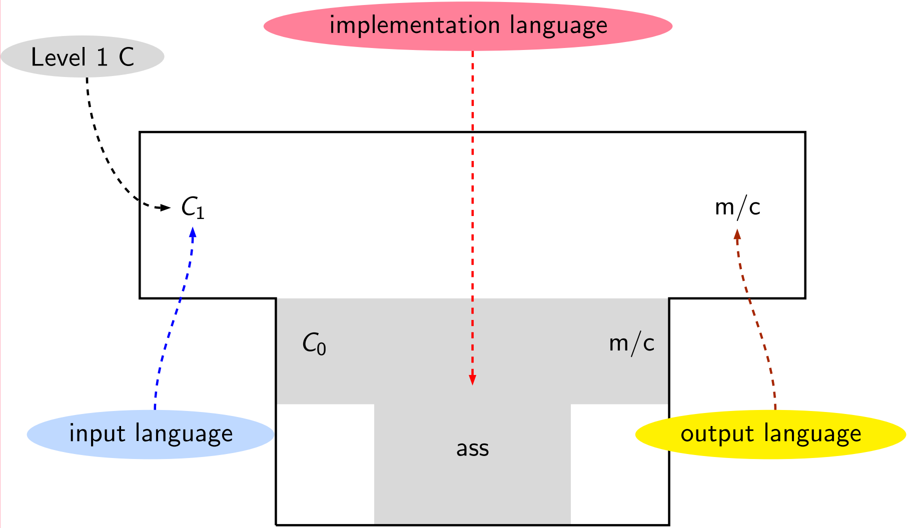
Bootstrapping: The Conventional View



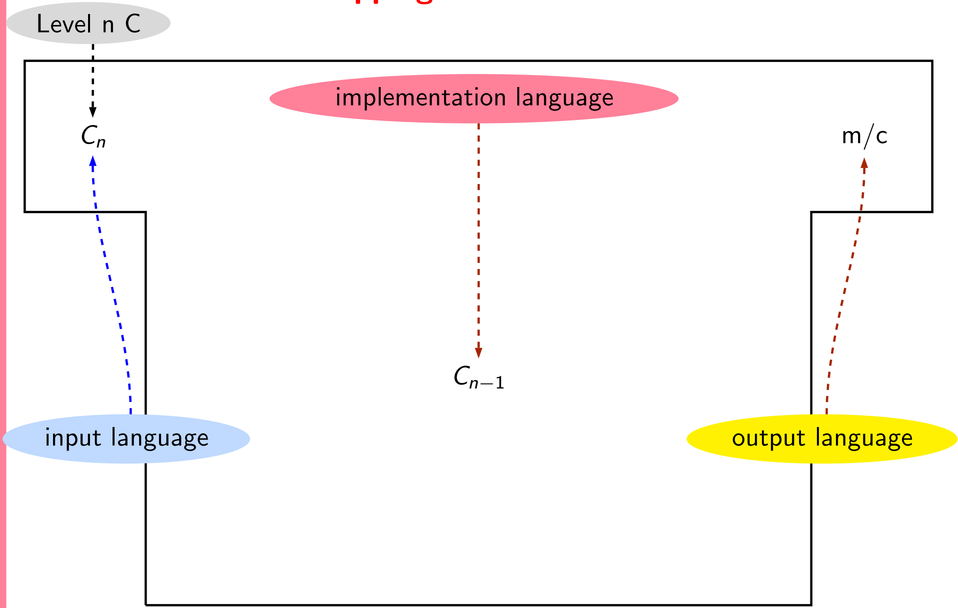
Bootstrapping: The Conventional View



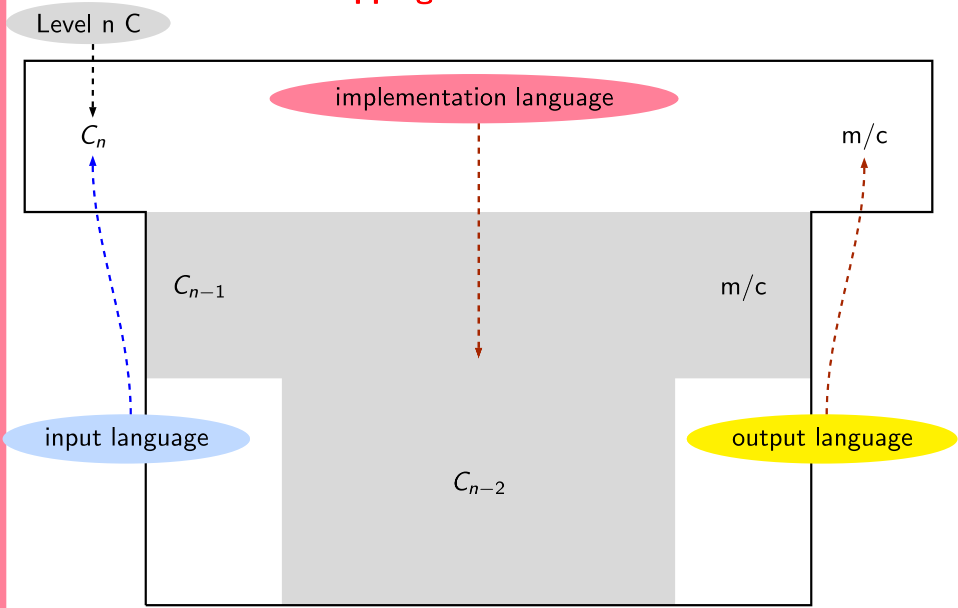
Bootstrapping: The Conventional View



Bootstrapping: The Conventional View



Bootstrapping: The Conventional View



Demo