An Overview of Compilation

Uday Khedker

(www.cse.iitb.ac.in/˜uday)

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

January 2020

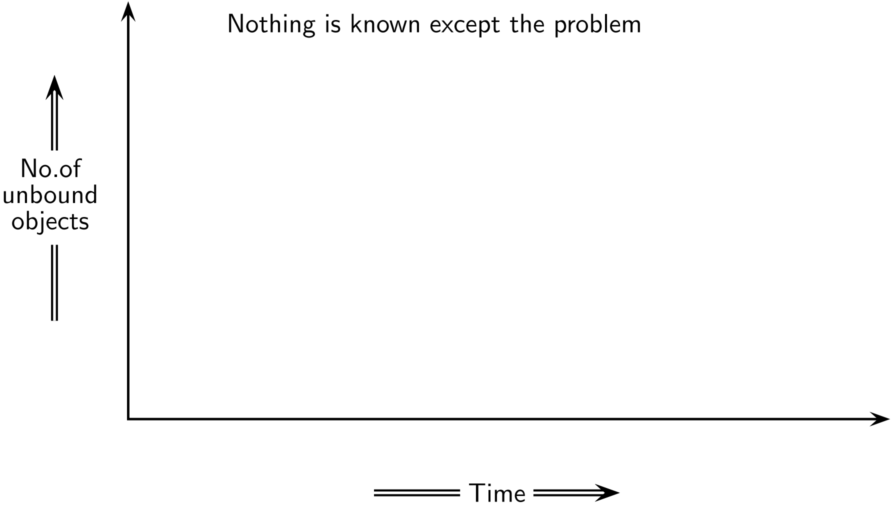
Outline

- Introduction
- Compilation phases
- Compilation models
- Incremental construction of compilers

Part 1

Introduction to Compilation

Binding

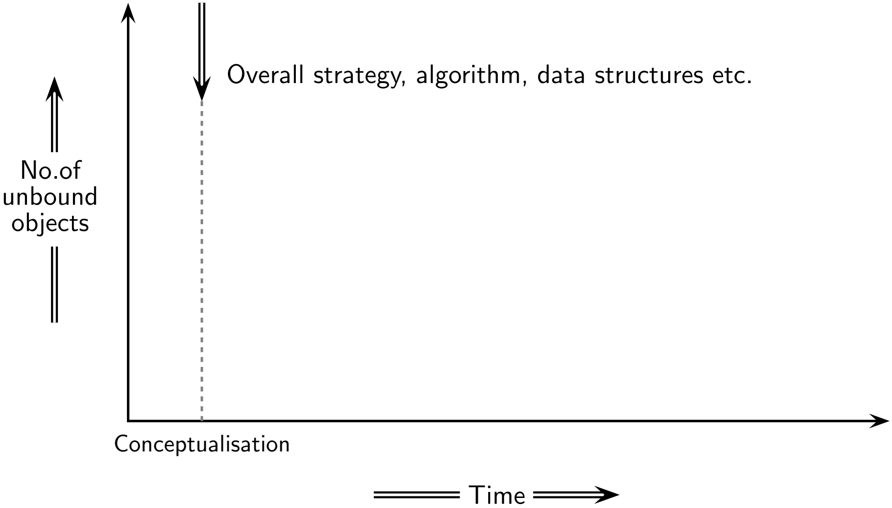


Nothing is known except the problem

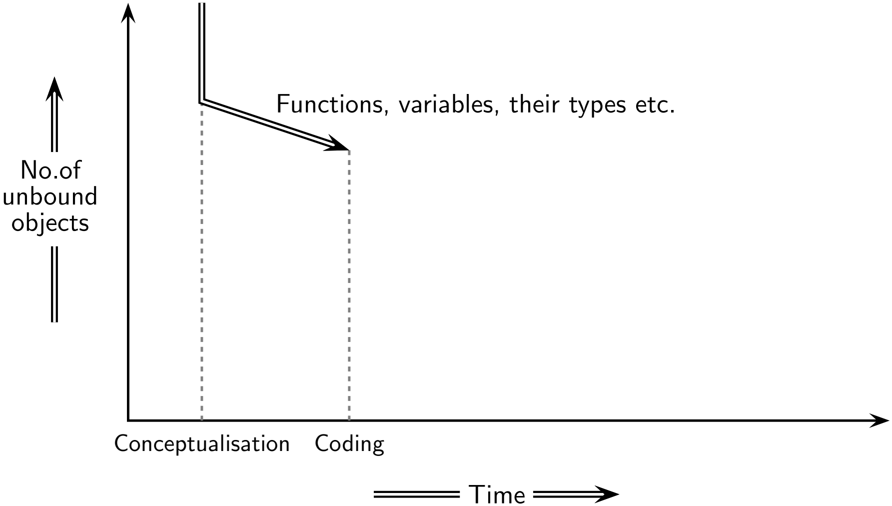
No.of unbound objects

Time

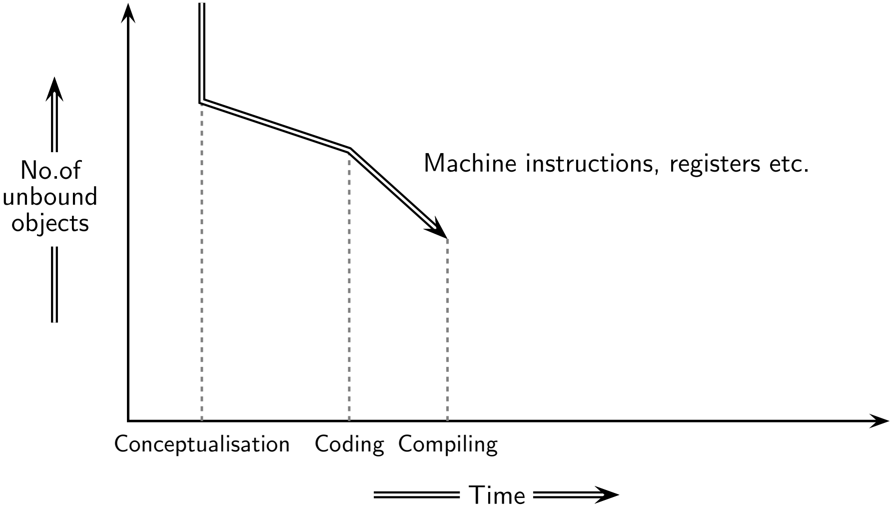
Binding


Overall strategy, algorithm, data structures etc.
No.of unbound objects
Conceptualisation
Time

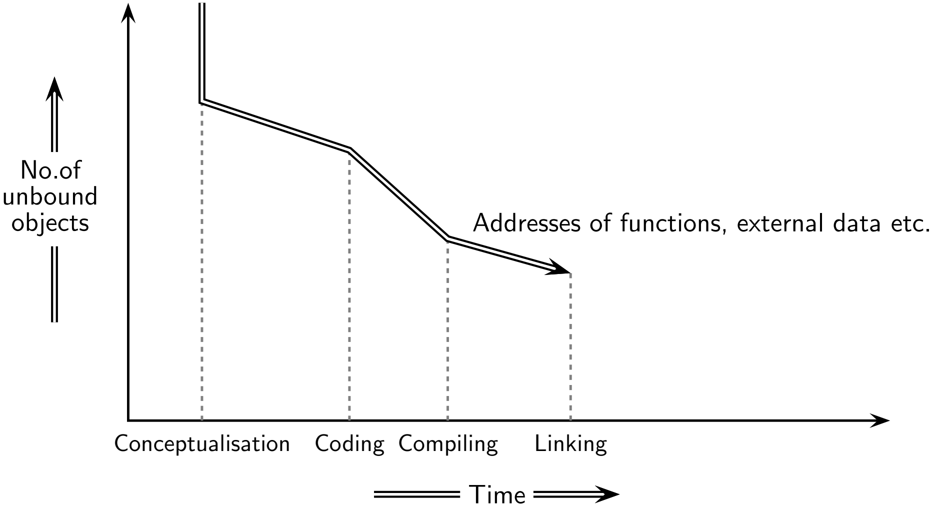
Binding


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

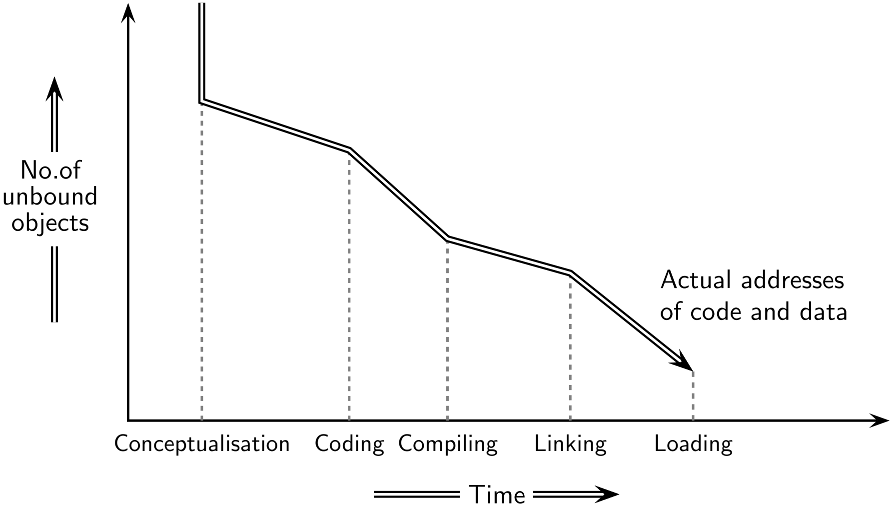
Binding


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

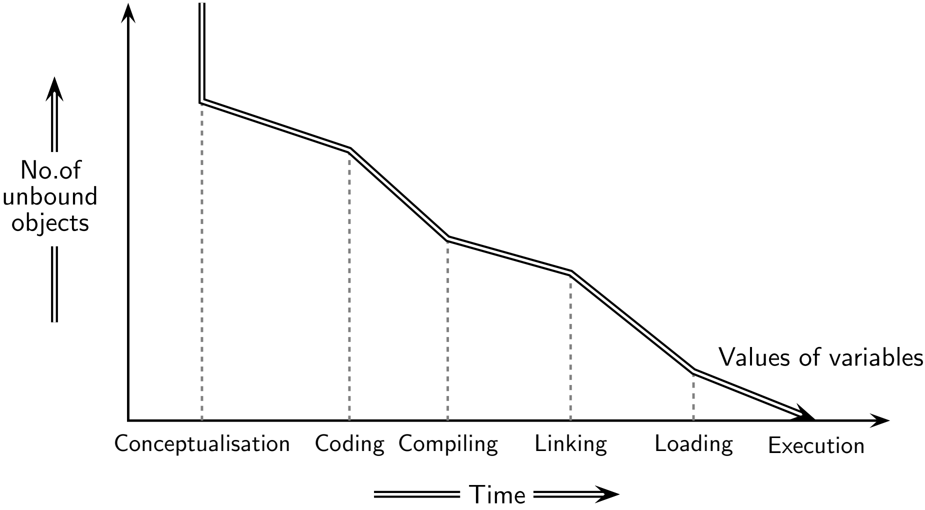
Binding


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

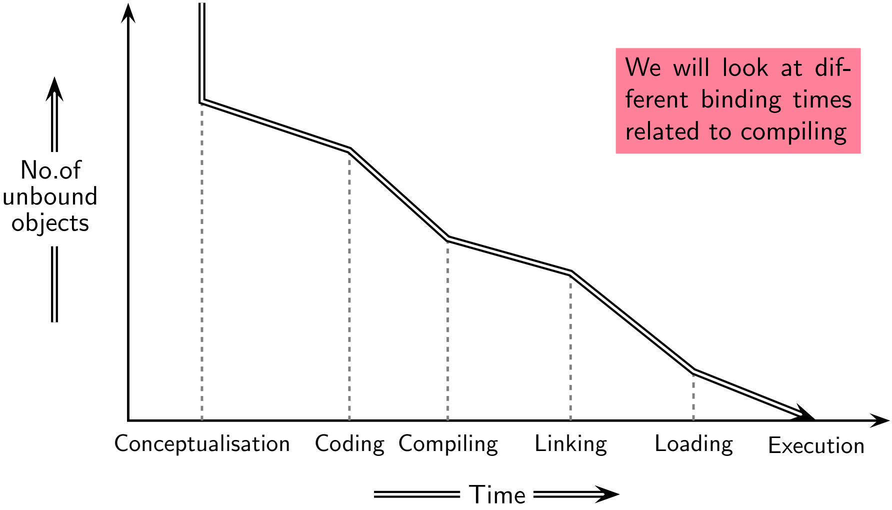
Binding


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

Binding

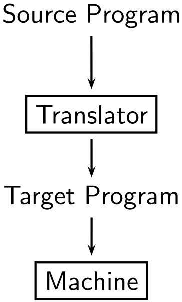

No.of unbound objects
Values of variables
Conceptualisation
Coding
Compiling
Linking
Loading
Execution
Time

Binding



We will look at different binding times related to compiling

Implementation Mechanisms

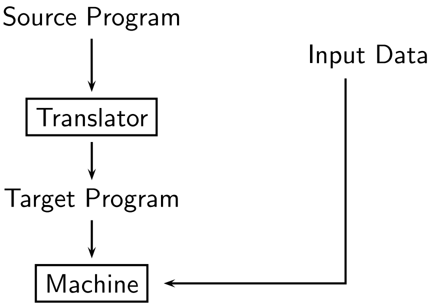

Source Program

Translator

Target Program

Machine

Implementation Mechanisms



Source Program

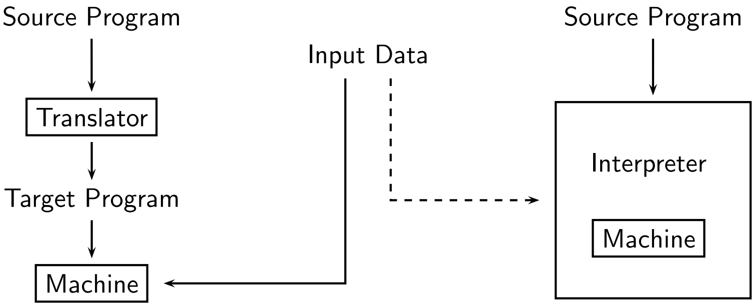
Translator

Target Program

Machine

Input Data

Implementation Mechanisms



Source Program

Translator

Target Program

Machine

Input Data

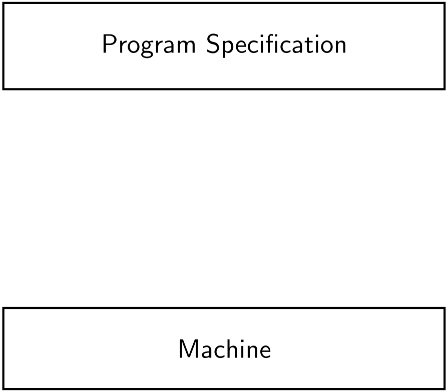
Source Program

Interpreter

Machine

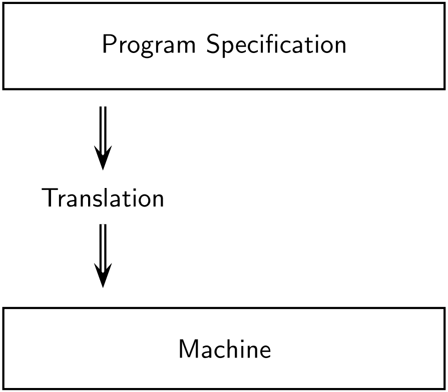
Implementation Mechanisms as "Bridges"

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



Implementation Mechanisms as “Bridges”

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



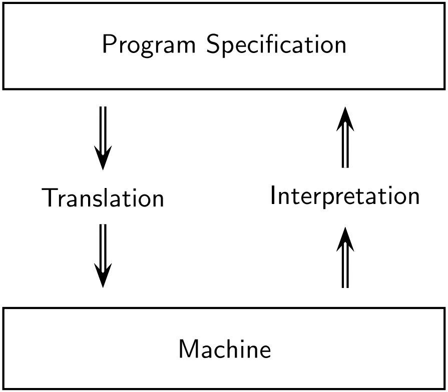
Program Specification

Translation

Machine

Implementation Mechanisms as "Bridges"

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


Program Specification

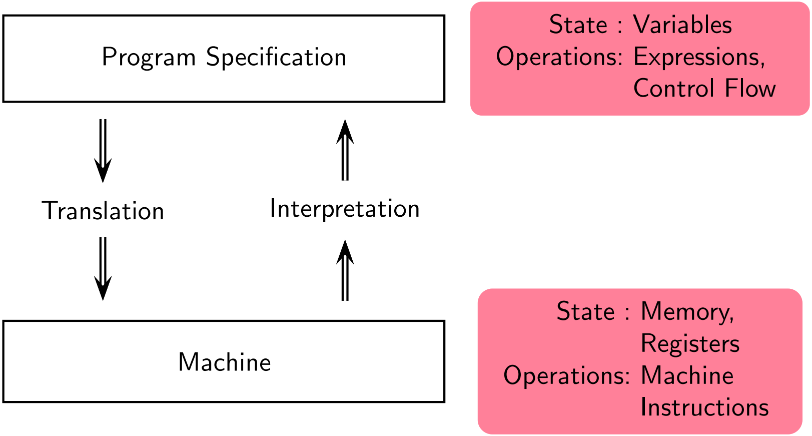
Translation

Interpretation

Machine

Implementation Mechanisms as "Bridges"

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



Program Specification

Translation

Interpretation

Machine

State : Variables
Operations: Expressions,
Control Flow

State : Memory,
Registers
Operations: Machine
Instructions

High and Low Level Abstractions

Input C statement

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

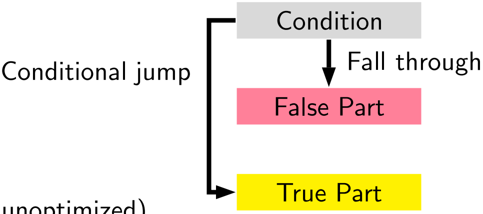
Spim assembly equivalent (unoptimized)

```
    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, $t4, 5  ;     t3 <- t4 + c         # NO
L1: sw     0($fp), $t3  ;L1: a <- t3
```

High and Low Level Abstractions

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, $t4, 5  ;     t3 <- t4 + c     # NO
L1:  sw     0($fp), $t3  ;L1:  a <- t3
```

High and Low Level Abstractions

NOT Condition

True Part

False Part

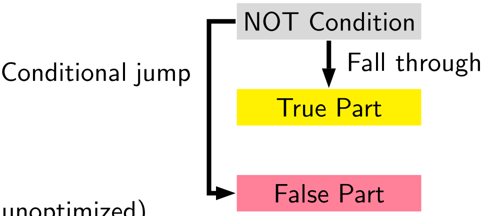
Input C statement

```
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, $t4, 5  ;     t3 <- t4 + c       # NO
L1: sw    0($fp), $t3  ;L1: a <- t3
```

High and Low Level Abstractions


Conditional jump

NOT Condition

Fall through

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, $t4, 5  ;     t3 <- t4 + c     # NO
L1: sw    0($fp), $t3  ;L1:  a <- t3
```

Implementation Mechanisms

- Translation = Analysis + Synthesis
  Interpretation = Analysis + Execution

Implementation Mechanisms

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

- Translation Instructions $\Longrightarrow$ Equivalent Instructions

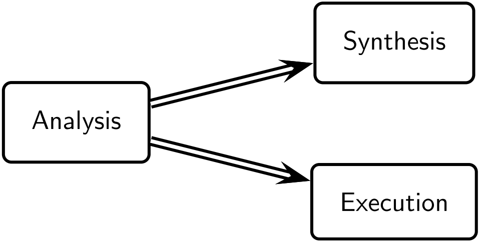
Implementation Mechanisms

- Translation = Analysis + Synthesis
  Interpretation = Analysis + Execution

- Translation Instructions $\Longrightarrow$ Equivalent Instructions

  Interpretation Instructions $\Longrightarrow$ Actions Implied by Instructions

Language Implementation Models



Analysis

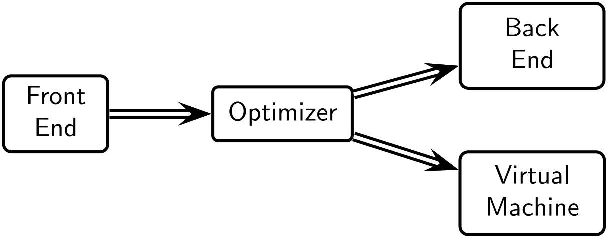
Synthesis

Execution

Compilation

Interpretation

Language Processor Models



C,C++

Java, C#

Part 2

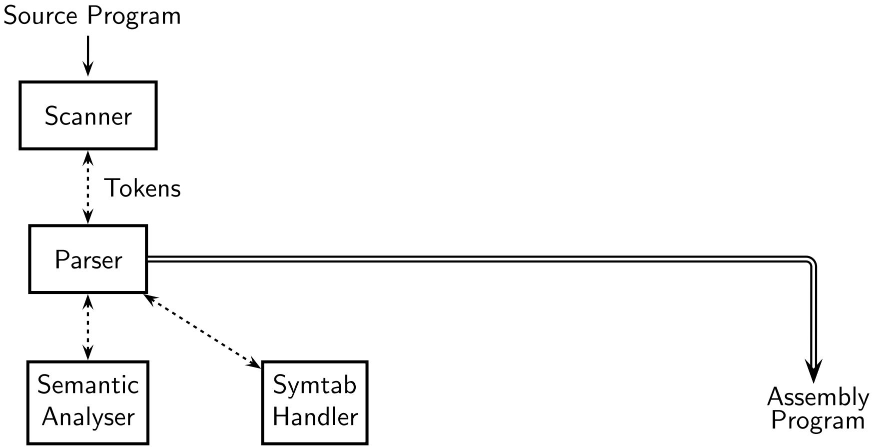
An Overview of Compilation Phases

The Structure of a Simple Compiler

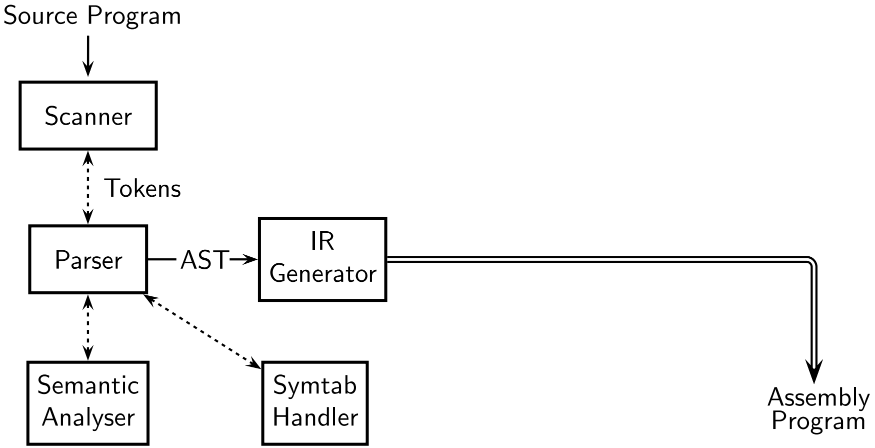

Source Program

Assembly Program

The Structure of a Simple Compiler



The Structure of a Simple Compiler


Source Program

Scanner

Tokens

Parser

AST

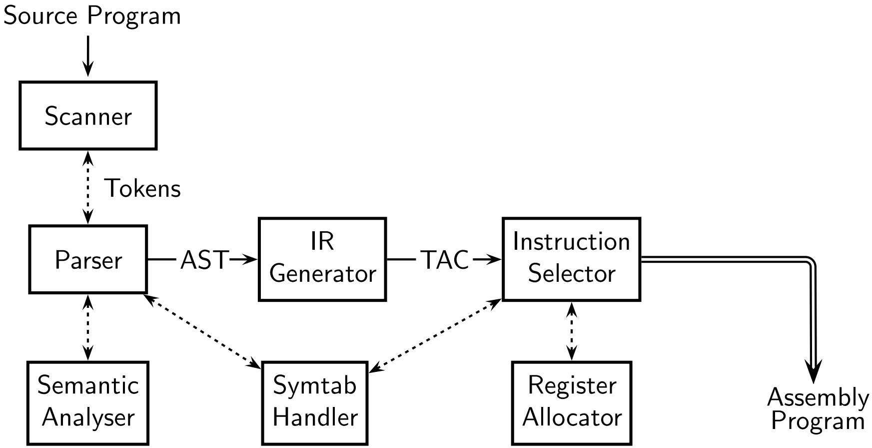
IR Generator

Semantic Analyser

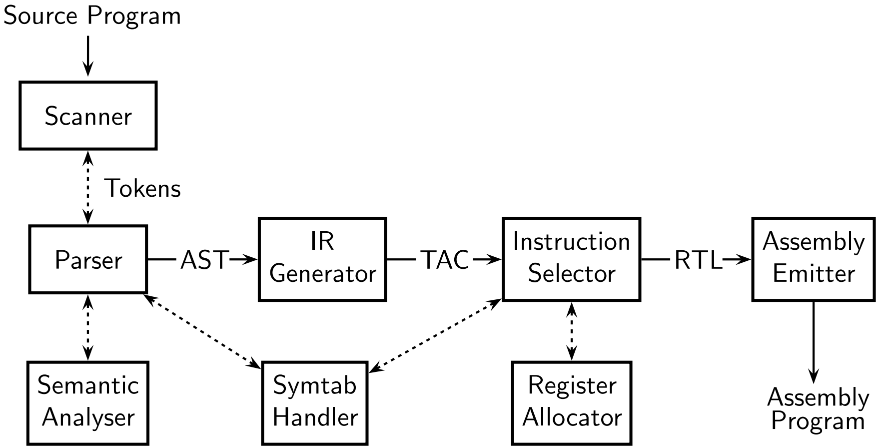
Symtab Handler

Assembly Program

The Structure of a Simple Compiler



The Structure of a Simple Compiler

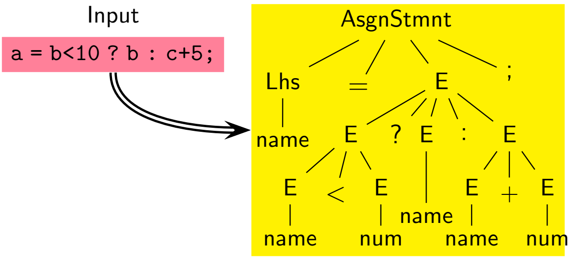


Translation Sequence in Our Compiler: Scanning and Parsing

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

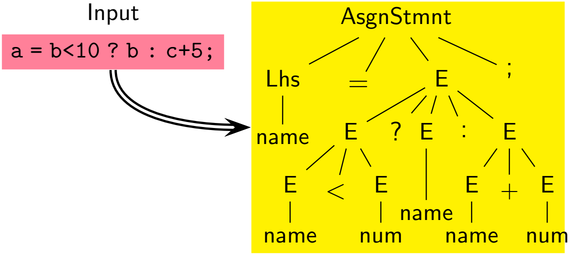
  eg. is it (`a = b<10?`) followed by (`b:c`)?
- Values of terminal symbols

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

Translation Sequence in Our Compiler: Semantic Analysis

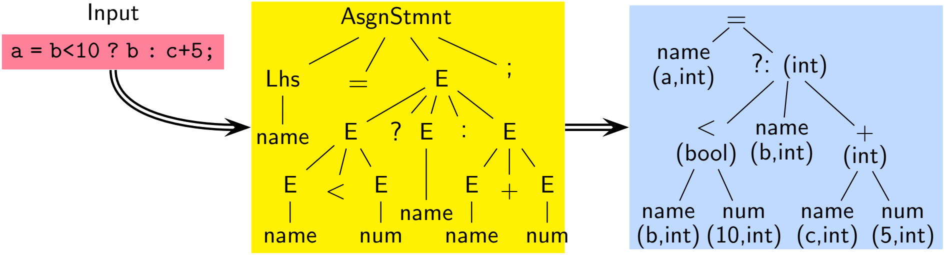
Input

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



Parse Tree

Translation Sequence in Our Compiler: Semantic Analysis



Parse Tree

Abstract Syntax Tree
(with attributes)

Issues:

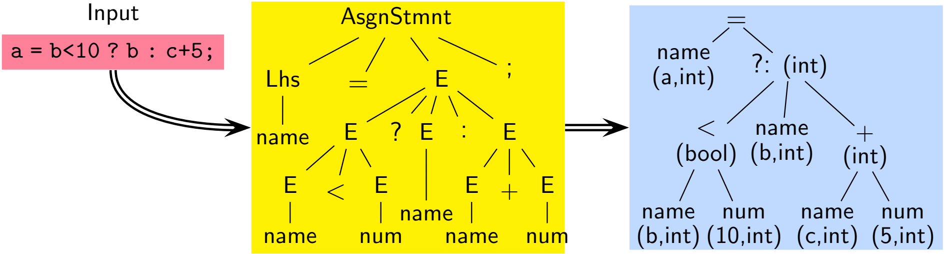
- Symbol tables

  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



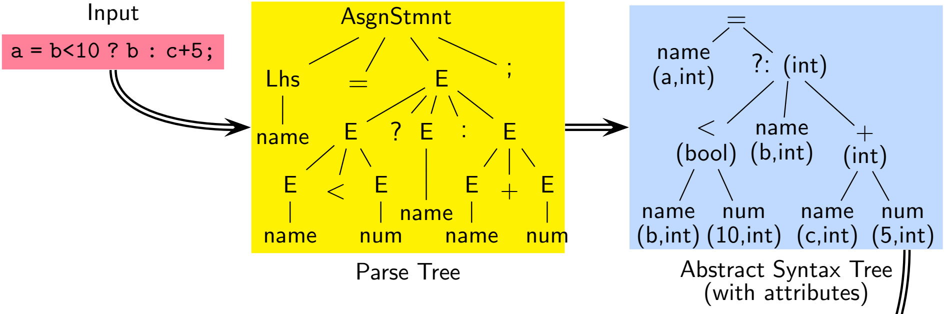
Parse Tree

Abstract Syntax Tree
(with attributes)

Translation Sequence in Our Compiler: IR Generation

Input

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



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: $a = T_3$

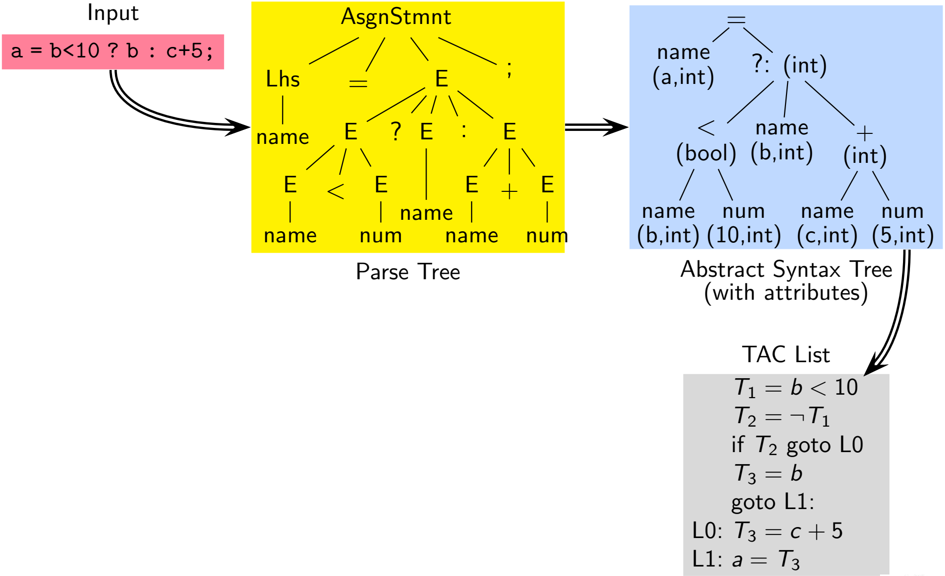
Issues:

- 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)

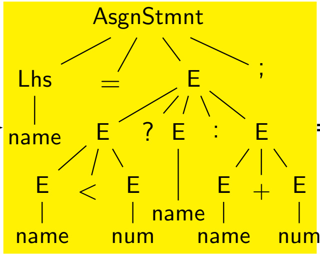
TAC List

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

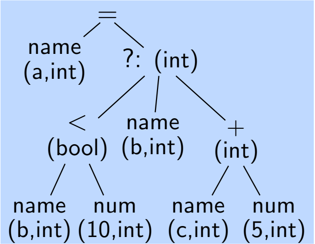
Translation Sequence in Our Compiler: Instruction Selection

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 $a \leftarrow t3$

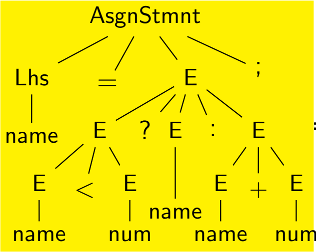
Issues:

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

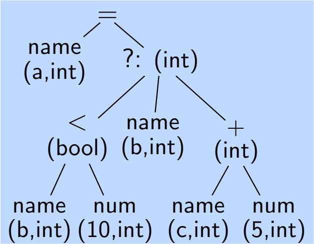
Translation Sequence in Our Compiler: Emitting Instructions

Input

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



Parse Tree



Abstract Syntax Tree
(with attributes)

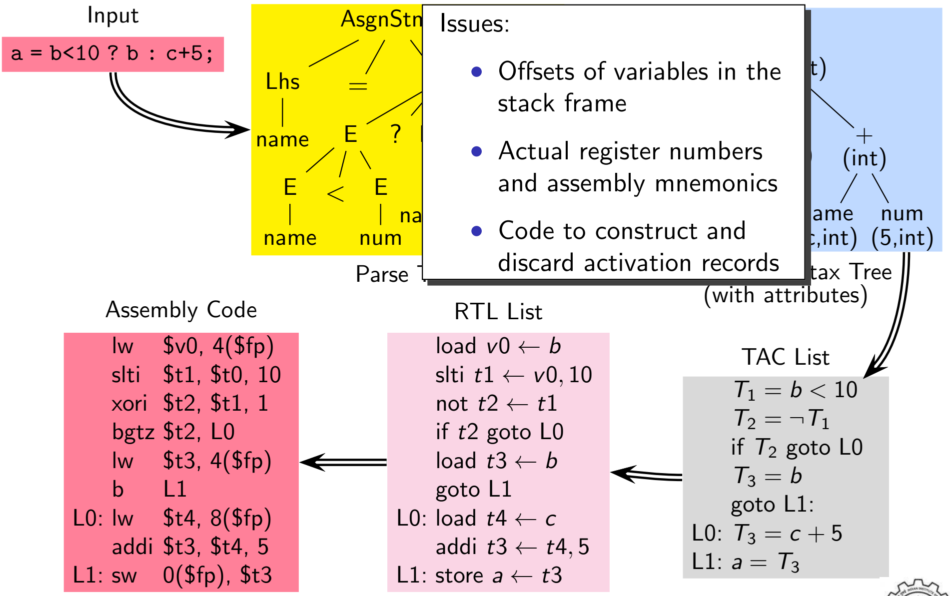
TAC List

$T_1 = b < 10$
$T_2 = \neg T_1$
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



Input

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

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: $a = T_3$

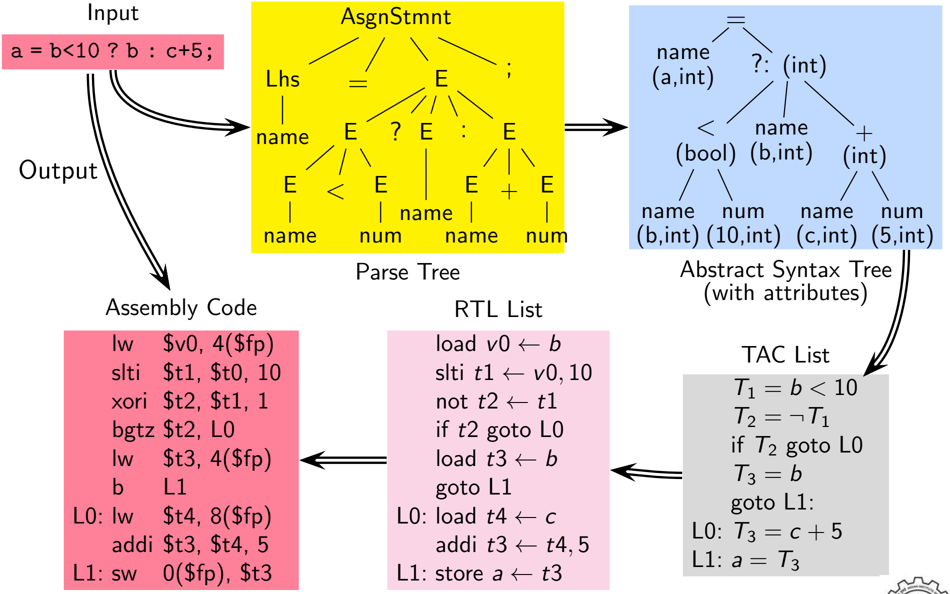
RTL List

load $v0 \leftarrow b$
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



Input

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

Output

Parse Tree

Abstract Syntax Tree (with attributes)

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
```

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
```

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: $a = T_3$

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?

Very few people write compilers any way

Why Is Compiler Construction a Relevant Course?

Very few people write compilers any way

- 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?

Very few people write compilers any way

- 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 abstractions and bridging abstraction gaps

Why Is Compiler Construction a Relevant Course?

Very few people write compilers any way

- 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 abstractions and bridging abstraction gaps
- Knowing compilers internals makes a person a much better programmer
  Writing programs whose data is programs

Part 3

Compilation Models

Compilation Models

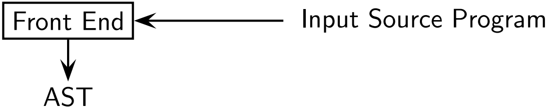
Aho Ullman Model

Davidson Fraser Model

Compilation Models

Aho Ullman Model

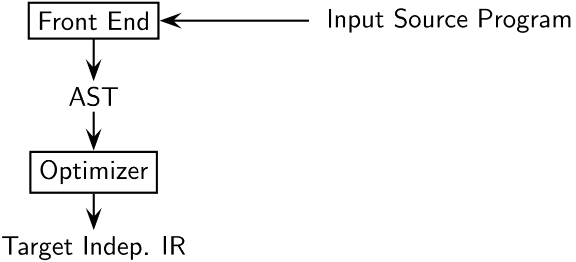
Davidson Fraser Model



Compilation Models

Aho Ullman Model

Davidson Fraser Model



Front End

Input Source Program

AST

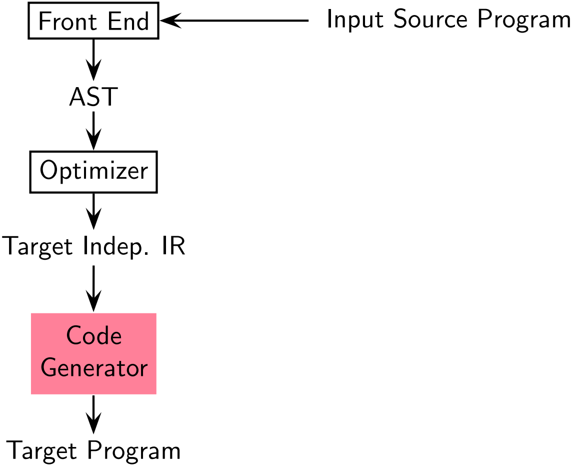
Optimizer

Target Indep. IR

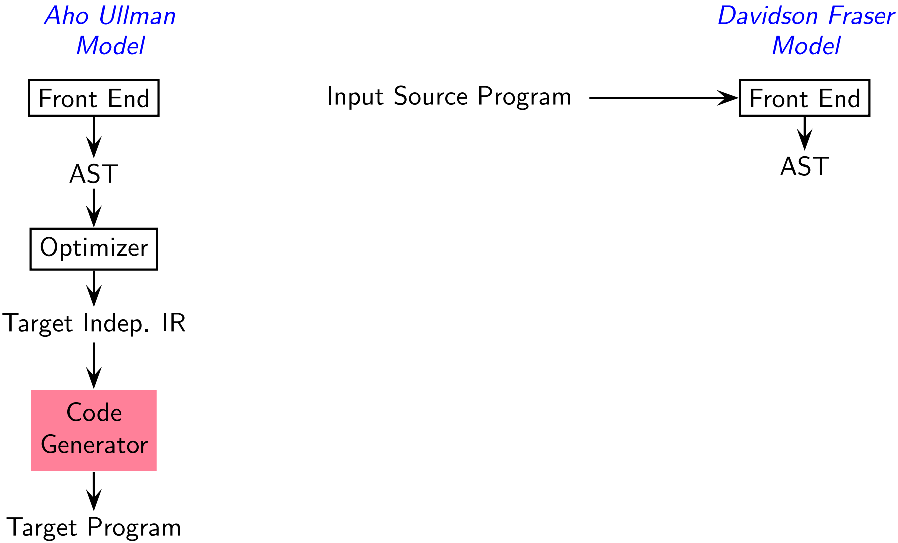
Compilation Models

Aho Ullman Model

Davidson Fraser Model



Compilation Models



Aho Ullman Model

Front End

AST

Optimizer

Target Indep. IR

Code Generator

Target Program

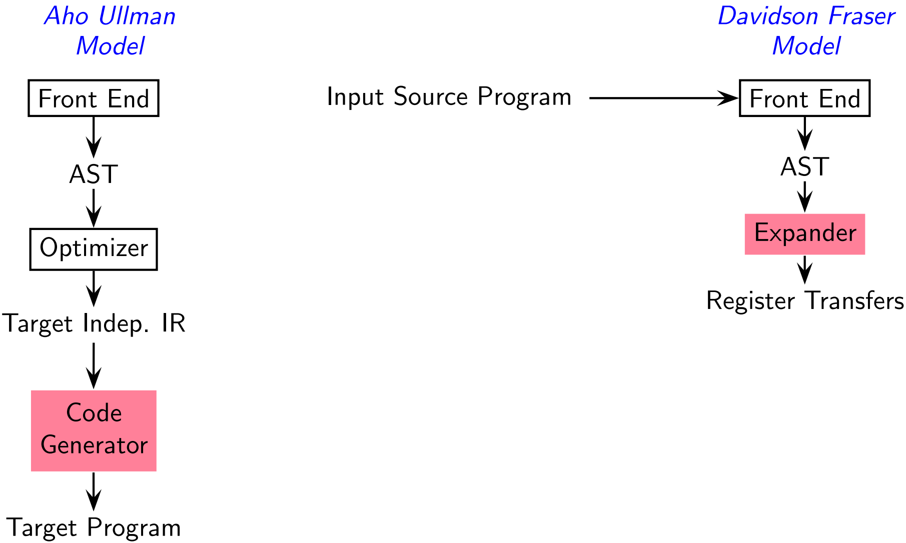
Davidson Fraser Model

Input Source Program

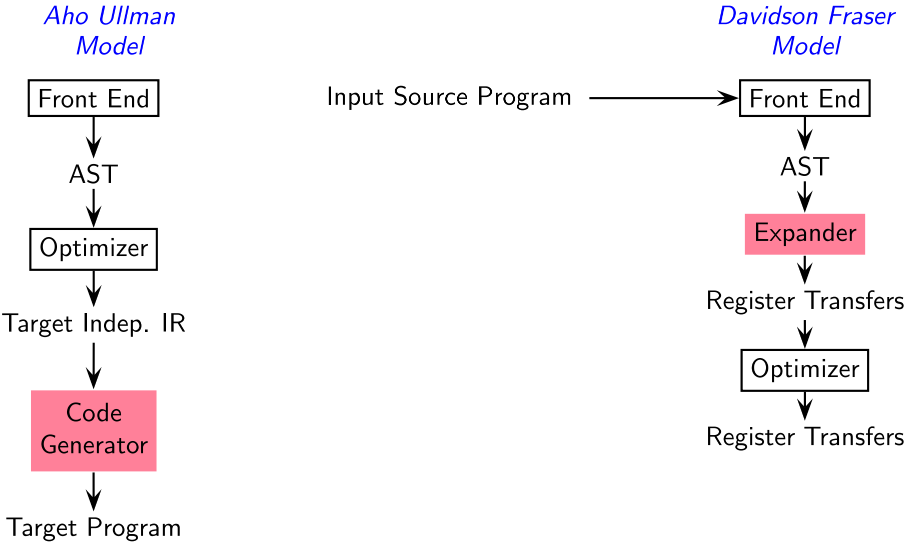
Front End

AST

Compilation Models



Compilation Models



Aho Ullman Model

Front End

AST

Optimizer

Target Indep. IR

Code Generator

Target Program

Input Source Program

Davidson Fraser Model

Front End

AST

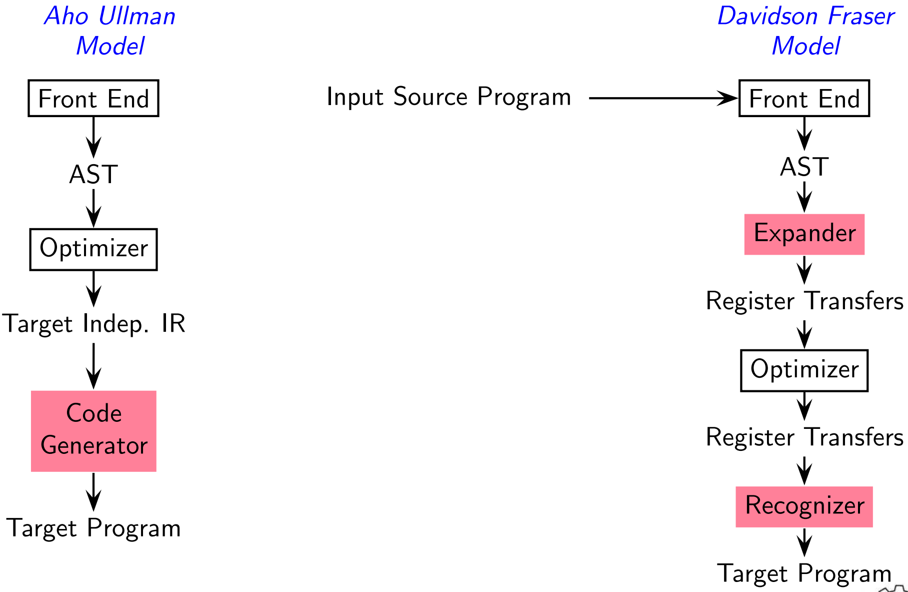
Expander

Register Transfers

Optimizer

Register Transfers

Compilation Models



Aho Ullman Model

Front End

AST

Optimizer

Target Indep. IR

Code Generator

Target Program

Input Source Program

Davidson Fraser Model

Front End

AST

Expander

Register Transfers

Optimizer

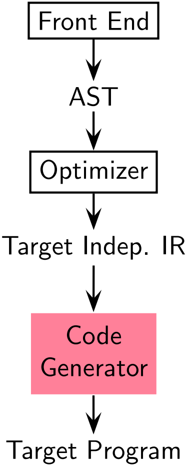
Register Transfers

Recognizer

Target Program

Compilation Models

Aho Ullman Model



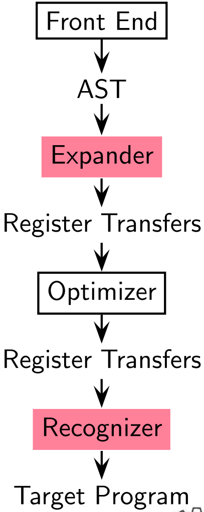
Aho Ullman: Instruction selection

- 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

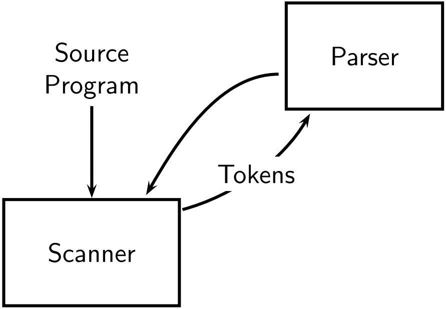
Davidson Fraser Model



Typical Front Ends

Parser

Typical Front Ends



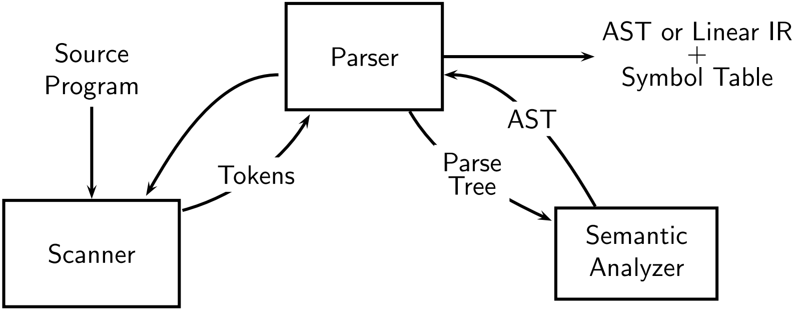
Source Program

Parser

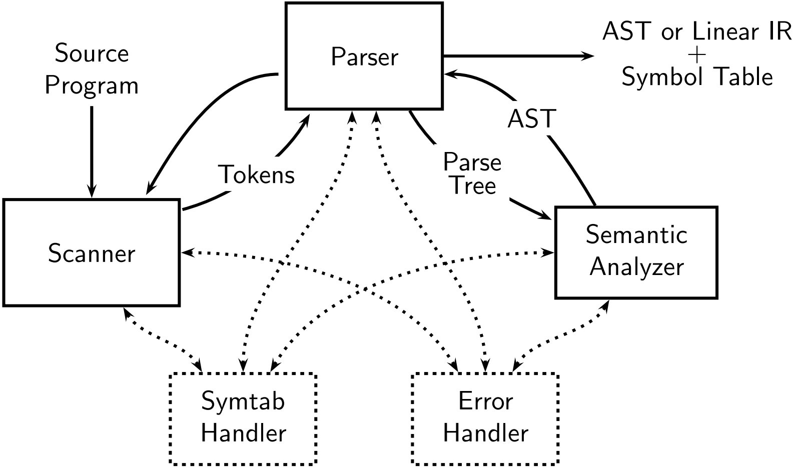
Tokens

Scanner

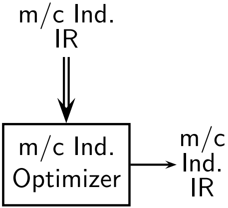
Typical Front Ends



Typical Front Ends

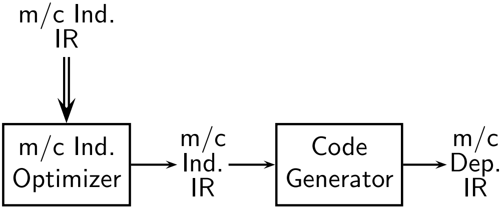


Typical Back Ends in Aho Ullman Model



- Compile time evaluations
- Eliminating redundant computations

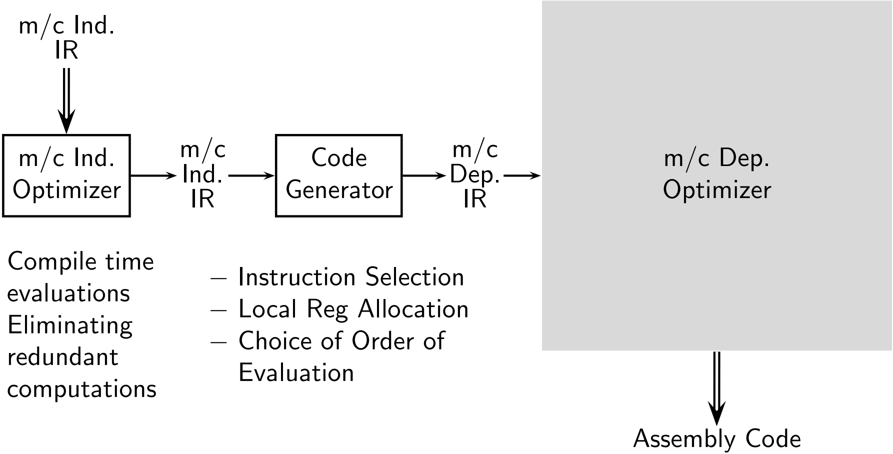
Typical Back Ends in Aho Ullman Model



- Compile time evaluations
- Eliminating redundant computations

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

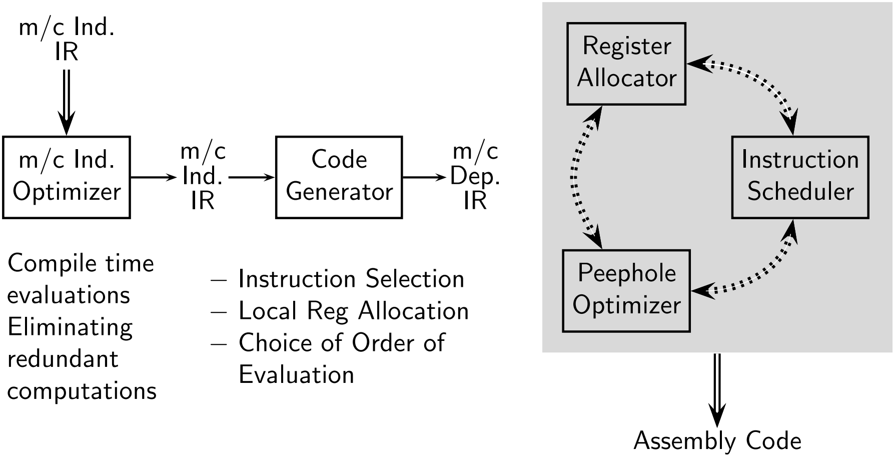
Typical Back Ends in Aho Ullman Model



- Compile time evaluations
- Eliminating redundant computations

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

Typical Back Ends in Aho Ullman Model



- Compile time evaluations
- Eliminating redundant computations

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

Retargetability in Aho Ullman and Davidson Fraser Models

<table>
<tr><th></th><th>Aho Ullman Model</th><th>Davidson Fraser Model</th></tr>
<tr><td rowspan="2">Instruction Selection</td><td colspan="2">• Machine independent IR is expressed in the form of trees
• Machine instructions are described in the form of trees
• Trees in the IR are “covered” using the instruction trees</td></tr>
<tr><td></td><td></td></tr>
<tr><td>Optimization</td><td></td><td></td></tr>
</table>

Retargetability in Aho Ullman and Davidson Fraser Models

<table>
<tr><th></th><th>Aho Ullman Model</th><th>Davidson Fraser Model</th></tr>
<tr><td rowspan="2">Instruction Selection</td><td colspan="2">• Machine independent IR is expressed in the form of trees
• Machine instructions are described in the form of trees
• Trees in the IR are “covered” using the instruction trees</td></tr>
<tr><td>Cost based tree pattern matching</td><td></td></tr>
<tr><td>Optimization</td><td></td><td></td></tr>
</table>

Retargetability in Aho Ullman and Davidson Fraser Models

<table>
<tr><th></th><th>Aho Ullman Model</th><th>Davidson Fraser Model</th></tr>
<tr><td rowspan="2">Instruction Selection</td><td colspan="2">• Machine independent IR is expressed in the form of trees
• Machine instructions are described in the form of trees
• Trees in the IR are “covered” using the instruction trees</td></tr>
<tr><td>Cost based tree pattern matching</td><td>Structural tree pattern matching</td></tr>
<tr><td>Optimization</td><td></td><td></td></tr>
</table>

Retargetability in Aho Ullman and Davidson Fraser Models

<table>
<tr><th></th><th>Aho Ullman Model</th><th>Davidson Fraser Model</th></tr>
<tr><td rowspan="2">Instruction Selection</td><td colspan="2">• Machine independent IR is expressed in the form of trees
• Machine instructions are described in the form of trees
• Trees in the IR are “covered” using the instruction trees</td></tr>
<tr><td>Cost based tree pattern matching</td><td>Structural tree pattern matching</td></tr>
<tr><td>Optimization</td><td>Machine independent</td><td></td></tr>
</table>

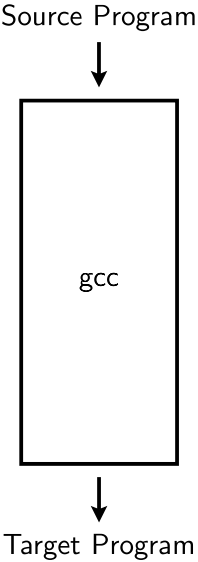
Retargetability in Aho Ullman and Davidson Fraser Models

<table>
<tr><th></th><th>Aho Ullman Model</th><th>Davidson Fraser Model</th></tr>
<tr><td rowspan="2">Instruction Selection</td><td colspan="2">• Machine independent IR is expressed in the form of trees
• Machine instructions are described in the form of trees
• Trees in the IR are “covered” using the instruction trees</td></tr>
<tr><td>Cost based tree pattern matching</td><td>Structural tree pattern matching</td></tr>
<tr><td rowspan="2">Optimization</td><td rowspan="2">Machine independent</td><td>Machine dependent</td></tr>
<tr><td></td></tr>
</table>

Retargetability in Aho Ullman and Davidson Fraser Models

	Aho Ullman Model	Davidson Fraser Model
Instruction Selection	• Machine independent IR is expressed in the form of trees • Machine instructions are described in the form of trees • Trees in the IR are "covered" using the instruction trees	
	Cost based tree pattern matching	Structural tree pattern matching
Optimization	Machine independent	Machine dependent
		Key Insight: *Register transfers are target specific but their form is target independent*

The GNU Tool Chain for C

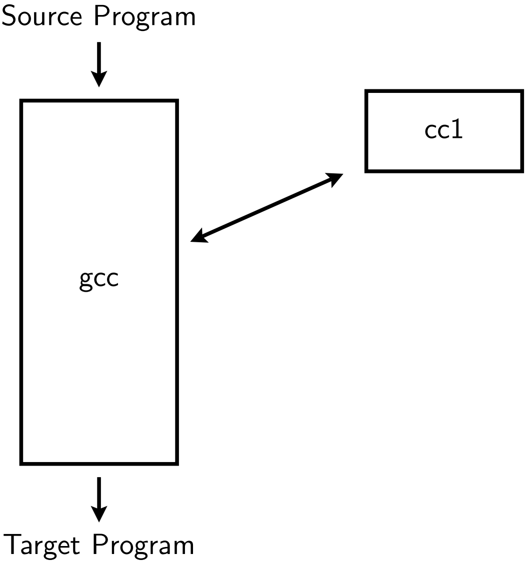


Source Program

gcc

Target Program

The GNU Tool Chain for C



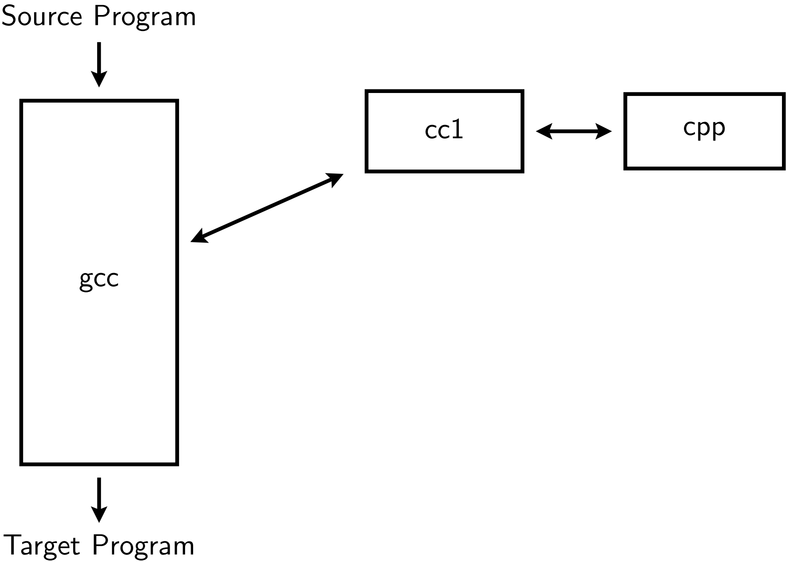
Source Program

gcc

cc1

Target Program

The GNU Tool Chain for C



Source Program

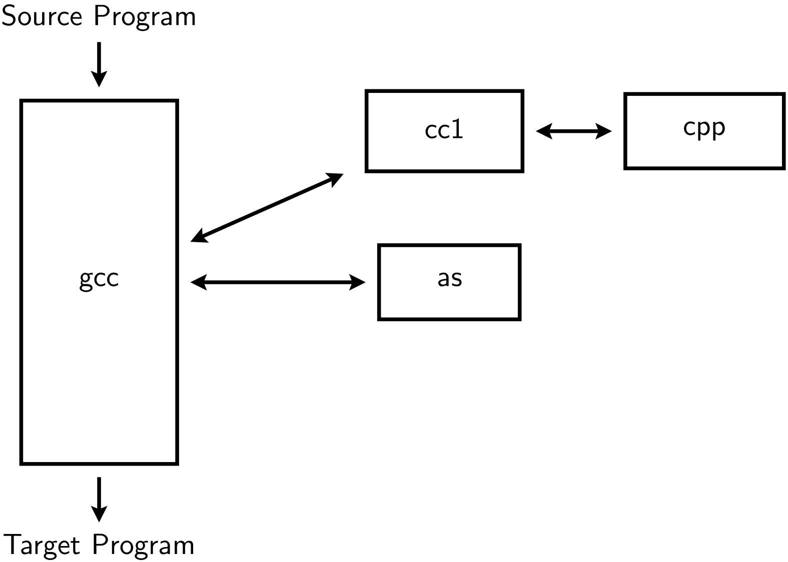
gcc

cc1

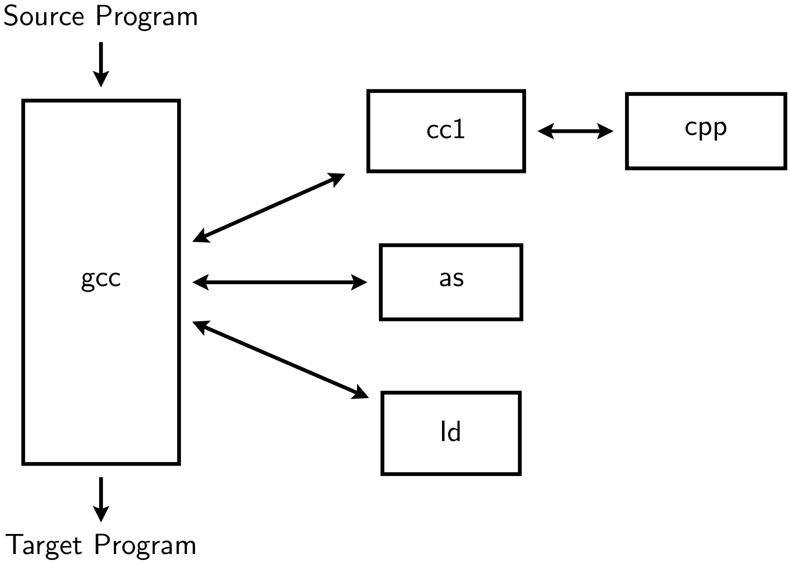
cpp

Target Program

The GNU Tool Chain for C



The GNU Tool Chain for C



Source Program

gcc

cc1

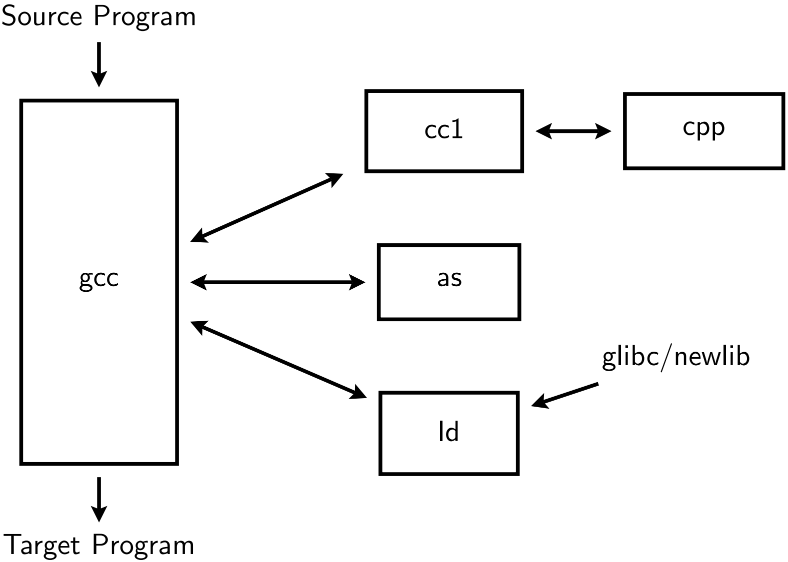
cpp

as

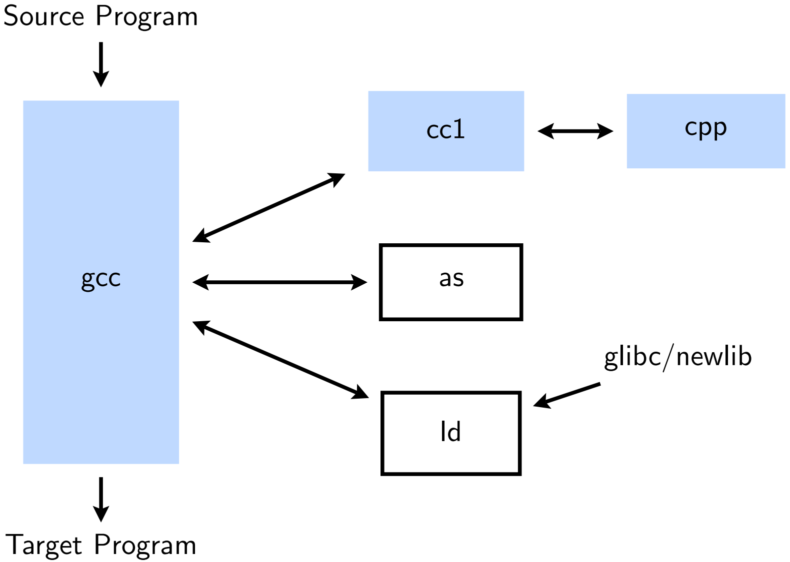
ld

Target Program

The GNU Tool Chain for C



The GNU Tool Chain for C



Source Program

gcc

cc1

cpp

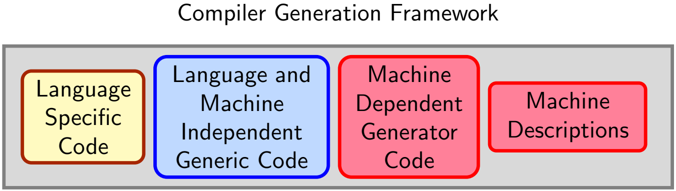
as

ld

glibc/newlib

Target Program

The Architecture of GCC



Compiler Generation Framework

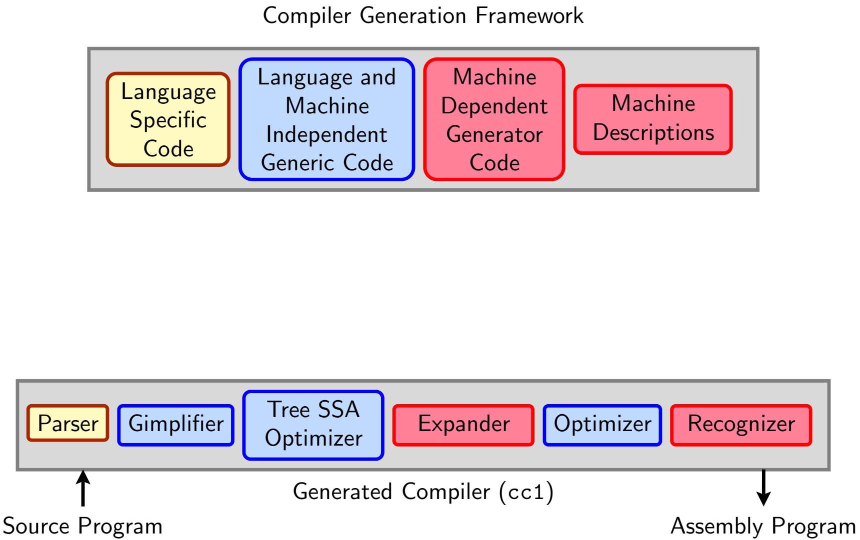
Language Specific Code

Language and Machine Independent Generic Code

Machine Dependent Generator Code

Machine Descriptions

The Architecture of GCC



Compiler Generation Framework

Language Specific Code

Language and Machine Independent Generic Code

Machine Dependent Generator Code

Machine Descriptions

Parser

Gimplifier

Tree SSA Optimizer

Expander

Optimizer

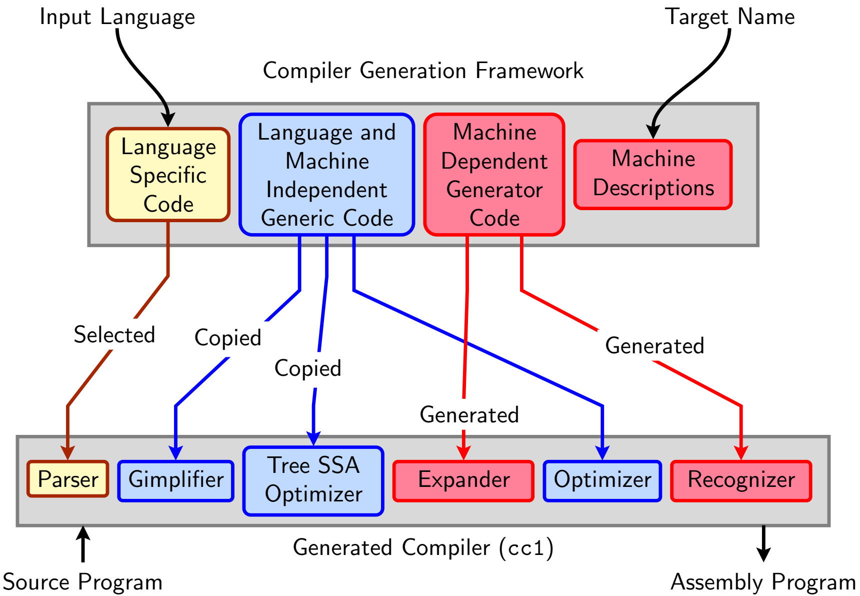
Recognizer

Generated Compiler (cc1)

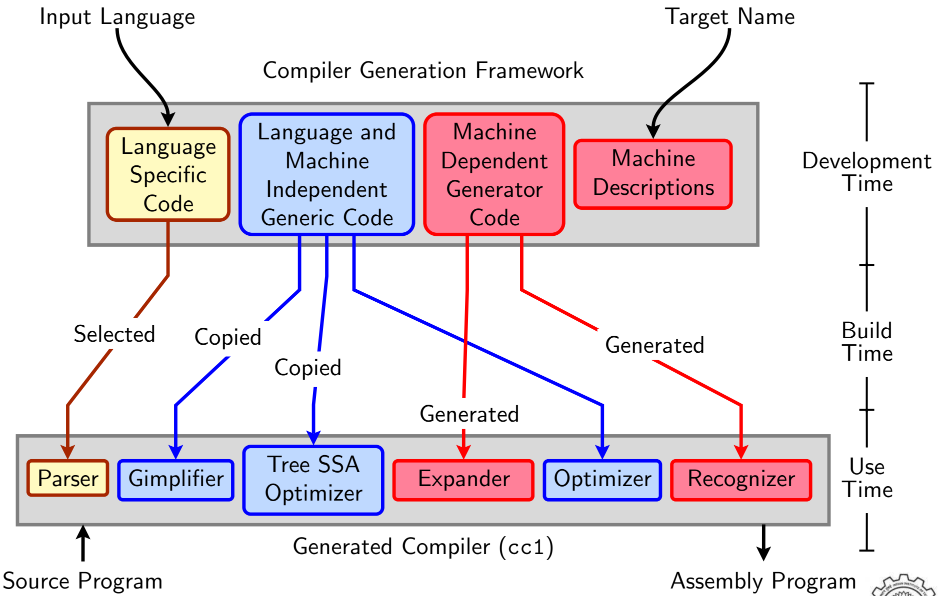
Source Program

Assembly Program

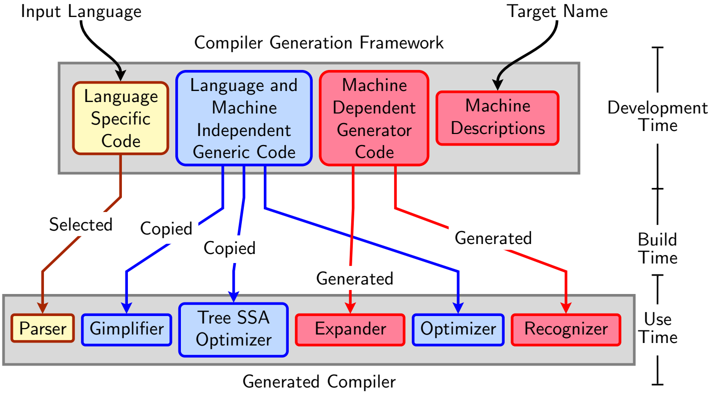
The Architecture of GCC



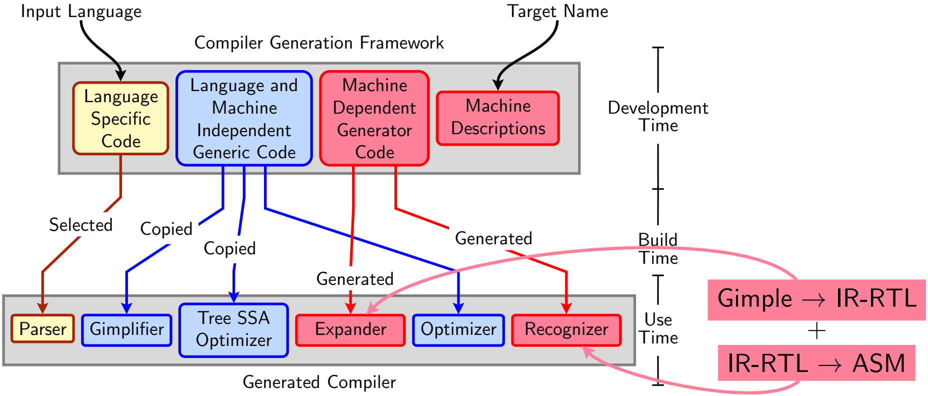
The Architecture of GCC



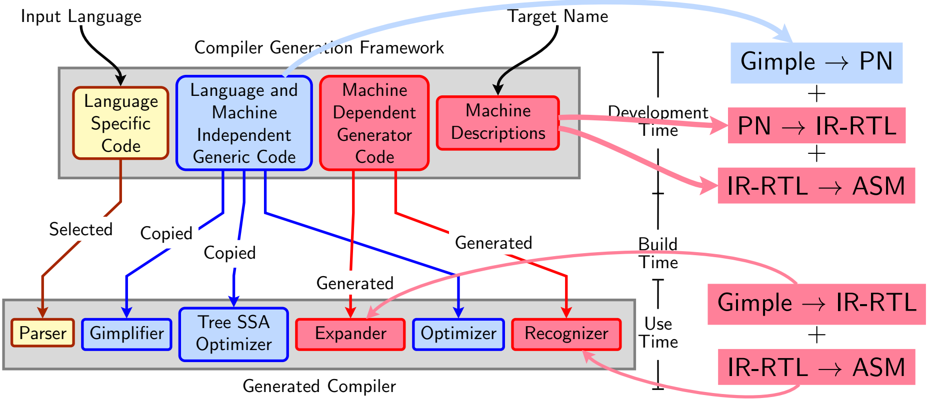
GCC Retargetability Mechanism



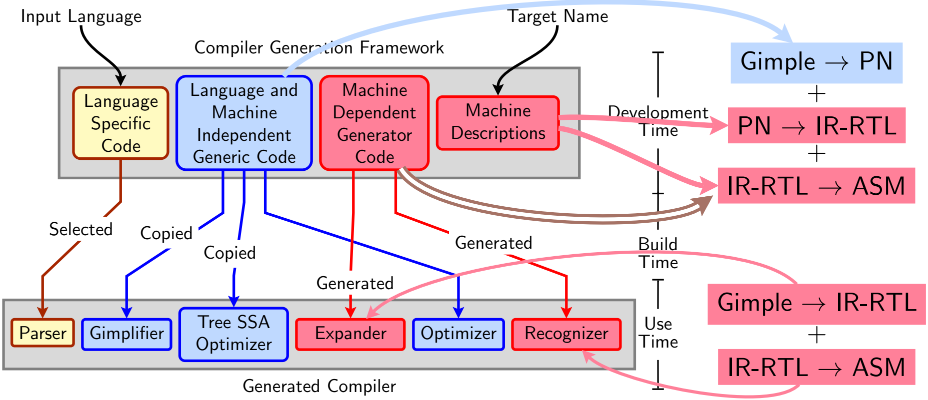
GCC Retargetability Mechanism



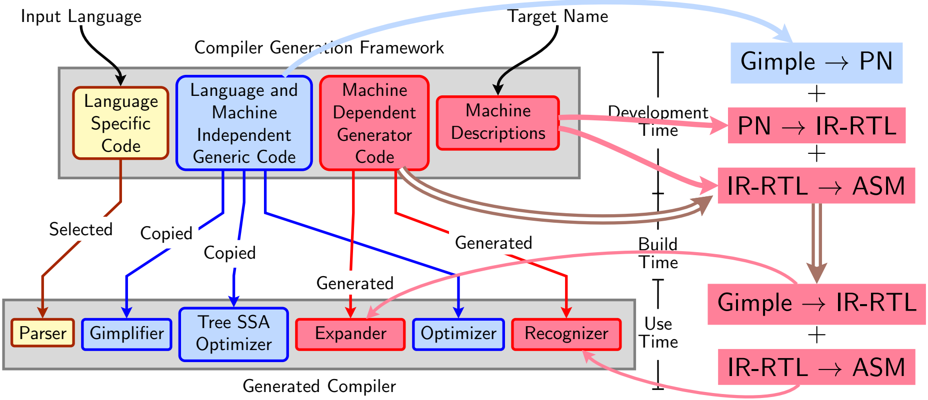
GCC Retargetability Mechanism



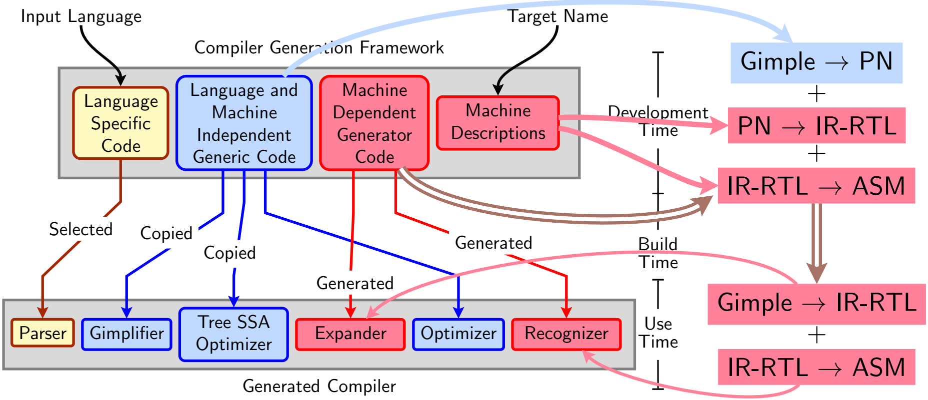
GCC Retargetability Mechanism



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

Part 4

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, Parallism Discovery, Heap Analysis

Modern Challenges: Design issues

- The IR interface

  What to export? What to hide?
- Retargetability

  Extending to the new version of a processor?

  Extending to a new processor?

Modern Challenges: Improving Program

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

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

Modern Challenges: Target Machine Issues

How to exploit

- Pipelines? (Spectre bug)
- Multiple execution units (pipelined)
- Cache hierarchy
- Parallel processing
  (Shared memory, distributed memory, message-passing)
- Data parallelism support in GPUs
- Vector operations
- VLIW and Superscalar instruction issue

Modern Challenges: Target Machine Issues

How to exploit

- Pipelines? (Spectre bug)

The crux of the matter

- Hardware is parallel, (conventional) software is sequential
- Software view of memory model: Strong consistency
  Architecture gives weak consistency
- Software view is stable, hardware is disruptive

Modern Challenges: Providing Guarantees

- Correctness of optimizations
    - Hard even for machine independent optimizations
    - Verification of a production optimizing compiler is a pipe dream
      Requires roving 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

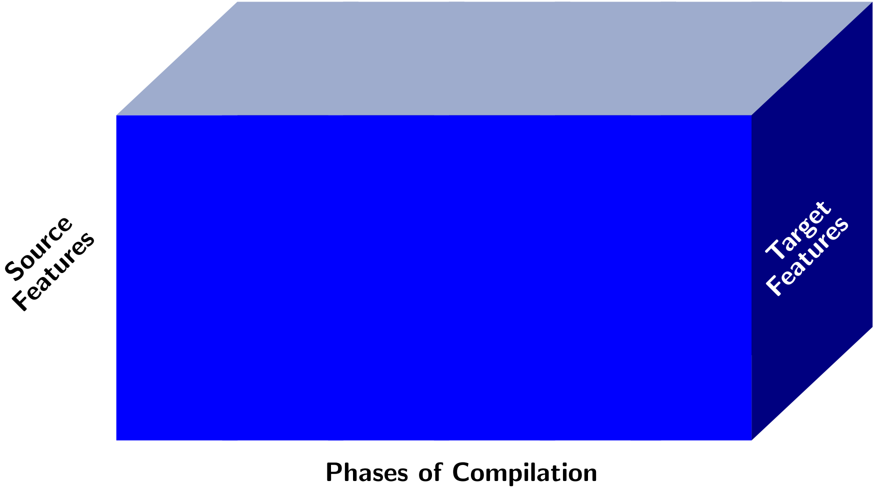
Modern Challenges: New Expectations

- 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?
  - Langueg 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?
  - Can compilers learn from the programs they have compiled and become “better” over time?

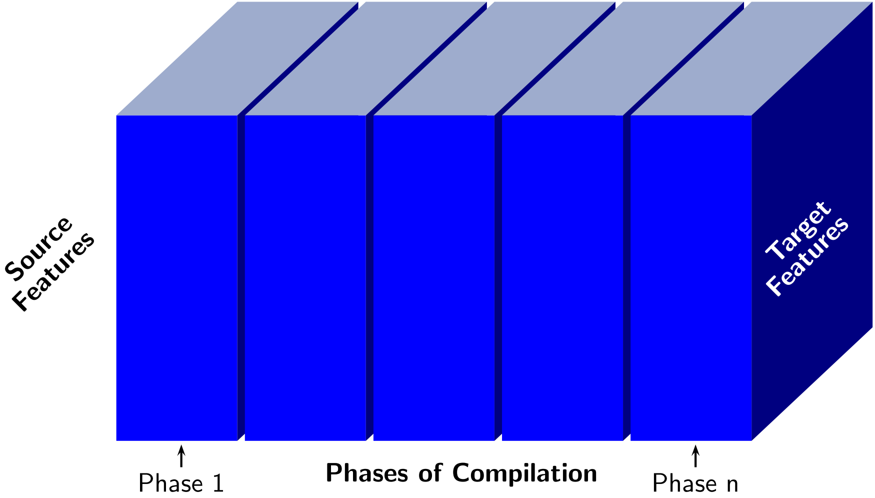
Part 5

Incremental Construction of Compilers

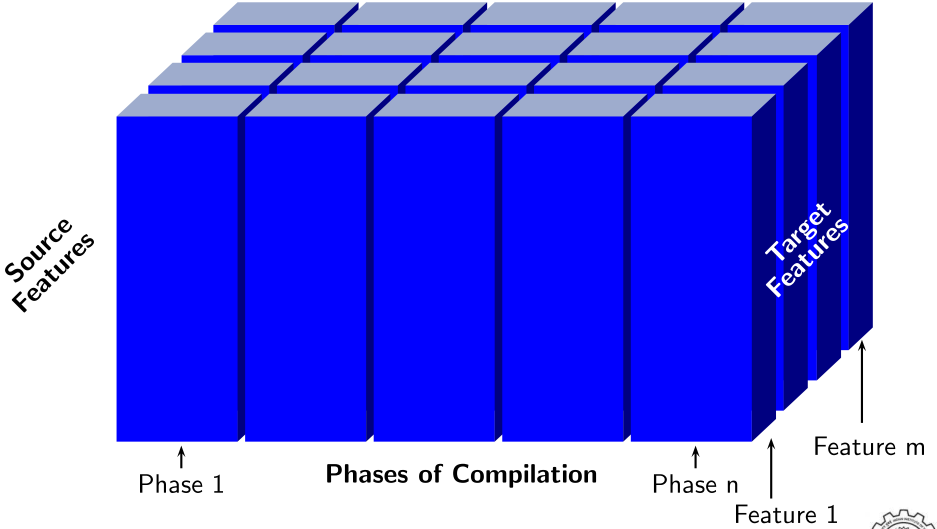
In Search of Modularity in Compilation



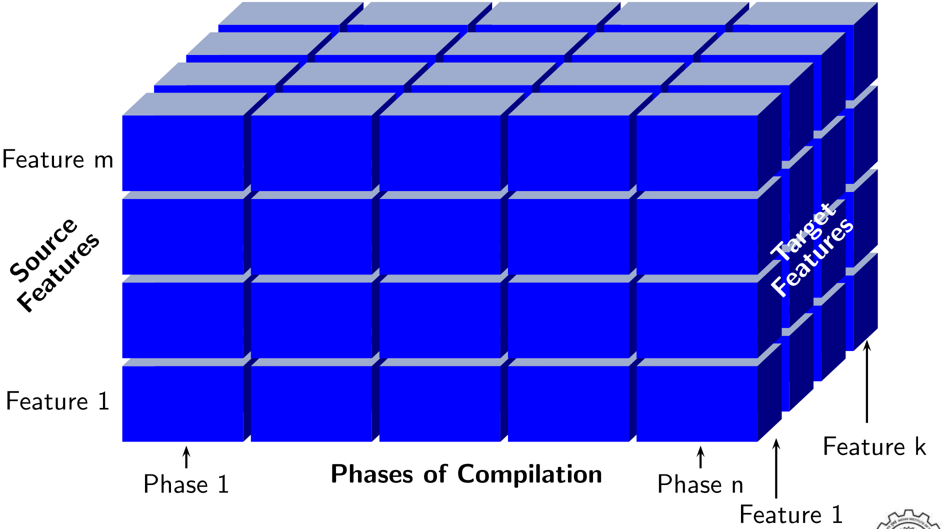
In Search of Modularity in Compilation



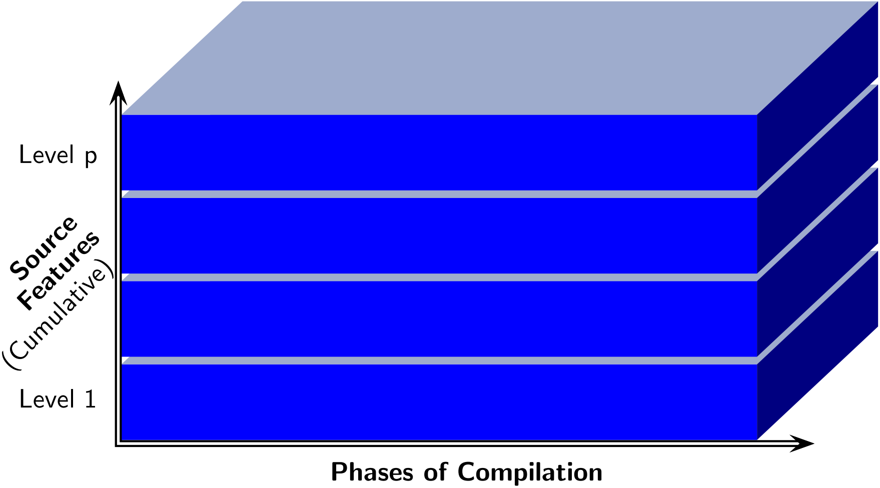
In Search of Modularity in Compilation



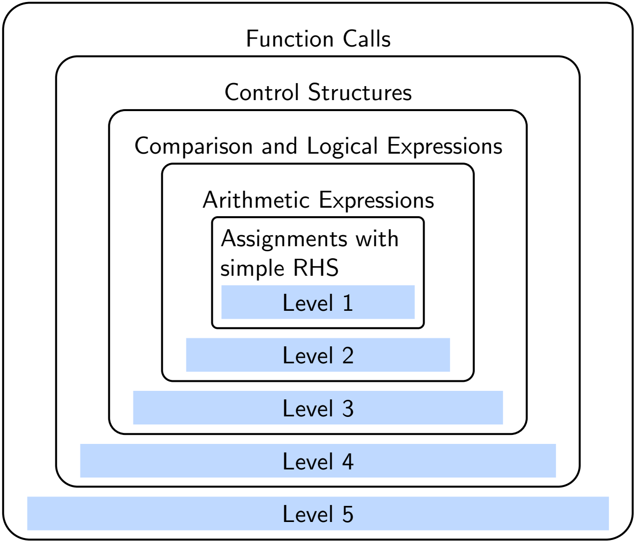
In Search of Modularity in Retargetable Compilation



In Search of Modularity in Compilation



Language Increments



Function Calls

Control Structures

Comparison and Logical Expressions

Arithmetic Expressions

Assignments with simple RHS

Level 1

Level 2

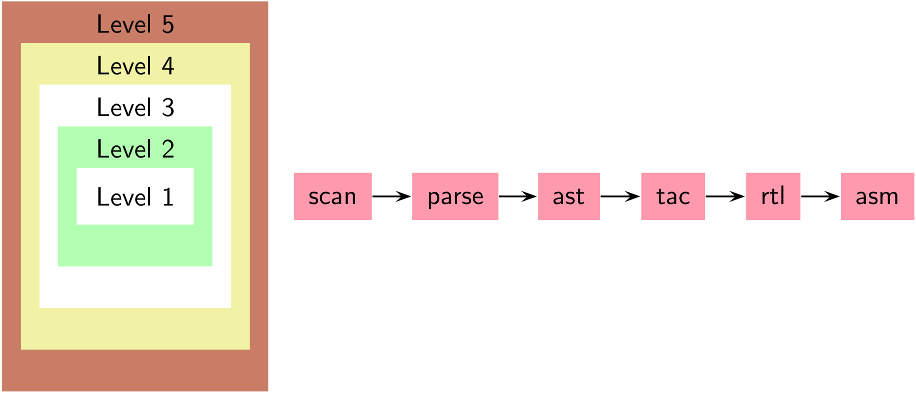
Level 3

Level 4

Level 5

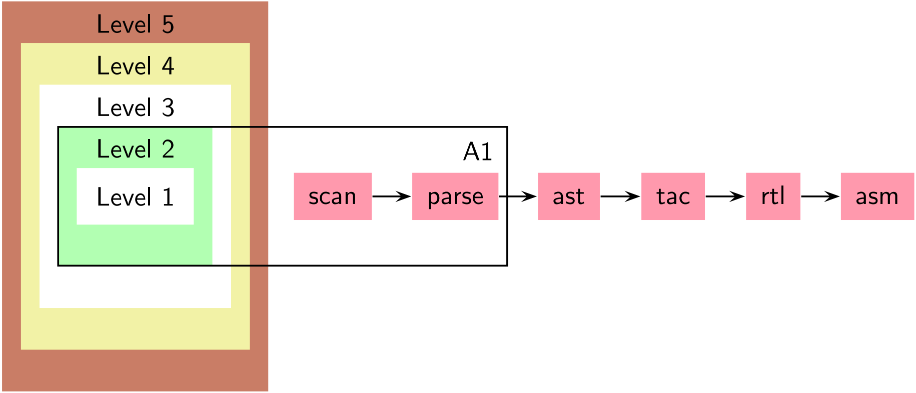
Proposed Assignment Plan

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



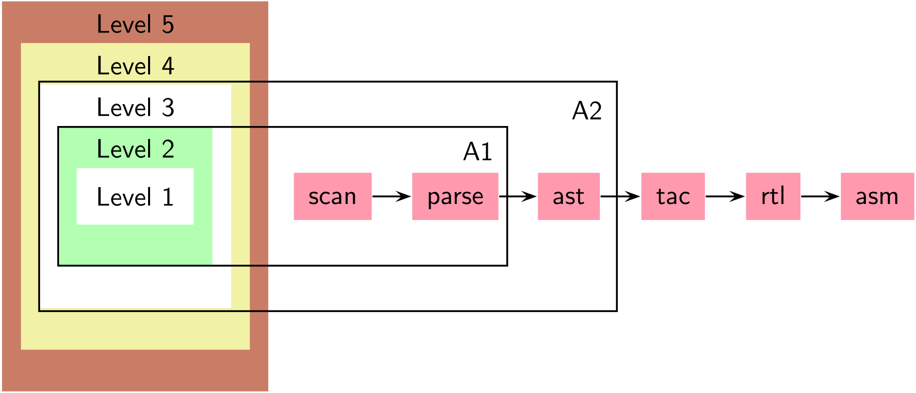
Proposed Assignment Plan

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



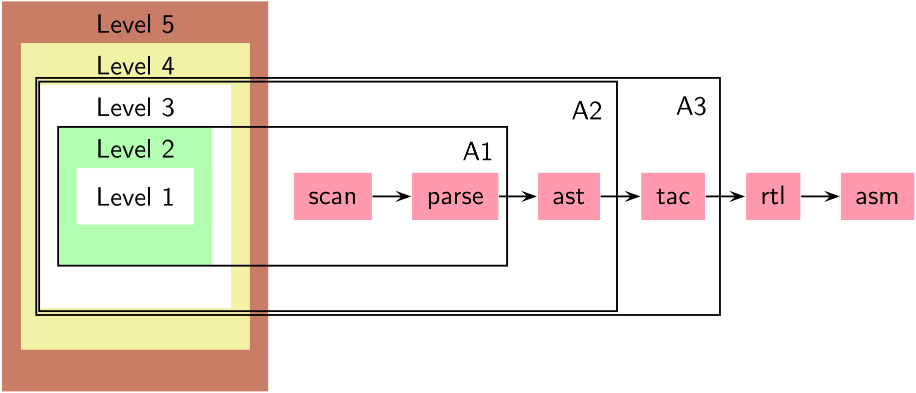
Proposed Assignment Plan

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



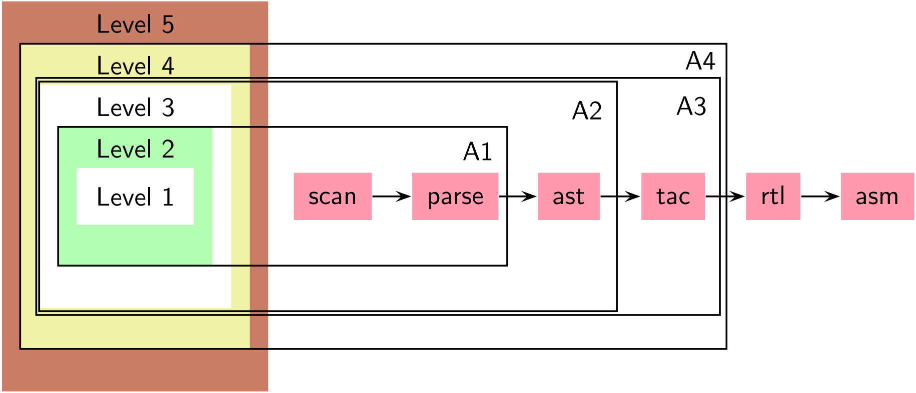
Proposed Assignment Plan

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



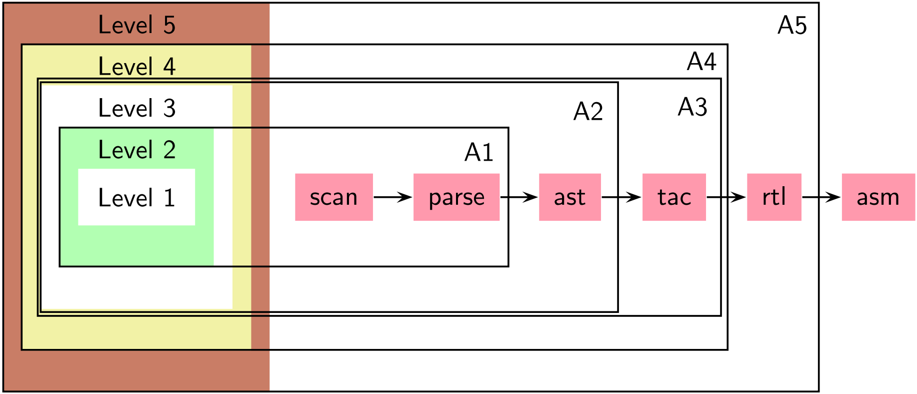
Proposed Assignment Plan

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



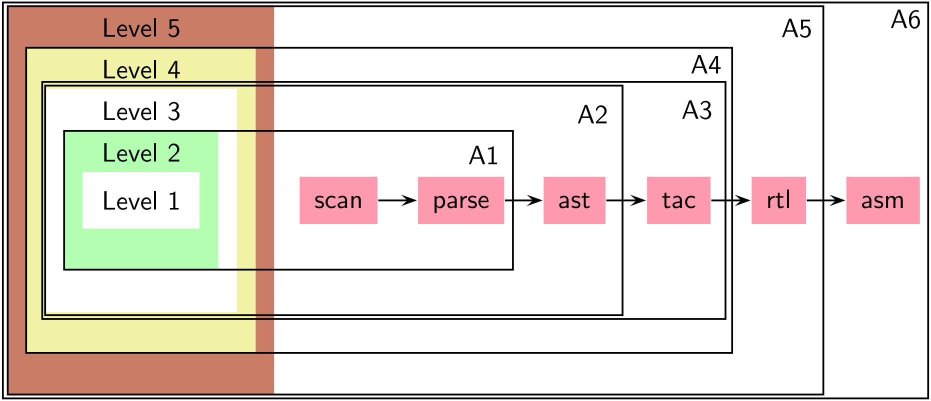
Proposed Assignment Plan

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



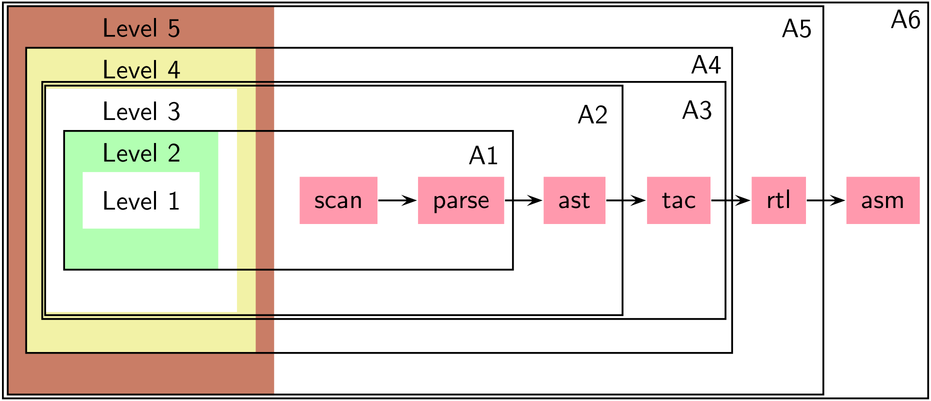
Proposed Assignment Plan

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



Proposed Assignment Plan

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



Further details will be provided in the lab session