An Overview of Compilation

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

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

January 2014
Outline

• Introduction
• compilation sequence
• compilation models
Part 1

Introduction to Compilation
Nothing is known except the problem

![Diagram](image-url)
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

- 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

Diagram showing the progression of binding stages from conceptualisation to loading:

- Conceptualisation
- Coding
- Compiling
- Linking
- Loading

No. of unbound objects decreases over time (Time). Actual addresses of code and data are assigned during the loading stage.
Binding

No. of unbound objects

Conceptualisation  Coding  Compiling  Linking  Loading  Execution

Values of variables

Time
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

Source Program → Input Data → Interpreter → Machine
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

```
Program Specification

Translation

Machine
```
Implementation Mechanisms as “Bridges”

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

Diagram:

```
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

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

Spim Assembly Equivalent

```assembly
lw    $t0, 4($fp) ; t0 <- b      # Is b smaller
slti  $t0, $t0, 10 ; t0 <- t0 < 10 # than 10?
not   $t0, $t0 ; t0 <- !t0
bgtz  $t0, L0: ; if t0>0 goto L0
lw    $t0, 4($fp) ; t0 <- b      # YES
b     L1: ; goto L1
L0: lw $t0, 8($fp) ;L0: t0 <- c  # NO
L1: sw 0($fp), $t0 ;L1: a <- t0
```
High and Low Level Abstractions

Input C statement

\[ a = b < 10 ? b : c; \]

Spim Assembly Equivalent

\[
\begin{align*}
  \text{lw} & \quad $t0, 4($fp) ; \quad t0 \leftarrow b \quad \# \text{Is } b \text{ smaller} \\
  \text{slti} & \quad $t0, $t0, 10 ; \quad t0 \leftarrow t0 < 10 \quad \# \text{than } 10? \\
  \text{not} & \quad $t0, $t0 ; \quad t0 \leftarrow \neg t0 \\
  \text{bgtz} & \quad $t0, L0: ; \quad \text{if } t0>0 \text{ goto } L0 \\
  \text{lw} & \quad $t0, 4($fp) ; \quad t0 \leftarrow b \quad \# \text{YES} \\
  \text{b} & \quad L1: ; \quad \text{goto } L1 \\
  \text{L0: lw} & \quad $t0, 8($fp) ; \quad \text{L0: } t0 \leftarrow c \quad \# \text{NO} \\
  \text{L1: sw} & \quad 0($fp), $t0 ; \quad \text{L1: } a \leftarrow t0
\end{align*}
\]
High and Low Level Abstractions

Input C statement
\[ a = b < 10 ? b : c; \]

Spim Assembly Equivalent

```
lw   $t0, 4($fp) ;   t0 <- b  # Is b smaller
slti $t0, $t0, 10 ;   t0 <- t0 < 10  # than 10?
not  $t0, $t0  ;     t0 <- !t0
bgtz $t0, L0:  ;     if t0>0 goto L0
lw   $t0, 4($fp) ;   t0 <- b  # YES
b    L1:  ;         goto L1
L0: lw   $t0, 8($fp) ;L0: t0 <- c  # NO
L1: sw  0($fp), $t0 ;L1: a <- t0
```

NOT Condition

True Part

False Part
High and Low Level Abstractions

Input C statement
\[ a = b < 10 ? b : c; \]

Spim Assembly Equivalent

```
lw $t0, 4($fp) ; t0 <- b  # Is b smaller
slti $t0, $t0, 10 ; t0 <- t0 < 10  # than 10?
not $t0, $t0 ; t0 <- !t0
bgtz $t0, L0: ; if t0>0 goto L0
lw $t0, 4($fp) ; t0 <- b  # YES
b L1: ; goto L1
L0: lw $t0, 8($fp) ; L0: t0 <- c  # NO
L1: sw 0($fp), $t0 ; L1: a <- t0
```

NOT Condition

Conditional Jump

Fall through

True Part

False Part
Implementation Mechanisms

- Translation $= \text{Analysis} + \text{Synthesis}$
- Interpretation $= \text{Analysis} + \text{Execution}$
Implementation Mechanisms

- Translation = Analysis + Synthesis
  Interpretation = Analysis + Execution

- Translation Instructions $\rightarrow$ Equivalent Instructions
Implementation Mechanisms

- Translation = Analysis + Synthesis
- Interpretation = Analysis + Execution

- Translation Instructions $\rightarrow$ Equivalent Instructions
- Interpretation Instructions $\rightarrow$ Actions Implied by Instructions
Language Implementation Models

Analysis → Synthesis

Analysis → Execution

Synthesis → Compilation

Execution → Interpretation
Language Processor Models

Front End → Optimizer → Back End

Front End → Optimizer → Virtual Machine

C, C++

Java, C#
Part 2

An Overview of Compilation Phases
The Structure of a Simple Compiler
The Structure of a Simple Compiler
The Structure of a Simple Compiler

Front End

- Scanner
- Semantic Analyser
- Parser

Back End

- Instruction Selector
- Register Allocator
- Assembly Emitter

AST

Insn

Source Program

Assembly Program

Uday Khedker
Translation Sequence in Our Compiler: Parsing

\[ a = b < 10 ? b : c; \]

Input
Translation Sequence in Our Compiler: Parsing

Input

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

Parse Tree

```
AsgnStmt

Lhs = E ;

name E ? E : E

E < E name name

name num
```

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;
```

Parse Tree

```
AsgnStmtnt

Lhs = E ;

name E ? E : E

E < E name name

name num
```
Translation Sequence in Our Compiler: Semantic Analysis

Input

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

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

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

Input

Parse Tree

Abstract Syntax Tree (with attributes)
Translation Sequence in Our Compiler: IR Generation

Input:

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

Tree List:

```
T₀ = Not
   /\  <
  / \  b 10
```

Parse Tree:

```
AsgnStmt
   Lhs = E ;
      |   |   |   |
      E ? E : E
      |   |   |   |
      E < E name name
      |   |   |   |
      name num name
```

Abstract Syntax Tree (with attributes):

```
= name (a, int) ?: (int)
  < (bool) name (b, int) name (c, int)
  name (b, int) num (10, int)
```

Issues:

- Convert to maximal trees which can be implemented without altering control flow
  Simplifies instruction selection and scheduling, register allocation etc.

- Linearise control flow by flattening nested control constructs
Translation Sequence in Our Compiler: Instruction Selection

\[ a = b < 10 ? b : c; \]

Input

Tree List

\[
\begin{array}{c}
T_0 = \text{Not} \\
\quad \downarrow \\
\quad b \\
\quad \text{<} \\
\quad 10 \\
\end{array}
\]

\[
\begin{array}{c}
T_0 \rightarrow \text{IfGoto} \\
\quad \downarrow \\
\quad L0: \\
\end{array}
\]

\[
\begin{array}{c}
T_1 = \text{=} \\
\quad \downarrow \\
\quad b \\
\end{array}
\]

\[
\begin{array}{c}
\text{Goto} \\
\quad \uparrow \\
\quad L1: \\
\end{array}
\]

\[
\begin{array}{c}
L0: = \\
\quad \downarrow \\
\quad T_1 \\
\quad \text{=} \\
\quad c \\
\end{array}
\]

\[
\begin{array}{c}
L1: = \\
\quad \downarrow \\
\quad a \\
\quad \text{=} \\
\quad T_1 \\
\end{array}
\]

AsgnStmtnt

\[
\begin{array}{c}
\text{Lhs} = \text{E} \\
\quad \downarrow \\
\quad \text{?} \\
\quad \text{E} \\
\quad \text{:} \\
\quad \text{E} \\
\end{array}
\]

\[
\begin{array}{c}
\text{name} \\
\quad \downarrow \\
\quad \text{E} \\
\quad \text{<} \\
\quad \text{E} \\
\quad \text{name} \\
\quad \text{name} \\
\end{array}
\]

\[
\begin{array}{c}
\text{name} \\
\quad \downarrow \\
\quad (\text{int}) \\
\quad \text{name} \\
\quad \text{name} \\
\quad \text{name} \\
\quad (\text{bool}) \\
\quad \text{name} \\
\quad \text{name} \\
\quad \text{name} \\
\quad (\text{c},\text{int}) \\
\end{array}
\]

\[
\begin{array}{c}
\text{name} \\
\quad \downarrow \\
\quad (\text{b},\text{int}) \\
\end{array}
\]

\[
\begin{array}{c}
\text{name} \\
\quad \downarrow \\
\quad (\text{num}) \\
\quad (10,\text{int}) \\
\end{array}
\]

Abstract Syntax Tree (with attributes)
Translation Sequence in Our Compiler: Instruction Selection

\[ a = b < 10 ? b : c; \]

Input

Tree List

\[ T_0 \quad \text{Not} \quad b \quad 10 \]

Parse Tree

\[ \text{AsgnStmtnt} \]

\[ \begin{array}{c}
\text{Lhs} = E \\
\text{name} E ? E : E \\
E < E \quad \text{name name} \\
E \quad \text{name num} \\
\end{array} \]

\[ \begin{array}{c}
\text{name (a, int)} \quad \text{?: (int)} \\
\quad \text{(bool)} \quad \text{name (b, int)} \\
\quad \text{name (c, int)} \\
\quad \text{name (b, int)} \quad \text{num (10, int)} \\
\end{array} \]

Abstract Syntax Tree (with attributes)

Instruction List

\[ \begin{array}{c}
T_0 \leftarrow b \\
T_0 \leftarrow T_0 < 10 \\
T_0 \leftarrow \neg T_0 \\
\text{if } T_0 > 0 \text{ goto } L0: \\
T_1 \leftarrow b \\
\text{goto } L1: \\
L0: T_1 \leftarrow c \\
L1: a \leftarrow T_1 \\
\end{array} \]

Issues:

- Cover trees with as few machine instructions as possible
- Use temporaries and local registers
Translation Sequence in Our Compiler: Instruction Selection

a = b < 10 ? b : c;

Input

Tree List

\[
T_0 \quad \text{Not} \quad b \quad 10
\]

Parse Tree

\[
Lhs = E \quad ;
\]

\[
ame \quad E \quad ? \quad E \quad : \quad E
\]

\[
E < E \quad \text{name} \quad \text{name}
\]

\[
\text{name} \quad \text{name} \quad \text{num}
\]

Abstract Syntax Tree (with attributes)

\[
\text{name} \quad (\text{a}, \text{int}) \quad ? : (\text{int}) \quad < \quad \text{name} \quad (\text{b}, \text{int}) \quad \text{name} \quad (\text{c}, \text{int})
\]

Instruction List

\[
T_0 \leftarrow b
\]

\[
T_0 \leftarrow T_0 < 10
\]

\[
T_0 \leftarrow \neg T_0
\]

if \( T_0 > 0 \) goto L0:

\[
T_1 \leftarrow b
\]

goto L1:

\[
L0: \quad T_1 \leftarrow c
\]

\[
L1: \quad a \leftarrow T_1
\]

Issues:

- Cover trees with as few machine instructions as possible
- Use temporaries and local registers
Translation Sequence in Our Compiler: Emitting Instructions

Input

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

AsgnStmtnt

Parse Tree

Abstract Syntax Tree (with attributes)

Instruction List

```
T_0 ← b
T_0 ← T_0 < 10
T_0 ← ! T_0
if T_0 > 0 goto L0:
T_1 ← b
goto L1:
L0: T_1 ← c
L1: a ← T_1
```
Translation Sequence in Our Compiler: Emitting Instructions

Input

AsgnStmnt

Lhs = E
E ? E : E
E < E
name (a, int)
? : (int)
< (bool)
name (b, int)
name (c, int)

Parse Tree

= name (a, int)
? : (int)
< (bool)
name (b, int)
num (10, int)

Abstract Syntax Tree (with attributes)

Tree List

T₀ Not
b 10

IfGoto

T₀ L0:

= T₁ b

Goto

L1: T₁ c

L0: = T₁

L1: = a T₁

Issues:

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

Instruction List

[T₀ ← b]
[T₀ ← T₀ < 10]
[T₀ ← ! T₀]
if [T₀ > 0 goto L0:]
[T₁ ← b]
goto L1:
L0: [T₁ ← c]
L1: [a ← T₁]

Assembly Code

lw $t0, 4($fp)
slti $t0, $t0, 10
not $t0, $t0
bgtz $t0, L0:
lw $t0, 4($fp)
b L1:
L0: lw $t0, 8($fp)
b L1:
L1: sw 0($fp), $t0
Part 3

Compilation Models
Compilation Models

Aho Ullman Model

Davidson Fraser Model
Compilation Models

Aho Ullman Model

Front End

AST

Input Source Program

Davidson Fraser Model
Compilation Models

Aho Ullman Model

Front End

AST

Optimizer

Target Indep. IR

Davidson Fraser Model

Input Source Program
Compilation Models

Aho Ullman Model

Input Source Program

Front End

AST

Optimizer

Target Indep. IR

Code Generator

Target Program

Davidson Fraser Model
Compilation Models

Aho Ullman Model

1. Front End
2. AST
3. Optimizer
4. Target Indep. IR
5. Code Generator
6. Target Program

Davidson Fraser Model

1. Front End
2. AST
3. Input Source Program

Uday Khedker
IIT Bombay
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

Expander

Register Transfers
Compilation Models

**Aho Ullman Model**

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

**Davidson Fraser Model**

- Front End
- AST
- Expander
- Register Transfers
- Optimizer
- Register Transfers

Input Source Program
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
- Expander
- Register Transfers
- Optimizer
- Register Transfers
- Recognizer
- Target Program
Compilation Models

**Aho Ullman Model**
- Front End
  - AST
  - Optimizer
- Target Indep. IR
- Code Generator
- Target Program

**Davidson Fraser Model**
- Front End
  - AST
  - Expander
  - Register Transfers
  - Optimizer
  - Register Transfers
  - Recognizer
- Target Program

**Aho Ullman: Instruction selection**
- over optimized IR using
decision based tree tiling matching

**Davidson Fraser: Instruction selection**
- over AST using
decision full tree matching based
algorithms that generate
- naive code which is
target dependent, and is
optimized subsequently
Typical Front Ends

Parser
Typical Front Ends

Source Program → Scanner → Tokens → Parser
Typical Front Ends

Source Program

Scanner

Parser

Tokens

Parse Tree

AST

AST or Linear IR + Symbol Table

Semantic Analyzer
Typical Front Ends

Source Program -> Scanner

Parser

AST or Linear IR + Symbol Table

Tokens -> AST

Parse Tree

Semantic Analyzer

Symtab Handler

Error Handler
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

m/c Ind. 
IR

m/c Ind. 
Optimizer

m/c Ind. 
IR

Code Generator

m/c Dep. 
IR

m/c Dep. 
Optimizer

Assembly Code
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>
<thead>
<tr>
<th></th>
<th>Aho Ullman Model</th>
<th>Davidson Fraser Model</th>
</tr>
</thead>
</table>
| **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 |                                                            |
| **Optimization**     |                                                                                  |                                                            |
## Retargetability in Aho Ullman and Davidson Fraser Models

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

- **Aho Ullman Model**
- **Davidson Fraser Model**

---

Uday Khedker

IIT Bombay
## Retargetability in Aho Ullman and Davidson Fraser Models

<table>
<thead>
<tr>
<th>Instruction Selection</th>
<th>Aho Ullman Model</th>
<th>Davidson Fraser Model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Machine independent IR is expressed in the form of trees</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Machine instructions are described in the form of trees</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Trees in the IR are “covered” using the instruction trees</td>
<td></td>
</tr>
<tr>
<td>Cost based tree pattern matching</td>
<td></td>
<td>Structural tree pattern matching</td>
</tr>
<tr>
<td>Optimization</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Retargetability in Aho Ullman and Davidson Fraser Models

<table>
<thead>
<tr>
<th>Instruction Selection</th>
<th>Aho Ullman Model</th>
<th>Davidson Fraser Model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Machine independent IR is expressed in the form of trees</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Machine instructions are described in the form of trees</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Trees in the IR are “covered” using the instruction trees</td>
<td></td>
</tr>
<tr>
<td>Optimization</td>
<td>Cost based tree pattern matching</td>
<td>Structural tree pattern matching</td>
</tr>
<tr>
<td></td>
<td>Machine independent</td>
<td></td>
</tr>
</tbody>
</table>
# Retargetability in Aho Ullman and Davidson Fraser Models

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

Uday Khedker

IIT Bombay
## Retargetability in Aho Ullman and Davidson Fraser Models

<table>
<thead>
<tr>
<th></th>
<th>Aho Ullman Model</th>
<th>Davidson Fraser Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instruction Selection</strong></td>
<td>• Machine independent IR is expressed in the form of trees</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Machine instructions are described in the form of trees</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Trees in the IR are “covered” using the instruction trees</td>
<td></td>
</tr>
<tr>
<td><strong>Cost based tree pattern matching</strong></td>
<td></td>
<td>Structural tree pattern matching</td>
</tr>
<tr>
<td><strong>Optimization</strong></td>
<td>Machine independent</td>
<td>Machine dependent</td>
</tr>
<tr>
<td></td>
<td>Key Insight: <em>Register transfers are target specific but their form is target independent</em></td>
<td></td>
</tr>
</tbody>
</table>